Deep Software Variability: Towards Handling Cross-Layer Configuration

Luc Lesoil, Mathieu Acher, Arnaud Blouin, Jean-Marc Jézéquel

Deep Software Variability: Towards Handling Cross-Layer Configuration

Luc Lesoil
Univ Rennes, CNRS, Inria, IRISA

Mathieu Acher
Univ Rennes, CNRS, Inria, IRISA

Arnaud Blouin
Univ Rennes, INSA Rennes, CNRS, Inria, IRISA

Jean-Marc Jézéquel
Univ Rennes, CNRS, Inria, IRISA

ABSTRACT

Configuring software is a powerful means to reach functional and performance goals of a system. However, many layers (hardware, operating system, input data, *etc.*), themselves subject to variability, can alter performances of software configurations. For instance, configurations' options of the x264 video encoder may have very different effects on x264's encoding time when used with different input videos, depending on the hardware on which it is executed. In this vision paper, we coin the term **deep software variability** to refer to the interaction of all external layers modifying the behavior or non-functional properties of a software. Deep software variability challenges practitioners and researchers: the combinatorial explosion of possible executing environments complicates the understanding, the configuration, the maintenance, the debug, and the test of configurable systems. There are also opportunities: harnessing all variability layers (and not only the software layer) can lead to more efficient systems and configuration knowledge that truly generalizes to any usage and context.

ACM Reference Format:
Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. Deep Software Variability: Towards Handling Cross-Layer Configuration. *In 15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21), February9–11, 2021, Krems, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3442391.3442402

ACKNOWLEDGMENTS

This research was funded by the ANR-17-CE25-0010-01 VaryVary project.

1 INTRODUCTION

Software systems can often be configured to reach specific functional and performance goals, either statically at compile time or through the choice of command line options at runtime. Beyond having a strong impact on non-functional properties [12], this configuration stage may introduce bugs [1, 4, 45], or even prevent the software from starting altogether [42]. Several researchers have been investigating the use of testing or statistical learning to determine whether a specific configuration would raise an error [36, 44], or to predict performances [25, 32], typically in a controlled environment. However, it has been shown that the software environment (in a broad sense, see Figure 1) can interact with its configuration, thus changing its validity [7] or its performances [10, 16].

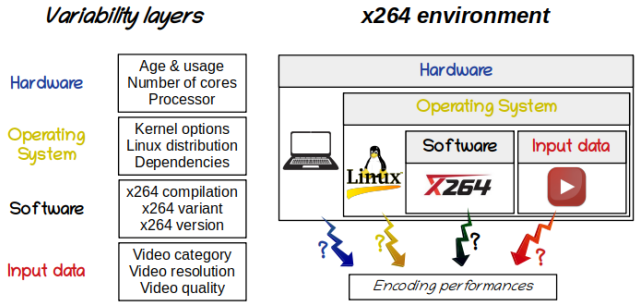


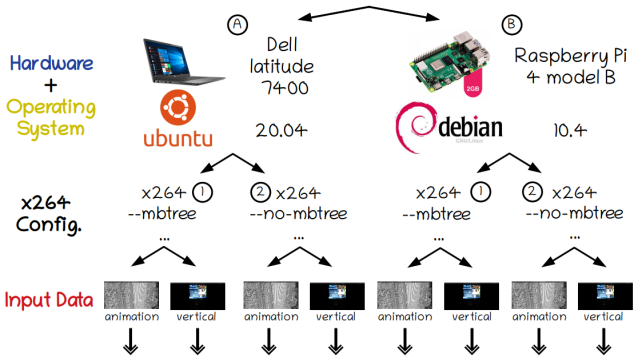
Figure 1: *x264*'s deep variability

That is, only considering the software layer might hide important bugs or provide non-optimal values for configurations' options of the software. Environment properties (hardware, operating system, *etc.*) can be organized into variability layers, as those depicted on Figure 1. We call **deep software variability** the interaction of all variability layers that could modify the behavior or non-functional properties of a software.

We illustrate this concept of deep software variability with concrete examples in Section 2. We emphasize the challenges and opportunities of configuring a software w.r.t. this deep software variability in Section 3. We highlight existing related work in Section 4.

2 EXAMPLE

Let us illustrate the concept of deep software variability with *x264*, a popular, highly-configurable video encoder. In Figure 2, we show the duration of the video compression (in seconds) and the size of the encoded video (in *MB*), for two configurations[1] under two different environments[2]. Performances were measured with two different input videos[3] and the same version of *x264*[4]. The remainder of this section describes five challenges related to *x264*'s deep variability.



Duration (s)	6	22	6	25	73	351	72	359
Size (MB)	33	28	21	34	33	28	21	34

Figure 2: Combinatorial explosion of cross-layer configurations for measuring *x264*'s deep variability

2.1 Impacts of variability layers

According to Figure 2, changing environment *A* for environment *B* would make the compression approximately *15* times longer. However, the compression durations of configurations *1* and 2 remain stable, and roughly proportional between *A* and *B* for both videos. There are more complicated cases, where the **variability layers change the relative importance of configuration options**. For an example, switching from configuration *1* to configuration *2* (*i.e.*, deactivating the feature *mbtree*) decreases the size of the encoded *animation* video ($33MB \searrow 21MB$), while it increases the size of the encoded *vertical* video ($28MB \nearrow 34MB$). It becomes difficult to determine the effect that *mbtree* may have on the size of the output video. Similar results have been reported in the literature [3, 16, 38]: hardware, input data, software version (or a combination thereof) can impact performance properties of *x264*.

[1] *1* : *mbtree* activated, *2* : *mbtree* deactivated

[2] *A* : Ubuntu *20.04* LTS on a Dell Latitude *7400*, *B* : Debian *10.4* on a Raspberri Pi 4 model B

[3] *animation* for the $Animation\_1080P-3d67.mkv$ video, *vertical* for the $VerticalVideo\_1080P-2195.mkv$ video, both extracted from the Youtube User General Content Dataset [40]

[4] *x264*version 0.155.2917, git commit 0a84d98

2.2 Deep testing and benchmarking

To go further than the example of Figure 2, let us consider a general *x264*'s performance model that has to work in environments *A* and *B*, but for *10* input videos and *20* boolean features of *x264*; assuming that each video is compressed in ten seconds, testing exhaustively the *21* millions of possible configurations would take about *8* months of computation. Regardless of the time needed to compute these measurements, it would require both **a lot of human and computational resources to test multiple environments** and a framework allowing to measure performances on an higher level than the software layer (*e.g.*, automatically testing multiple workloads and input videos).

2.3 Improve documentation

Occasionally, requirements between layers can break the build of *x264*. For instance, manually compiling *x264*(version 0.152.2854, git commit e9a5903) on Ubuntu *18.04* LTS requires the upgrade of the Linux package *nasm* (version $>$ *2.13*)[5], unless one deactivates *asm* during the compilation (*i.e.*, specifying the configuration option *–disable-asm*). To the best of our knowledge, this **information is neither centralized nor included in the official documentation** ; we have to search in forums, where developers gave the answer to this problem. Add: (1) software dependencies (2) the effects of a variability layer on software performances would certainly improve *x264*'s documentation.

2.4 Generalize the configuration knowledge

Between environments *A* and *B*, both the hardware and its operating system are changed. What would be *x264*'s performances on the environment *C*, composed of *A*'s hardware and *B*'s operating system? Can we estimate them without measuring configurations in environment C (*i.e.*, just with measurements of environments *A* and *B*)? As long as we are unable to quantify the marginal effect of each layer, **changing a single property of the environment makes us incapable of predicting *x264*'s performance**.

2.5 Cross-layer tuning

By generalizing the measures of Figure 2 on other environments (*cf.* Section 2.2), a streaming website could then automatically choose an environment, and configure a server fully dedicated to the encoding of the *animation* video. In order to achieve this specialization, experts would have to **fix the layers' variability**, selecting the optimal configuration corresponding to this setting, and hopefully increasing the overall performances of *x264*.

[5] We cross-checked the information provided in this webpage

3 CHALLENGES AND OPPORTUNITIES

We propose to break the deep software configuration challenge down to five steps:

(1) **Identify** influential variability layers that impact software's non-functional properties
(2) **Test** software's performances in multiple environments
(3) **Improve knowledge** about how configurable systems interact with their environments
(4) **Transfer** performances across environments and build performance models robust to environment changes
(5) **Specialize** the environment for the software, *i.e.*, configure each layer to improve software performances

For each step, we describe the current problems encountered by users or developers, the challenges that researchers could tackle, and the potential benefits of facing them.

3.1 Impacts of variability layers

We identify at least four potential variability layers:

- **Hardware**: material part of the computer on which the software is executed. *Examples of properties*: the computer or server model and brand, the amount and type of Random-Access Memory available or the Graphics Processing Unit frequency and memory, *etc.*
- **Operating system**: the operating system and dependencies that could be used by the software. *Examples of properties* : the distribution of an operating system, its version, the linux kernel version and modules, the packages that are installed on the operating system, *etc.*
- **Input data**: the raw data fed and processed by the software. *Examples of properties*: a video for *x264*, a word or a sentence for an online translator, a C program for a compiler, *etc.*
- **Software**: how the software was configured and built. *Examples of properties:* any configuration option (*e.g.*, *–no-cabac* for *x264*), how the software was compiled (compiler and version), the version of the software, *etc.*

Figure 1 shows an example of these layers and possible impacts on two performances (encoding time and size).

3.1.1 Problem. Software practitioners may not be aware of the impacts of some layers over some performance properties. In fact, it is hard to know *a priori* which and to what extent configurations' layers do have an impact on software. For software variability researchers *e.g.*, seeking to train predictive performance models, these variability layers can disrupt the model's training and weaken the results, preventing their generalization and making them useless to operate under different conditions.

3.1.2 Challenges. Can we predict how much (combination of) variability layers will change non-functional properties of a software? Owing to the cost of executing cross-layer configurations', identifying the influential layers remains a challenge; their importances might vary depending on the subject system, the targeted non-functional property as well as the software configurations considered. While characterizing the impact of variability layers individually is necessary, it is not sufficient. As observed for software options, we conjecture that cross-layer configurations interact and can alter software performances and configuration option's effects.

3.1.3 Opportunities. Once their effects have been identified and quantified, the influential variability factors can be leveraged to improve software performance, while the others remain fixed and can be forgotten (*e.g.*, when benchmarking or transfer learning, see hereafter).

3.2 Deep testing and benchmarking

3.2.1 Problem. Since each user applies the software on his input data, with his dependencies on his laptop, there are almost as many possible environments as users. Because of the combinatorial explosion of possible environments, it becomes time-consuming to test all possible environment performances (*cf.* Section 2.2). Moreover, testing all variability layers requires to test variability on the system level, which is not supported by existing tools.

3.2.2 Challenges. How to sample a good benchmark of environments? A good benchmark should be representative of the real usage of the software. It has to contain not only a wide variety of possibilities within a layer (*e.g.*, different brands or models for the hardware layer), but also cross-layer diversity (*e.g.*, several variants of software running in different versions of operating systems). These two rules should be followed, in order to avoid missing corner case environments, in which combination of layers interact with software features, causing performance bugs.

Can we build a framework that tests configurable systems in multiple environments? In general, testing software configurations requires to automatically observe their properties, being functional or related to performance. Deep software variability further challenges the automation: (1) each layer (*e.g.*, operating system) may come with its own specific tooling; (2) integrating all layers together to test the software is not straightforward: for instance, instrumenting the tests of a software configuration in variants of hardware and operating systems. Another open challenge is: Can we detect corner cases configurations of layers that eventually cause software's performance bugs? As exemplified in Figure 2, a change in the input data or hardware can cause performance differences. There are two hypotheses worth investigating.

First, such differences are expected and can be explained. For example, it is normal that a given software option has more or less effect when processing over a specific input. A second hypothesis is that performance differences among variability layers are in fact a manifestation of a software bug that causes an accidental and unexpected decrease in performance. For validating or refuting the "bug or feature" hypothesis, software developers and domain experts of the different layers should be involved *e.g.*, by reviewing performance results or formalizing expected impacts of layers.

3.2.3 Opportunities. Some software bugs, especially related to performance, never manifest in a fixed environment. The diversification of the different layers is an opportunity to test the robustness and resilience of the software layer in multiple environments. Another observation is that an absolute number for *e.g.*, execution time has little meaning in itself and in a fixed environment. On the contrary, some performance issues are noticeable under the conditions the distributions are put in perspective and compared to others. That is, another opportunity to detect unexpected performance differences relatively to different configuration layers. Overall, developers and operationals can exploit deep software variability to detect more (performance) bugs.

3.3 Improve documentation

3.3.1 Problem. Assisting users in charge of configuring software is a still an open issue *per se*; deep software variability exacerbates this problem. As described in Section 2.3, impacts of different layers are only partially reported in the documentation. Due to the lack of documentation on the relative effects of configuration options, practitioners have to ask directly to experts in forums or open topics to make their software work, or just to configure it properly.

3.3.2 Challenges. Can developers include deep software variability constraints and effects in the documentation? The challenge is to develop user-friendly documentation that alerts users to the effects of deep software variability, its requirements, and gives them an overview of how the software should be configured for their environment. This documentation could be used as a pretext to centralize measurements from volunteers, benefiting to the community members ; users could compare their usage of the software to others running it in similar environments.

3.3.3 Opportunities. Building such a documentation would encourage the fine-grained, informed customization of configurable systems. When the effects of software options are captured, users may try to optimize and conveniently replace the default software configuration. Advanced users could even build their own variant of the software, working in their environment and fulfilling their own needs. Furthermore, different experts can participate to the customization process, each focusing on their domain of expertise (*e.g.*, hardware, operating system) with the help of a precise documentation.

3.4 Generalize the configuration knowledge

3.4.1 Problem. Changing a simple property of its executing environment can alter the performances of a software.

For instance, operating systems evolve frequently, introducing numerous changes in software environments; performance models has to evolve and adapt their prediction following the same rule. Unfortunately, practitioners cannot afford to train one new model per environment.

3.4.2 Challenges. Is it possible to accurately predict software performance for any environment?

The goal is to train a general model that aims at predicting software performances for any user's environment (*i.e.*, robust to major changes, like changing simultaneously the hardware and the input data). Another achievement is to identify the limits of the model, and include the tests (*cf.*Section 3.2) as a part of the learning process when transferring performances leads to poor predictions.

3.4.3 Opportunities. Being able to transfer performance from one environment to another avoids the need to build a performance model for each environment, thus saving time and energy resources. From the user's point of view, it allows to estimate the performance of a configuration without testing it, facilitating the configuration of the software for its environment.

3.5 Cross-layer tuning

3.5.1 Problem. Some operating systems and packages include options that may be over-adapted to the environment in which they run. In a sense, these layers are too general-purpose. It can lead to performance bugs when the configuration options misidentified the software environment. In the same vein, a software always configured with the default values will not benefit from the optimization coming from its environment (*e.g.*, parallelization if the default sticks to sequential tasks), wasting energy and computing time.

3.5.2 Challenges. How can we tune the software for one specific executing environment? Instead of enduring deep software variability, the goal is to pre-select the right environment for the software, tuning each layer separately in such a way it improves the overall software performances. The tuning process may have a cost since it requires to know which variability layer has an effect on software, what will be its interactions with other layers, and how to configure it in symbiosis with the rest of the environment. This cost should be confronted to the underlying benefits.

3.5.3 Opportunities. Since cross-layer tuning works on multiple variability layers, in comparison to a simple software tuning (*i.e.*, mono-layer optimization), we expect it to largely outperform default values configurations in terms of performances (*e.g.*, energy consumption, execution time, *etc.*) while keeping the same level of service. Cross-layer tuning should be more efficient at accomplishing a unique task; however, it requires to restrict some flexibility (*e.g.*, at runtime) and forget some variability of each layer.

4 RELATED WORK

For each step defined in Section 3, we refer to existing research papers that have identified and addressed parts of the deep software variability.

4.1 Impacts of variability layers

Numerous research papers study the effects of variability layers on software's performances or on configuration options: hardware [10, 35, 43], workloads [19, 20, 24, 30, 43], variants [22, 37], versions [13, 29, 37, 41], compilation options [15, 27] and input data [3, 8]. Such studies provide evidence that some layers have a noticeable impact on the software (configuration) layer. Deep software variability provides a conceptual, unified framework to systematically investigate the significance of the problem. To the best of our knowledge, there are two open research directions: (1) a systematic and comprehensive assessment of the influence of individual layers onto the distribution of software configurations: only a few hardware, input data, or software configurations are usually considered; (2) empirical knowledge about how layers composed and interact each other, for example how hardware and input data both influence the performance properties of a configurable system (*e.g.*, *x264*).

4.2 Deep testing and benchmarking

Related work proposes several methods to test the software layer, typically to improve coverage of the software configuration space: combinatorial interaction testing [7], t-wise [26, 36], genetic algorithms [14], adversarial machine learning [33], to name a few. These approaches are usually applied at the software layer, but need to be adapted at the system level (*e.g.*, operating system) and integrated into a comprehensive testing or benchmarking pipeline. The effectiveness of existing testing methods (*e.g.*, differential testing [21], metamorphic testing [31], multimorphic testing [34]) in the context of deep software variability is another research direction.

4.3 Improve documentation

State of the art provides evidences that documentation sometimes lacks of details [2, 11], and give insights on how to build a configurable system's documentation [5, 6, 28]. Deep software variability further challenges developers of software configurable systems: the configuration documentation should mention the potential effects of other layers on the tuning of configuration options.

4.4 Generalize the configuration knowledge

Several research papers have shown that it is possible to transfer software performances from a *source* environment to a *target* environment, using either the similarities between environments [9, 16, 39], or selecting the best source environment [23]. Including causalities in the performance model to improve the transfer [17, 18] is another promising research direction to explore. Deep software variability provides a concrete case for transfer learning: a change in a specific layer (*e.g.*, hardware) is a new target environment and calls to adapt the prediction model obtained out of a source (*i.e.*, original environment).

4.5 Cross-layer tuning

Various research papers [15, 19, 20, 24, 27, 43] configure their variability layer to optimize non-functional properties of software, while others, as [8, 30], adapt configuration options of software to variability layers. Such works are in line with the vision exposed in Section 3.5 though only one layer at a time is usually considered as part of the specialization process.

5 CONCLUSION

In this vision paper, we proposed deep software variability and described its many challenges and opportunities. As illustrated with x264 and partly observed in the literature, many layers (hardware, operating system, input data, *etc.*), themselves subject to variability, can alter the configurable software layer. Deep software variability calls to investigate how to systematically handle cross-layer configuration.

At this step of the research, we ignore to what extent deep software variability impacts performances of software. May configurable systems be more or less sensitive to deep variability, depending on quantitative properties of interest (*e.g.*, energy consumption, execution time)?

In fact, deep software variability concerns all scientists that aim at publishing reproducible research works; if your results are not consistent from one environment to another, your conclusions could be either weakened or invalidated. Many scientific domains are potentially impacted. Applied to ethical decisions, deep software variability could also have dramatic consequences; what would you do if a machine learning model deciding on the release of a prisoner changes its prediction when modifying its executing environment?

To make the vision of this paper more explicit, we recall the general idea of this paper: when considering software variability, we encourage you to broaden the scope of your research, and include parts of software environments in the study. Lastly, we illustrate how we could use the different steps detailed throughout the paper to concretely handle deep variability for a given software:

(1) First, we have to identify influential variability layers for this software. For instance, to estimate how the input data layer interacts with its configuration options, we could consider a large amount of inputs and measure their effects on performances. If the effects of features remain the same across all inputs, we could argue that input data is a negligible layer of variability, and could remain fixed when benchmarking the software. If not, this layer is influential, and a representative set of inputs should be extracted. This process should be applied to each layer.
(2) Each influential layer of variability is now represented by elements of reference condensing its inner variability (*e.g.*, for the hardware layer, at least a server, a laptop and an Arduino). Then, we propose to measure software performances in different environments, composed of different layer's elements (*e.g.*, a laptop extracted of the hardware layer, an operating system coming from the operating system's layer, etc.).
(3) Computing these measurements will allow us to detect some bugs and reference them in the documentation.
(4) With this sample of measurements, we could build a performance model working for all environments. A novel environment would be associated to a set of reference elements. With transfer learning and sampling techniques, we would be able to build a dedicated model designed for this new environment.
(5) Being able to predict accurately the performance of each environment would allow us to improve the default configuration of software w.r.t. their environment, *i.e.*, tune them for their environment.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 421–432. https://doi.org/10.1145/2642937.2642990

[2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Montreal, Quebec, Canada, 1199–1210. https://doi.org/10.1109/ICSE.2019.00122

[3] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. (Feb. 2020). https://hal.inria.fr/hal-02356290 working paper or preprint.

[4] Mona Attariyan and Jason Flinn. 2008. Using Causality to Diagnose Configuration Bugs. In *USENIX 2008 Annual Technical Conference* (Boston, Massachusetts) *(ATC'08)*. USENIX Association, USA, 281–286.

[5] Alexander Bakman, Daniel Sabin, Tudor Hulubei, and Shalom Wertsberger. 2005. Automatic documentation of configurable systems by outputting explanatory information of configuration parameters in a narrative format and configuration parameters differences. US Patent 6,981,207.

[6] Alexander Bakman, Daniel Sabin, Tudor Hulubei, and Shalom Wertsberger. 2018. Method and system for automatic documentation of configurable systems. US Patent 9,959,115.

[7] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. Association for Computing Machinery, New York, NY, USA, 129–139. https://doi.org/10.1145/1273463.1273482

[8] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. *SIGPLAN Not.* 50, 6 (June 2015), 379–390. https://doi.org/10.1145/2813885.2737969

[9] Johannes Dorn, Sven Apel, and Norbert Siegmund. 2020. Generating Attributed Variability Models for Transfer Learning. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems* (Magdeburg, Germany) *(VAMOS '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/3377024.3377040

[10] Johannes K. Fichte, Markus Hecher, and Stefan Szeider. 2020. A Time Leap Challenge for SAT-Solving. In *Principles and Practice of Constraint Programming*, Helmut Simonis (Ed.). Springer International Publishing, Cham, 267–285.

[11] Andrew Forward and Timothy C. Lethbridge. 2002. The Relevance of Software Documentation, Tools and Technologies: A Survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering* (McLean, Virginia, USA) *(DocEng '02)*. Association for Computing Machinery, New York, NY, USA, 26–33. https://doi.org/10.1145/585058.585065

[12] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Ciudad Real, Spain) *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 23, 10 pages. https://doi.org/10.1145/2961111.2962602

[13] Johannes Hasreiter. 2019. Evolution of Performance Influences in Configurable Systems. In *Evolution of Performance Influences in Configurable Systems*. University of Passau, Passau, 77 pages. https://www.se.cs.uni-saarland.de/theses/JohannesHasreiterMA.pdf

[14] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Multi-Objective Test Generation for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference* (Tokyo, Japan) *(SPLC '13)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/2491627.2491635

[15] Kenneth Hoste and Lieven Eeckhout. 2008. Cole: Compiler Optimization Level Exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Boston, MA, USA) *(CGO '08)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/1356058.1356080

[16] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to sample: exploiting similarities across environments to learn performance models for configurable systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Association for Computing Machinery, New York, 71–82. https://dl.acm.org/doi/pdf/10.1145/3236024.3236074

[17] Mohammad Ali Javidian, Pooyan Jamshidi, and Marco Valtorta. 2019. Transfer Learning for Performance Modeling of Configurable Systems: A Causal Analysis. arXiv:1902.10119 [cs.AI]

[18] Rahul Krishna, Md Shahriar Iqbal, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2020. CADET: A Systematic Method For Debugging Misconfigurations using Counterfactual Reasoning. arXiv:2010.06061 [cs.SE]

[19] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. *SIGPLAN Not.* 52, 7 (April 2017), 15–29. https://doi.org/10.1145/3140607.3050757

[20] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 461–472. https://doi.org/10.1145/2490301.2451167

[21] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[22] Stefan M. Muehlbauer. 2020. Identifying Software Performance Changes Across Variants and Versions. In *Identifying Software Performance Changes Across Variants and Versions*. University of Passau, Passau, 12 pages. https://www.se.cs.uni-saarland.de/publications/docs/MAS+20.pdf

[23] Vivek Nair, Rahul Krishna, Tim Menzies, and Pooyan Jamshidi. 2018. Transfer Learning with Bellwethers to find Good Configurations. *CoRR* abs/1803.03900 (2018), 1–11. arXiv:1803.03900 http://arxiv.org/abs/1803.03900

[24] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 59–73. https://doi.org/10.1145/3313808.3313817

[25] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2019. Learning Software Configuration Spaces: A Systematic Literature Review. *ArXiv* abs/1906.03018 (2019), 1–44. https://arxiv.org/abs/1906.03018

[26] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. l. Traon. 2010. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, Paris, France, 459–468. https://doi.org/10.1109/ICST.2010.43

[27] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. 2013. Automatic Tuning of Compiler Optimizations and Analysis of their Impact. *Procedia Computer Science* 18 (2013), 1312–1321. https://doi.org/10.1016/j.procs.2013.05.298

[28] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd, and E. Wong. 2017. On-demand Developer Documentation. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Canada, 479–483. https://doi.org/10.1109/ICSME.2017.17

[29] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker. 2013. Performance evolution blueprint: Understanding the impact of software evolution on performance. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, Eindhoven, NL, 1–9. https://doi.org/10.1109/VISSOFT.2013.6650523

[30] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. 2017. On Cross-Stack Configuration Errors. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Buenos Aires, Argentina, 255–265. https://doi.org/10.1109/ICSE.2017.31

[31] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[32] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 284–294. https://doi.org/10.1145/2786805.2786845

[33] Paul Temple, Mathieu Acher, Battista Biggio, Jean-Marc Jézéquel, and Fabio Roli. 2018. Towards Adversarial Configurations for Software Product Lines. arXiv:1805.12021 [cs.LG]

[34] P. Temple, M. Acher, and J. M. Jezequel. 2019. Empirical Assessment of Multimorphic Testing. *IEEE Transactions on Software Engineering* 1, 1 (2019), 1–1. https://doi.org/10.1109/TSE.2019.2926971

[35] X. Teng, H. Pham, and D. R. Jeske. 2006. Reliability Modeling of Hardware and Software Interactions, and Its Applications. *IEEE Transactions on Reliability* 55, 4 (2006), 571–577. https://doi.org/10.1109/TR.2006.884589

[36] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages. https://doi.org/10.1145/2580950

[37] Thomas Thüm, André van Hoorn, Sven Apel, Johannes Bürdek, Sinem Getir, Robert Heinrich, Reiner Jung, Matthias Kowal, Malte Lochau, Ina Schaefer, and Jürgen Walter. 2019. *Performance Analysis Strategies for Software Variants and Versions*. Springer International Publishing, Cham, 175–206. https://doi.org/10.1007/978-3-030-13499-0_8

[38] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. 2015. Empirical comparison of regression methods for variability-aware performance prediction. In *SPLC'15*. ACM, New York, NY,United States, 186–190. https://dl.acm.org/doi/10.1145/2791060.2791069

[39] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring Performance Prediction Models Across Different Hardware Platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/3030207.3030216

[40] Yilin Wang, Sasi Inguva, and Balu Adsumilli. 2019. YouTube UGC Dataset for Video Compression Research. In *2019 IEEE 21st International Workshop on Multimedia Signal Processing (MMSP)*. IEEE, United States, 1–5. https://doi.org/10.1109/mmsp.2019.8901772

[41] Niklas Werner, S. Apel, and C. Kaltenecker. 2019. Energy and Performance Evolution of Configurable Systems: Case Studies and Experiments. In *Energy and Performance Evolution of Configurable Systems: Case Studies and Experiments*. University of Passau, Passau, 97 pages. https://www.se.cs.uni-saarland.de/theses/NiklasWernerMA.pdf

[42] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems*

Principles (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 159–172. https://doi.org/10.1145/2043556.2043572

[43] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. *SIGPLAN Not.* 51, 4 (March 2016), 545–559. https://doi.org/10.1145/2954679.2872375

[44] S. Zhang and M. D. Ernst. 2013. Automated diagnosis of software configuration errors. In *2013 35th International Conference on Software Engineering (ICSE)*. ACM, New York, 312–321. https://doi.org/10.1109/ICSE.2013.6606577

[45] Sai Zhang and Michael D. Ernst. 2014. Which Configuration Option Should I Change?. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 152–163. https://doi.org/10.1145/2568225.2568251