Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size

Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, Djamel Eddine Khelladi

Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size

Hugo Martin*, Mathieu Acher*†, Juliana Alves Pereira‡, Luc Lesoil*,
Jean-Marc Jézéquel*, Djamel Eddine Khelladi*
*Univ Rennes, Inria, CNRS, IRISA, France
{hugo.martin, mathieu.acher, luc.lesoil, jean-marc.jezequel, djamel-eddine.khelladi}@irisa.fr
†Institut Universitaire de France (IUF)
‡PUC-Rio, Rio de Janeiro, Brazil
jpereira@inf.puc-rio.br

Abstract—With large scale and complex configurable systems, it is hard for users to choose the right combination of options (i.e., configurations) in order to obtain the wanted trade-off between functionality and performance goals such as speed or size. Machine learning can help in relating these goals to the configurable system options, and thus, predict the effect of options on the outcome, typically after a costly training step. However, many configurable systems evolve at such a rapid pace that it is impractical to retrain a new model from scratch for each new version. In this paper, we propose a new method to enable transfer learning of binary size predictions among versions of the same configurable system. Taking the extreme case of the Linux kernel with its $\approx 14,500$ configuration options, we first investigate how binary size predictions of kernel size degrade over successive versions. We show that the direct reuse of an accurate prediction model from 2017 quickly becomes inaccurate when Linux evolves, up to a 32% mean error by August 2020. We thus propose a new approach for transfer evolution-aware model shifting (TEAMS). It leverages the structure of a configurable system to transfer an initial predictive model towards its future versions with a minimal amount of extra processing for each version. We show that TEAMS vastly outperforms state of the art approaches over the 3 years history of Linux kernels, from 4.13 to 5.8.

Index Terms—Software Product Line, Software Evolution, Machine Learning, Transfer Learning, Performance Prediction

1 Introduction

Configurable systems form a vast class of software systems that encompasses: Software Product Lines, operating systems kernels, web development frameworks/stacks, e-commerce configurators, code generators, software ecosystems (e.g., Android's "Play Store"), autonomous systems, etc. While being very different in their goals and implementations, configurable systems see their behaviour affected by the activation or deactivation of one or more *configuration options*. Configurable software systems offer a multitude of configuration options that can be combined to tailor the systems' functional behavior and performance (e.g., execution time, memory consumption, etc.). Options often have a significant influence on performance properties that are hard to know and model a priori. There are numerous possible options values, logical constraints between options, and subtle interactions among options.

Numerous works [1], [2], [3], [4], [5] have shown that quantifying the performance influence of each individual option is not meaningful in most cases. The performance influence of n options, all jointly activated in a configuration, is not easily deducible from the performance influence of each individual option [5]. Since some options are interacting with others, they have non-linear effects and cannot be reduced to "additive" effects. Even something apparently as simple as predicting the binary size of an application based on the selected options is then surprisingly difficult.

Measuring all configurations of a configurable system is the most obvious path to, e.g., find a well-suited configuration. However, it is too costly or even infeasible in practice, because n binary options could yield up to 2^n configurations. Machine-learning techniques address this issue by measuring only a subset of configurations (known as sample) and then using these configurations' measurements to build a performance model capable of predicting the performance of other configurations (i.e. configurations not measured before). Several works, thus, follow a *"sampling, measuring, learning"* process (e.g., [1], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]). Obtaining a low prediction error typically requires a large sample, that should be distributed as uniformly as possible over the configuration space.

As any software system, configurable systems evolve with many commits that may modify the entire architecture and source code. In addition, options may be added or removed during evolution. All these modifications can have an impact on the performance distribution of the configuration space: the effects of individual options may change as well as the interactions among them.

Thus, for large and complex configurable systems, one has to manage both the combinatorial explosion of possibly thousands of options (yielding *variants*, *i.e.,* variability in space) and the continuous rapid evolution (yielding *versions*, *i.e.,* variability in time). Learning variability in both space and time is indeed challenging.

At each new release, the problem of obtaining an accurate performance model of a configurable system may arise again due to the variability in time. For configurable systems evolving at a rapid pace, **sampling again each new version is impractical**. Since it requires such a large amount of computational resources to measure the performance of each configuration to be used as input to a machine learning algorithm, this process could even take more time than the release period.

To overcome this issue, we propose to transfer the learning across versions. Since the feature space (i.e., the set of configuration options) can change across versions, our approach falls under the category known as *heterogeneous* transfer learning, opposite to *homogeneous* transfer learning, that assumes the feature space remains unchanged during evolution. Identifying how to efficiently apply transfer knowledge of the learned model as the systems evolve is challenging. This is indeed a well-known general problem in machine learning [16], [17], made even more difficult because of the heterogeneity of configuration spaces (due to the fact that features come and go across versions) may cause bias on cross-version feature representation [18].

Existing works [7], [11], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28] have attempted to investigate the transfer challenge. However, they investigated it in the context of homogeneous feature spaces and not for the case of system code evolving across different versions, i.e. heterogeneous feature spaces. Thus, to the best of our knowledge, no work has shown evidence that transfer learning can model variability in space and time of configurable systems accurately. Moreover, existing works do not consider very complex systems that have thousands of options and at the same time evolve frequently with large deltas between releases.

In this paper, we purposely consider the Linux kernel, because it is one of the most complex representative case of this problem. It has thousands of options (*e.g.,* around 15,000 for version 5.8) and hundreds of releases over more than two decades. We selected seven releases spanning over three years from version 4.13 (2017) to 5.8 (2020). On this dataset we first answer the following research question: *RQ1. To what extent does Linux evolution degrade the accuracy of a performance prediction model trained on a specific version?*

To this end, we first perform an experiment to quantify the impact of Linux evolution (*i.e.,* the release of a new kernel version) on configuration performance (specifically: kernel binary size for a Linux configuration). To the best of our knowledge, no prior work has attempted to check whether a learnt configuration performance model can be used regardless of the applied system evolution over time. Our experiments show that evolution does indeed impact the learnt model by degrading the prediction (by a ratio of 4 to 6) down to the point where it cannot effectively be used on subsequent releases.

After that, we propose a new automated transfer *Evolution-aware Model Shifting* (tEAMS) approach that consists of applying a tree-based model shifting using gradient boosting trees, backed up with feature alignment to handle the problem of heterogeneity in configuration spaces. The goal of tEAMS, as any transfer learning technique, is to beat approaches that learn at every release ("from scratch"), as illustrated in Figure 1. We then answer the next research question: *RQ2. What is the accuracy of* tEAMS *compared to learning from scratch and other transfer learning techniques?*

Our results show that tEAMS outperforms state-of-the-art approaches by reaching a low and constant 5.6% to 7.1% error rate rather than 8.3 to 9.2% for learning from scratch method. Our contributions are summarized as follows:

1) We design and implement a large-scale study of the effectiveness of transfer learning to model kernels' variability in space and time.
2) We show empirical evidence for the degradation of binary size predictions from version 4.13 onward.
3) We propose a novel technique (tEAMS) for transfer learning across versions, with two variants: one shot transfer and incremental transfer.
4) We evaluate our results over seven kernel versions and a dataset of 243K+ configurations spanning over three years: tEAMS vastly outperforms the accuracy of state-of-the-art transfer learning strategies. Moreover, it is cost-effective since it works with the addition of a reduced set of training samples over future versions.

We provide a replication package with all artifacts (including datasets and learning procedures): https://zenodo.org/record/4960172.

The rest of the paper is organized as follows. Section 2 gives background on the Linux Kernel, its size, evolution, and on machine learning. Section 3 investigates how performance predictions degrade over evolution releases. Section 4 presents a new approach called tEAMS. Section 5 evaluates tEAMS on the history of Linux kernels from 4.13 to 5.8 and compares it to state-of-the-art. Sections 6 and 7 discuss threats to validity and related work. Finally, Section 8 concludes this paper.

2 Background

2.1 Linux Kernel

In this paper, we chose the Linux kernel case for several reasons. First, it is a prominent example of a highly-configurable system. It is extremely complex both in terms of kernel *variants*, i.e., with its thousands of configuration options and in terms of code size, i.e., millions of Lines Of Code (LOC). Other existing cases considered in the literature (*e.g.,* see [1]) vary from 5 to a few hundreds options and from 2,595 LOC to 305,191 LOC, while, *e.g.,* Linux version 4.13 has > 12,797 options and > 13M LOC. Thus, those subject systems exhibit far less options which question whether proposed techniques can scale and obtain accurate results for a huge configuration space like the Linux one. Second, the Linux kernel has been rapidly evolving over a long period of time (*versions*) and continues until this day thanks to a large community of active developers. Thus, the large scale combination of both variants and versions makes it an ideal case study as we aim to investigate transferability over different releases spanning several years of active development of a configurable system.

To generate a variant of the Linux kernel, users set values to options (*e.g.,* through a configurator [29]) and obtain a so-called .config file. The majority of options has either boolean values (*'y'* or *'n'* for activating/deactivating an option) or

tri-state values (*'y'*, *'n'*, and *'m'* for activating an option as a module). We consider that a *configuration* is an assignment of a value to each option, either by the user, or automatically with a default value. Based on a configuration, the build process of a Linux kernel can start and involves different layers, involving different programming languages (asm, C), configuration languages (Kconfig), preprocessors and compilers (cpp, gcc), and build tools (make).

For instance, if a user wants to add support for certain type of network cards, she might activate the option HAMACHI. She should also take options' dependencies declared within Kconfig files into account. The option HAMACHI depends on the option PCI.

```
config HAMACHI
tristate "Packet Engines Hamachi GNIC-II support"
depends on PCI
    ...
help
  If you have a Gigabit Ethernet card
  of this type, say Y here.
  To compile this driver as a module,
  choose M here.
  The module will be called hamachi.
```

Configurators are available to support the configuration activity, automatically activate (or deactivate) other implied or excluded options, and interactively ask users what options' values to set. This results in a .config file that is used as input to eventually build the kernel with make and gcc. The process is illustrated below:

```
> make menuconfig # configurator
> cat .config
#
# Automatically generated file; DO NOT EDIT.
# Linux/x86 4.15.0 Kernel Configuration
#
CONFIG_64BIT=y
CONFIG_X86_64=y
...
CONFIG_HAMACHI=y
CONFIG_PCI=y
CONFIG_BFS_FS=m
CONFIG_ARCH_SUPPORTS_DEBUG_PAGEALLOC=y
# CONFIG_IRQ_DOMAIN_DEBUG is not set
> make # build out of a .config
> ls -lh vmlinux # binary size
... 29.3Mb
```

In the .config example, the option BFS_FS is activated as a module. The option IRQ_DOMAIN_DEBUG is deactivated ('n' value). The options HAMACHI, PCI, and ARCH_SUPPORTS_DEBUG_PAGEALLOC are activated with 'y' value.

2.2 Kernel size

Linux kernels are used in a wide variety of systems, ranging from embedded devices, up to cloud services or powerful supercomputers [30]. Many of these systems have strong requirements on the *kernel size* due to constraints such as limited memory or instant boot [31], [32], [33]. Obtaining a suitable trade-off between kernel size, functionality, and other non-functional concerns (*e.g.*, security) is an extremely hard problem. For instance, activating an individual option can increase the kernel size so much that it becomes impossible to deploy it on small devices. Our experimental data show that kernel sizes highly vary depending on options' values, ranging from a 7.3Mb for tinyconfig to 2,134Mb (2GB) for a random configuration.

As elaborated in [33], community effort exists to document the influence of options on the Linux Kernel size, such as with the Wiki https://elinux.org/Kernel_Size_Tuning_Guide and the project tinyconfig http://tiny.wiki.kernel.org. However, they are not maintained over time, respectively, since 2011 and 2015. Acher *et al.* [33] report on several use cases (*e.g.*, software debloating [34]), Kconfig documentation, options values for default configurations, as well as past and ongoing initiatives. All provide evidence that options related to kernel size are an important issue for the Linux community but their specific effects are hard to document or model for developers and contributors. Moreover, this effort is impractical to maintain over time for the rapid Linux evolution. Therefore, it is crucial to be able to automate it so that it is valuable for not only Linux developers, but also integrators of the kernel in several deployment settings.

Besides, numerous works have shown that quantifying the performance influence of each individual option is not meaningful in most cases [1], [12], [13], [20], [35]. That is, the performance influence of n options, all jointly activated in a configuration, is not easily deducible from the performance influence of each individual option. The Linux kernel binary size is not an exception: options such as `CONFIG_DEBUG_*` or `CC_OPTIMIZE_FOR_SIZE` have cross-cutting, non-linear effects and cannot be reduced to additive effects. For example, basic linear regression models, which are unable to capture interactions among options, give poorly accurate results.

2.3 Predicting kernel size with machine learning

We aim to predict the effects of options w.r.t. size in such a way that developers and end-users can then make informed and guided configuration decisions. Because building all configurations is infeasible (estimated number of Linux kernel configurations : 10^{6000}), the principle is to learn from a *sample* of measured configurations. Predicting the size of a kernel is then a supervised, regression problem. Out of a combination of options values (*i.e.*, a configuration), the learning model should be able to predict a quantitative value (the size) without actually building and measuring the kernel.

The challenge is to apply the right algorithm with the right balance between *accuracy* (i.e., the prediction is close enough to reality) and the *cost* of gathering the measurements' samples. **A key issue is that the compilation of one kernel configuration and the measurement of its size already requires 261 seconds** on average in our dataset (see Table 1 column 6 - [Seconds/config]), fully using 16 cores of a recent machine. **Therefore, a given machine can compile less than 14 configurations per hour**, which is extremely low in comparison to the space of all possible configurations to compile. Hence, the cost of computing thousands of configurations' measurements can be very high.

This is where machine learning comes in handy. Recent approaches show the usefulness of machine learning techniques for learning performance models based on a sample

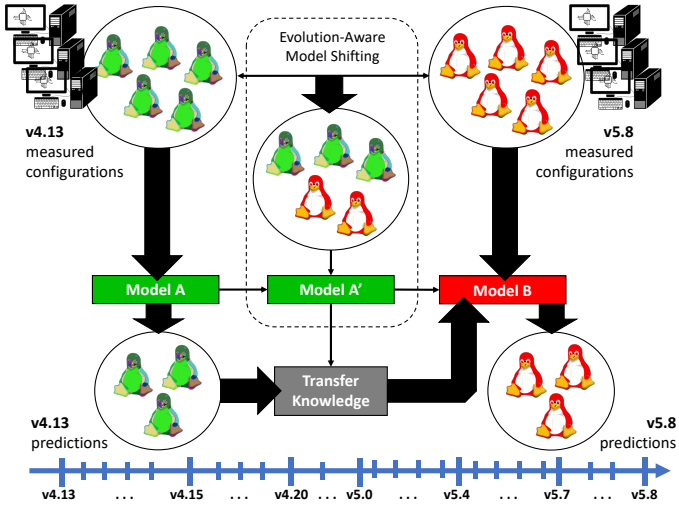

Figure 1: An overview on how to predict the performance of Linux kernel configurations over versions 4.13 and 5.8.

set of configurations [1], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. Researchers have been experimenting with different techniques, e.g., decision trees, linear regression, neural networks, etc. Figure 1 illustrates, when ignoring the middle part, how traditional machine learning models work. For example, suppose we aim to predict the size of Linux kernel over versions 4.13 and 5.8. To do so, we train two different models A and B based on a sample of configuration measurements in this same version, respectively for 4.13 and 5.8, as depicted in the left and right parts of Figure 1. Basically, the use of traditional machine learning techniques requires a completely new model to be retrained for each very specific target scenario. Knowing that Linux has hundreds of releases in each major release, makes this process impracticable and too expensive to compute. This paper tackles this issue.

2.4 Linux kernel rapid evolution

As mentioned before, one of the significant reasons why Linux represents an ideal case study is its frequent evolution over a long period of time. For example, just the major release 4.x contains 20 minor releases (from 4.1.x to 4.20.x) and thousands of patch releases (e.g., 4.20.1 to 4.20.17). With every release, code changes cover bugs reparation and/or the introduction of new options with new functionalities.

Linux kernel evolution steps are significant in terms of frequency but also in terms of size. For example, between the two releases 4.13 and 4.20, 468 features are deleted and 1,189 features are added, along with 104,691 commit changes and almost two million of file changes. This is also the case for other releases that we consider in our study between the source release and the target releases to which we perform transfer learning, as illustrated in Table 1. All consist of thousands of changes in each dimension: options, commits, source files.

3 Impacts of Evolution on Configuration Performance

In this section, we aim to quantify the impact of Linux evolution (*i.e.*, the release of a new kernel version) on configuration binary size.

Mühlbauer *et al.* [28] investigated the history of software performances to isolate when a performance shift happens over time. If we know evolution can impact the performance of a configurable software, we do not actually know if and how much it can impact a performance prediction model.

An hypothesis is that the evolution has no significant impact and the Linux community can effectively reuse a binary size prediction model across all versions. The counter hypothesis is that the evolution changes the binary size distributions: in this case, a measurable and practical consequence would be that a binary size model becomes inaccurate for other versions. In other words, if the degradation of the accuracy of a prediction model is to be expected, it is necessary to know whether such degradation is sharp enough to be a problem for the Linux community. However, none of these hypotheses has been investigated in the literature. Therefore, quantifying the impacts of evolution is crucial and boils down to address the following research question:

(RQ1) To what extent does Linux evolution degrade the accuracy of a binary size prediction model trained on a specific version? To address it, we measure the accuracy of a performance prediction model, specifically a model predicting the binary size of the kernel image, trained in one specific version (*i.e.*, 4.13), when applied to later versions (*e.g.*, up to 5.8).

3.1 Experimental Settings

We now present the datasets we gathered on different Linux configurations and versions; the learning algorithms we used to build the prediction models, as well as the accuracy metric.

3.1.1 Dataset

We compiled and measured Linux kernels on seven different versions. Table 1 further details each considered release version:

- 4.13: this release was the starting point of our work with huge investments (builds and measurements of 90K+ configurations);
- 4.15: the release was the first to deal with the serious chip security problems meltdown/spectre [36] that mainly apply to Intel-based processor (x86 architecture). A broad set of mitigations has been included in the kernel, which can have an effect on kernel sizes;
- 4.20: the last version before 5.0, with several x86/x86_64 optimizations. As part of the in-depth analysis on the evolution of core operation performance in Linux [37], Ren *et al.* identified several changes in latency for versions between 4.15 and 4.20;
- 5.0: a major release. Interestingly, there have been some debates about the decrease of kernel performance on some macro-benchmarks (*e.g.*, see [38]);
- 5.4: it is a long term support release that will be maintained 6 years. This version also includes modifications for dealing with Linux performance [38], [39];
- 5.7: a recent version, more than half-year after 5.4;

- 5.8: Linus Torvalds commented[1] *"IOW, 5.8 looks big. Really big."* and reported *"over 14k non-merge commits (over 15k counting merges), 800k new lines, and over 14 thousand files changed"*, suggesting an important and challenging evolution to tackle.

As depicted in Table 1, the continuous evolution from 4.13 to 5.8 is significant in terms of numbers of added/deleted options, delta of the commits and the changes files. Note that those changes are computed for each release w.r.t. 4.13.

For all versions, we specifically targeted the x86-64 architecture, *i.e.,* technically, all configurations have values CONFIG_X86=y and CONFIG_X86_64=y. Overall, we span different periods during 3 years, with some modifications (security enhancements, new features) suggesting possible impacts on kernel non-functional properties (*e.g.,* size).

For each version, we build thousands of random configurations (see Table 1 column 5 - [Examples]). Owing to the computational cost, we balance the budget to measure at least and around 20K+ configurations per version. Such data is used to test the accuracy of a prediction model. We used TUXML[2], a tool to build the Linux kernel in the large *i.e.,* whatever options are combined. TUXML relies on Docker to host the numerous packages needed to compile and measure the Linux kernel. Docker offers a reproducible and portable environment – clusters of heterogeneous machines can be used with the same libraries and tools (*e.g.,* compilers' versions). Inside Docker, a collection of Python scripts automates the build process. We rely on randconfig to randomly generate Linux kernel configurations. randconfig has the merit of generating valid configurations that respect the numerous constraints between options. It is also a mature tool that the Linux community maintains and uses [40]. Though randconfig does not produce uniform, random samples (see Section 7), there is a diversity within the values of options (being *'y'*, *'n'*, or *'m'*). Given .config files, TUXML builds the corresponding kernels. Throughout the process, TUXML can collect various kinds of information, including the build status and the size of the kernel. We concretely measure vmlinux, a statically linked executable file that contains the kernel in object file format.

The distribution of binary size in our dataset varies depending on the version. While the mean binary size on version 4.13 is 47 MiB, for other versions that mean value is between 89 MiB and 118 MiB. The minimum size for all version is around 10 MiB and the maximum around 2 GiB

3.1.2 Preprocessing

We distinguish between *options* and *features*, as an *option* is a variable in Linux kernel configuration, and a *feature* an independent variable from which the machine learning algorithm creates a model. The distinction is important as we made some manipulations over the dataset, in order to facilitate the learning. The first one is to put aside non-tristate options, which represent a very small subset (between 300 and 320 depending on the version). Most Linux kernel options values are *"yes"*, *"no"*, or *"module"*, hence the name "tristate" options. The second manipulation was to encode these option values into numbers to be processed by the algorithm. We observed that the values *"no"* and *"module"* had the same effect on the kernel size, so we encoded them as 0, and *"yes"* as 1. That is, we do not use "module" as an option value and we are not interested in module size. Then we put aside the options only having one value in the whole dataset, as it would bring no information from a statistical point of view and only make it longer for a algorithm to learn. Last but not least, we added a custom feature which is the sum of all activated options in the configuration, as it has proved important and helpful in a previous study of Acher *et al.* [33].

3.1.3 Performance Prediction Models

Extensive experiments on 4.13 (the oldest version of the dataset) showed what learning algorithms and hyperparameters were effective and for which training set size [33]. Specifically, we chose gradient boosting trees (GBTs) that, according to [33], obtain the best results whatever the training set size. GBTs proved to be superior than linear regression models, decision trees, random forest, and neural networks for relatively small training set size (*e.g.,* 20K) but also for larger budgets (*e.g.,* 80K+). We trained GBTs with 85.000 examples on version 4.13. We took care of finding the best hyperparameters, using grid-search, as it proved itself a quite important factor in the accuracy of the models. We relied on scikit-learn [41], a Python library that implements state-of-the-art machine learning algorithms. As a performance model only matches a specific set of features (here: the features of 4.13), we deleted features only contained in further, target versions (*e.g.,* 4.15).

3.1.4 Accuracy Measurement

Due to the wide distribution of Linux kernel binary sizes (from 7MB to around 2GB in our dataset), relying on some metrics such as Mean Absolute Error or Mean Squared Error can be biased, as an error of a few MB can be a big error for small kernels, and negligible for the biggest kernels. As numerous existing works (*e.g.,* [1], [4], [7], [21], [42]), we rely on a variant of Mean Absolute Error, but normalized in a percentage error, known as Mean Absolute Percentage Error (MAPE), computed as follows : $MAPE = \frac{100}{t}\sum_{i=1}^{t}\frac{\left|f(c_i)-\hat{f}(c_i)\right|}{f(c_i)}\%$, t being the number of predictions, $\hat{f}(c_i)$ the predicted values, and $f(c_i)$ the measured values (ground truth). Another advantage of this metric is that it is easily understandable and comparable, being a percentage.

To limit the impact of randomness in the experimentation, we performed the process of learning 5 times, leading to different training and test sets; we report the average error in our results.

3.1.5 Insights about evolution of options' importance

To better understand the evolution and possible degradation w.r.t. accuracy, we extract relevant information from the learning models created on the version 4.13 and the ones created on later versions. Specifically, we analyze the evolution of the features within prediction models that

1. https://lore.kernel.org/lkml/CAHk-=whfuea587g8rh2DeLFFGYxiVuh-bzq22osJwz3q4SOfmA@mail.gmail.com/
2. https://github.com/TuxML/tuxml

Version	Release Date	LOC	Files	Examples	Seconds/config	Options	Features	Deleted features	New features	ΔCommits	Files changes
4.13	2017/09/03	16,616,534	60,530	92,562	271†	12,776	9,468	-	-	-	-
4.15	2018/01/28	17,073,368	62,249	39,391	263†	12,998	9,425	342	299	31,052	934,628
4.20	2018/12/23	17,526,171	62,423	23,489	225	13,533	10,189	468	1,189	104,691	1,972,020
5.0	2019/03/03	17,679,372	63,076	19,952	247	13,673	10,293	494	1,319	118,778	2,170,935
5.4	2019/10/24	19,358,903	67,915	25,847	285	14,159	10,813	663	2,008	181,308	3,827,025
5.7	2020/05/31	19,358,903	67,915	20,159	258	14,586	11,338	715	2,585	225,804	4,393,117
5.8	2020/08/02	19,729,197	69,303	21,923	289	14,817	11,530	730	2,792	242,381	4,681,313

Table 1: Dataset properties for each version. The number of deleted/new features, delta commits, files changes are w.r.t. 4.13. † for versions 4.13 and 4.15, the build time (number of seconds to build one configuration) should be interpreted with caution since we used heterogeneous machines and did not seek to control their workload

participate most in the prediction of binary size. We rely on *feature importance* a model agnostic, widely considered score for computing the increase in the prediction error of the model after we permuted the feature's values [43]. Feature importance provides an integrated and global insight into the prediction model of a given version: the score takes into account both the main feature effect and the interaction effects on model prediction. We perform the computation out of GBTs. Once the feature importance is computed, a so-called *feature ranking list* can be created and list orders' features by importance [44]. Note that we create 20 models on each version and average out the ranking to tamper with the randomness of the tree building process. By comparing two feature ranking lists from two different versions, we can track which features have their importance evolving. We compare 4.13 version with (1) the 4.15 corresponding to an important drop in accuracy and subject to many changes due to meltdown/spectre [45], [46]; (2) the 5.8 the most recent version at our disposal.

3.2 Results

Figure 2 shows the degradation of models trained on the Linux Kernel version 4.13 by plotting their error rate (meaning lower is better) on later versions. The models get on average 5% MAPE on 4.13, and less than two versions after, on 4.15, the error rate is 4 times higher at 20%. It keeps this error rate for multiples version, at least up to 5.0, and goes even higher, at 32% for the version 5.7 and 5.8, i.e., an error rate 6 times higher. Note that the degradation do occur independently from the training set size, i.e., both with 20K and 85K. This is a crucial result that confirms the hypothesis of degradation over time, regardless of the training set size. Hence, calling for a more sustainable solution like transfer learning [7], [47], [48].

It is worth noting though that the error rate stabilizes between 4.15 and 5.0. Hence, an hypothesis is that the evolution might be more affordable between some versions. Unfortunately, a direct reuse of the prediction model is inaccurate for the early version 4.15 and subsequent ones (4.20, and 5.0). Moreover, the degradation slightly decreases between 5.7 and 5.8. A possible explanation is that the binary size distributions of 5.8 is closer to 4.13, at least for the way the basic transfer is performed. It also suggests an effect of the evolution between 5.7 and 5.8. Besides model reuse with 20K is more accurate than model reuse with much more budget (85K) for all target versions, except 5.8. It is not what we would have expected for a learning model: a larger training set for the source model should lead to improved accuracy. This shows that despite the evolution changes both at the code and options between the releases (see Table 1), the use of model reuse does not follow a logical or explainable reason from a machine learning point of view. Thus, overall, simply transferring a prediction model is neither accurate nor reliable: the evolution of the configurations binary size is not captured.

We also measured the degradation of prediction models trained on other versions with 15K (see Figure 2). We can observe that the degradation is less immediate than with the version 4.13 but is still happening, especially on version 5.8 as accuracy is raising to 40% - 50%.

Insights about evolution and options. We first compare feature ranking lists from models trained on versions 4.13 and 4.15. We notice the following evolution patterns:

- **Features unchanged:** Numerous influential features do not change in importance from one version to another. Specifically, out of the top 50 features from both list (the top 50 representing 95% of the feature importance), 29 are the same. It is a key observation that allows one to envision the use of a model from one version to another, even partly;
- **Appearing important features:** 3 important features from the top 50 in version 4.15 did not exist in 4.13, such as CHASH (ranked #18), NOUVEAU_DEBUG_MMU (#21) or BLK_MQ_RDMA (#44);
- **Features losing importance:** 21 features from the top 50 that were important in 4.13 became unimportant in 4.15, meaning they do not impact kernel size anymore, such as RTL8723BE (from #48 to #8892), BT_BNEP_MC_FILTER (#38 to #4182) or AQUANTIA_PHY (#43 to #3348);
- **Features gaining importance:** 18 features from the top 50 that were unimportant in 4.13 became important in 4.15, such as HIPPI (from #6882 to #27), HAMACHI (from #7197 to #50) or NFC_MEI_PHY (from #4921 to #26).

We also compare feature ranking lists from models trained on versions 4.13 and 5.8 and find similar patterns:

- **Features unchanged:** Out of the top 50 features from both list, 21 are the same. It is less than between versions 4.13 and 4.15, but the overlap is still important;
- **Appearing important features:** 12 features from the top 50, such as DMA_COHERENT_POOL (ranked #12), DEBUG_INFO_COMPRESSED(#6) or AMD_MEM_ENCRYPT(#9);
- **Features losing importance :** 24 features from the top 50, such as ATH5K_TRACER (from #20 to #10423), DQL (#40 to #4153) or FDDI (#32 to #2532);

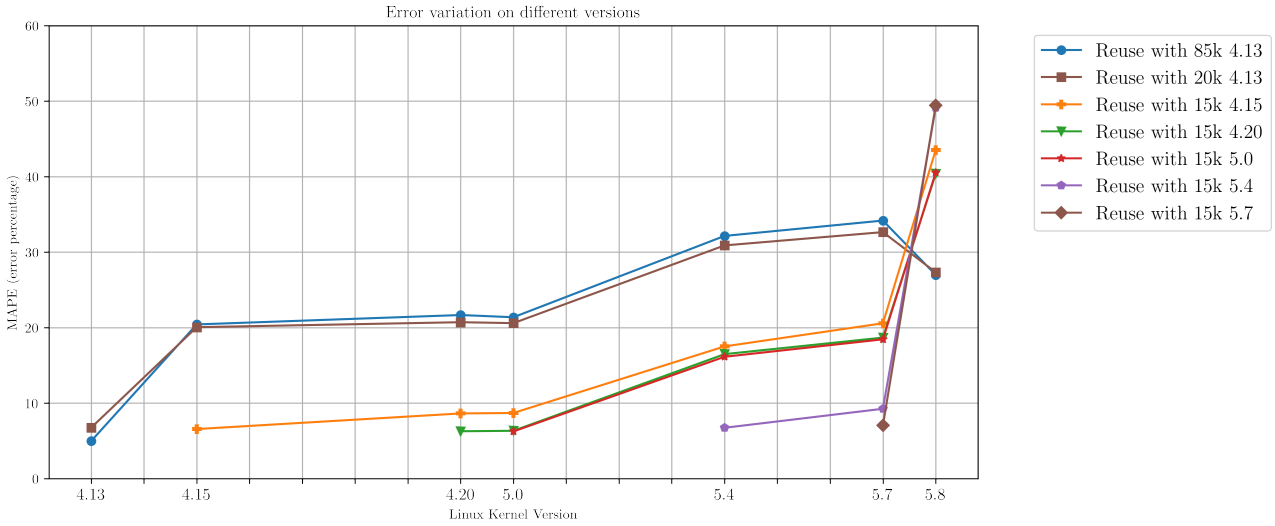

Figure 2: Accuracy of prediction models, trained on 4.13, with training set size 20K and 85K, when applied on later versions.

- **Features gaining importance :** 17 features from the top 50, such as LOCKDEP (#5639 to #22), PROVE_LOCKING (#2362 to #19) or SND_SOC_SI476X (#7075 to #31)

Our observations show that numerous features involved in different evolution patterns can cause the degradation of a prediction model. The impact of new features, unknown by the old model, but also the changes in importance of known features, challenge prediction model trained on a specific version. Interestingly, we did not find important features that were removed between version 4.13 and 4.15. Another important insight is that a large subset of features remain important across versions: one can leverage this knowledge for building a prediction model. Overall, there is a potential to transfer a model from one version to another under the conditions new features together with the effects of important features are correctly handled. In fact, the insights drive the design of TEAMS: more details in the next section.

The **evolution does impact configuration prediction and the degradation quickly occurs: from less than 5% to 20% (only after 4 months of evolution) and up to 32% for recent versions.** The reuse of a prediction model on different versions is not a satisfying solution, calling to other approaches.

4 Evolution-aware Model Shifting

In order to deal with the degradation issue highlighted in the previous section, we now present *evolution-aware model shifting* (TEAMS), a method to transfer a prediction model for a given source version onto a target version.

4.1 Heterogeneous Transfer Learning Problem

Owing to the huge configuration space of the Linux kernel, it is impractical to re-learn at every release. To alleviate this issue, transfer learning was proposed as a new machine learning paradigm. It transfers the knowledge of a model built for a specific source domain (where sufficient measured/labeled data are available) to a related target domain (with little or no additional data) [18]. Figure 1 gives the general principle about how transfer learning works. Here, to make predictions over the Linux kernel version 5.8, we could directly reuse the performance model A built from the source version 4.13. Basically, the source model A is adapted to consider the aligned set of features from both source and target domains (i.e., model A'). Finally, the target model B is trained with only a few measured configurations in the target domain B plus the knowledge from the modified source model A'.

However, the evolution of Linux brings a specific scenario for transfer learning: configuration options for the source version (*e.g.*, 4.13) are not the same as further, target versions (*e.g.*, 5.8). It is worth noting in Table 1 that the number of options keeps increasing over versions. Between two versions, numerous options appear while some others disappear. If a model trained on a specific version cannot handle this new set of options, it will most likely have a negative effect on the accuracy of the size prediction. In terms of machine learning, since options are encoded as features (see Section 3), the feature spaces between the source and target version are non-equivalent.

It is an instance of an *heterogeneous* transfer learning problem [18]. This case is potentially more challenging than *homogeneous* transfer learning in which the set of configuration options remains fixed over time. It can be a serious issue to not consider *e.g.*, new options that can have impacts on kernels' sizes. Technically, prediction models assume that values (*e.g.*, 0 or 1) for a pre-defined set of features are given. If the set of features changes (as it is the case for the evolution of a configurable system), the predictions cannot be done. Hence numerous transfer techniques developed for configurable systems are simply not applicable.

The problem of heterogeneous transfer learning has recently caught attention in different domains (*e.g.*, image processing) with different assumptions and "gap" between the source and the target [18]. The intuition is that, for Linux, the shared set of features can be exploited to effectively transfer predictions.

4.2 Principles

The major challenge is to bridge the gap between the feature spaces. We rely on two steps: (1) feature alignment, which deals with the differences between features' sets among two versions; (2) the learning of a transfer function that map features' source onto target size. For realizing *feature alignment*, we distinguish three cases:

- **commonality**: options that are common across versions (*i.e.,* options have the same names) are encoded as unique, shared features. There are two benefits: we can reuse a prediction model obtained over a source version *"as is"*, without having to retrain it with another feature scheme; we do not double the number of features, something that would increase the size of the learning model up to the point some learning algorithms might not scale. The anticipated risk is that some Linux options, though common across versions, may drastically differ at the implementation level, thus having different effects on sizes. We deal with this risk through the learning of a transfer function that aims to find the correspondences between the source and the target (see below);
- **deleted features**: options that are in the source version, but no longer in the target version: we add features in the target version with one possible value, *"0"* or *"1"*. Observations show that putting *"1"* as the default always gives slightly better accuracy.
- **new features**: options that are not in the source version, but only introduced in the target version: we ignore them when predicting the performance value since the source model cannot handle them, but we keep them in the target dataset.

Feature alignment alone is not sufficient; it is mainly a syntactical mapping at this step. There is a need to capture the *transfer function*, *i.e.,* the relationship between the source features, the source labels (kernel size of each configuration under source version), the target features, and the target labels. This transfer function should be learned. Owing to the complexity of the evolution, a "simple" linear function is unlikely to be accurate – our empirical results confirm the intuition, see next section. In contrast to existing works that rely on linear regression models for "shifting" the prediction models [7], [47], [48], we rely on more expressive learning models, capable of capturing interactions between source and target information.

Note that the feature alignment is a completely automated process and relies on the high similarity between the features spaces. In case of too disjoint features spaces, this solution would likely fail, and other solutions should be considered [18].

4.3 Algorithm

Figure 3 outlines the process for the tEAMS algorithm that gives the four key significant steps:

- ① Target dataset and Source model acquisition: Train or acquire a robust model on measurements from the source version and a dataset from the target version;
- ② Feature alignment: If the source and the target do not have the same set of options, an alignment of the feature spaces is applied (*e.g.,* as described in Section 4.2);
- ③ Target prediction: Using the source model, predict the value of the target data and add this prediction as a new feature in the target dataset;
- ④ Shifting model training: Using the enhanced target dataset, train a new model (*e.g.,* with a Gradient Boosting Tree algorithm capable of handling interactions).

Note that the source model is usually already trained beforehand, and its training step can be skipped in this case. Overall, our solution is fairly easy to implement and deploy. Moreover, once we train a source version the learning of the transfer function scales well (see next section).

4.4 Variant: Incremental Transfer

A possible variant of this technique is to use it in an incremental fashion, and to replace the source model by an already transferred model for a previous version. In the end, such a model consists of a source model, shifted multiple times in a row through multiple intermediate target versions until the final target version.

This variant could potentially give more accurate results, since the complexity of the transfer is spread over multiple models. The farther two versions are from each other, in terms of software evolution, the more performances-impacting changes can happen. A transfer model that handles two distant versions has to deal with all changes between these two versions at once, while in an incremental process, each model only has to deal with a fraction of the changes. On the other hand, we know that machine learning models are imperfect and error prone, even if the error is limited. Relying on a series of machine learning models can turn out to be risky, as these errors can be spread and amplified over the multiple models.

5 Effectiveness of Transfer Learning

Our goal is to evaluate the cost-effectiveness of our approach tEAMS in the context of Linux evolution. The effectiveness is the accuracy of the prediction model and its ability to minimize prediction errors as much as possible. If the source version and the target version has very little in common, configuration performance knowledge may not transfer well. In such situations, transfer learning can be unsuccessful and can lead to a "negative" transfer [18]. Specifically, we consider that the transfer is negative when learning from scratch directly onto the target version – without transfer and using only the limited measured target data available for transfer – leads to better accuracy than a transfer model with the same budget. Thus, we aim to answer the following research question:

(RQ2) What is the accuracy of our evolution-aware model shifting (tEAMS) compared to learning from scratch and other transfer learning techniques? The accuracy of tEAMS depends on the investments realized for creating or updating the prediction models. Specifically:

- the number of configurations' measurements over the target model used to train the prediction model:

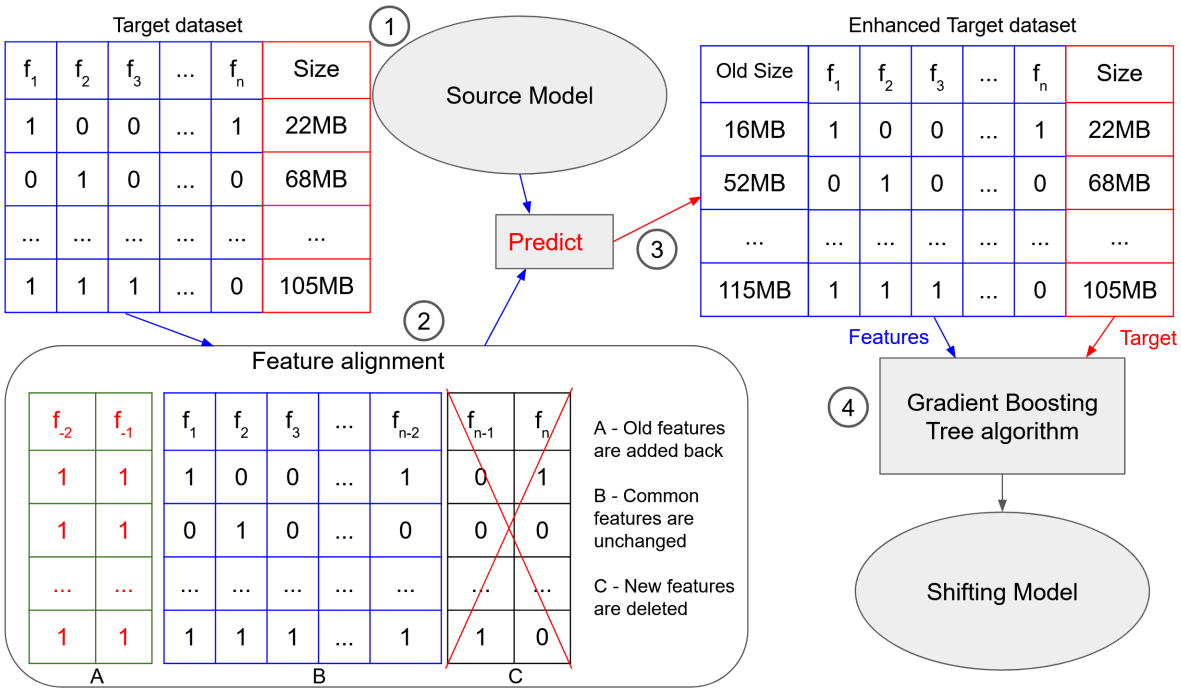

Figure 3: Model shifting process: ① Target dataset and Source model acquisition; ② Feature alignment for source model compatibility; ③ Enhance the dataset with predicted size; ④ Shifting model training

non-transfer learning (*i.e.,* from scratch) uses the same training set and we can confront our results;

- the number of configurations' measurements over the source version used to train the prediction model: from large training sets to relatively small ones;

Hence, we address RQ2 through different cost scenarios and we can identify for which investments TEAMS is effective.

5.1 Experimental settings

5.1.1 Dataset

For training and validating the prediction models, we use the same kernels' versions and configurations' measurements as in Section 3 and Table 1.

5.1.2 Training size for targeted versions

We vary the number of configurations' measurements amongst the following values $\{1K, 5K, 10K\}$. $5K$ corresponds to around 5% of the 95K configurations in the dataset of 4.13: it is representative of a scenario in which a relatively small fraction is used to update the model for a target version. As we have invested around 20K per version, we needed to take care of having a sufficiently large testing set for computing the accuracy. In particular, we cannot use the whole configuration measurements since otherwise we cannot simply compute the accuracy of the models. We stop at $10K$ since then the testing set can be set to around $10K$ too. Moreover, we repeat experiments 5 times with different training sets and report on standard deviations.

5.2 Baseline Methods and Parameters

5.2.1 Source prediction model for TEAMS

We use a prediction model trained with 4.13. It is the oldest version in our dataset and as such, we investigate an extreme scenario for the evolution and potentially the most problematic for transfer learning. As in Section 3, we rely on GBTs, the most accurate solution whatever the training set size is. We train GBTs over 4.13 with two different budgets: 85K configurations and 20K configurations. Hereafter, we call these prediction models 4.13_85K and 4.13_20K respectively.

5.2.2 Incremental TEAMS

We use the incremental method in the same way as without increment, only changing the base model for each increment by the model trained on the previous version. We also have two different series of increment, one based on the 4.13_85K model, the other on the 4.13_20K model. For instance, the first series starts with the transfer from model 4.13_85K to version 4.15 with a shifting model T4.15. This process creates a prediction model 4.15 composed of the two models : 4.15 = T4.15(4.13_85K). The next step is to transfer that model to version 4.20 : 4.20 = T4.20(4.15). At the end of that series, we have a model looking like this :

5.8 = T5.8(T5.7(T5.4(T5.0(T4.20(T4.15(4.13_85K))))))

5.2.3 Learning from scratch

The most simple way to create a prediction model for a given version is to learn from scratch with an allocated budget. We use the GBTs algorithm to create prediction models from scratch, for each version of our dataset. As previously stated, the study in [33] shows that GBTs were a scalable and accurate solution compared to other state-of-the-art solutions. Furthermore, the superiority of GBTs is more apparent when small training sets are employed. This quality of GBTs is even more important when learning for the target version where the budget for updating the model is typically limited – we investigate budget with less

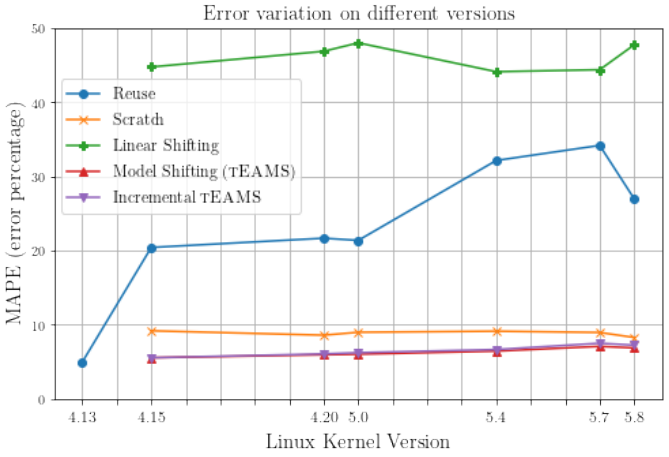

Figure 4: Accuracy of TEAMS 4.13_85K compared to other techniques using 5K examples for the target

than 20K measurements (see above *"Training size for targeted versions"*). We replicated the experiments of 4.13 on other versions: linear models, decision trees, random forests, and neural networks give inferior accuracy compared to GBTs, especially for small sampling size (*e.g.*, 10K). Thus, we do not report results of other learning algorithms and keep only GBTs, the strongest baselines for learning from scratch or for transfer learning techniques.

5.2.4 TEAMS *with linear-based transfer function*

In most state-of-the-art cases, model shifting processes use a simple linear learning algorithm to create a shifting model and they are performing quite well (e.g. [7], [23]). We rely on such a linear transfer function and also apply feature alignment as part of TEAMS.

5.3 Results

Figure 4 depicts the evolution of the MAPE for the reuse of the model 4.13_85K (*i.e.*, 4.13 with a 85K of training set), and the 4 studied techniques trained using 5K examples. We can quickly see that linear model shifting has more than 40% MAPE over all versions and is not accurate at all. It is surprisingly the worst by far, in particular, in comparison with the direct reuse of the prediction model. The standard deviation for Linear model shifting is between 1.5 and 3, while all other techniques are much more stable with a standard deviation always at 0.1 or less. Moreover, learning from scratch with 5K examples allows to create models having an MAPE between 8.2% and 9.2% quite consistently. On the other side, TEAMS with the same budget offers a lower MAPE from 5.6% on version 4.15 to 6.8% in version 5.8 with a peak at 7.1 in version 5.7. It is worth noting that TEAMS MAPE increases a little bit at each version. However, the increase is not significant and remains low.

Comparing TEAMS with its incremental variant shows a very slight but constant advantage over the variant, which also show better results then learning from scratch.

Table 2 gives the results for MAPE with combinations of different models used (4.13_20K and 4.13_85K) and training set size (1K, 5K, 10K) for the scratch baseline,TEAMS and Incremental TEAMS. The other techniques already performed poorly, Table 2 hence focuses on the three competing solutions in Figure 4. We now report on these results.

Impact of training set size over target. As illustrated in Tables 2, when decreasing the training transfer set for the newer versions to 1K example (1% of the original set), the MAPE increases to 14.9%-16.7% depending on the version. Whereas, the MAPE for TEAMS only increases to 6.7%-10.6%, with the same trend consistently increasing MAPE over time (and versions). For Incremental TEAMS, the error rate increases faster up to 13.3 on version 5.8. On the other hand, if we increase the training set to 10K (10% of the original set), accuracy when learning from scratch gets better, with 7.0% to 7.7% MAPE. For TEAMS, the accuracy also gets better, from 5.2% MAPE on version 4.15 to 6.1% on version 5.7 and then slightly further improves to 6.1% on version 5.8. We observe the same trend for Incremental TEAMS, going up to 6.5% on 5.7 and then to 6.2 on 5.8.

Impact of TEAMS source model. We measured the same variations using model 4.13_20K as the source model, which was built from 20,000 examples instead of 85,000. This affects TEAMS by slightly increasing the MAPE. In particular, we observe that for TEAMS: 1) with 1K, the MAPE varies from 8.5% to 11.6%, 2) With 5K, it varies from 6.7% to 7.9%, and 3) with 10K, it varies from 6.2% to 6.7%. Whereas for learning from scratch, we observe that: 1) with 1K, the MAPE varies from 14.9% to 16.7%, 2) With 5K, it varies from 8.3% to 9.2%, and 3) with 10K, it varies from 7.04% to 7.67%. Therefore, our results show that TEAMS outperforms the two baselines, regardless of the size of training sets. In this situation, Incremental TEAMS also shows slightly better results than TEAMS in some cases. At 10k, Incremental TEAMS beats TEAMS on all versions except 5.7, and at 5k, only for versions 4.20 and 5.0. Given the fair increase in error rate at 1k, Incremental TEAMS seems to be very sensitive to higher error rate from previous versions.

Computational cost of training. We performed our experiments on a machine with an Intel Xeon 4c/8t, 3,7 GHz, 64GB memory. Training from scratch with 1K, 5K and 10K examples take respectively 21, 195 and 407 seconds. Learning with TEAMS takes a little more time with 60, 288 and 604 seconds for the same number of examples. The training time of TEAMS for updating a prediction model is thus affordable and negligible compared to the time taken to build and measure the kernel configurations. The overall cost of training is mainly due to the training of the source model (details can be found in [33]) which is done only once. As a final note, building kernels and gathering configurations data (see Table 1) is by far the most costly activity – the time needed to train the prediction model out of data through either transfer learning or from scratch (a few minutes) is negligible.

Version	Scratch			tEAMS						Incremental tEAMS					
				4.13_20K			4.13_85K			4.13_20K			4.13_85K		
	1k	5k	10k	1k	5k	10k	1k	5k	10k	1k	5k	10k	1k	5k	10k
4.15	16.72	9.19	7.46	8.46	6.69	6.21	6.73	5.56	5.19	8.46	6.69	6.21	6.73	5.56	5.19
4.20	16.39	8.60	7.12	8.85	6.94	6.22	7.64	5.96	5.44	9.49	6.89	6.15	8.39	6.08	5.46
5.0	15.50	8.99	7.07	9.14	7.04	6.34	7.80	6.03	5.48	10.32	6.99	6.15	8.84	6.24	5.63
5.4	16.06	9.14	7.67	9.76	7.06	6.39	9.01	6.45	5.71	11.64	7.23	6.10	10.56	6.66	6.07
5.7	15.63	8.96	7.59	11.56	7.85	6.69	10.13	7.09	6.12	13.77	7.90	6.75	12.57	7.51	6.50
5.8	14.91	8.29	7.04	11.47	7.27	6.41	10.62	6.88	6.06	13.82	7.58	6.39	13.29	7.26	6.19

Table 2: MAPE for Scratch and tEAMS, with varying source models and training set sizes for the target

tEAMS and Incremental tEAMS are more accurate solutions than learning from scratch and linear model shifting to predict Linux Kernel size on different versions. Also, Incremental tEAMS shows results mostly worse than tEAMS and without significant improvement. Even with different source models and training set sizes, **tEAMS keeps better and acceptable accuracy with 6.9% MAPE on the latest 5.8 version leveraging model trained on 3 years old data.**

6 Discussions

Integration of tEAMS in the Linux project. Our results suggest that we can now provide accurate prediction tools that can be used on 7 versions spanning 3 years of period. In fact, it is an additional advantage of tEAMS compared to non-transfer learning methods that stick to a specific version. This ability is important, since older versions of the kernel are still widely used. Long term support (LTS) releases are particularly relevant since (1) they are maintained over a period of 6 years (2) they are considered by other related communities, like the Android one. The version 5.4 of our dataset is an LTS release and tEAMS can be used in this context.

Linux practitioners interested in a specific release, not present in our dataset, could well invest some resources to obtain a new model. For example, we have not considered version 4.19 (a LTS release). Non-transfer learning methods would be unable to reuse their models while tEAMS provide state-of-the-art cost-accuracy results.

We envision to integrate tEAMS as part of the ongoing continuous integration effort on the Linux kernel. We have released a tool, called kpredict[3], that predicts the size of the kernel binary size given only a .config file. kpredict is written in Python, available on pip, and supports all kernel versions mentioned in this article. A usage example is as follows:

```
> curl -s http://tuxml-data.irisa.fr/data/
configuration/167950/config -o .config
> kpredict .config
> Predicted size : 68.1MiB
```

whereas the actual size of the configuration (see http://tuxml-data.irisa.fr/data/configuration/167950/ for more details) is 67.82 MiB.

Recently KernelCI [49], the major community-effort supported by several organizations (Google, Redhat, etc.), has added the ability to compute kernel sizes and this functionality is activated by default, for any build. Hence tEAMS will benefit from such data. Besides, the current focus of KernelCI and many CI effort is mostly driven by controlling that the kernels build (for different architectures and configurations). It is not incompatible with the prediction of kernel sizes since we did not employ a sampling strategy specifically devoted to this property. We rely on random configurations that are used to cover the kernel and find bugs (see *e.g.,* [50]). In passing tEAMS can benefit from kernel sizes' data while the CI effort continues to track bugs.

Besides, the development cost of integrating this process is fairly small and requires little maintenance. All steps in the process, including feature alignment and adaptation of the learning model to the new version, are fully automated and do not require the intervention of kernel developers or maintainers. Our engineering experience with kpredict is that the overhead of implementing transfer learning steps is not much higher than learning from scratch.

Is the cost of tEAMS affordable for Linux? Under the same cost settings, tEAMS shows superior accuracy compared to non-transfer learning and other baselines. Still, one can wonder whether measuring thousands of configurations is practically possible, especially when there is a new release. Specifically, results show that with 5K measurements we can obtain almost stable accuracy. So, is the build of 5K affordable for Linux? To address this question, we investigated the number of builds realized by KernelCI and asked the leaders of the project. At the time of the publication, KernelCI is able to build 200 kernels in one hour. We then analyze the retrospective cost of measuring 5K as part of our experiments. On average, our investments sum around 260 seconds per machine whatever the version (see Table 1, page 6). That is, with 15 machines (16 cores) full time during 24 hours, we can already measure 5K configurations. As a side note, the numbers should be put in perspective: Linux exhibits thousands of options (see Table 1) and has a large community with many contributors and organizations involved. Overall, though the cost itself is affordable from a computational point of view, there is a tradeoff to find between (1) accuracy; (2) value for the community. This tradeoff should be considered for any configurable system. In any case tEAMS has demonstrated being the most cost-effective approach.

tEAMS versus Incremental tEAMS As we investigated the differences in accuracy between the two techniques, we observed that both show very similar results when using a significant amount of examples, while Incremental tEAMS drops in accuracy when having a reduced training set. This shows that it is very sensitive to low accuracy models in series. On the other hand, if the investment in measurements is sizeable, it would be wise to consider it.

3. https://github.com/HugoJPMartin/kpredict/

When it comes to using and sharing a model, one should consider the cost of having an incremental solution as it comes at a cost. The first is the size of the model, as a solution composed of a multitude of models also means a larger size of the file to share. The other is the time taken to predict a value, which can be a critical point depending on the context. If using one or two models in series usually take less than a second, an incremental solution can take few seconds. If the model is used in a user interface, such an high response time can be considered a severe drawback.

TEAMS and transfer functions As part of TEAMS several transfer functions can be considered for shifting a prediction model. Results show that simple linear regression is clearly not effective. An alternative is to add interactions' terms as part of the linear regression. As stated in [4], for p options, there are p possible main influences, $p \times (p-1)/2$ possible pairwise interactions, and an exponential number of higher-order interactions among more than two options. In the worst case, all 2-wise or 3-wise interactions among the 9K+ options are included in the model, which is computationally intractable. Even if a subset of options is kept, there is a combinatorial explosion of possible interactions. Another possibility is to consider linear models with regularized regression. However, prior experiments [33] for training prediction models over Linux 4.13 showed that Lasso [51] or ElasticNet have severe limitations in terms of accuracy. We chose GBTs for their inherent ability to capture non-linear behavior or options' interactions and empirical results show high accuracy. However we do not claim that the use of GBTs is the best solution. In fact, as TEAMS is capable of handling other transfer functions, we plan to consider other learning techniques (*e.g.,* neural networks) and investigate tradeoffs between accuracy, computational cost, and interpretability.

When and how should we update TEAMS? Results show that our solution of transfer learning works well over a long period of three years from 2017 to 2020. Meaning that up to today, transferring is better than re-learning from scratch. However, eventually, in future evolution and releases, say in 2021 or 2022, transfer learning might degrade. Therefore, at some point, one must learn a new prediction model as the source on which to apply further transfer learning with our solution. Quantifying exactly when one must re-learn the performance model is out of the scope of this paper and is left for future work. In practice, developers can set a limit of acceptable error rate under which transfer can be still applied and beyond which re-learning a prediction model is necessary.

Is TEAMS applicable to other configurable systems? We investigated in this paper a case of heterogeneous transfer learning where our considered configurable system evolves in time and space, i.e., the set of options changes on every release over time. Therefore, other configurable systems that evolve while the set of options also changes can be ideal candidates to leverage our results and TEAMS. For example, `curl` is a widely used configurable system that has now more than 200 options. `curl` does evolves frequently [52], and has performance concerns. GCC [53] and Clang [54] (compilers), Apache Cassandra [55] (database), Amazon EC2 (Web cloud service), OpenCV [56] (image processing), Kafka [57] (distributed systems) are other examples of configurable systems that continuously add or remove options, maintain their code base at a fast pace, and with strong performance requirements. These systems have active user communities and have already been considered in the context of performance prediction of configurable systems (see [1], [19], [58], [59], [60], [61]), but without considering the evolution of their options' sets. In the future, we plan to replicate our study for such subject systems.

7 Threats to Validity

Threats to *internal validity* are related to the sampling used as training set for all experiments. We deliberately used random sampling to diversify our datasets of configurations. To mitigate any bias, for each sample size, we repeat the prediction process 5 times. For each process, we split the dataset into training and testing which are independent of each other. To assess the accuracy of the algorithms, we used MAPE that has the merit of being comparable among versions and learning approaches. Most of the state-of-the-art works use this metric for evaluating the effectiveness of performance prediction algorithms [1]. Another threat to internal validity concerns the (lack of) uniformity of randconfig. Indeed randconfig does *not* provide a perfect uniform distribution over valid configurations [40]. The strategy of randconfig is to randomly enable or disable options according to the order in the Kconfig files. It is thus biased towards features higher up in the tree. The problem of uniform sampling is fundamentally a satisfiability (SAT) problem. However, to the best of our knowledge, there is no robust and scalable solution capable of comprehensively translating Kconfig files of Linux into a SAT formula, especially for the new versions of Linux. Second, uniform sampling either does not scale yet or is not uniform at the scale of Linux [62], [63], [64], [65].

Looking at the decrease gap in accuracy between learning from scratch and TEAMS the more examples are given for training, we can expect that at some point, learning from scratch would be better than TEAMS. We did not investigate further as it would require an important effort to gather so much more examples on all the studied versions, but it would be interesting to know *if* such turning point exists and *where* is that turning point in term of number of examples. While we highlight how good transfer is with limited data, we did not investigate enough how good it is with a lot of data.

Regarding the analysis with the feature ranking list, some problems may arise that can threaten results of Section 3.2. First, the importance is based on a tree structure, and even if we average the ranking out of multiple models, the way the tree is built can create a bias. Second, the importance does not show if the feature has a positive or negative impact. Hence, there is a risk of having a feature evolving that gets the same importance value but with inverse impact from one version to another. Other interpretable techniques are needed to gain further insights.

Threats to *external validity* are related to the targeted kernel versions, targeted architecture (x86), targeted quantitative property (size), and used compilers (gcc 6.3). We have spanned different periods of the kernel from 2017 to 2020, considering stable versions and covering numerous

evolutions (see Table 1). We could consider minor releases, but the practical interest for the Linux community would be less obvious. We are expecting similar results since the evolution in minor release is mostly based on patches and bug fixings. We have considered the most popular and active architecture (x86). Another architecture could lead to different options' effects on size. The idea of transfer learning could well be applied in this context and is left as future work. A generalization of the results for other non-functional properties (*e.g.,* compilation time, boot time) would require additional experiments with further resources and engineering efforts. In contrast to *e.g.,* compilation time, the size of a the kernel is not subject to hardware variations and we can scale the experiments with distributed machines. Hence, we focused on a single property to be able to make robust and reliable statements about whether learning approaches can be used in such settings. There is also a clear interest for the Linux community (see Section 6). Our results suggest we can now envision to perform an analysis over other properties to generalize our findings. Finally, as we used Linux kernel, we do not generalize to all configurable systems, especially to those with few options and small-medium code size. However, this is not the goal of this study since we purposely targeted Linux due to its complexity (in options and LOC). Further experimentation on other case studies remains necessary.

8 Related Work

Several empirical studies [40], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75] have considered different aspects of Linux (build system, variability implementation, constraints, bugs, compilation warnings). However, most of the works did not concretely build configurations in the large, a necessary and costly step for training a prediction model. There are two exceptions. Melo *et al.* [76] compiled 21K valid random Linux kernel configurations over a fixed version with the goal of analyzing configuration-dependent warnings. Acher *et al.* [33] compiled 95K+ configurations over version 4.13. In contrast, our study targets different versions of the kernel and many more configurations.

Numerous works have investigated the idea of learning performance of configurable systems [4], [5], [7], [10], [12], [19], [58], [77], [78], [79], [80], [81], [82], [83], [84], [85], spanning different application domains [1], such as compression libraries, database systems, or video encoding. However, only dozens of options over a few configurations are usually considered [1]. Linux has received little attention in a learning context, certainly due to its comparatively high complexity. Only a few approaches try to predict or understand non-functional properties (*e.g.,* size) of Linux kernel configurations. Siegmund *et al.* [86] only considered 25 options and 100 random configurations' sizes. In this study, we make no assumptions about the supposed influence of some options; our experiments consider the entire 12K+ options of the Linux kernel on the x86_64 architecture. We also use 243K+ configurations over seven versions.

Besides the problem of optimizing performance of configurable systems (*i.e.,* finding the best configuration) has attracted attention [1], [10], [47], [87]. In this article, we specifically target a regression problem (*i.e.,* predicting binary size for any configuration). However, tEAMS is providing an accurate regressor that can be used for finding the smallest kernels (*e.g.,* as in [47]). An open question is whether regressor-based optimization is competing with dedicated optimization approaches (*e.g.,* learning to rank [1]) at the scale of Linux.

White-box approaches [88], [89], [90] have been proposed to inspect the implementation of a configurable system in order to guide the performance analysis. Static dataflow analysis or profiling are typically used to help understanding options and their interactions in a fine-grained way. It can provide valuable insights that are complementary to what reported in Section 3.2. Though such investigations are interesting to conduct, white-box approaches should account for the challenges raised by Linux (*e.g.,* high number of options and interactions, difficulties of analysing source code, build files, and linker/compiler behavior).

Transfer learning has attracted interest with the promise to save resources by reusing the knowledge of performance under different settings (*e.g.*, hardware settings) [7], [11], [19], [20], [21], [22], [23], [24], [25], [26], [27], [91]. Existing approaches have considered homogeneous transfer learning, *i.e.,* the set of options remain stable under different settings. However, homogeneous transfer learning does not apply to variability in space and time. Hence, beyond the limitation in the number of options and configurations considered in previous studies, most existing approaches of transfer learning do not take into account the software evolution and the impacts on performance configurations. In this context, we propose a *heterogeneous transfer learning* solution. To the best of our knowledge, we provide the first concrete evidence that heterogeneous transfer learning can model variability in space and time of configurable systems accurately. Our insights (see Section 3.2) suggest that there is a common configuration knowledge across versions *e.g.,* some features remain important predictor variables. This knowledge can be used as part of the transfer function and the sampling. For instance, L2S (Learning to Sample) [21] employs targeted sampling as opposed to random sampling. L2S concentrates on interesting regions of the configuration space and could be used to target features that are important whether the Linux version is. It is an open question whether L2S can scale to Linux, deal with the numerous logical constraints (see Section 7) and be effective within the heterogeneous transfer learning setting.

Beyond software systems, transfer learning is subject to intensive research in many domains (*e.g.,* image processing, natural language processing) [16], [17], [92]. Different kinds of data, assumptions, and tasks are considered. The evolution of Linux calls to specifically tackle a regression problem with hetereogenous feature spaces. Negative transfer is an important concern [92]: divergence measures between source and target have been proposed as well as dedicated algorithms [16], [17], [93], [94]. Our empirical results show that tEAMS does not suffer from negative transfer and Section 3.2 provides some insights about possible reasons (*e.g.,* similarity among important options). A challenging research direction is to develop theories providing guarantees about the transfer.

There are many studies in the literature of software engineering applying heterogeneous transfer learning for

defect prediction [95], [96], [97], [98], [99]. They are used for handling a classification problem instead of a regression problem as in our case. Moreover, researchers use software quality metrics as features to predict cross-software defects. Instead, we use configuration options to predict cross-version performance (*i.e.*, size).

Finally, another line of research investigates Linux evolution, but without learning techniques. She *et al.* investigated the Linux variability model and how it evolves over time [100]. Dintzer *et al.* [71], [101] proposed an approach to analyze the delta in the Linux feature model. They further studied how feature model evolution leads to co-evolution of other Linux artefact. Passos et al. [102] further investigated co-evolution patterns between Linux feature model and Linux artefacts make files and C code files. Ren et al. [37] presented an analysis of how Linux kernel performance has evolved over seven years. The study considers properties different than size (*e.g.*, latency), default configurations, and stops at version 4.20. In line with our findings (RQ1), the authors reported noticeable changes in the performance for some evolution of Linux (*e.g.*, after version 4.15). Lawall et al. [103] investigated Linux Kernel evolution over 10 years, and in particular patch application. Lu et al. [104] also conducted a comprehensive study of file-system code evolution of Linux Kernel evolution over 8 years. Our current work distinguishes from [37], [71], [84], [85], [100], [101], [102], [103], [104] by focusing on another angle of Linux kernel evolution, namely performance prediction of configurations.

9 Conclusion

In this paper, we showed that the evolution of Linux has a noticeable impact on kernel sizes of configurations with a large-scale study spanning 7 versions over 3 years and 240K+ configurations. As a result, a size prediction model learned from one specific version quickly becomes obsolete and inaccurate, despite a huge initial investment (15K hours of computation for building a training set of 90K configurations). We developed a heterogeneous transfer evolution-aware model shifting (tEAMS) learning technique. It is capable of handling new options that appear in new versions while learning the function that maps the novel effects of shared options among versions. Our results showed that the transfer of a prediction model leads to accurate results (the prediction error (MAPE) remains low and constant) without the need of collecting a very large corpus of measurements' configurations. With only 5K configurations, we can transfer the model made in September 2017 for the 5.8 version released in August 2020.

Linux is an extreme case of a highly complex configurable system that rapidly evolves. Though not all systems have ≈ 14.500 options and such a frequency of commits, many software systems face the same problem of dealing with variability in space (variants) and time (versions). The increasing adoption of continuous integration in software engineering has exacerbated the problem of adding, removing, or maintaining configuration options (being realized as feature toggles, command line parameters, conditional compilation, etc.). The fact that transfer learning works for Linux is reassuring, which has a great potential impact on Linux developers and integrators. There is hope that the effort made for one version of a software system can be reused through transfer.

However, we cannot generalize at this step of this research. As future work, we plan to investigate or revisit several research questions in other software engineering contexts. Compilers (*e.g.*, GCC [53] and Clang [54]), database systems (*e.g.*, Apache Cassandra [55]), Web cloud service (*e.g.*, Amazon EC2), image processing (*e.g.*, OpenCV [56]), distributed streaming platforms (*e.g.*, Kafka [57]) or data transfer tools (*e.g.*, curl [52]) are examples of software systems that continuously add or remove options, maintain their code base at a fast pace, and with strong performance requirements. Owing to the continuous evolution of options' sets of modern software systems, our study calls to replicate our effort: To what extent evolution degrades the effects of options on a non-functional property like execution time, energy consumption, or security? Is transfer learning (*e.g.*, tEAMS) cost-effective for updating a prediction model?

References

[1] J. A. Pereira, H. Martin, M. Acher, J.-M. Jézéquel, G. Botterweck, and A. Ventresque, "Learning software configuration spaces: A systematic literature review," 2019.

[2] 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 Control*, vol. 20, no. 3-4, pp. 487–517, Sep. 2012.

[3] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *ICSE*, 2012, pp. 167–177.

[4] S. Kolesnikov, N. Siegmund, C. Kästner, A. Grebhahn, and S. Apel, "Tradeoffs in modeling performance of highly configurable software systems," *SoSyM*, vol. 18, no. 3, pp. 2265–2283, Jun 2019.

[5] Y. Zhang, J. Guo, E. Blais, K. Czarnecki, and H. Yu, "A mathematical model of performance-relevant feature interactions," in *Proceedings of the 20th International SPLC*. ACM, 2016, pp. 25–34.

[6] J. Guo, J. H. Liang, K. Shi, D. Yang, J. Zhang, K. Czarnecki, V. Ganesh, and H. Yu, "SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines," *Software & Systems Modeling*, Jul 2017.

[7] P. Valov, J.-C. Petkovich, J. Guo, S. Fischmeister, and K. Czarnecki, "Transferring performance prediction models across different hardware platforms," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2017, pp. 39–50.

[8] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Faster discovery of faster system configurations with spectral learning," *ASE*, pp. 1–31, 2018.

[9] S. A. Safdar, H. Lu, T. Yue, and S. Ali, "Mining cross product line rules with multi-objective search and machine learning," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 1319–1326.

[10] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *TSE*, 2018.

[11] P. Jamshidi, J. Cámara, B. Schmerl, C. Kästner, and D. Garlan, "Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots," *arXiv preprint arXiv:1903.03920*, 2019.

[12] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based sampling of software configuration spaces," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE). ACM*, 2019.

[13] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wąsowski, "Variability-aware performance prediction: A statistical learning approach." IEEE, 2013, pp. 301–311.

[14] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *ASE'15*, 2015.

[15] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 284–294.

[16] S. J. Pan and Q. Yang, "A survey on transfer learning," *TKDE*, vol. 22, no. 10, pp. 1345–1359, 2009.

[17] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big data*, vol. 3, no. 1, p. 9, 2016.

[18] O. Day and T. M. Khoshgoftaar, "A survey on heterogeneous transfer learning," *Journal of Big Data*, vol. 4, no. 1, p. 29, 2017.

[19] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar, "Transfer learning for improving model predictions in highly configurable software," in *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Los Alamitos, CA: IEEE Computer Society, 5 2017, pp. 31–41.

[20] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer learning for performance modeling of configurable systems: An exploratory analysis," in *2017 32nd ASE*. IEEE, 2017, pp. 497–508.

[21] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, "Learning to sample: exploiting similarities across environments to learn performance models for configurable systems," in *Proceedings of the 2018 ACM ESEC/SIGSOFT*. ACM, 2018, pp. 71–82.

[22] P. Valov, J. Guo, and K. Czarnecki, "Transferring pareto frontiers across heterogeneous hardware environments," in *Proceedings of the ACM/SPEC ICPE*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 12–23.

[23] M. S. Iqbal, L. Kotthoff, and P. Jamshidi, "Transfer learning for performance modeling of deep neural network systems," in *2019 USENIX OpML*. Santa Clara, CA: USENIX Association, May 2019, pp. 43–46.

[24] X. Wang, T.-K. Huang, and J. Schneider, "Active transfer learning under model shift," in *ICML*, 2014, pp. 1305–1313.

[25] H. Chen, W. Zhang, and G. Jiang, "Experience transfer for the configuration tuning of large scale computing systems," Nov. 20 2012, uS Patent 8,315,960.

[26] E. Kocaguneli, T. Menzies, and E. Mendes, "Transfer learning in effort estimation," *EMSE*, vol. 20, no. 3, pp. 813–843, 2015.

[27] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *2013 35th ICSE*. IEEE, 2013, pp. 382–391.

[28] S. Mühlbauer, S. Apel, and N. Siegmund, "Identifying software performance changes across variants and versions," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 611–622.

[29] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *34th International Conference on Software Engineering*, 06/2012 2012.

[30] L. Torvalds, "The linux edge," *Communications of the ACM*, vol. 42, no. 4, pp. 38–38, 1999.

[31] https://tiny.wiki.kernel.org/, "Linux kernel tinification."

[32] M. Opdenacker, "Bof: Embedded linux size," 2018, embedded Linux Conference North-America.

[33] M. Acher, H. Martin, J. Pereira, A. Blouin, J.-M. Jézéquel, D. Khelladi, L. Lesoil, and O. Barais, "Learning very large configuration spaces: What matters for linux kernel sizes," 2019.

[34] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu, "Set the configuration for the heart of the os: On the practicality of operating system kernel debloating," *POMACS*, vol. 4, no. 1, pp. 1–27, 2020.

[35] J. Alves Pereira, M. Acher, H. Martin, and J.-M. Jézéquel, "Sampling Effect on Performance Prediction of Configurable Systems: A Case Study," in *International Conference on Performance Engineering (ICPE 2020)*, 2020. [Online]. Available: https://hal.inria.fr/hal-02356290

[36] A. Prout, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein *et al.*, "Measuring the impact of spectre and meltdown," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–5.

[37] X. Ren, K. Rodrigues, L. Chen, C. Vega, M. Stumm, and D. Yuan, "An analysis of performance evolution of linux's core operations," in *Proceedings of the 27th ACM SOSP*, 2019, pp. 554–569.

[38] https://www.phoronix.com/scan.php?page=article&item=linux-416-54&num=1, "The Disappointing Direction Of Linux Performance From 4.16 To 5.4 Kernels," November 2019.

[39] https://kernelnewbies.org/Linux_5.4, "TKernelNewbies: Linux_5.4," November 2019.

[40] J. Melo, E. Flesborg, C. Brabrand, and A. Wasowski, "A quantitative analysis of variability warnings in linux," in *Proceedings of the Tenth VaMoS*, ser. VaMoS '16. ACM, 2016, pp. 3–8.

[41] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[42] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, "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*, 2018, pp. 71–82.

[43] C. Molnar, *Interpretable Machine Learning*, 2019, https://christophm.github.io/interpretable-ml-book/.

[44] H. Martin, J. A. Pereira, M. Acher, and J. Jézéquel, "A comparison of performance specialization learning for configurable systems," in *SPLC '21: 25th ACM International Systems and Software Product Line Conference*. ACM, 2021.

[45] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[46] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[47] R. Krishna, V. Nair, P. Jamshidi, and T. Menzies, "Whence to learn? transferring knowledge in configurable systems using beetle," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[48] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, "Learning to sample: exploiting similarities across environments to learn performance models for configurable systems," in *Proceedings of the 2018 ACM ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 71–82.

[49] *KernelCI*. [Online]. Available: https://kernelci.org/

[50] M. Acher, H. Martin, J. Alves Pereira, A. Blouin, D. Eddine Khelladi, and J.-M. Jézéquel, "Learning from thousands of build failures of linux kernel configurations," Inria ; IRISA, Technical Report, Jun. 2019. [Online]. Available: https://hal.inria.fr/hal-02147012

[51] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[52] *cURL*. [Online]. Available: https://curl.haxx.se/docs/releases.html

[53] *GCC*. [Online]. Available: https://gcc.gnu.org/releases.html

[54] *Clang*. [Online]. Available: https://clang.llvm.org/

[55] *Apache Cassandra*. [Online]. Available: https://cassandra.apache.org/doc/latest/new/index.html

[56] *OpenCV*. [Online]. Available: https://github.com/opencv/opencv_contrib

[57] *Kafka*. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/Index

[58] P. Temple, M. Acher, J. Jézéquel, and O. Barais, "Learning contextual-variability models," *IEEE Software*, vol. 34, no. 6, pp. 64–70, 2017.

[59] N. Siegmund, S. Sobernig, and S. Apel, "Attributed variability models: Outside the comfort zone," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 268–278.

[60] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th {USENIX} {NSDI} 17*, 2017, pp. 469–482.

[61] L. Bao, X. Liu, Z. Xu, and B. Fang, "Autoconfig: automatic configuration tuning for distributed message systems." ACM, 2018, pp. 29–40.

[62] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform sampling of sat solutions for configurable systems: Are we there yet?" in *ICST 2019 - 12th International Conference*

on Software Testing, Verification, and Validation, Xian, China, Apr. 2019, pp. 1–12.

[63] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable and nearly uniform generator of sat witnesses," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 608–623.

[64] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform SAT witness generation," in *Tools and Algorithms for the Construction and Analysis of Systems TACAS'15 2015, London, UK, April 11-18, 2015. Proceedings*, 2015, pp. 304–319.

[65] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of SAT solutions for testing," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 549–559.

[66] I. Abal, J. Melo, x. Stănciulescu, C. Brabrand, M. Ribeiro, and A. Wasowski, "Variability bugs in highly configurable systems: A qualitative analysis," *TSE Methodol.*, vol. 26, no. 3, pp. 10:1–10:34, Jan. 2018.

[67] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: a qualitative analysis," in *ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 421–432.

[68] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann, "Linux variability anomalies: What causes them and how do they get fixed?" in *Proceedings of the 10th MSR*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 111–120.

[69] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Trans. Software Eng.*

[70] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A study of feature scattering in the linux kernel," *IEEE Transactions on Software Engineering*, 2018.

[71] N. Dintzner, A. van Deursen, and M. Pinzger, "Analysing the linux kernel feature model changes using fmdiff," *Software and Systems Modeling*, vol. 16, no. 1, pp. 55–76, 2017.

[72] C. Bezemer, S. McIntosh, B. Adams, D. M. Germán, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *EMSE*, vol. 22, no. 6, pp. 3117–3148, 2017.

[73] J. Lawall and G. Muller, "Jmake: Dependable compilation for kernel janitors," in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, 2017, pp. 357–366.

[74] M. Zhou, Q. Chen, A. Mockus, and F. Wu, "On the scalability of linux kernel maintainers' work," in *Proceedings of the 2017 11th Joint Meeting on FSE*, ser. ESEC/FSE 2017, 2017, pp. 27–37.

[75] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the linux kernel," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, 2018, pp. 601–614.

[76] J. Melo, E. Flesborg, C. Brabrand, and A. Wąsowski, "A quantitative analysis of variability warnings in linux," in *Proceedings of the Tenth VaMoS*. ACM, 2016, pp. 3–8.

[77] P. Temple, J. A. Galindo Duarte, M. Acher, and J.-M. Jézéquel, "Using Machine Learning to Infer Constraints for Product Lines," in *Software Product Line Conference (SPLC)*, Beijing, China, Sep. 2016.

[78] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *EMSE*, pp. 1–42, 2017.

[79] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using bad learners to find good configurations," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 257–267.

[80] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)." IEEE, 2015, pp. 342–352.

[81] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer learning for performance modeling of configurable systems: an exploratory analysis." IEEE Press, 2017, pp. 497–508.

[82] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1009–1024.

[83] N. Siegmund, A. Grebhahn, C. Kästner, and S. Apel, "Performance-influence models for highly configurable systems," in *ESEC/FSE'15*.

[84] A. M. Sharifloo, A. Metzger, C. Quinton, L. Baresi, and K. Pohl, "Learning and evolution in dynamic software product lines," in *2016 IEEE/ACM 11th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2016, pp. 158–164.

[85] C. Quinton, M. Vierhauser, R. Rabiser, L. Baresi, P. Grünbacher, and C. Schuhmayer, "Evolution in dynamic software product lines," *Journal of Software: Evolution and Process*, p. e2293, 2020.

[86] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines," in *Software Product Line Conference (SPLC), 2011 15th International*, 2011, pp. 160–169.

[87] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 794–811, 2020.

[88] M. Weber, S. Apel, and N. Siegmund, "White-box performance-influence models: A profiling and learning approach," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1059–1071.

[89] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "White-box analysis over machine learning: Modeling performance of configurable systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1072–1084.

[90] M. Velez, P. Jamshidi, F. Sattler, N. Siegmund, S. Apel, and C. Kästner, "Configcrusher: Towards white-box performance analysis for configurable systems," *Automated Software Engineering*, vol. 27, no. 3, pp. 265–300, 2020.

[91] R. Krishna, T. Menzies, and W. Fu, "Too much automation? the bellwether effect and its implications for transfer learning," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 122–131.

[92] W. Zhang, L. Deng, L. Zhang, and D. Wu, "Overcoming negative transfer: A survey," *arXiv preprint arXiv:2009.00909*, 2020.

[93] W. Dai, Q. Yang, G.-R. Xue, and Y. Yu, "Boosting for transfer learning," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 193–200. [Online]. Available: https://doi.org/10.1145/1273496.1273521

[94] Y. Yao and G. Doretto, "Boosting for transfer learning with multiple sources," in *2010 IEEE computer society conference on computer vision and pattern recognition*. IEEE, 2010, pp. 1855–1862.

[95] H. Chen, X.-Y. Jing, Z. Li, D. Wu, Y. Peng, and Z. Huang, "An empirical study on heterogeneous defect prediction approaches," *IEEE Transactions on Software Engineering*, 2020.

[96] Z. Li, X.-Y. Jing, F. Wu, X. Zhu, B. Xu, and S. Ying, "Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction," *ASE*, vol. 25, no. 2, pp. 201–245, 2018.

[97] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 9, pp. 874–896, 2017.

[98] J. Chen, Y. Yang, K. Hu, Q. Xuan, Y. Liu, and C. Yang, "Multiview transfer learning for software defect prediction," *IEEE Access*, vol. 7, pp. 8901–8916, 2019.

[99] A. Wang, Y. Zhang, H. Wu, K. Jiang, and M. Wang, "Few-shot learning based balanced distribution adaptation for heterogeneous defect prediction," *IEEE Access*, vol. 8, pp. 32 989–33 001, 2020.

[100] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the linux kernel variability model," in *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, ser. LNCS, J. Bosch and J. Lee, Eds., vol. 6287. Springer, 2010, pp. 136–150.

[101] N. Dintzner, A. van Deursen, and M. Pinzger, "FEVER: an approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems," *EMSE*, vol. 23, no. 2, pp. 905–952, 2018.

[102] L. T. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, and J. Guo, "Coevolution of variability models and related software artifacts - A fresh look at evolution patterns in the linux kernel," *Empirical Soft. Eng.*, vol. 21, no. 4, pp. 1744–1793, 2016.

[103] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the linux kernel," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 601–614.

[104] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," in *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, 2013, pp. 31–44.

Hugo Martin is a Ph.D. Student at the University of Rennes 1, France. His research focuses on using interpretable machine learning to better understand configurable systems, especially on systems with colossal configuration spaces such as Linux.

Mathieu Acher is Associate Professor at University of Rennes 1/Inria, France and junior research fellow at Institut Universitaire de France (IUF). His research focuses on reverse engineering, modeling, and learning variability of software intensive systems with different contributions published at ICSE, ASE, ESEC/FSE, SPLC, MODELS, IJCAI, or TSE, JSS, ESEM journals. He was PC co-chair of SPLC 2017 and VaMoS 2020. He is currently leading a research project, VaryVary, on machine learning and variability.

Juliana Alves Pereira is a Postdoctoral researcher at PUC-Rio, Brazil. She was a Postdoctoral researcher at the University of Rennes 1/Inria, France. Juliana received a Ph.D. degree in Computer Science from the University of Magdeburg, Germany, in 2018. Her research thrives to automate software engineering by combining methods from software analysis, machine learning, and meta-heuristic optimization. In recent years, she has published and revised research papers in premier software engineering conferences, symposiums, and journals.

Luc Lesoil is currently a doctoral candidate at the University of Rennes 1, France. For his research work, he combines machine learning and software variability to study deep software variability, with a focus on the limitations of performance models trained on configurable systems.

Jean-Marc Jézéquel is a Professor at the University of Rennes and vice-director of Informatics Europe. He was Director of IRISA, one of the largest public research lab in Informatics in France, from 2012 to 2020. He is also head of research of the French Cyber-defense Excellence Cluster and the director of the Rennes Node of EIT Digital. In 2016, he received the Silver Medal from CNRS. His interests include model driven software engineering for software product lines, and specifically component based, dynamically adaptable systems with quality of service constraints, including security, reliability, performance,timeliness, etc. He is the author of 4 books and more than 250 publications in international journals and conferences. He was a member of the steering committees of the AOSD and MODELS conference series. He also served on the editorial boards of IEEE Computer, IEEE Transactions on Software Engineering, the Journal on Software and Systems, the Journal on Software and System Modeling and the Journal of Object Technology. He received an engineering degree from Telecom Bretagne in 1986, and a Ph.D. degree in Computer Science from the University of Rennes, France, in 1989.

Djamel Eddine Khelladi is a CNRS researcher in the IRISA research lab in the DIVERSE team, Université Rennes 1. Before that he was a Postdoctoral researcher in the Institute for Software Systems Engineering (ISSE) at the Johannes Kepler University Linz. He hold a Ph.D. in the Laboratoire d'Infromatique de Paris 6 (LIP6) at the university of Piere et Marie Curie (UPMC). Research interests include Software engineering, Model-Driven Engineering, Software Evolution, Evolution impacts, Co-evolution, Software Processes.