The Interplay of Compile-time and Run-time Options for Performance Prediction

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

The Interplay of Compile-time and Run-time Options for Performance Prediction

Luc Lesoil, Mathieu Acher, Xhevahire Tërnava, Arnaud Blouin, Jean-Marc Jézéquel
Univ Rennes, INSA Rennes, CNRS, Inria, IRISA
Rennes, France
firstname.last@irisa.fr

ABSTRACT

Many software projects are configurable through compile-time options (*e.g.,* using ./configure) and also through run-time options (*e.g.,* command-line parameters, fed to the software at execution time). Several works have shown how to predict the effect of run-time options on performance. However it is yet to be studied how these prediction models behave when the software is built with different compile-time options. For instance, is the best run-time configuration always the best w.r.t. the chosen compilation options? In this paper, we investigate the effect of compile-time options on the performance distributions of 4 software systems. There are cases where the compiler layer effect is linear which is an opportunity to generalize performance models or to tune and measure runtime performance at lower cost. We also prove there can exist an interplay by exhibiting a case where compile-time options significantly alter the performance distributions of a configurable system.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software product lines**; **Software performance**; • **Computing methodologies → Machine learning**.

1 INTRODUCTION

An ideal software system is expected to deliver the right functionality on time, using as few resources as possible, in every possible circumstances whatever the hardware, the operating system, the compiler, or the data fed as input. For fitting such a diversity of needs, it is increasingly common that software comes in many variants and is highly configurable through configuration options.

Numerous works have studied the effects of configuration options on performance. The outcome is a performance model that can be used to predict the performance of any configuration, to find an optimal configuration, or to identify, debug, and reason about influential options of a system [19, 23, 28, 29, 38–40, 44, 49, 53, 56, 63].

There are however numerous mechanisms to implement and deliver options: configuration files, command-line parameters, feature toggles or flags, plugins, *etc.* A useful distinction to make is between compile-time and run-time options. On the one hand, compile-time options can be used to build a custom system that can then be executed for a variety of usages. The widely used "./configure && make" is a prominent example for configuring a software project at compile-time. On the other hand, run-time options are used to parameterize the behavior of the system at load-time or during the execution. For instance, users can set some values to command-line arguments for choosing a specific algorithm or tuning the execution time based on the particularities of a given input to process. Both compile-time and run-time options can be configured to reach specific functional and performance goals.

Existing studies consider either compile-time or run-time options, but not both and the possible interplay between them. For instance, all run-time configurations are measured using a unique executable of the system, typically compiled with the default compile-time configuration (*i.e.,* using *./configure* without overriding compile-time options' values). Owing to the cost of measuring configurations, this is perfectly understandable but it is also a threat to validity. In particular, we can question the generality of these models if we change the compile-time options when building the software system: Do compile-time options change the performance distribution of run-time configurations? If yes, to what extent? Is the best run-time configuration always the best? Are the most influential run-time options always the same whatever the compile-time options used? Can we reuse a prediction model whatever the build has been? In short: **(RQ1) Do compile-time options change the performance distributions of configurable systems?**

In this paper we investigate the effect of compile-time options together with runtime options on the performance distributions of 4 software systems. For each of these systems, we measure a relevant performance metrics for a combination of nb_c compile time options and nb_r run-time options (yielding $2^{nb_c+nb_r}$ possible configurations) over a number of different inputs. We show that the compile-time options can alter the run-time performance distributions of software systems, and that it is worth tuning the compile-time options to improve their performances. We thus address a second research question: **(RQ2) How to tune software performances at the compile-time level?**

Our contributions are as follows :

- We construct a protocol and carry out an empirical study investigating the effects of compile-time options on run-time performances, for 4 software systems, various workloads, and non-functional properties;
- We provide a Docker image, a dataset of measurements, as well as analysis scripts[1];
- We exhibit a case (namely `nodeJS` and its operation rate) for which the compile-time options interact with the run-time options. We also present and implement a simple case of cross-layer tuning, providing a set of configuration options (run-time & compile-time) improving the operation rate of the default configuration of `nodeJS`.

Remainder. Section 2 discusses the motivation of this paper; Section 3 presents the experimental protocol; Section 4 evaluates and analyses the results w.r.t. this protocol; Section 5 discusses

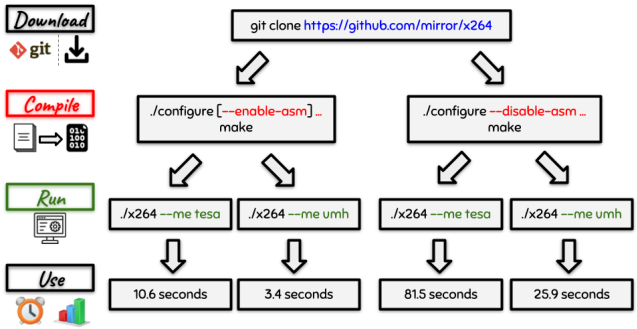
[1] Companion repository; Zenodo data: https://zenodo.org/record/4706963; Docker: https://hub.docker.com/r/anonymicse2021/splc21

the results, their limitations and future work; Section 6 describes research works related to the topic of this paper; Section 7 details the threats to validity; Section 8 concludes our paper.

2 BACKGROUND AND MOTIVATION

2.1 Compile-time and run-time options

Many software systems are highly-configurable both at compile time and at run time. For instance, the "./configure && make" is widely used to build a custom system *i.e.,* an executable binary. Considering the `x264` video encoder (see Figure 1), the compile-time option *–disable-asm* can be used to disable "*platform-specific assembly optimizations*". There are compile-time options to activate the needed functionality (*e.g., –disable-avs*), to fit requirements related to the hardware or operating system, *etc.* Some of these options have a suggested impact on performance. For the same subject system `x264`, it is also possible to use run-time options such as *me*, *mbtree* and *cabac* with possible effects on the size and encoding time. Run-time options are used to parameterize the behavior of the system at load time or during the execution.



This paper investigates how compile-time options can affect software performances and how compile-time options interact with run-time options.

Figure 1: Cross-layer variability of `x264`

Both compile-time and run-time options can be configured to reach specific functional and performance goals. Beyond `x264`, many projects propose these two kinds of options. A configuration option, being compile-time or run-time, can take different possible values. Boolean options have two possible values: "–activate" or "–deactivate" are typical examples of Boolean compile-time options. There are also numerical options with a range of integer or float values. Options with string values also exist and the set of possible values is usually predefined.

From a terminology point of view, we consider that a *compile-time (resp. run-time) configuration* is an assignment of values to compile-time options (resp. run-time options). It is possible to use "./configure" without setting explicit values. In this case, default values are assigned and form a default configuration. Similarly, when `x264` is called, default run-time options' values are internally used and constitute a default run-time configuration.

2.2 Performance prediction

Performance model. Given a software system with a set of run-time configurations, a performance model maps each run-time configuration to the performance of the system. A performance model has many applications and use-cases [4, 19, 22–24, 28, 29, 38–40, 44, 49, 53, 56, 60, 63, 65]: optimization (tuning) and finding of the best configuration, specialization of the configuration space, understanding and debugging of the effects of configuration options, verification (*e.g.,* non-regression testing), *etc.* The performance of a system, such as execution time, is usually measured using the same compiled executable (binary). **In our case, we consider that the way the software is compiled is subject to variability; there are two configuration spaces.** Formally, given a software system with a compile-time configuration space C and a run-time configuration space $\mathcal{R}$, a performance model is a black-box function $f : C \times \mathcal{R} \to \mathbb{R}$ that maps each run-time configuration $r \in \mathcal{R}$ to the performance of the system compiled with a compile-time configuration $c \in C$. The construction of a performance model consists in running the system in a fixed compile-time setting $c \in C$ on various configurations $r \in \mathcal{R}$, and record the resulting performance values $p = f(c, r)$.

Learning performance models. Measuring all configurations of a configurable system is the most obvious path to, for example, find a well-suited configuration. It is however too costly or infeasible in practice. 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). Research work thus follow a "sampling, measuring, learning" process [19, 22–24, 29, 38–40, 44, 49, 53, 60, 63, 65]. The training data for learning a performance model of a system compiled with a configuration $c \in C$ is then $D_c = \{(\mathcal{R}\#i, \mathcal{P}\#i) \mid i \in [\![1 : n]\!]\}$ where n is the number of measurements. Statistical learning techniques, such as linear regression [51], decision trees [48], or random forests [42], use this training set to build prediction models.

Generalization and transfer. Performance prediction models pursue the goal of generalizing beyond the training distribution. A first degree of generalization is that the prediction model is accurate for unmeasured and unobserved run-time configurations – it is the focus of most of previous works. However, it is not sufficient since the performance model may not generalize to the compile-time configuration space. Considering Figure 1, **one can question the generalization of a performance prediction model learned with a default compile-time configuration: will this model transfer well when the software is compiled differently?**

2.3 Research questions

The use of different compile-time configurations may change the raw and absolute performance values, but it can also change the overall distribution of configuration measurements. Given the vast variety of possible compile-time configurations that may be considered and actually used in practice, the generalization of run-time performance should be carefully studied. We aim to address two main research questions, each coming with their hypothesis.

(RQ1) Do compile-time options change the performance distributions of configurable systems? An hypothesis is that two performance models f_1 and f_2 over two compile-time configurations $c_1 \in C$ (resp. c_2) are somehow related and close. In its simplest form, there is a linear mapping: $f_1 = \beta \times f_2 + \alpha$. In this case,

System $\mathcal{CS}$	Commit	Compile-time #$\mathcal{C}$	Runtime #$\mathcal{R}$	Inputs #$\mathcal{I}$	# Measurements	Docker	Performances $\mathcal{P}$
nodeJS	78343bb	50	30	10	15000	Link	operation rate (ops)
poppler	42dde68	15	16	10	2400	Link	output size, time
x264	b86ae3c	50	201	8	80400	Link	output size, time, fps, kbs
xz	e7da44d	30	30	12	10800	Link	output size, time

Table 1: Table of considered configurable systems (see Algorithm 1 for the notations)

the performance of the whole run-time configurations increases or decreases; we aim to quantify this gain or lose. More complex mappings can exist since the underlying performance distributions differ. Such differences can impact the ranking of configurations and the statistical influence of options on performance. Owing to the cost of compiling and measuring configurations, we aim to characterize what configuration knowledge generalizes and whether the transfer of performance models is immediate or requires further investment.

A follow up research question is: **(RQ2) How to tune software performances at the compile-time level?** There are certainly compile-time options with negative or positive impacts on performance (*e.g.,* debugging options). Hence an idea is to tune the right compile-time options to eventually select optimal run-time configurations. An hypothesis is that compile-time options interact with run-time options, which can further challenges the tuning. Depending on the relationship between performance distributions (RQ1), the tuning strategy of compile-time options may differ.

3 EXPERIMENTAL PROTOCOL

To answer these research questions, we built the following experimental protocol. All the materials are freely available in the companion repository.

3.1 Selecting the subject systems

The objects of this experiment are a set of software systems that respect the following criteria: 1. The system must be open-source, so we can download the source code, compile and execute it; 2. The system must provide at least 5 compile- and run-time options; 3. Ideally, the software system should have been considered by research papers on software variability. The selected software systems must cover various application domains to make our conclusions generalizable. As a baseline for searching for software systems, we used: 1/ research papers on performance and/or variability; 2/ the website *openbenchmarking*[2] that conducts a large panel of benchmarks on open-source software systems; 3/ our own knowledge in popular open-source projects.

We selected 4 open-source software systems listed in Table 1. In addition, we have made sure that the software processes input data $\mathcal{I}$ (*e.g.,* a performance test suite), so we can experiment the options in different and realistic scenarios.

3.1.1 nodeJS. nodeJS[3] is a widely-used JavaScript execution environment (78k stars on Github) [9, 10, 21]. **Inputs $\mathcal{I}$.** We execute different JavaScript programs extracted from the nodeJS benchmark test suite. **Configurations $\mathcal{C}$ and $\mathcal{R}$.** We manually selected compile-time options (*e.g., v8-lite-mode* or *enable-lto*) that could change the way the executable behaves. We also selected run-time options that were supposed to impact the performances according to the documentation[4], like *jitless* or *node-memory-debug*. Additionally, we added experimental features (*e.g., experimental-wasm-modules* or *experimental-vm-modules*). **Performances $\mathcal{P}$.** We measured the rate of operations (measured in operations per seconds) performed by the selected test suite.

3.1.2 poppler. poppler[5] is a library for rendering PDF files [27, 34]. We focus on the poppler tool *pdfimages* that extracts images from PDF files[6]. **Inputs $\mathcal{I}$.** We tested *pdfimages* on a series of ten PDF files containing ten books on computer science. These books also have different: number of pages; illustration-text ratio; image resolutions; numbers of images; sizes (from 0.8 MB to 40.3 MB). **Configurations $\mathcal{C}$ and $\mathcal{R}$.** We selected the compile-time options that select the compression algorithm to be used at run time (*e.g., libjpeg* vs *openjpeg*): those different compression algorithms may be sensitive to the selected run-time options. The run-time options we selected are related to compression formats to maximize the potential variation impacts on performance. **Performances $\mathcal{P}$.** We systematically measured: the user time; the time needed to extract the pictures (with the tool *time*) in seconds; the size of the output pictures, in bytes.

3.1.3 x264. x264[7] (version 0.161.3048) is a video encoder that uses the H264 format [1, 35]. **Inputs $\mathcal{I}$.** As input data, we selected eight videos extracted from the Youtube UGC Dataset [68], having different categories (Animation video, Gaming video, *etc.*), different resolutions (*e.g.,* 360p, 420p) and different sizes. This dataset is intended for the study of the performances of compression software, which is adapted to our case. **Configurations $\mathcal{C}$ and $\mathcal{R}$.** For x264, we selected compile-time options related to performance (*e.g., enable-asm*) or to libraries related to hardware capacities (*e.g., disable-opencl*). Those compile-time options may interact with run-time options, linked to the different stages of video compression. **Performances $\mathcal{P}$.** We measured five different non-functional properties of x264: the time needed to encode the video (with *time*); the size (in bytes) of the encoded video; the *bitrate* (in bytes per second); the number of frames encoded per seconds.

3.1.4 xz. xz[8] (version 5.3.1alpha) is a data compression tool that uses the *xz* and *lzma* formats [6, 7, 37]. **Inputs $\mathcal{I}$.** As input data we use the Silesia corpus [12], that provides different types of file (*e.g.,*

[2] openbenchmarking.org
[3] https://nodejs.org/en/
[4] https://nodejs.org/api/cli.html
[5] https://poppler.freedesktop.org
[6] https://manpages.debian.org/testing/poppler-utils/pdfimages.1.en.html
[7] https://www.videolan.org/developers/x264.html
[8] https://tukaani.org/xz/

binary files, text files) with various sizes w.r.t. memory. **Configurations C and $\mathcal{R}$.** We manually selected compile-time options that can slow down the execution of the program (*e.g., disable-threads* or *enable-debug*), to state whether these options interact with run-time options. We selected specific run-time options related to the compression level, the format of the compression (*e.g., xz* or *lzma*), and the hardware capabilities (*e.g., -memory*=50 %). **Performances $\mathcal{P}$.** Like with `poppler`, we measured the time needed to compress the file (in seconds) and the size of the encoded files (in bytes).

3.2 Measuring performances

3.2.1 Protocol. For each of these systems we measured their performances by applying the protocol detailed in Algorithm 1.

Algorithm 1 - Measuring performances of the chosen systems

```
1:  Input S a configurable system
2:  Input C compile-time configurations
3:  Input R run-time configurations
4:  Input I system inputs
5:  // The inputs choices for each system are listed in Table 1
6:  Init P performance measurements of S
7:  Download the source code of S
8:  for each compile-time configuration c ∈ C do
9:      Compile source code of S with c arguments
10:     for each input i ∈ I do
11:         for each run-time configuration r ∈ R do
12:             Execute the compiled code with r on the input i
13:             Assign P[c, r, i] the performance of the execution
14:         end for
15:     end for
16: end for
17: Output P
```

Lines 1-4. First, we define the different inputs fed to the algorithm, the first one being the configurable system $\mathcal{S}$ we study. Then, we provide a set of compile-time configurations C, as well as a set of run-time configurations $\mathcal{R}$ related to the configurable system $\mathcal{S}$. Finally, we consider a set of input data $\mathcal{I}$, processed by the configurable system $\mathcal{S}$. **Lines 5-6.** Then, we initialize the matrix of performances P. **Line 7.** We download the source code of $\mathcal{S}$ (via the command line *'git clone'*), w.r.t. the link and the commits referenced in Table 1. We keep the same version of the system for all our experiments. If needed, we ran the scripts (*e.g., autogen.sh* for `xz`) generating the compilation files, thus enabling the manual configuration of the compilation. **Lines 8-16.** We apply the following process to all the compile-time configurations of C: based on a compile-time configuration c, we compile the software $\mathcal{S}$ (de-)activating the set of options of c. Then, we measured the performances of the compiled S when executing it on all inputs of $\mathcal{I}$ with the different run-time configurations of $\mathcal{R}$. **Line 17.** We store the results in the matrix of performances $\mathcal{P}$ (in CSV files). We then use these measurements to generate the results for answering the research questions (see Section 4).

3.2.2 Replication. To allow researchers to easily reproduce our experiments, we provide docker containers for each configurable system. The links are listed in Table 1 in the "Docker" column.

3.2.3 Hardware. To avoid introducing a bias in the experiment, we measure all performances sequentially on the same server (model Intel(R) Xeon(R) CPU D-1520 @ 2.20GHz, running Ubuntu 20.04 LTS). This server was dedicated to this task, so we can ensure there is no interaction with any other processes running at the same time.

3.3 Analyzing run-time performances

We split **RQ1. Do compile-time options change the performance distributions of configurable systems?** into two sub-questions.

RQ1.1. Do the run-time performances of configurable systems vary with compile-time options? A first goal of this paper is to determine whether the compile-time options affect the run-time performances of configurable systems. To do so, we compute and analyze the distribution of all the different compile-time configurations for each run-time execution of the software system (*i.e.,* given a input i and a run-time configuration r, the distribution of $\mathcal{P}[c, r, i]$ for all the compile-time configurations c of $\mathcal{S}$). All else being equal, if the compile-time options have no influence over the different executions of the system, these distributions should keep similar values. In other words, the bigger the variation of these distributions, the greater the effect of the compile-time options on run-time performances. To visualize these variations, we first display the boxplots of several run-time performances and few systems in Figure 2. Note that for `x264` (Figures 2a and 2c), only an excerpt of 30 configurations is depicted. We then comment the values of the InterQuartile Range (*i.e.,* IQR, the difference between the third and the first quartile of a distribution) for each couple of system $\mathcal{S}$ and performance $\mathcal{P}$. In order to state whether these variations are consistent across compile-time configurations, we then apply Wilcoxon signed-rank tests [52] (significance level of 0.05) to distributions of run-time performances, and report on the performance leading to significant differences. This test is suited to our case since our performance distributions are quantitative, paired, not normally distributed and have unequal variances.

RQ1.2. How many performance can we gain/lose when changing the default compile-time configuration? As an outcome of RQ1.1, we isolate a few couples (system, performance) for which the way we compile the system significantly changes its run-time performances. But how much performance can we expect to gain or lose when switching the default configuration (*i.e.,* the compilation processed without argument, with the simple command line *./configure*) to another fancy configuration? In other words, RQ1.2 states if it is worth changing the default compile-time configuration in terms of run-time performances. Moreover, RQ1.2 tries to estimate the benefit of manually tuning the compile-time options to increase software performances. To quantify this gain (or loss), we compute the ratios between the run-time performances of each compile-time configuration and the run-time performances of the default compile-time configuration. A ratio of 1 for a compile-time option suggests that the run-time performances of this compile-time option are always equals to the run-time performances of the default compile-time configuration. Intuitively, if the ratio is close to 1, the effect of compile-time options is not important. An average performance ratio of 2 corresponds to a compile-time option whose

run-time performances are in average twice the default compile-time option's performances. Section 4 details the average values and the standard deviations of these ratios for each input (row) and each couple of system and performance (column) kept in RQ1.1. We add the standard deviation of run-time performance distributions to estimate the overall variations of run-time performances due to the change of compile-time options.

To complete this analysis, and as an extreme case, we also computed the best ratio values in Section 4. By best ratio, we refer to the minimal ratio for the time (*e.g.,* reduction of encoding time for `x264` or the compression time for `xz`) and the maximal ratio for the operation rate (*i.e.,* increase of the number of operation executed per second for `nodeJS`) and the number of encoded fps (`x264`). As for Section 4, the best ratios are displayed for each input.

3.4 Studying the interplay of compile- and run-time options

RQ1 highlights few systems and performances for which we can increase the performances by tuning their compile-time options. Now, how to achieve this tuning process, and choose the right values to tune the performances of a software system is a problem to address. In short: **RQ2. How to tune software performances at the compile-time level?** Again, we split this question into two parts. **RQ2.1.** ***Do compile-time options interact with the run-time options?*** Before tuning the software, we have to deeply understand how the different levels (here the run-time level and the compile-time level) interact with each other. The protocol of RQ1 states whether the compile-time options change the performances, but the compilation could just change the scale of the distribution, *i.e.,* without really interacting with the run-time options. To discover such interactions, we compute the Spearman correlations [26] between the run-time performance distributions of software systems compiled with different configurations. The Spearman correlation allows us to measure if the way we compile the system change the rankings of the run-time performances. All else being equal, finding that two performance distributions, having the same run-time configurations but different compile-time configurations, are uncorrelated proves the existence of an interplay between the compile- and the run-time options. We depict a correlogram in Figure 3a. Each square$_{(i,j)}$ represents the Spearman correlation between the run-time performances of the compile-time configurations C#i and C#j. The color of this square respects the top-left scale: high positive correlations are red; low in white; negative in blue. Because we cannot describe each correlation individually, we added a table describing the distribution of the correlations (diagonal excluded). We apply the Evans rule [13] when interpreting these correlations. In absolute value, we refer to correlations by the following labels; very low: 0-0.19, low: 0.2-0.39, moderate: 0.4-0.59, strong: 0.6-0.79, very strong: 0.8-1.00.

To complete this analysis, we train a Random Forest Regressor [42] on our measurements so it predicts the operation rate of `nodeJS` for a given input $\mathcal{I}$. We fed this ensemble of trees with all the configuration options *i.e.,* all the compile-time options C and the run-time options $\mathcal{R}$ related to the performances $\mathcal{P}$ are used as predicting variables in this model. We then report the feature importances [8, 36, 43] for the different options (run-time or compile-time) in Figure 3b. Intuitively, a feature is important if shuffling its values increases the prediction error. Note that each Random Forest only predicts the performances $\mathcal{P}$ for a given input $\mathcal{I}$. The idea of this graph is to show the relative importances of the compile-time options, compared to the run-time options.

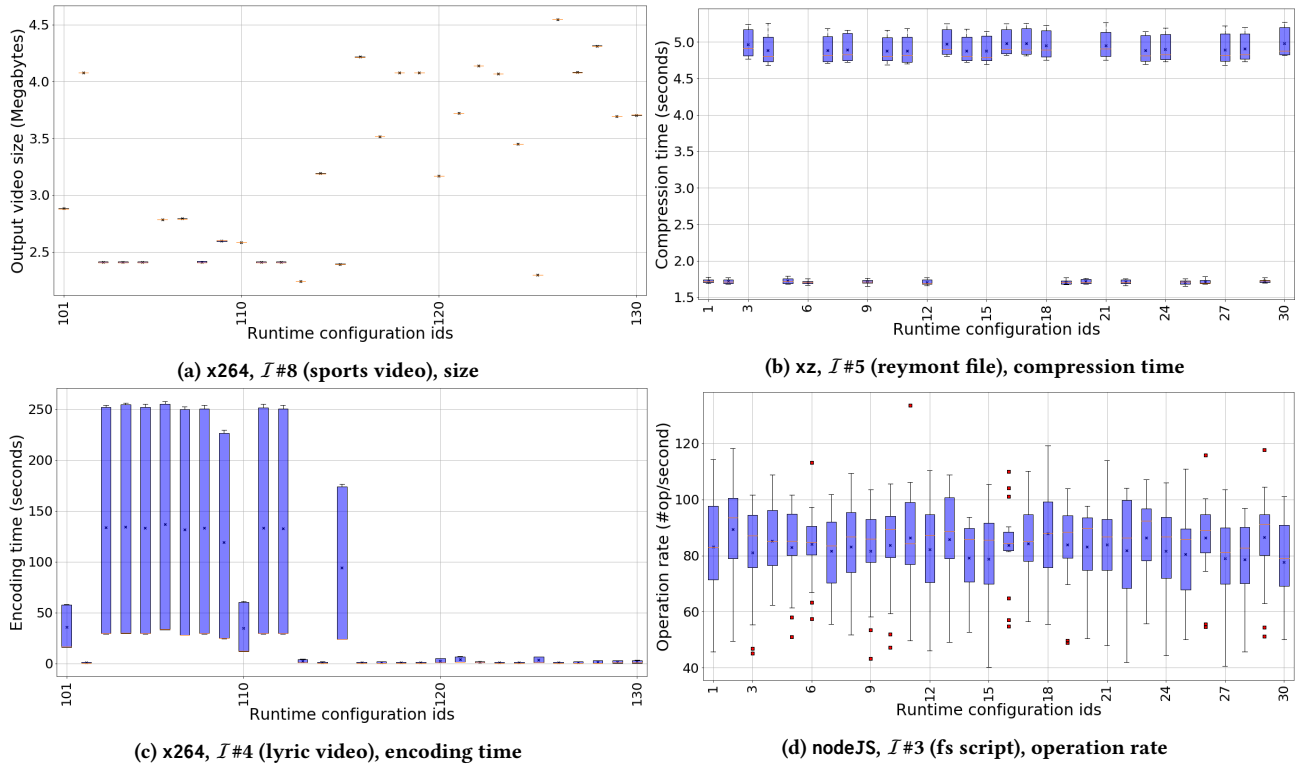
RQ2.2. ***How to use these interactions to find a set of good compile-time options and tune the configurable system?*** RQ2.1. exhibits interactions between the compile-time options and the run-time options. Now, the goal is to be able to use these interactions to find a good configuration in order to optimize the performances. As in RQ2.1., we used a Random Forest Regressor [42] to predict the operation rate of `nodeJS`. For this research question, we split our dataset of measurements in two parts, one training part and one test part. The goal is then to use the Random Forest Regressor to predict the performance of the configurations of the test set, and then keep the one leading to the best performances. In order to estimate how many measurements we need to predict a good configuration, we vary the training size (with 1 %, 5 % and 10 % of the data). We also compute the best configuration of our dataset, that would be predicted by an oracle. We then compare the obtained configuration with the default configuration of `nodeJS` (*i.e.,* the mostly used command-line, without argument, using a compiled version of `nodeJS` without argument). We plot the performance ratios between the predicted configuration and the default configuration of node for each input in Section 4. A performance ratio of 1.5 suggests that we found a configuration increasing the performance of default configuration by $1.5 - 1 = 50$ %.

4 EVALUATION

Let us answer ***RQ1.1. Do the run-time performances of configurable systems vary with compile-time options?*** by distinguishing performance properties.

First, the *size* is an extreme case of a stable performance that does not vary at all with the different compile-time options. As shown for the size of the encoded sports video in Figure 2a (boxplots), it stays the same for all compile-time configurations, leading to an average IQR of 2.3kB, negligible in comparison to the average size (3.02 MB). This conclusion applies for all the sizes we measured over the 4; the size of the compressed file for `xz` (*e.g.,* 2.6B of IQR for $\mathcal{I}$#8 having an average size of 2.85 MB) and the size of the image folder for `poppler` (*e.g.,* IQR = 16B, avg = 2.36 MB for $\mathcal{I}$#2). For the size of `x264`, 46 % of the Wilcoxon tests do not compute because the run-time distributions were equals (*i.e.,* same values for all sizes).

The variation of the *time* depends on the considered system. Overall, for the execution time of `poppler`, it is stable (*e.g.,* 29ms, for a execution of 2.6 s). For `xz`, and as depicted in Figure 2b, it seems to also depend on the run-time configurations. For instance, the distribution of the first run-time configuration (*i.e.,* $\mathcal{R}$#1) executed on $\mathcal{I}$#5 has an IQR of 40ms but this number increases to 0.37 s for the distribution of $\mathcal{R}$#9. For the encoding time of `x264` and the $\mathcal{I}$#4, we can draw the same conclusion ; suddenly, for a given run-time configuration (*e.g.,* from $\mathcal{R}$#102 to $\mathcal{R}$#103) the execution times increases not only in average (from 0.9 s to 133 s), but also in terms of variations w.r.t. the compile-time options (IQR from 1.0 s to 223 s). Since the number of frames for a given video is fixed, these conclusion are also valid for the number of encoded *fps* (`x264`).



(a) x264, $\mathcal{I}$#8 (sports video), size

(b) xz, $\mathcal{I}$#5 (reymont file), compression time

(c) x264, $\mathcal{I}$#4 (lyric video), encoding time

(d) nodeJS, $\mathcal{I}$#3 (fs script), operation rate

Figure 2: Boxplots of runtime performance distributions for different compile-time configurations. Each boxplot ($\mathcal{S}$, $\mathcal{I}$, $\mathcal{P}$) is related to a system $\mathcal{S}$, an input $\mathcal{I}$ and a performance $\mathcal{P}$.

For the *number of operations* executed per second (nodeJS), the IQR values of performance distribution are high : *e.g.*, for $\mathcal{I}$#3 as shown in Figure 2d in average 19 per second, for an average of 83 operations per second. However, unlike x264, these variations are quite stable across the different run-time configurations. A Wilcoxon test confirms a significant difference between run-time distributions of C#1 and C#11 ($p = 4.28 * 10^{-6}$) or C#4 and C#7 ($p = 1.24 * 10^{-5}$).

Key findings for RQ1.1. The sizes of software systems do not depend on the compile-time options for our 3 cases. However, other performance properties such as the time or the number of operations per second can vary with different compile-time options.

RQ1.2. How many performance can we gain/lose when changing the default compile-time configuration?

As a follow-up of RQ1.1., we computed the gain/lose ratios for the sizes of poppler, x264, xz. They are all around 1.00 in average, and less than 0.01 in terms of standard deviations, whatever the input is. The same applies with poppler or with xz and their execution times, as shown in Section 4. There are few variations, less than 3 % for the standard deviation of all inputs for poppler. For xz and time, we can observe the same trend. But we can observe an input sensitivity effect: for some inputs, like $\mathcal{I}$#2 or $\mathcal{I}$#11, the performances vary in comparison to the default one (stds at 0.48 and 0.23). Maybe the combination of an input and a compile-time option can alter the software performances.

Overall, there is room for improvement when changing the default compile-time options. For an example, with the operation rate of nodeJS, the average performance ratio is under 1 (*e.g.*, 0.86 for $\mathcal{I}$#3, 0.8 for $\mathcal{I}$#1 and $\mathcal{I}$#10). Compared to the default compile-time configuration of nodeJS, our choices of compilation options decrease the performances, by about 20 %. Besides, the standard variations are relatively high : we can expect the run-time performance ratios to vary from 41 % between different compile-time options for $\mathcal{I}$#5, or 11 % for $\mathcal{I}$#4. So it can be worse than losing only 20 % of operation per second. However, for $\mathcal{I}$#8, there exists a run-time configuration for which we can double (*i.e.*, multiply by 2.28) the number of operation per second) just by changing the compile-time options of nodeJS. We can draw the same conclusions for the execution time of x264: our compile-time configurations are not effective : it takes more than three times as long as the default configuration for all the inputs. But in this case, the best we can get is a decrease of 10 % of the execution time, which will not have a great impact on the overall performances. We can formulate an hypothesis with Figure 2c to explain these bad results: maybe few run-time configurations (*e.g.*, $\mathcal{R}$#103 to $\mathcal{R}$#109) take a lot of time to execute, thus increasing the overall average of performance ratios. Here, it would be an interaction between the run-time options and the compile-time options.

Key findings for RQ1.2. Depending on the performance we consider, it is worth to change the compile-time options or not. For nodeJS, it can increase the operation rate up to 128 % when

$\mathcal{S}$	nodeJS	poppler	x264		xz
$\mathcal{P}$	ops	time	fps	time	time
$\mathcal{I}$#1	0.8 ± 0.34	1.0 ± 0.02	0.59 ± 0.4	3.33 ± 2.4	1.01 ± 0.03
$\mathcal{I}$#2	0.79 ± 0.36	1.0 ± 0.01	0.59 ± 0.39	3.5 ± 2.53	1.16 ± 0.48
$\mathcal{I}$#3	0.86 ± 0.2	1.0 ± 0.01	0.59 ± 0.4	3.5 ± 2.57	1.11 ± 0.32
$\mathcal{I}$#4	1.01 ± 0.11	1.0 ± 0.01	0.6 ± 0.39	3.26 ± 2.37	1.01 ± 0.02
$\mathcal{I}$#5	0.73 ± 0.41	1.0 ± 0.01	0.59 ± 0.4	3.53 ± 2.62	1.02 ± 0.03
$\mathcal{I}$#6	1.05 ± 0.21	1.0 ± 0.02	0.6 ± 0.4	3.35 ± 2.49	1.01 ± 0.02
$\mathcal{I}$#7	0.98 ± 0.01	1.0 ± 0.07	0.58 ± 0.4	3.75 ± 2.8	1.01 ± 0.03
$\mathcal{I}$#8	0.84 ± 0.38	1.0 ± 0.01	0.59 ± 0.39	3.32 ± 2.37	1.01 ± 0.03
$\mathcal{I}$#9	1.01 ± 0.02	1.0 ± 0.02			1.01 ± 0.02
$\mathcal{I}$#10	0.8 ± 0.34	0.99 ± 0.03			1.04 ± 0.11
$\mathcal{I}$#11					1.08 ± 0.23
$\mathcal{I}$#12					1.02 ± 0.04

(a) Average ± standard deviation

$\mathcal{S}$	nodeJS	poppler	x264		xz
$\mathcal{P}$	ops	time	fps	time	time
$\mathcal{I}$#1	1.06	0.95	1.12	0.94	0.95
$\mathcal{I}$#2	1.08	0.98	1.14	0.93	0.98
$\mathcal{I}$#3	1.48	0.98	1.12	0.95	0.97
$\mathcal{I}$#4	1.68	0.97	1.27	0.83	0.96
$\mathcal{I}$#5	1.18	0.97	1.1	0.94	0.96
$\mathcal{I}$#6	2.3	0.95	1.68	0.51	0.97
$\mathcal{I}$#7	1.01	0.84	1.35	0.94	0.94
$\mathcal{I}$#8	2.28	0.97	1.12	0.93	0.97
$\mathcal{I}$#9	1.04	0.95			0.97
$\mathcal{I}$#10	1.09	0.92			0.97
$\mathcal{I}$#11					0.97
$\mathcal{I}$#12					0.91

(b) Best (min for time, max for ops & fps)

Table 2: Table of run-time performance ratios (compile-time option/default) per input. An average performance ratio of 1.4 suggests that the run-time performances of a compile-time option are in average 1.4 times greater than the run-time performance of the default compile-time configuration.

changing the default configuration. For x264, we can gain about 10 % of execution time with the tuning of compile-time options.

> **RQ1. Do compile-time options change the performance distributions of configurable systems?** Properties like size are extremely stable when changing the compile-time options. Performance models predicting the sizes can be generalized over different compile-time configurations. However, we found other performances, like the operation rate for nodeJS, or the execution time for x264, that are sensitive to compile-time options. It is worth to tune the compile-time options to optimize these performances.

RQ2.1. Do compile-time options interact with the run-time options?

For x264 and xz, there are few differences between the run-time distributions. As an illustration of this claim, for all the execution time distributions of x264, and all the input videos, the worst correlation is greater than 0.97 (>0.999 for x264 and encoded size, 0.55 for xz and time). This result proves that, if the compile-time options of these systems change the scale of the distribution, they do not change the rankings of run-time configurations (*i.e.,* they do not truly interact with the run-time options).

Then, we study the rankings of the run-time operation rate for nodeJS for different compile-time configurations, and details the Figure 3a. The first results are also positive. There is a large amount of compile-time configurations (top-left part of the correlogram) for which the run-time performances are moderately, strongly or even very strongly correlated. For instance, the compile-time option C#16 is very strongly (0.91) correlated with C#24 in terms of run-time performances. Similarly, compile-time options C#40 and C#23 are strongly correlated (0.73). There are less favorable cases when looking at the middle and right parts of the correlogram. For an example, C#27 and C#13 are uncorrelated (*i.e.,* a very low correlation of 0.01). Worse, switching from compile-time option C#8 to C#29 changes the rankings to such an extent that their run-time performances are negatively correlated (−0.35). In between, poppler's performance distributions are overall not sensitive to the change of compile-time options, except for the input $\mathcal{I}$#3 (for which the correlations can be negative).

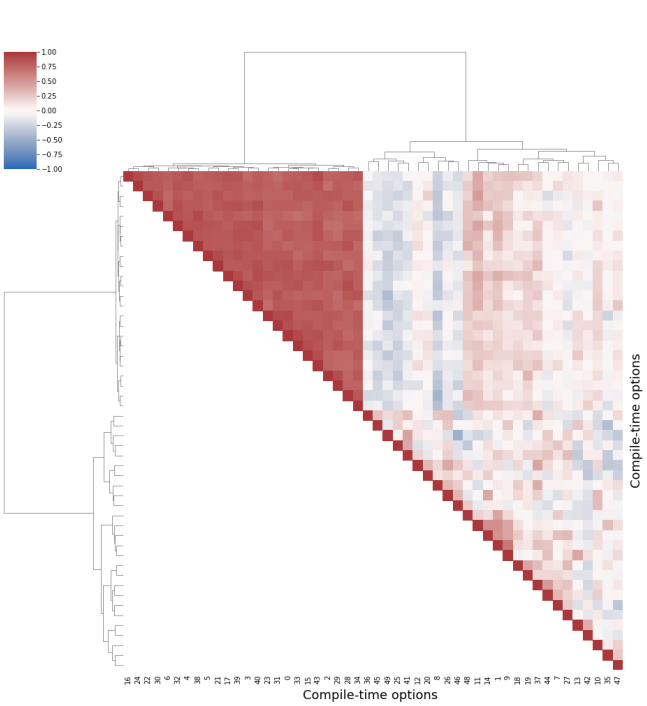
To complete this analysis, we discuss Figure 3b. The feature importances for predicting the operation rate of nodeJS for $\mathcal{I}$#3 are distributed among the different options, both the run-time and compile-time options. If it does not prove any interaction, it signifies that to efficiently predict the operation, the algorithm has to somehow combine the different levels of options. For the input $\mathcal{I}$#10, it is a bit different, since the only influential run-time option (*i.e.,* the one that has a great importance) is *jitless*. When looking at a decision tree (see additional results in the companion repository), the first split of the tree uses in fact this run-time option *jitless*, and then split the other branches with the compile-time options *–v8-lite-mode* and *–fully-static*.

Key findings for RQ2.1. xz, poppler and x264's performances rankings are not sensitive to the change of compile-time options. On the other hand, nodeJS's performances changes at run time with different the compile-options, *i.e.,* nodeJS run-time options interact with nodeJS's compile-time options.

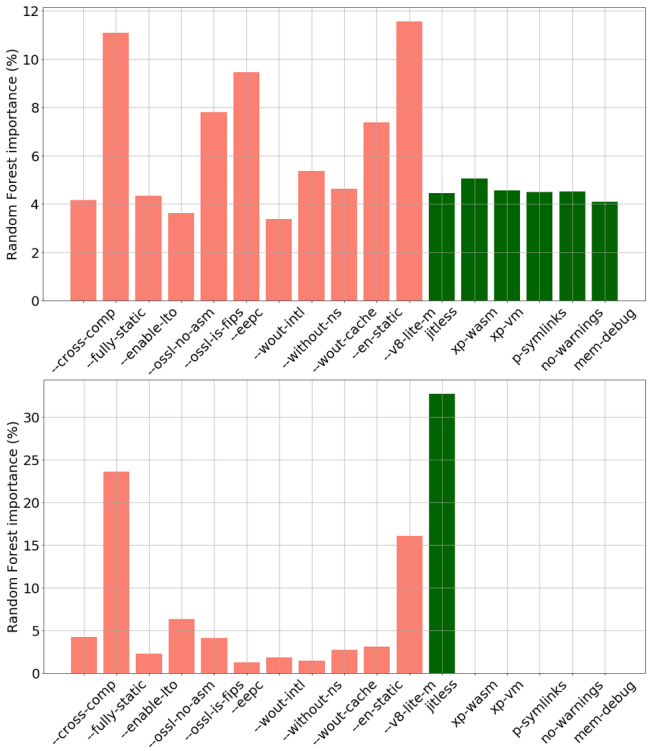
	Training Size			
Inputs	0.01	0.05	0.1	Oracle
$\mathcal{I}$#1	0.94	1.039	1.045	1.06
$\mathcal{I}$#2	1.051	1.086	1.085	1.099
$\mathcal{I}$#3	1.167	1.37	1.386	1.505
$\mathcal{I}$#4	1.123	1.226	1.232	1.251
$\mathcal{I}$#5	0.96	1.004	1.005	1.007
$\mathcal{I}$#6	1.005	1.069	1.09	1.104
$\mathcal{I}$#7	0.986	0.987	0.987	0.988
$\mathcal{I}$#8	1.035	1.047	1.05	1.054
$\mathcal{I}$#9	1.034	1.037	1.038	1.039
$\mathcal{I}$#10	1.003	1.021	1.021	1.044

Table 3: Performance ratios between the best predicted configuration and the default configuration for nodeJS and the operation rate, for different training sizes and inputs

RQ2.2. How to use these interactions to find a set of good compile-time options and tune the configurable system?



(a) Correlogram (Spearman) of the same run-time performance distributions for different compile-time configurations for $\mathcal{I}$#10



(b) Random forest importances (top $\mathcal{I}$#3, bottom $\mathcal{I}$#10) for predicting $\mathcal{P}$ - both compile- $\mathcal{C}$ (red) and run-time $\mathcal{R}$ (green) options matter

Figure 3: Illustration of the interplay between the runtime and the compile-time configurations ($\mathcal{S}$ = `nodeJS`, $\mathcal{P}$=operation rate)

Our results on `x264`, `xz`, and `poppler` show that their performance distributions are remarkably stable whatever their compile-time options. That is, interactions between the two kinds of options are easy to manage. This is a very good news for all approaches that try to build performance models using machine learning: if nb_c and nb_r are the number of boolean options present in $\mathcal{R}$ and C, it makes it possible to reduce the learning space to something proportional to $2^{nb_c} + 2^{nb_r}$ instead of $2^{nb_c+nb_r} = 2^{nb_c} \times 2^{nb_r}$. There are three practical opportunities (that apply to `x264`, `xz`, and `poppler`):

Reuse of configuration knowledge: transfer learning of prediction models boils down to apply a linear transformation among distributions. Users can also trust the documentation of run-time options, consistent whatever the compile-time configuration is.

Tuning at lower cost: finding the best compile-time configuration among all the possible ones allows one to immediately find the best configuration at run time. It is no longer necessary to measure many configurations at run time: the best one is transferred because of the linear relationship between compile-time configuration. Finding the best compile-time configuration is like solving a one-dimensional optimization problem: we simply compare the performances of a compilation operating on a fixed set of run-time configurations. Intuitively, it is enough to determine whether a compilation improves the performance of a limited set of run-time configurations. Theoretically, it is possible to compare the compilations' performance on a single run-time configuration. In practice, we expect to measure r' run-time configurations with $r' \ll 2^{nb_r}$

Measuring at lower cost: a common practice to measure run-time configurations is to use a default compile-time configuration. However, RQ1 results showed that it is possible to accelerate the execution time and thus drastically reduce the computational cost of measurements. That is, instead of using a default ./configure, we can use a compile-time configuration that is optimal w.r.t. cost. Then, owing to the results of RQ2.1, the measurements will transfer to any compile-time configuration and are representative of the run-time configuration space. The minimisation of the time is an example of a cost criteria; other properties such as memory or energy consumption can well be considered. It is even possible to use two compile-time configurations and executable binaries: (1) a first one to measure at lower cost and gather training samples; (2) a second one that is optimal for tuning a performance.

However, for `nodeJS`, it requires additional (as shown in Section 4). If we had access to an oracle, we could search for the best configuration of our dataset (in terms of performance), and replace the default configuration by this one. Depending on the input script, it will improve (or not) the performances. For instance, with the input $\mathcal{I}$#2, we can expect to gain about 10 % of performance, while for $\mathcal{I}$#9 and $\mathcal{I}$#10, it would be only 4 %. The worst case is without contest $\mathcal{I}$#7, for which we lose about 1 % of operation rate. But for inputs $\mathcal{I}$#3 and $\mathcal{I}$#4, it increases the performances by respectively 50 % and 25 %. We see these cases as proofs of concept; we can use the variability induced by the compile-time options to increase the overall performances of the default configuration. And if we do not have much data, it is possible to learn from it: with only 1 % of

the measurements, we can expect to gain 16 % of performance on $\mathcal{I}$#3. It steps up to 1.37 % for 5 % of the measurements used in the training. The same applies for the input $\mathcal{I}$#4 : 12 % of gain for 1 % of the measurements and 23 % for 5 %.

Key findings for RQ2.2. We can use the interactions between the compile-time and the run-time options to increase the default configuration's operation rate of `nodeJS` (up to 50 % for $\mathcal{I}$#3).

RQ2. How to tune software performances at the compile-time level? Two types of systems are emerging: if the run-time options of a software system are not sensitive to compile-time options (*e.g.,* for `x264`, `xz`, and most of the time `poppler`), there is an opportunity to tune "once and for all" the compilation layer for both improving the runtime performances and reducing the cost of measuring. However, for `nodeJS` and one specific input of `poppler`, we found interactions between the run-time and compile-time options, changing the rankings of their run-time performances distributions. We prove we can overcome this problem with a simple performance model using these interactions to outperform the default configuration of `nodeJS`.

5 DISCUSSION

Impacts for practitioners and researchers. For the benefit of software variability practitioners and researchers, we give an estimate of the potential impact of tuning software during compilation. We also provide hints to choose the right values of options before compiling software systems (see RQ2.2). This may be of particular interest to developers responsible for compiling software into packages (*i.e., apt*, *dnf*, *etc.*). For engineers who build performance models or test the performance of software systems, we show there are opportunities to decrease the underlying cost of tuning or measuring runtime configurations. We also warn that performance models may not generalize, depending on the software and the performance studied (as shown in our study). At this step of the research, it is hard to anticipate such situations. However we recommend that practitioners verify the sensitivity of performance models w.r.t. compile-time options. Our results are also good news for researchers who build performance models using machine learning. Many works have experiments with `x264` [3, 19, 53] and we show that for this system the performance is remarkably stable. `xz` considered in [37] also enters in this category. That is, there is no threat to validity w.r.t. compile-time options. To the best of our knowledge, other systems (`nodeJS` and `poppler`) have not been considered in the literature of configurable systems [44]. Hence we warn researchers there can be cases for which this threat applies.

Understanding the interplay. Our results suggest that compile-time options affect specific non-functional properties of software systems. The cause of this interplay between compile-time and run-time options is unclear and remains shallow for the authors of this paper. The results could be related to the system domain, or the way it processes input data; trying to characterize the software systems sensitive to compile-time options (*i.e.,* without measuring their performances) is challenging, but worth looking at. We are looking forward discussing with developers to know more about why it appears in these cases, and not for the other software systems (left as future work).

Other variability factors. Compile-time options are a possible layer that can affect performance, but not the only one. Could we in the same way prove the existence of the effect of the operating system on software performances? On the hardware level? This paper is also a way for us to alert researchers to other factors of variability beyond software, which may interact with their performance. We encourage researchers to highlight these factors in their work. Similarly, the 4 software systems are evolving, with constantly new commits and features. Then, a question arises naturally: will this interplay evolve with the software? And if it changes with time and versions, how to automate the update of our understanding of these interactions? This is another challenging layer and direction to explore. In our study design we consider compile-time options and not the compiler flags. Though there is an overlap, there are many compile-time options specific to a domain and system. As future work, we plan to investigate how compiler flags (*e.g.,* −02 and −03 for *gcc*) relate to run-time configurations. More generally, the variability of interpreters and virtual machines [25, 32, 55] can be considered as yet another variability layer [33] on which we encourage researchers to perform experiments.

6 RELATED WORK

Machine learning and configurable systems. Machine learning techniques have been widely considered in the literature to learn software configuration spaces and non-functional properties of software product lines [15, 22, 23, 30, 38, 39, 41, 44, 47, 66, 67]. Several works have proposed to predict performance of configurations, with several use-cases in mind for developers and users of configurable systems: the maintenance and interpretability of configuration spaces [54], the selection of an optimal configuration [15, 39, 41], the automated specialization of configurable systems [59], *etc.* Studies usually support learning models restrictive to specific static settings (*e.g.,* inputs, hardware, and version) such that a new prediction model may have to be learned from scratch or adapted once the environment change. The studies of Valov *et al.* [64, 66] suggest that changing the hardware has reasonable impacts since linear functions are highly accurate when reusing prediction models. Netflix conducts a large-scale study for comparing the compression performance of `x264`, *x265*, and *libvpx* over different inputs [1]. However, only two run-time configurations were considered on a fixed compile-time configuration. Alterations in software version [16] and changes in operating system [18] have both shown to cause variation in the results of a neuroimaging study. Jamshidi *et al.* [22] conducted an empirical study on four configurable systems (including `x264`), varying software configurations and environmental conditions, such as hardware, input, and software versions. Pereira*et al.* [2] and Lesoil *et al.* [33] empirically report that inputs can change the performance distributions of a configurable system (`x264`). In our study, we purposely fix the hardware and the version in order to isolate the effects of compile-time options on run time. To the best of our knowledge, our large-scale study is the first to systematically investigate the effects of compile-time options on performance of run-time configurations. The use of transfer learning techniques [23, 30, 38, 66] can be envisioned to adapt prediction models w.r.t. compile-time options. A key contribution of our study is to show that compile-time options can

change the rankings of run-time options, thus preventing the reuse of a model predicting the best run-time configuration.

Input sensitivity. There are works addressing the performance analysis of software systems [11, 14, 17, 31, 46, 56] depending on different inputs (also called workloads). In our study, we also consider different inputs when measuring performances. In contrast to our work, existing studies either consider a limited set of compile-time and run-time configurations (*e.g.,* only default configurations). It is also a threat to validity since compile-time options may change the performance distribution. In response, we perform an in-depth, controlled study of different systems to make it vary in the large, both in terms of compile-time and run-time configurations as well as inputs. In our study, we show a huge effect of inputs in the interplay of compile- and run-time options; for few inputs, the interactions of the compilation and the execution layers will not change anything, while for others, it would be a disaster to ignore them.

Compiler optimizations. The problem of choosing which optimizations of a compiler to apply has a long tradition [50]. Since the mid-1990s, machine-learning-based or evolutionary approaches have been investigated to explore the configuration space of compiler [5, 20, 45, 57, 58, 61, 62]. Such works usually consider a limited set of run-time configurations in favor of a broad consideration of inputs (programs). The goal is to understand the cost-effectiveness of compiler optimizations or to find tuning techniques for specific inputs and architectures. In contrast, our goal is to understand the interplay between compile-time options and run-time options, with possible consequences on the generalization of the configuration knowledge. As discussed in Section 5, compiler flags are worth considering in future work in addition to compile-time options.

7 THREATS

Construct validity. While constructing the experimental protocol and measuring the performances of software systems, we only kept a subset of all their compile-time and run-time options. The study we conducted focuses on performance measurements. The risk was to handle options that have no impact on performance, letting the results irrelevant. So we drove our selection on options which documentation gives indications about potential impacts on performance. The relevance of the input data provided to software systems during the experiment is crucial. To mitigate this threat we rely on: performance tests (and the input data they use) developed and used by `nodeJS`; widely-used input data sets (`xz` and `x264`); a large and heterogeneous data set of PDF files (`poppler`).

Internal Validity. Measuring non-functional properties is a complex process. During this process, the dependencies of the operating system can interact with the software system. For instance, the version of *gcc* could alter the way the source code is compiled, and change our conclusion. To mitigate this threat, we provided one docker container and fixed the configuration of the operating system for each subject system. However, and due to the measurement cost, we did not repeat the measurements several times. To gather measurements, we use the same dedicated server for the different subject systems. Thus, we can guarantee it was the only process running. The performances of the software systems can depend on the hardware they are executed on. To mitigate this threat we use the same hardware and provided its specifications for comparison during replications. Another threat to validity is related to the performances measured per second (*e.g.,* the number of fps for `x264`). For fast run-time executions, tiny variations of the time can induce high variations of the ratio over time. To alleviate this threat, we make sure the average execution time stays always greater than one second for all the input. To learn a performance model and predict which configuration was optimal, we used a machine learning algorithm in RQ2.2, namely Random Forest. These algorithms can produce unstable results from one run to the next, which could be a problem for the results related to this research question. In order to mitigate this threat, we have kept the average value over 10 throws.

8 CONCLUSION

Is there an interplay between compile-time and run-time options when it comes to performance? Our empirical study over 4 configurable software showed that two types of systems exist.

In the most favorable case, compile-time options have a linear effect on the run-time configuration space. We have observed this phenomenon for two systems and several non-functional properties (*e.g.,* execution time). There are then opportunities: the configuration knowledge generalizes no matter how the system is compiled; the performance can be further tuned through the optimisation of compile-time options and without thinking about the run-time layer; the selection of a custom compile-time configuration can reduce the cost of measuring run-time configurations. We have shown we can improve the run-time performance of these two systems, at compile-time and at lower cost.

However, our study also showed that there is a subject system for which there are interactions between run-time and compile-time options. This challenging case changes the rankings of run-time performances configurations and the performance distributions. We have shown we can overcome this problem with a simple performance model using these interactions to outperform the default compile-time configuration. The fourth subject of our study is in-between: the compile-time layer strongly interacts with run-time options only when processing a specific input. For the 9 other inputs of our experiment, we can take advantage of the linear interplay. Hence it is possible but rare that there is an interplay between compile-time options, run-time options, and inputs fed to a system.

Our work calls to further investigate how variability layers interact. We encourage researchers to replicate our study for different subject systems and application domains.

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

REFERENCES

[1] Jan De Cock, Aditya Mavlankar, Anush Moorthy, and Anne Aaron. 2016. A Large-Scale Comparison of x264, x265, and libvpx – a Sneak Peek. netflix-study-link.

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

[3] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2019. *Learning Software Configuration Spaces: A Systematic Literature Review (submitted).* Research Report. Univ Rennes, Inria, CNRS, IRISA. https://doi.org/10.1145/nnnnnnn.nnnnnnn

[4] Benoit Amand, Maxime Cordy, Patrick Heymans, Mathieu Acher, Paul Temple, and Jean-Marc Jézéquel. 2019. Towards Learning-Aided Configuration in 3D Printing: Feasibility Study and Application to Defect Prediction. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems* (Leuven, Belgium) *(VAMOS '19).* Association for Computing Machinery, New York, NY, USA, Article 7, 9 pages. https://doi.org/10.1145/3302333.3302338

[5] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sept. 2018), 42 pages. https://doi.org/10.1145/3197978

[6] Dominic Berz, Marco Engstler, Moritz Heindl, and Florian Waibel. 2015. Comparison of lossless data compression methods. *TECHNICAL REPORTS IN COMPUTING SCIENCE* 1, 1 (2015), 1–13.

[7] Born de Oliveira, Augusto. 2015. Measuring and Predicting Computer Software Performance: Tools and Approaches. http://hdl.handle.net/10012/9259

[8] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[9] Steven Bucaille. 2020. Gadolinium: Monitoring Non-Functional Properties of REST APIs.

[10] Agustina Buccella, Matias Pol'la, Esteban Ruiz de Galarreta, and Alejandra Cechich. 2018. Combining Automatic Variability Analysis Tools: An SPL Approach for Building a Framework for Composition. In *Computational Science and Its Applications – ICCSA 2018*, Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Elena Stankova, Carmelo M. Torre, Ana Maria A.C. Rocha, David Taniar, Bernady O. Apduhan, Eufemia Tarantino, and Yeonseung Ryu (Eds.). Springer International Publishing, Cham, 435–451.

[11] Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. 2014. Estimating the Empirical Cost Function of Routines with Dynamic Workloads. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. ACM, New York, NY, USA, Article 230, 10 pages. https://doi.org/10.1145/2581122.2544143

[12] Sebastian Deorowicz. 2014. Silesia compression corpus.

[13] James D Evans. 1996. *Straightforward statistics for the behavioral sciences*. Thomson Brooks/Cole Publishing Co, Book News, Inc. Portland, Or.

[14] Hany FathyAtlam, Gamal Attiya, and Nawal El-Fishawy. 2013. Comparative Study on CBIR based on Color Feature. *International Journal of Computer Applications* 78, 16 (Sept. 2013), 9–15. https://doi.org/10.5120/13605-1387

[15] Wei Fu and Tim Menzies. 2017. Easy over Hard: A Case Study on Deep Learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/3106237.3106256

[16] Tristan Glatard, Lindsay B Lewis, Rafael Ferreira da Silva, Reza Adalat, Natacha Beck, Claude Lepage, Pierre Rioux, Marc-Etienne Rousseau, Tarek Sherif, Ewa Deelman, et al. 2015. Reproducibility of neuroimaging analyses across operating systems. *Frontiers in neuroinformatics* 9 (2015), 12.

[17] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. 2007. Measuring Empirical Computational Complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 395–404. https://doi.org/10.1145/1287624.1287681

[18] Ed HBM Gronenschild, Petra Habets, Heidi IL Jacobs, Ron Mengelers, Nico Rozendaal, Jim Van Os, and Machteld Marcelis. 2012. The effects of FreeSurfer version, workstation type, and Macintosh operating system version on anatomical volume and cortical thickness measurements. *PloS one* 7, 6 (2012), e38234.

[19] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Silicon Valley, CA, USA, 301–311. https://doi.org/10.1109/ASE.2013.6693089

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

[21] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. 2017. Software performance self-adaptation through efficient model predictive control. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana-Champaign IL USA, 485–496. https://doi.org/10.1109/ASE.2017.8115660

[22] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 497–508.

[23] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to Sample: Exploiting Similarities across Environments to Learn Performance Models for Configurable Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3236024.3236074

[24] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. 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)* (Buenos Aires). IEEE Computer Society, Los Alamitos, CA, 31–41. https://doi.org/10.1109/SEAMS.2017.11

[25] Sanath Jayasena, Milinda Fernando, Tharindu Rusira, Chalitha Perera, and Chamara Philips. 2015. Auto-tuning the java virtual machine. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, Hyderabad,India, 1261–1270.

[26] Maurice George Kendall. 1948. *Rank correlation methods*. Griffin, Michigan.

[27] Mark J. Kilgard. 2020. Anecdotal Survey of Variations in Path Stroking among Real-world Implementations. arXiv:2007.12254 [cs.GR]

[28] Alexander Knüppel, Thomas Thüm, Carsten Immanuel Pardylla, and Ina Schaefer. 2019. Understanding Parameters of Deductive Verification: An Empirical Investigation of KeY. In *Software Engineering and Software Management, SE/SWM 2019, Stuttgart, Germany, February 18-22, 2019 (LNI, Vol. P-292)*, Steffen Becker, Ivan Bogicevic, Georg Herzwurm, and Stefan Wagner (Eds.). GI, Stuttgart, Germany, 165–166. https://doi.org/10.18420/se2019-51

[29] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in Modeling Performance of Highly Configurable Software Systems. *Software & Systems Modeling* 18, 3 (01 Jun 2019), 2265–2283. https://doi.org/10.1007/s10270-018-0662-9

[30] Rahul Krishna, Vivek Nair, Pooyan Jamshidi, and Tim Menzies. 2019. Whence to Learn? Transferring Knowledge in Configurable Systems using BEETLE. *CoRR* abs/1911.01817 (2019), 1–16. arXiv:1911.01817 http://arxiv.org/abs/1911.01817

[31] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.* 16, 3, Article 15 (April 2016), 23 pages. https://doi.org/10.1145/2885497

[32] Philipp Lengauer and Hanspeter Mössenböck. 2014. The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering* (Dublin, Ireland) *(ICPE '14)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/2568088.2568091

[33] Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. Deep Software Variability: Towards Handling Cross-Layer Configuration. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems* (Krems, Austria) *(VaMoS'21)*. Association for Computing Machinery, New York, NY, USA, Article 10, 8 pages. https://doi.org/10.1145/3442391.3442402

[34] D. Maiorca and B. Biggio. 2019. Digital Investigation of PDF Files: Unveiling Traces of Embedded Malware. *IEEE Security Privacy* 17, 1 (2019), 63–71. https://doi.org/10.1109/MSEC.2018.2875879

[35] A. Maxiaguine, Yanhong Liu, S. Chakraborty, and Wei Tsang Ooi. 2004. Identifying "representative" workloads in designing MpSoC platforms for media processing. In *2nd Workshop onEmbedded Systems for Real-Time Multimedia, 2004. ESTImedia 2004.* IEEE, Stockholm, Sweden, 41–46. https://ieeexplore.ieee.org/document/1359702

[36] Christoph Molnar. 2020. *Interpretable Machine Learning*. Lulu.com, Munich.

[37] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2020. Identifying Software Performance Changes Across Variants and Versions. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Melbourne, Australia, 611–622.

[38] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. 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*. ACM, Paderborn, Germany, 257–267. https://doi.org/10.1145/3106237.3106238

[39] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel. 2020. Finding Faster Configurations Using FLASH. *IEEE Transactions on Software Engineering* 46, 7 (2020), 794–811. https://doi.org/10.1109/TSE.2018.2870895

[40] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 61–71. https://doi.org/10.1145/3106237.3106273

[41] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 61–71. https://doi.org/10.1145/3106237.3106273

[42] Thais Mayumi Oshiro, Pedro Santoro Perez, and Jose Augusto Baranauskas. 2012. How Many Trees in a Random Forest?. In *Machine Learning and Data Mining in Pattern Recognition*, Petra Perner (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–168.

[43] Terence Parr, Kerem Turgutlu, Christopher Csiszar, and Jeremy Howard. 2018. Beware Default Random Forest Importances. last access: july 2019.

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

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

[46] Suporn Pongnumkul, Chaiyaphum Siripanpornchana, and Suttipong Thajchayapong. 2017. Performance Analysis of Private Blockchain Platforms in Varying Workloads. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, Vancouver, Canada, 1–7. https://doi.org/10.1109/icccn.2017.8038517

[47] Clément Quinton, Michael Vierhauser, Rick Rabiser, Luciano Baresi, Paul Grünbacher, and Christian Schuhmayer. 2020. Evolution in dynamic software product lines. *Journal of Software: Evolution and Process* 33 (2020), e2293.

[48] S Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics* 21, 3 (1991), 660–674.

[49] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Press, Lincoln, Nebraska, 342–352. https://doi.org/10.1109/ASE.2015.45

[50] Paul B. Schneck. 1973. A Survey of Compiler Optimization Techniques. In *Proceedings of the ACM Annual Conference* (Atlanta, Georgia, USA) *(ACM '73)*. Association for Computing Machinery, New York, NY, USA, 106–113. https://doi.org/10.1145/800192.805690

[51] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 329. John Wiley & Sons, North America.

[52] Sidney Siegel. 1956. Nonparametric statistics for the behavioral sciences. *The Journal of Nervous and Mental Disease* 125 (1956), 497.

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

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

[55] Jeremy Singer, Gavin Brown, Ian Watson, and John Cavazos. 2007. Intelligent Selection of Application-Specific Garbage Collectors. In *Proceedings of the 6th International Symposium on Memory Management* (Montreal, Quebec, Canada) *(ISMM '07)*. Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/1296907.1296920

[56] Urjoshi Sinha, Mikaela Cashman, and Myra B. Cohen. 2020. Using a Genetic Algorithm to Optimize Configurations in a Data-Driven Application. In *Search-Based Software Engineering - 12th International Symposium, SSBSE 2020, Bari, Italy, October 7-8, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12420)*, Aldeida Aleti and Annibale Panichella (Eds.). Springer, Bari, Italy, 137–152. https://doi.org/10.1007/978-3-030-59762-7_10

[57] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta optimization: Improving compiler heuristics with machine learning. *ACM sigplan notices* 38, 5 (2003), 77–90.

[58] Suprapto and Retantyo Wardoyo. 2013. Algorithms of the Combination of Compiler Optimization Options for Automatic Performance Tuning. In *Information and Communication Technology*, Khabib Mustofa, Erich J. Neuhold, A. Min Tjoa, Edgar Weippl, and Ilsun You (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 91–100.

[59] Paul Temple, Mathieu Acher, Jean-Marc Jezequel, and Olivier Barais. 2017. Learning Contextual-Variability Models. *IEEE Software* 34, 6 (Nov. 2017), 64–70. https://doi.org/10.1109/ms.2017.4121211

[60] Paul Temple, José A. Galindo, Mathieu Acher, and Jean-Marc Jézéquel. 2016. Using Machine Learning to Infer Constraints for Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference* (Beijing, China) *(SPLC '16)*. Association for Computing Machinery, New York, NY, USA, 209–218. https://doi.org/10.1145/2934466.2934472

[61] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. 2009. A Scalable Auto-Tuning Framework for Compiler Optimization. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '09)*. IEEE Computer Society, USA, 1–12. https://doi.org/10.1109/IPDPS.2009.5161054

[62] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. 2003. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, San Francisco California USA, 204–215. https://doi.org/10.1109/CGO.2003.1191546

[63] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. 2015. Empirical Comparison of Regression Methods for Variability-Aware Performance Prediction. In *Proceedings of the 19th International Conference on Software Product Line* (Nashville, Tennessee) *(SPLC '15)*. Association for Computing Machinery, New York, NY, USA, 186–190. https://doi.org/10.1145/2791060.2791069

[64] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. 2020. Transferring Pareto Frontiers across Heterogeneous Hardware Environments. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/3358960.3379127

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

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

[67] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. arXiv:2101.05362 [cs.SE]

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