Deep software variability for resilient performance models of configurable systems

Luc Lesoil

L'UNIVERSITÉ DE RENNES

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Luc LESOIL

Deep Software Variability for Resilient Performance Models of Configurable Systems

Thèse présentée et soutenue à Rennes, le 17 avril 2023
Unité de recherche : Équipe DiverSE, IRISA
Thèse N° :

Rapporteurs avant soutenance :

Lidia FUENTES	Full Professor, Universidad de Málaga, ITIS Software, Málaga, SPAIN
Rick RABISER	Full Professor, Johannes Kepler Universität, Linz, AUSTRIA

Composition du Jury :

Présidente :	Elisa FROMONT	Professeure des Universités, UR1, IUF, Rennes, FRANCE
Examinateurs :	Maxime CORDY	Research Scientist, University of Luxembourg, LUXEMBOURG
	Lidia FUENTES	Full Professor, Universidad de Málaga, ITIS Software, Málaga, SPAIN
	Pooyan JAMSHIDI	Assistant Professor, University of South Carolina, UNITED STATES
	Rick RABISER	Full Professor, Johannes Kepler Universität, Linz, AUSTRIA
Dir. de thèse :	Mathieu ACHER	Professeur des Universités, INSA Rennes, IUF, Rennes, FRANCE
	Arnaud BLOUIN	Maître de Conférences (HDR), INSA Rennes, Rennes, FRANCE
Co-dir. de thèse :	Jean-Marc JÉZÉQUEL	Professeur des Universités, Université de Rennes 1, Rennes, FRANCE

Remerciements

À François Bodin, le premier à m'avoir ouvert les portes de la recherche. À Laurent et Olivier, qui ont grandement contribué au décollage d'Eole Eyes.

À ceux qui ne figurent pas dans la suite de ces remerciements... Désolé!

À tous les services de l'IRISA et de l'Université de Rennes 1 qui permettent le bon déroulement de nos activités de recherche. À la patience infinie de Sophie Maupile qui a sauvé tant de procédures administratives sur le point de dérailler.

À l'ensemble des membres de l'équipe: pour nos discussions, pendant et —souvent jusqu'à longtemps— après les pauses; pour l'ouverture d'esprit qui caractérise nos échanges; pour cette densité incroyable de professionnels passionnés par leur métier et prêts à le partager. DiverSE est un environnement exceptionnel pour mener sa propre recherche et donner vie à ses idées.

À cette folle liberté dont jouissent ceux se risquant sur la tortueuse voie du doctorat.

À mes voisins de bureau: à Aaron, à l'apaisante assurance du stoïcisme et à la beauté de Lisp; à Quentin, à l'amour du débat et à l'efficacité de Rust. À vos explications qui ont comblé nombre de mes lacunes sur l'informatique, et à vos disputes qui me manquent déjà!

Aux chercheurs avec lesquels j'aurais aimé collaborer plus longtemps, en particulier à Clément Quinton de Spirals, et à Paul, arrivé en poste pendant le chassé-croisé hivernal.

To the RESIST associate team and especially to Arnaud and Helge from Simula, for their kind advices, their support when preparing our submission and for sharing their human vision of research.

À mes co-auteurs qui m'ont accompagné ou emporté avec eux pour défendre des idées qui nous tenaient à cœur. À Hugo et à nos discussions qui, tout bien réfléchi, auraient sans doute dû s'accompagner de paracétamol. À Aaron et à cet ouvrage scientifique, rédigé reclus de l'humanité pour plusieurs mois, que j'attends et ne manquerai pas de lire. À Xhevahire, à la vision à la fois lucide et espiègle que tu portes sur le monde, et à la singularité des directions de recherche que tu choisis d'emprunter.

À la conversation-dont-on-ne-peut-prononcer-le-nom peuplée d'émotifs anonymes. Au million de blagues/références de Gauthier et Gwendal qui m'ont survolé et auxquelles j'ai quand même ri. À June et Paul, mes aînés dans la grande famille de la recherche, qui ont toujours suivi les progrès de ma thèse d'assez près. Ils ne s'en rendent pas encore compte, mais vos futurs doctorants ont beaucoup de chance!

À mes superviseurs, sans lesquels ce document n'existerait pas. Jean-Marc, merci de m'avoir prodigué tous ces conseils empreints de bon sens et de pragmatisme tout au long de ces trois années. J'espère ne pas (trop) te vieillir en te disant être honoré d'avoir travaillé avec un monument de la recherche tel que toi. Arnaud, merci d'avoir mis ton humour décapant et la finesse de ta plume au service de la brillance de cette thèse. Avant de croiser ta route, je m'estimais organisé; depuis, plus tant que ça. Mathieu, merci pour tout. J'ai appris ce qu'était un très bon

chercheur en te regardant faire. Je ne sais pas dans quelle source tu puises toute cette énergie, mais pour le bien de l'ingénierie logicielle, je souhaite qu'elle soit intarissable. Merci de m'avoir embarqué dans cette thèse et d'avoir partagé tes idées avec moi, j'espère que l'on aura l'occasion de retravailler ensemble.

A mes parents et ma frangine, que j'aime.

A l'équipe des bfis et celle du mercredi soir, à nos fugaces moments de libre créativité qui rendent cette existence supportable.

Résumé en français

Contexte

La configuration est le ciment du monde numérique. Malléable initialement, elle se façonne et se tord au gré des envies de l'utilisateur, devenant l'unique support de ses choix au sein de l'environnement virtuel qu'il se construit ; souvent à l'origine de défaillances, comme en témoignent récemment OVH et Facebook, la configuration assure ou non la solidité des lignes de produits logiciels, colmatant ou exposant leurs failles de sécurité ; enfin, une fois choisie, elle fige les préférences de l'utilisateur et virtualise fidèlement chacune de ses caractéristiques, garantissant l'intégrité de son identité numérique.

De manière plus terre à terre, une configuration permet de modifier le comportement d'un logiciel en fonction des besoins de son utilisateur. En pratique, ce dernier dispose d'un jeu d'options, auxquelles il peut attribuer des valeurs. Pour reprendre un exemple d'actualité : est-ce que la visioconférence doit être enregistrée ? (oui ou non) Quelle taille d'icône pour afficher ses participants ? (petite, moyenne ou grande) Combien de participants au maximum ? (12, 15, 23, *etc.*) L'ensemble de ces choix forme une configuration. Une fois fournie au logiciel, celui-ci adapte son exécution conformément aux valeurs d'options qui compose cette configuration. Le logiciel est alors qualifié de configurable.

Qu'ils soient basiques ou complexes, tous les logiciels modernes sont plus ou moins configurables. Même un programme aussi simple que *hello world*, dont l'unique but est de saluer celui qui l'exécute, devrait en pratique proposer une option pour s'adapter à la langue de ses utilisateurs, et donc être configurable. À l'extrême inverse, Linux et ses 20 000 options de configuration mettent à l'épreuve notre compréhension du logiciel. Il est impossible de simplement lister l'ensemble des 2^{20000} configurations acceptées par le noyau Linux.

Mais toutes les configurations logicielles ne se valent pas. Suivant celle choisie, le logiciel s'exécutera d'une manière plus ou moins efficace en terme de performance. Par exemple, le temps et l'énergie moyens nécessaires à l'encodage d'une vidéo augmenteront lorsque l'utilisateur exigera d'obtenir une meilleure qualité d'image. Du fait de la complexité grandissante des logiciels et des nombreuses interactions existant entre leurs options, il est difficile pour un humain de déterminer quelle configuration doit être choisie pour optimiser les performances logicielles.

En revanche, l'apprentissage automatique s'est révélé particulièrement efficace pour capturer les interactions au sein de l'espace de configuration d'un logiciel. En particulier, dériver les modèles (prédictifs) de performance, associant une valeur de performance à chaque configuration, permet de configurer automatiquement les logiciels, c'est-à-dire de déterminer quelle configuration doit être choisie (1) par défaut ou (2) afin de le spécialiser pour une tâche définie.

Problème

Le logiciel s'exécute au sein d'un environnement composé d'éléments variables, comme le matériel utilisé, l'architecture du système, le système d'exploitation, la nature des entrées du logiciel, *etc.* Tous ces éléments peuvent influer sur la couche logicielle, modifiant l'impact des options sur les performances. Nous définissons l'ensemble de ces interactions, et leur influence sur les logiciels configurables comme la *variabilité profonde*. Ces interactions peuvent avoir une influence directe sur notre manière de configurer les logiciels. Si l'effet des configurations change avec l'environnement d'exécution des logiciels, alors une configuration optimale pour un environnement ne le sera peut-être pas pour un autre environnement. En conséquence, nous ne devrions pas fournir une configuration par défaut fixe pour tous, mais au contraire l'adapter à chaque utilisateur.

La variabilité profonde impacte directement la validité des modèles de performance: si les distributions de performance changent avec l'environnement d'exécution, alors les modèles qui les prédisent doivent être ré-entraînés à chaque modification d'environnement. Par exemple, admettons qu'un développeur entraîne un modèle de performance avec des configurations mesurées directement sur son environnement de travail. S'il n'a pas le même système d'exploitation que l'utilisateur final et que son choix d'environnement (modèle d'ordinateur, système d'exploitation, etc.) influe sur l'effet des configurations qui ont servi à entraîner le modèle, alors l'utilisateur final ne pourra ou ne devra pas utiliser le modèle de performance du développeur, invalide dans son cas. Dans ces conditions, pré-entraîner des modèles de performance semble complètement inutile, car les développeurs n'auront jamais exactement le même environnement d'exécution que l'utilisateur final. Autrement dit, les modèles de performance entraînés par tous les chercheurs ne seront jamais pleinement opérationnels pour les utilisateurs des logiciels configurables.

Objectifs

À travers cette thèse, nous souhaitons d'abord envoyer un message à destination des chercheurs en variabilité logicielle, à savoir que **la variabilité profonde représente un problème majeur rencontré par les praticiens développant et configurant nos logiciels mais pas (encore suffisamment) traité par la recherche. Elle peut menacer la validité de certaines techniques proposées dans l'état de l'art, notamment les modèles de performances.**

L'objectif de cette thèse est double.

Premièrement, il s'agit de démontrer, quantifier et illustrer empiriquement l'existence de la variabilité profonde. Si ce problème est connu par les ingénieurs et les développeurs confrontés à l'optimisation de performance, il doit être remonté au niveau des chercheurs en variabilité logicielle, afin que ces derniers s'en emparent, le lient aux progrès technologiques récents et les adaptent en conséquence, diminuant ainsi le fossé existant entre l'état de l'art de la recherche et son application pratique.

Nous cherchons ensuite à limiter les conséquences de la variabilité profonde, et même à en profiter pour définir de nouvelles techniques. En particulier, nous souhaitons exploiter la variabilité profonde pour étendre la durée de vie des modèles de performance existants. Enfin, nous

l'utilisons afin d'entraîner des modèles de performance qui seront résilients et se généraliseront à différents environnement d'exécution.

Contributions

Tout d'abord, nous définissons la variabilité profonde, concept que nous avons défendu dans différentes communautés de recherche, notamment les systèmes configurables et l'ingénierie des performances.

Puis, pour démontrer que la variabilité profonde constitue un problème de taille, nous analysons et rassemblons des preuves empiriques témoignant de l'existence et de l'ampleur de la variabilité profonde. Par exemple, nous démontrons l'existence d'interactions non linéaires existant entre les options de configurations des systèmes logiciels et les données qu'ils prennent en entrée. Nous montrons également l'existence d'interactions linéaires et non-linéaires entre les options choisies lors de la compilation du logiciel et celles choisies lors de son exécution.

Pour arriver à cela, nous collectons des données sur les performances des systèmes configurables, mesurées dans divers environnements d'exécutions. Ces données sont ouvertes, disponibles et libres d'utilisation. Nous fournissons également des containers docker pour faciliter la réplication du protocole de mesure.

Nous explorons également de nouvelles techniques permettant de faire avancer l'état de l'art des modèles de performance, en y intégrant la variabilité profonde. Par exemple, nous démontrons l'usage de l'apprentissage par transfert entre différents systèmes logiciels exécutant la même tâche (par exemple entre deux encodeurs vidéos comme x264 et x265).

Enfin, nous apportons des pistes de solutions pour résoudre le problème de la variabilité profonde. Nous proposons des profils d'environnements logiciels homogènes en performance, qui permettent de réduire le coût des mesures nécessaires à la compréhension de la variabilité profonde. Puis, nous proposons d'enrichir les modèles de performance en y intégrant des informations sur l'environnement d'exécution du logiciel, comme par exemple des caractéristiques sur les données d'entrée. Appliquées à plus large échelle, ces contributions pourraient changer notre manière d'entraîner des modèles de performance.

Contenu du manuscrit

Le chapitre 1 présente et définit les éléments nécessaires à la compréhension de cette thèse, puis le chapitre 2 propose un panorama de l'état de l'art concernant la variabilité logicielle et les modèles de performance.

Nous découpons ensuite le contenu original du manuscrit suivant trois axes :

1. Le premier axe se focalise sur l'impact de différentes couches de variabilité sur les distributions de performance, et l'interaction entre ces couches et les options des logiciels configurables. Nous décomposons cet axe en trois chapitres, le chapitre 3 qui définit la variabilité profonde, le chapitre 4 s'intéressant aux données fournies en entrée des logiciels configurables, et le chapitre 5 aux interactions entre options choisies à la compilation et à l'exécution.

2. Le deuxième axe propose d'étendre la durée de vie et l'applicabilité des modèles de performances, en profitant notamment de la variabilité profonde. Une nouvelle fois, nous le décomposons en deux chapitres distincts, le chapitre 6 exploitant les similitudes entre environnements et le chapitre 7 exploitant les similitudes entre logiciels pour réutiliser au maximum les modèles de performance.
3. Le troisième et dernier axe explore l'entraînement de modèles de performance robustes à la variabilité profonde. Nous prouvons le concept sur les données d'entrée des logiciels dans le chapitre 8.

Enfin, le chapitre 9 conclue le manuscrit et présente des idées pour poursuivre sur ce sujet.

Summary

Context

All software systems are more or less configurable. In practice, this implies that they accept configurations. A configuration makes it possible to modify the behaviour of a software according to the needs of its user. In practice, the user has a set of options, to which he can assign values. To take a example: should the video conference be recorded? (yes or no) What size icon should be used to display the participants? (small, medium or large) What is the maximum number of participants? (12, 15, 23, *etc.*) All these choices form a configuration. Once provided to the software, it adapts its execution in accordance with the values of the options that make up this configuration. The software is then qualified as configurable.

But not all software configurations are equal. Depending on which one is chosen, the software will run more or less efficiently in terms of performance. For example, the average time and energy required to encode a video will increase when the user demands better image quality. Because of the increasing complexity of software and the many interactions between its options, it is difficult for a human to determine which configuration should be chosen to optimise software performance.

In contrast, machine learning has proven to be particularly effective in capturing the interactions within the configuration space of software. In particular, deriving (predictive) performance models, associating a performance value with each configuration, allows software to be automatically configured, *i.e.,* to determine which configuration should be chosen (1) by default or (2) in order to specialise it for a defined task.

Problem

The software runs in an environment made up of variable elements, such as the hardware used, the system architecture, the operating system, the nature of the software inputs, *etc.* All these elements can influence the software layer, changing the impact of the options on performance. We define all of these interactions, and their influence on configurable software as "deep variability". These interactions can have a direct influence on how we configure software. If the effect of configurations changes with the environment in which the software is run, then an optimal configuration for one environment may not be optimal for another environment. As a result, we should not provide a fixed default configuration for all, but instead adapt it to each user.

Deep variability directly impacts the validity of performance models: if performance distributions change with the execution environment, then the models that predict them must be re-trained with each environment change. For example, let us say a developer trains a performance model with configurations measured directly on his working environment. If he does not

have the same operating system as the end user and his choice of environment (computer model, operating system, *etc.*) influences the effect of the configurations used to train the model, then the end user will not be able to or should not use the developer's performance model, which is invalid in his case. Under these conditions, pre-training performance models seems completely useless, as developers will never have exactly the same execution environment as the end user. In other words, the performance models trained by all researchers will never be fully operational for users of configurable software.

Goal

Through this thesis, we first send a message to researchers in software variability, namely that **deep variability represents a major problem encountered by practitioners developing and configuring our software** but not yet sufficiently addressed by research. It can threaten the validity of some of the techniques proposed in the state of the art, especially performance models. The objective of this thesis is twofold.

First, we exhibit, quantify and empirically demonstrate the existence of deep variability. If this problem is known by engineers and developers confronted with performance optimisation, it must be brought up to the level of software variability researchers, so that the latter can take hold of it, link it to recent technological advances and adapt them accordingly, thus reducing the gap between the state of the art of research and its practical application.

We then seek to limit the consequences of deep variability and even to use it to define new techniques. In particular, we aim to exploit deep variability to extend the lifetime of existing performance models. Finally, we use it to train performance models that will be resilient and generalise to different execution environments.

Contributions

We first define deep variability, a concept that we have advocated in different research communities, including configurable systems and performance engineering.

Then, to prove that deep variability is a significant problem, we analyse and gather empirical evidence of the existence and magnitude of deep variability. For example, we demonstrate the existence of non-linear interactions between the configuration options of software systems and the data they take as input. We also show the existence of linear and non-linear interactions between the options chosen during the compilation of the software and those chosen during its execution.

To achieve this, we collect data on the performance of configurable systems measured in various execution environments. This data is open, available and free to use. In each case, we provide docker containers to facilitate the replication of the measurement protocol.

We also explore new techniques for advancing the state of the art of performance models by incorporating deep variability. For example, we demonstrate the use of transfer learning between different software systems performing the same task (*e.g.,* between two video encoders like `x264` and `x265`).

Finally, we provide insights to address the problem of deep variability. We propose software environment profiles that are homogeneous in performance. Using them would reduce the cost of the measurements required to understand deep variability when benchmarking software systems. Then, we propose to enhance the performance models by integrating information about the software execution environment, such as features on the input data. Applied on a larger scale, these contributions could change the way we train performance models.

Content

Chapter 1 introduces and defines the elements necessary for the understanding of this thesis, then Chapter 2 proposes an overview of the state of the art concerning software variability and performance models.

We then divide the original content of the manuscript along three axes:

1. The first axis focuses on the impact of different layers of variability on performance distributions, and the interaction between these layers and configurable software options. We decompose this axis into three chapters: Chapter 3 defines deep variability, Chapter 4 is interested in the data provided as input to the configurable software and Chapter 5 in the interactions between options chosen at compile time and at run time.
2. The second axis proposes to extend the lifetime and applicability of performance models, taking advantage of deep variability. Once again, we break it down into two distinct chapters, Chapter 6 exploiting the similarities between environments and Chapter 7 exploiting the similarity between distinct software systems to reuse the performance models as much as possible.
3. The third and last axis explores the training of performance models robust to deep variability. We prove the concept on the input data layer in Chapter 8.

Finally, Chapter 9 concludes the manuscript and presents ideas for further work on this topic.

Table of Contents

List of acronyms

BEETLE	BEllwEther Transfer LEarner	State-of-the-art Technique
DRPC	Daily Relative Percentage Change	Methodology
DT	Decision Tree	Learning Algorithm
DTW	Dynamic Time Warping	Methodology
GB	Gradient Boosting (trees)	Learning Algorithm
IQR	InterQuartile Range	Methodology
KNN	K-Nearest Neighbors	Learning Algorithm
LR	Linear Regression	Learning Algorithm
L2S	Learning to Sample	State-Of-The-Art Technique
MAPE	Mean Absolute Percentage Error	Methodology
ML	Machine Learning	Approach
MS	Model Shift	State-Of-The-Art Technique
NN	Neural Network	Learning Algorithm
PCA	Principal Component Analysis	Learning Algorithm
RF	Random Forest	Learning Algorithm
SKU	Stock Keeping Unit	Other
SPL	Software Product Lines	Approach
TEAMs	Transfer Evolution-Aware Model shifting	State-of-the-art Technique
TL	Transfer Learning	Approach

List of Figures

List of Tables

Part I

State Of The Art

This part defines the context (Chapter 1 in page 22) and the recent state-of-the-art progresses (Chapter 2 starting at page 32) related to this thesis.

Chapter 1

Background

This chapter introduces the context and the key concepts needed to smoothly address the remainder of the manuscript. It is a combination of knowledge extracted out of these different books [1, 2, 3] that we summarise and illustrate with various original and concrete examples. As this thesis, this background lies at the intersection of three domains; configurable systems, performance engineering and machine learning.

1.1 Configurable Systems

The core topic of this thesis is the study of software systems that can be customised and modified according to the needs of their users. We start by introducing software product lines to present how they should be conceived. Then, we present how to customise them. Finally, we detail the consequences of this customisation on their performance.

1.1.1 Software Product Lines

Software Product Lines are a fundamental concept on which we rely for this thesis, since it historically allowed to include various functionalities in software systems and to build configurable systems. We start by presenting the concept of product line in real life before defining its version applied to software systems.

A product line is an industrial technique designed to increase the productivity of machines or employees in charge of building a family of products — assets or goods sharing similar properties — as the hot beverages drawn in the bottom right of Figure 1.1.

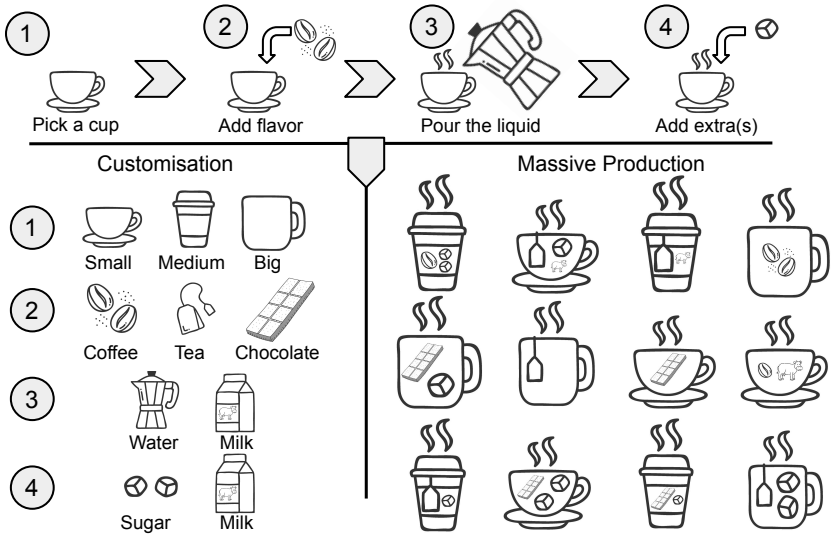


Figure 1.1 – A Hot Beverage Product Line

The first strength of product lines lies in the reuse of the same protocol. In the proposed example, the common protocol to prepare a hot drink — on top of Figure 1.1 — consists in four steps: 1. grab a cup; 2. put flavor; 3. pour the liquid; 4. add extra ingredients in the chosen cup. Apart from any debate on the consequences of integrating human beings as part of these protocols, following them factually diminishes the times required to switch between the different steps of the product conception and between each product. As a result, it significantly speeds up the individual creation of a product [2]. But in practice, the major advantage of a common protocol is a reduced cost of production, since similar infrastructures are not duplicated anymore but shared for the production of multiple products.

The second strength of product lines is the possibility to customise the different steps of the creation. Everyone is different in a family of products and deserves a special treatment. If the order of the steps are the same for all, each step can be adapted for each product. For instance, in our example (bottom left of Figure 1.1) one could change: 1. the size of the cup, *small*, *medium* or *big*; 2. the plant used to bring the flavor of the beverage, *coffee*, *tea* or *chocolate*; 3. the liquid used in the beverage, *water* or *milk*; 4. the kind and number of extra ingredients to add in the final drink, *sugar* or *milk*. In the end, it provides a way to produce variable products, just by selecting a combination of different steps.

A third and last strength of product lines is their ability to handle and embed constraints during the design of the infrastructure. Though product lines allow to select different choices in practice, as defined earlier, we may not be able to produce all the combinations of options. In our example, some combinations do not make sense; one could argue that pouring milk in a cup flavored with coffee is a gastronomical nonsense that should be explicitly forbidden. In the design of the product line, its designer can restrain the possibility to use milk when coffee has been chosen before, embedding the constraint directly into the supply chain.

In summary, product lines conciliate the standardisation of the creation of products and the need to adapt them to the final customer. Although its initial cost — in terms of design and implementation — is higher compared to the basic uncustomisable supply chain, it is worth deploying to allow the massive production of various and customized assets [3].

In a similar way, the concept of product lines has been successfully applied to software systems. Instead of producing a set of goods, the idea is to create a variant of software systems. According to Clemens *et al.* [4], the so-called Software Product Line (SPL) is "*a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*"

We retrieve the three main advantages brought by physical product lines. First, SPL maximise the amount of reused components among a family of software systems [1] and provide a unique codebase to implement and debug, which facilitates their deployment and reduces their cost of maintenance. Second, they make it possible the massive customisation of software systems through the use of software options. From a user's perspective, it allows to able or disable some functionalities without manually changing the source code, which could be frightening for beginners. In the end, the user just has to modify the value of some run-time parameters to execute another software.

Third, they integrate the way to develop and use these options by-design, abstracting the complexity of dealing with their interactions and offering a simple and friendly user interface to manipulate them.

SPL have been applied to deploy some modern systems, ensuring that the resulting software is both working and configurable.

1.1.2 Software Variability

According to Svahnberg *et al.* [5], "*software variability is the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context.*" The more unalike the variants generated out of a SPL and the more diversified their functionalities, the larger the variability. Extending variability increases the range of what we can do with the SPL: since we want to maximize the possibilities of customisation for end-users, variability is often of great value. But with -great- variability comes a great cost; more maintenance [1], a higher attack surface [6], several sources of bugs [7], a higher requirement of knowledge to master the software that can repel new users [8], higher risk of technical debt, *etc.* Therefore, researchers are interested in developing techniques to handle the cost of variability.

We will concretely illustrate the various notions presented in the rest of this chapter with our showcase software system, the video encoder `x264`.[1]

Configuration Options

To action this variability and make the freedom brought by SPL accessible to everyone, developers encapsulate the different functionalities into software features. These are often called configuration options in the text. A feature is defined as "*a prominent or distinctive user-visible aspect, quality, or characteristic of a software system*" [9]. A feature is a subpart of a software systems designed to deliver a functionality to the user via an increment of code. There are numerous mechanisms to deliver features: configuration files, command-line parameters, feature toggles or flags, plugins, *etc.* For instance, Figure 1.2a shows an excerpt of `x264` source code, namely the file *cabac.c*, in charge of implementing different functionalities related to the external option *--no-cabac*. This file also handles all the internal interactions between this option and the rest of the code.

Configuration options can be of three different kinds. As *--no-cabac*, some of them are Boolean, they have two possible values, activated or deactivated. There are also numerical options with a range of integer or float values, like *--ref* n, $n \in [|1 : 16|]$. Options with string values also exist and the set of possible values is usually predefined, as *--profile* or *--preset* in Figure 1.2c.[2]

To explain how to use the potential of options and communicate their knowledge to practitioners, developers rely on the documentation. As explained in the documentation [3] of `x264` detailed in Figure 1.2b, the option *--no-cabac* disables the CABAC functionality for the current

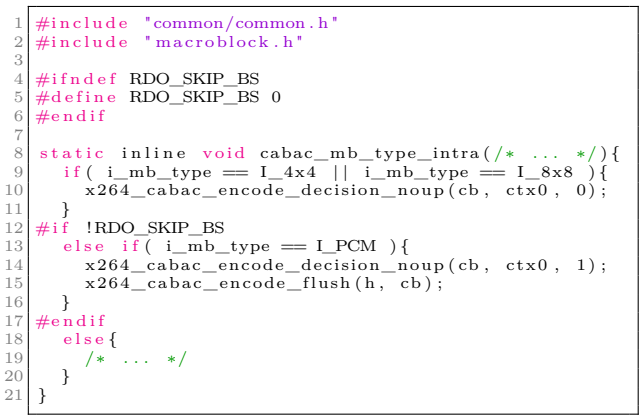
[1] See the website at `https://www.videolan.org/developers/x264.html` and the source code at `https://github.com/mirror/x264`

[2] This graph was made with FeatureIDE [10]

[3] See `https://en.wikibooks.org/wiki/MeGUI/x264_Settings#no-cabac`

execution of x264. Often, it is also common to include in the documentation (1) the potential effects of software options and (2) the interactions between software options. For example, and as shown in Figure 1.2c, setting the value 'ultrafast' of the *--preset* option implies the activation of *--no-cabac*,[4] while the 'Baseline' value of the *--profile* option implies its deactivation.[5]

Once in possession of all this knowledge, users are informed enough to know whether they should activate or deactivate *--no-cabac* in their case. Two executions of x264 are shown in Figure 1.2d, the top one without and the bottom one with *--no-cabac*.

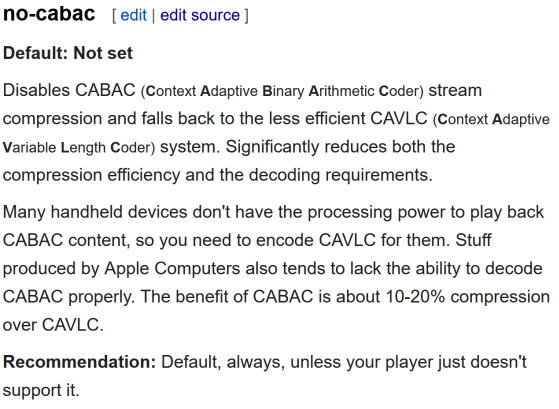


```c
#include "common/common.h"
#include "macroblock.h"

#ifndef RDO_SKIP_BS
#define RDO_SKIP_BS 0
#endif

static inline void cabac_mb_type_intra(/* ... */){
   if( i_mb_type == I_4x4 || i_mb_type == I_8x8 ){
      x264_cabac_encode_decision_noup(cb, ctx0, 0);
   }
#if !RDO_SKIP_BS
   else if( i_mb_type == I_PCM ){
      x264_cabac_encode_decision_noup(cb, ctx0, 1);
      x264_cabac_encode_flush(h, cb);
   }
#endif
   else{
      /* ... */
   }
}
```

(a) Excerpt of the source code of x264, file *cabac.c*



no-cabac [edit | edit source]

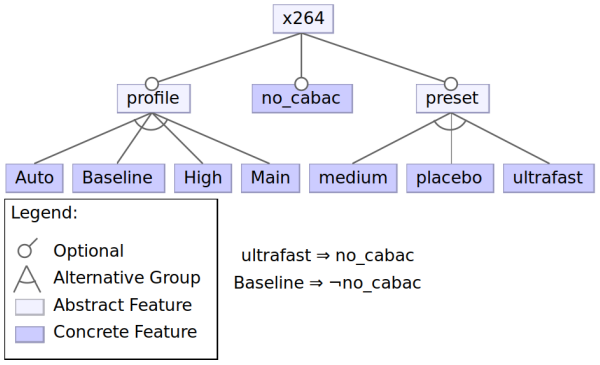
Default: Not set

Disables CABAC (Context Adaptive Binary Arithmetic Coder) stream compression and falls back to the less efficient CAVLC (Context Adaptive Variable Length Coder) system. Significantly reduces both the compression efficiency and the decoding requirements.

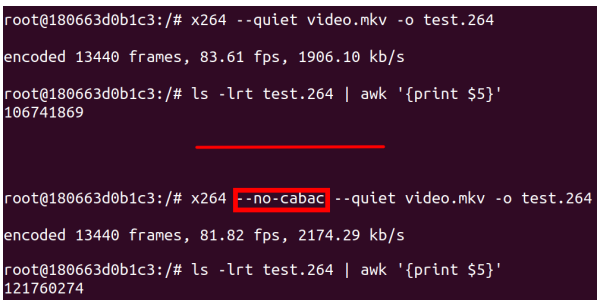
Many handheld devices don't have the processing power to play back CABAC content, so you need to encode CAVLC for them. Stuff produced by Apple Computers also tends to lack the ability to decode CABAC properly. The benefit of CABAC is about 10-20% compression over CAVLC.

Recommendation: Default, always, unless your player just doesn't support it.

(b) Documentation of x264, *--no-cabac* option



(c) Constraints of the *--no-cabac* option



(d) The *--no-cabac* option in practice

Figure 1.2 – Journey of the *--no-cabac* option, from the head of developers to the hands of x264 end-users

Configurations

Software systems are qualified as configurable when configurations can modify their executions. A configuration consists in assigning a value to each software feature. For example, in the bottom part of the Figure 1.2d, first command-line:

— the "x264" command specifies we want to execute the x264 video encoder;

— as described above, *--no-cabac* is an option disabling the CABAC functionality;

— *-quiet* is an option limiting the verbosity of the command-line output of the software;

[4] See http://dev.beandog.org/x264_preset_reference.html
[5] See https://sites.google.com/site/linuxencoding/x264-ffmpeg-mapping

— *video.mkv* indicates the location of the video to be compressed;

— the option *-o*, alias of *--output* here set to the *test.264* file, indicates the location of the compressed video.

The "x264" command aside, the whole line is part of the configuration accepted by `x264`. By simplicity, users only have to explicit the changes *w.r.t.* the default values. Other options are assumed to keep their default value. Considering the 118 options of `x264`, most of the configuration is hidden on this example. The software system does not accept all the configurations. Some options are mandatory to fill in *e.g., --output* the location of the video to be compressed in the previous example. A configuration without a value for a mandatory option should make the software crash. In the same way, some constraints, more restrictive than those exposed in Figure 1.2c, prevent the use of two values — or options — in the same configuration and can also provoke a bug. Although it is not the main purpose of this document, related research directions aim at modelling and express these constraints to only allow valid configurations.

1.1.3 Performance Properties

As long as the configuration is accepted and the software works, it is likely that no one will complain. But it does not necessarily imply that there is no room for optimisation. To measure the efficiency of a software, researchers measure its performance properties. In the case of `x264`, a performance property can be the size of the compressed video, the *encoding time*, the *power consumption*, the *bitrate*, *etc.* Configurations play a major role in the optimisation of configurable systems as depending on the configuration provided to the software, its performance values will be affected, as shown by Figure 1.2d. After two different video encodings, the first without and the second with *--no-cabac*, we can already observe a small difference in performance in terms of *bitrate* and *encoded sizes*, of about 14%. This small experiment validates the documentation (Figure 1.2b): disabling *--no-cabac* truly reduced the *size* of the encoded video in our case.

1.1.4 On the Complexity of Predicting Performance

But not all cases are as simple to understand as the previous one. The complexity of software systems tend to grow over the years, which includes an increase of their number of files and lines of code but also of their number of external options. For instance, the number of files used in the Linux kernel has been multiplied by 4.2 since 2005, resulting in 20 millions of lines of code added through the years, for a total of nearly 15 000 supplementary kernel options, as shown in Figure 1.3a. For colossal configuration spaces like this, it is not always possible to have access to a complete documentation for each couple of option and performance property or even to understand the individual effect of each option [11]. Moreover, options can interact with each other [12] and alter performance properties [13]. If they do so, it is not sufficient to only consider the individual effect of each option, but necessary to also include the combination of these effects to correctly configure the software [11].

1.2 Performance Models

Performance models originally referred to models predicting the performance value of microprocessors based on their architecture [14]. In 2013 however, Guo *et al.* [15] proposed to use machine learning to automate the performance prediction of software systems. So, since ten years in the configurable system community, performance models are known as machine learning algorithms linking the configurations of a software system to their performance value. In this section, we first present the notions related to machine learning, the toolbox hidden behind performance models. We then detail how to train and use a performance models in practice.

1.2.1 Machine Learning

According to Arthur Samuel [16], Machine Learning (ML) is a "*field of study that gives computers the ability to learn without being explicitly programmed.*" ML is a subpart of artificial intelligence automating the decision related to a task through the use of a mathematical model calibrated with data. This kind of models can achieve a broad range of tasks, like identify the content of images, translate and transcript text or even play games in an autonomous way. The recent increase of computing power allowed to exploit the progresses made (more than) twenty years ago like convolutional neural networks (1989) [17] or energy-based models (2006) [18], giving ML a second burst of life since 2012.

In this thesis, we place ourselves in the context of generalisation learning. Once the pattern of the data are recognised and embedded in the model, we expect that the model will be able to reproduce what has been learnt before when confronted to unseen situations. To ensure the generalisation of the prediction, we only fed the model with a part of the data (training set) and evaluate the validity of the model with another subset of the data (validation set) not overlapping with each other. If the accuracy of the model is not good enough on both parts, we say that the model is underfitted, it requires either more data to learn from or a more complex model to capture the patterns. If the predictions of the model are only accurate on the training set and not on the validation set, we say that it overfits on the data, the model is focused on mimicking the patterns of the training set and will not generalise well to unseen cases. The goal is to obtain the same level of accuracy on both sets. Alternatively, for the algorithms requiring a choice and an optimisation of model (hyper)parameters, a third part of the data — the test set — can be used to ensure that the chosen parameters for the model are not only valid for the training and validation sets (another kind of overfit). Although it exists, we do not adopt this paradigm in this document since it is not (yet) in the state-of-the-art practice of performance models. As a result, we consider the test and the validation sets as the same subpart of the data.

We often distinguish three types of learning: supervised, unsupervised, and reinforcement learning. For a supervised approach, we have access to the value of the property we are predicting during the training. The goal of the model is just to link a property value to the data. For instance, if the task involves the prediction of digits depicted in images, the label of the digit is known for all images of the training set. In an unsupervised approach, we do not know *a priori* the variable of interest and its value is intrinsically dependent of the model. A typical illustration of unsupervised learning is the clustering *i.e.,* how to separate the data into groups,

but it also contains more diverse tasks like associations between individuals, prediction in time, *etc.* With a reinforcement approach, the model is seen as an agent evolving in an environment, where the agent has to choose between several policies. Based on the output of his actions, we update the reward associated to each action, the goal being to converge to the best policy in this environment *w.r.t.* to the objectives of the agent. In this document, we mostly rely on supervised learning for the classical performance models and periodically use unsupervised algorithms. We mostly consider cases where the property to predict is a quantitative (or numerical) variable. In terms of vocabulary, we refer to it as a regression problem, as opposed to a classification problem, where the goal would be to predict a set of labels instead of a distribution of numerical values.

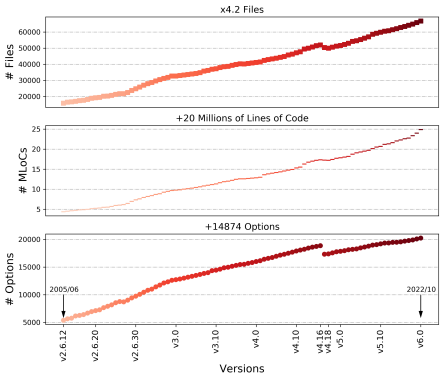
After presenting the general principles of machine learning, we present the main algorithms trained throughout the document, namely:

1. Linear Regression [19] (LR), estimating the performance value with a weighted sum of variables. It is not designed for complex cases;
2. Decision Tree [19] (DT), using decision rules to separate the configurations into sets and then predicting the performance separately for each set;
3. Random Forest [19] (RF) , an ensemble algorithm based on bagging, combining the knowledge of different decision trees to make its prediction;
4. Gradient Boosting Tree [19] (GB), also derived from Decision Tree. Unlike Random Forest training different trees, gradient boosting aims at improving one tree by specialising its rules of decision at each step;
5. Neural Networks [17] (NN), a combination of layers of perceptron units communicating with each other and combined together to predict the results.

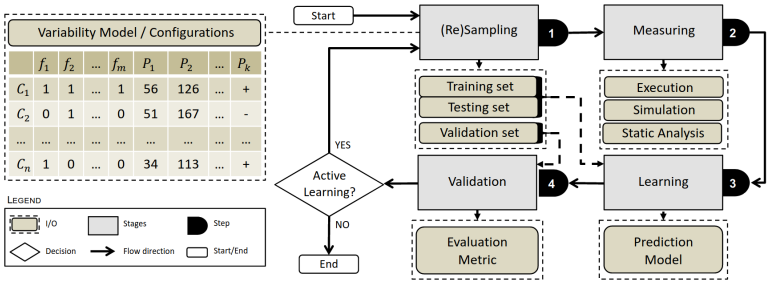
1.2.2 Sampling, Measuring, Learning

Machine learning techniques have been widely considered in the literature to learn software configuration spaces [15, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]. These works measure the performance of a configuration sample under specific settings to then build a machine learning model capable of predicting the performance of any other configuration. The resulting model is called a performance predictive model — or performance model in short.

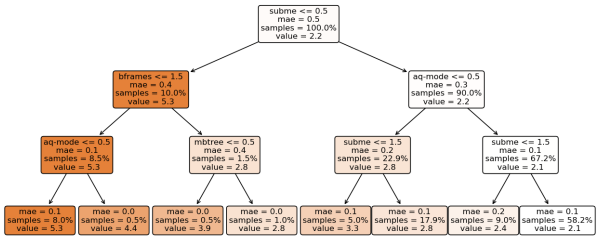
The training of a performance model typically requires three steps: sampling, measuring, learning [21]. First, this method first requires to gather data, in our case to sample configurations *i.e.,* listing a set of valid software configurations of interest. The current state-of-the-art solution to sample configurations involves the use of a feature model [9, 34], capturing the different constraints between software options and thus generating valid configurations, accepted by the software. One example of tiny feature model is presented in Figure 1.2c. Then, we measure this set of configurations. For each, we compute the performance property and store the configuration coupled with their performance values. We present the structure of a dataframe of configurations and performance properties in the left part of Figure 1.3b. In this dataframe, C_1, ..., C_n are the n configurations (rows), f_1, ..., f_m the values of the m features (columns) and P_1, ...,P_k the values of the k performance properties (columns) corresponding to these configurations. Finally, it is possible to learn from this data, as presented in the right part of Figure 1.3b. We train a



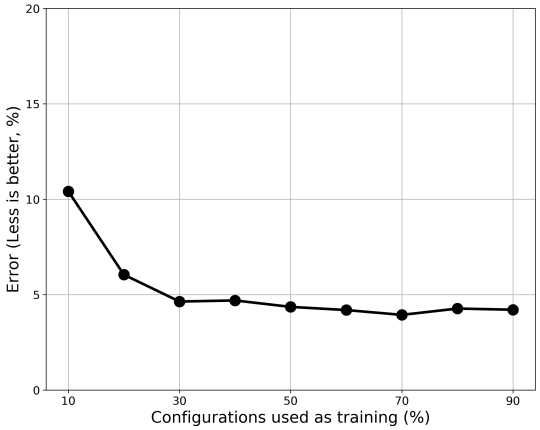
(a) Why? The growing complexity of Linux



(b) How? Sampling, Measuring, Learning (Credits to [21])



(c) Predicting performance out of configurations



(d) Prediction error of ML techniques

Figure 1.3 – Why and how to automate the performance prediction of software systems?

performance model on it, linking each configuration to its performance value, in the hope that the model can be generalised to a set of unknown configurations. For instance, in Figure 1.3c, we display the graphical representation of the decision tree trained on the data of Figure 1.3b to predict the output sizes of compressed videos out of `x264` configurations. We used is a decision tree here, but the principle would remain the same for other algorithms. Each node represents a subset of the configurations. To illustrate the different numbers, the node on the left of the second line represents a subset of 10% of the configurations, all having a *--subme* value under 0.5. Their compressed size is estimated at $5.3MB$, which results in a Mean Absolute Error (MAE) of $0.4MB$.

The training is done using different numbers of configurations in the training set, we then display the resulting MAE for the validation set in Figure 1.3d, using the model presented in Figure 1.3c. The more configurations fed to the model, the more accurate its estimation of the performance values; with 10% of the configurations used in the training set, the average error is estimated at 10%, while with 90% of the configurations, we can expect to reach about 4% of error. In general, practitioners seek to train performance models in a data-efficient way [35] *i.e.*, to measure the least possible configurations. In our case, we only need 30% of the configurations to train an accurate model, so the rest of the configurations may be a waste of resources in a real-world context.

1.2.3 Benefits

A medical study [36] has shown and quantified the limitations of the human brain in terms of concentration (20→30 seconds) and in the amount of information kept in the short-term memory (7 elements). Our brain is made to reason at high levels, which seems not compatible with the processing of numerous tabular data composed of millions of cells. Given the scale of some feature, most of our brains would not be able to just compute the weighted sum of all configuration options, as it is for instance required for linear regressions. So, we guess that for complex systems, with more than a dozen of options, we would not be able to achieve a performance prediction with the accuracy of the current state-of-the-art practice of performance models. Besides, the execution of a performance model is in general quicker than what we would need to predict performance. For small systems like `x264`, generating ten repetitions of the curve of Figure 1.3d only requires few seconds. So, the first and main benefit of performance models is to allow the performance prediction of software systems, a task that humans are bored to or can simply not achieve.

A second benefit of performance models is to allow us to understand the effect of the different configuration options on performance and interpret which option is responsible of performance changes [13, 31]. With this knowledge in mind, developers of software systems can focus on the influential features and users can change the values of interesting software options *w.r.t.* their performance goals. It can also be used to simplify the maintenance of the software and assign priorities to features based on their estimated impact on performance: if the impact of a feature is always negligible on software performance, maybe it is not a priority to maintain this feature as part of the software system.

Besides from increasing our understanding of software systems, performance models can be derived to automatically configure software systems [29, 30]. Out of a set of configurations, we can apply the performance model to predict their values of performance. Then, we can reverse the performance model: we select the configuration resulting in the best performance value and set it as the default configuration. Alternatively, we can specialise software systems thanks to performance models [37, 38]. We select a configuration leading to the best performance value meeting the user requirements or constraints. A third benefit of performance models is the automatic configuration or specialisation of software systems.

1.2.4 Drawbacks

The main drawback of performance models is the cost of measuring all the configurations [35] required to train the model. Typically, the sampling phase requires to measure at least hundreds of configurations, which can be time- and energy-consuming. The cost of measuring so many configurations can be prohibitive if the performance prediction is not essential for the end-user.

Moreover, if the measures are the responsibility of the developers, it requires to carefully set up the environment, remove the noise and ensure the validity of performance measurement so the values correspond to the production environment. This also requires an additional performance engineering effort, in addition to the complexity of mastering the machine learning techniques and tools needed to train the model.

Another potential weakness would be the lack of generalisation of the model, if it is trained

on the developer environment but then applied on the user side [39]. No one can guarantee that the accuracy of the initial model will hold when applied to the user environment. This last drawback is the starting point of this thesis, we develop it in the rest of the manuscript.

Typographic Convention. For the rest of this thesis, we adopt the following typographic convention: `emphasized` will be relative to a software system, *--slanted* to its configuration options and *small* to its performance properties.

Chapter 2

State-of-the-Art

In the current state-of-the-art of performance modelling, it is assumed that the only cause of software variability is the choice of configurations. And indeed, changing its configuration options makes a software performance vary. But other elements of the executing environment could also be an additional source of variability. Our message to the scientific community is two-folded. First, we want to acknowledge that the software environment is impacting its performance in a non trivial way with non-linear feature interactions. Second, if these elements change the performance distributions of software systems, our practice of performance models have to evolve accordingly and take these elements into account.

This chapter is divided into three sections. The first one proposes a literature analysis of research papers to study the current state-of-the-art practice. The second gathers research works showing the effect of the software environment on its performance. The third lists all the possible solutions (to the best of our knowledge) that can be used to improve the practice of performance modelling.

2.1 Browsing the Related Work

In this section, we explore the significance of the problem of the performance prediction under different executing environments in research. Specifically, we want to state whether researchers are aware of the issue and how they deal with it in their papers. Is the interaction between software configurations and the rest of the environment well-known in research?

Protocol

First, we aim at gathering research papers predicting the performance of configurable systems with a performance model. We focused on the publications of the last ten years. To do so, we analyzed the papers published (strictly) after 2011 from the survey of Pereira *et al.* [21] - published in 2019. We completed those papers with more recent papers (2019-2021), following the same procedure as in [21]. We have only kept research work that trained performance models on software systems. We read each selected paper and answered different questions: Q-A. Is there a software system processing input data in the study? If not, the impact of input sensitivity in the existing research work would be relatively low. The idea of this research question is to estimate the proportion of the performance models that could be affected by input sensitivity. Q-B. Does the experimental protocol include several inputs? If not, it would suggest that the performance model only captures a partial truth, and might not generalize for other inputs fed to the software system. Q-C. Is the problem of input sensitivity mentioned *e.g.,* in threat? This question aims to state whether researchers are aware of the input sensitivity issue, and estimate

Table 2.1 – Browsing research papers. ***Q-A.*** Is there a software system processing input data in the study? ***Q-B.*** Does the experimental protocol include several inputs? ***Q-C.*** Is the problem of input sensitivity mentioned *e.g.*, in threat? ***Q-D.*** Does the experimental protocol include several hardware platforms? ***Q-E.*** Does the paper train a performance model on compile-time options? ***Q-F.*** Does the paper train a performance model on run-time options? ***Q-G.*** Does the paper propose a solution to generalise performance models across environments? Justifications in the companion repository.

ID	Authors	Conference	Year	Title	Q-A	Q-B	Q-C	Q-D	Q-E	Q-F	Q-G
1	Guo *et al.* [35]	ESE	2017	Data-efficient performance learning for [...]	X					X	
2	Jamshidi *et al.* [40]	SEAMS	2017	Transfer learning for [...]	X	X	X	X		X	
3	Jamshidi *et al.* [24]	ASE	2017	Transfer learning for performance [...]	X	X	X	X		X	X
4	Oh *et al.* [41]	ESEC/FSE	2017	Finding near-optimal configurations [...]	X					X	
5	Kolesnikov *et al.* [42]	SoSyM	2018	Tradeoffs in modeling performance [...]	X			X		X	
6	Nair *et al.* [27]	ESEC/FSE	2017	Using bad learners to find good [...]	X	X		X		X	
7	Nair *et al.* [43]	TSE	2018	Finding Faster Configurations using FLASH	X	X	X	X		X	
8	Murwantara *et al.* [44]	iiWAS	2014	Measuring Energy Consumption for [...]	X	X	X			X	X
9	Temple *et al.* [45]	SPLC	2016	Using Machine Learning to [...]						X	
10	Temple *et al.* [38]	IEEE Soft.	2017	Learning Contextual-Variability Models	X					X	X
11	Valov *et al.* [26]	ICPE	2017	Transferring performance prediction [...]	X		X	X		X	X
12	Weckesser *et al.* [46]	SPLC	2018	Optimal reconfiguration of dynamic [...]						X	
13	Acher *et al.* [47]	VaMoS	2018	VaryLATEX: Learning Paper Variants [...]	X	X			X		
14	Sarkar *et al.* [20]	ASE	2015	Cost-Efficient Sampling for [...]	X					X	
15	Temple *et al.* [48]	Report	2018	Towards Adversarial Configurations for SPL						X	
16	Nair *et al.* [49]	ASE	2018	Faster Discovery of Faster System [...]	X				X	X	
17	Siegmund *et al.* [13]	ESEC/FSE	2015	Performance-Influence Models for [...]	X					X	
18	Valov *et al.* [50]	SPLC	2015	Empirical comparison of regression [...]	X					X	
19	Zhang *et al.* [51]	ASE	2015	Performance Prediction of Configurable [...]	X		X			X	
20	Kolesnikov *et al.* [52]	ESE	2019	On the relation of control-flow [...]	X				X		
21	Couto *et al.* [53]	SPLC	2017	Products go Green: Worst-Case Energy [...]	X		X			X	
22	Van Aken *et al.* [54]	SIGMOD	2017	Automatic Database Management [...]	X	X	X		X		X
23	Kaltenecker *et al.* [55]	ICSE	2019	Distance-based sampling of [...]	X					X	
24	Jamshidi *et al.* [25]	ESEC/FSE	2018	Learning to sample: exploiting [...]	X	X	X	X		X	X
25	Jamshidi *et al.* [56]	MASCOTS	2016	An Uncertainty-Aware Approach [...]	X	X	X			X	
26	Lillacka *et al.* [57]	Soft. Eng.	2013	Improved prediction of non-functional [...]	X	X	X			X	X
27	Zuluaga *et al.* [58]	JMLR	2016	ε-pal: an active learning approach [...]	X	X		X		X	
28	Amand *et al.* [59]	VaMoS	2019	Towards Learning-Aided Configuration [...]	X	X	X			X	
29	Alipourfard *et al.* [60]	NSDI	2017	Cherrypick: Adaptively unearthing the [...]	X	X	X	X		X	
30	Saleem *et al.* [61]	TSC	2015	Personalized Decision-Strategy based [...]	X	X				X	
31	Zhang *et al.* [62]	SPLC	2016	A mathematical model of [...]	X					X	
32	Ghamizi *et al.* [63]	SPLC	2019	Automated Search for Configurations [...]	X	X	X			X	
33	Grebhahn *et al.* [64]	CPE	2017	Performance-influence models of [...]						X	
34	Bao *et al.* [65]	ASE	2018	AutoConfig: Automatic Configuration [...]	X	X		X		X	
35	Guo *et al.* [15]	ASE	2013	Variability-aware performance [...]	X					X	
36	Švogor *et al.* [66]	IST	2019	An extensible framework for software [...]	X	X				X	
37	El Afia *et al.* [67]	CloudTech	2018	Performance prediction using [...]	X	X				X	
38	Ding *et al.* [68]	PLDI	2015	Autotuning algorithmic choice for [...]	X	X	X		X		X
39	Duarte *et al.* [69]	SEAMS	2018	Learning Non-Deterministic Impact [...]	X	X	X	X		X	X
40	Thornton *et al.* [70]	KDD	2013	Auto-WEKA: Combined selection and [...]	X	X	X			X	
41	Siegmund *et al.* [71]	ICSE	2012	Predicting performance via automated [...]	X	X	X	X	X	X	
42	Siegmund *et al.* [72]	SQJ	2012	SPL Conqueror: Toward optimization [...]	X	X				X	
43	Westermann *et al.* [73]	ASE	2012	Automated inference of goal-oriented [...]	X	X				X	
44	Velez *et al.* [74]	ICSE	2021	White-Box Analysis over Machine [...]	X	X				X	
45	Pereira *et al.* [75]	ICPE	2020	Sampling Effect on Performance [...]	X	X	X			X	
46	Shu *et al.* [76]	ESEM	2020	Perf-AL: Performance prediction for [...]	X				X	X	
47	Dorn *et al.* [77]	ASE	2020	Mastering Uncertainty in Performance [...]	X					X	
48	Kaltenecker *et al.* [78]	IEEE Soft.	2020	The Interplay of Sampling and [...]	X					X	
49	Krishna *et al.* [23]	TSE	2020	Whence to Learn? Transferring [...]	X	X	X	X		X	X
50	Weber *et al.* [79]	ICSE	2021	White-Box Performance-Influence [...]	X	X				X	
51	Mühlbauer *et al.* [80]	ASE	2020	Identifying Software Performance [...]	X	X				X	
52	Han *et al.* [81]	Report	2020	Automated Performance Tuning for [...]	X	X				X	
53	Han *et al.* [82]	ICPE	2021	ConfProf: White-Box Performance [...]	X		X			X	
54	Valov *et al.* [83]	ICPE	2020	Transferring Pareto Frontiers [...]	X			X		X	X
55	Liu *et al.* [84]	CF	2020	Deffe: a data-efficient framework [...]	X	X	X	X		X	X
56	Fu *et al.* [85]	NSDI	2021	On the Use of ML for Blackbox [...]	X	X	X			X	
57	Larsson *et al.* [86]	IFIP	2021	Source Selection in Transfer [...]	X	X	X	X		X	X
58	Chen *et al.* [87]	ICSE	2021	Efficient Compiler Autotuning via [...]	X	X	X		X		
59	Chen *et al.* [88]	SEAMS	2019	All Versus One: An Empirical [...]	X	X				X	
60	Ha *et al.* [89]	ICSE	2019	DeepPerf: Performance Prediction [...]	X					X	
61	Pei *et al.* [90]	Report	2019	DeepXplore: automated white box [...]	X	X				X	
62	Ha *et al.* [91]	ICSME	2019	Performance-Influence Model for [...]	X					X	
63	Iorio *et al.* [92]	CloudCom	2019	Transfer Learning for [...]	X	X	X			X	X
64	Koc *et al.* [93]	ASE	2021	SATune: A Study-Driven Auto-Tuning [...]	X	X	X			X	X
65	Ding *et al.* [31]	ESEC/FSE	2021	Generalizable and Interpretable [...]	X	X	X	X		X	X
Total					61	39	29	17	8	60	15

the proportion of the papers that mention it as a potential threat to validity. Q-D. Does the experimental protocol include several hardware platforms? We then verify if the experimental protocol of research papers includes different models of hardware, to see if the conclusion could be altered when reproducing the experiment with changed hardware platforms. Q-E. Does the paper train a performance model on compile-time options? Options of configurable systems can be changed during the compilation of the system. We check whether the paper is changing the default compile-time configurations *i.e.,* if the trained performance models are related to the compile-time variability of the system. Q-F. Does the paper train a performance model on run-time options? Options of configurable systems can be changed during the execution of the system. We check whether the paper is changing the default run-time configurations *i.e.,* if the trained performance models are related to the run-time variability of the system. Q-G. Does the paper propose a solution to generalise performance models across environments? Finally, we check whether the paper proposes a solution to predict the performance of configurable systems whatever their executing environments.

Evaluation

Table 2.1 lists the 65 research papers we identified following this protocol, as well as their individual answers to Q-A→Q-G. A checked cell indicates that the answer to the corresponding question (column) for the corresponding paper (line) is *yes*. Since answering Q-B and Q-C only makes sense if Q-A is checked, we grayed and did not consider Q-B and Q-C if the answer to Q-A is *no*. We also provide full references and detailed justifications in the companion repository.[1] We now comment the average results:

Q-A. Is there a software system processing input data in the study? Of the 65 papers, 60 (94%) consider at least one configurable system processing inputs.

Q-B. Does the experimental protocol include several inputs? 63% of the research work answering *yes* to Q-A include different inputs in their protocol. But what about the other 37%? It is understandable not to consider several inputs because of the cost of measurements. However, if we reproduce all experiments of Table 2.1 using other input data, will we draw the same conclusions for each paper? Based on the results of Chapter 4 (page 50), we encourage researchers to consider at least a set of inputs in their protocol.

Q-C. Is the problem of input sensitivity mentioned *e.g.,* in threat? Only half (47%) of the papers mention the issue of input sensitivity, mostly without naming it or using a domain-specific keyword *e.g.,* workload variation [26]. For the other half, we cannot guarantee with certainty that input sensitivity concerns all papers. But we shed light on this issue: ignoring input sensitivity can prevent the generalization of performance models across inputs. This is especially true for the 37% of papers answering *no* to Q-B and thus considering one input per system. Among them, only 14% of these research works mention it in their publication.

We discuss the state-of-the-art practice related to Q-A, Q-B and Q-C in Section 2.2.1.

Q-D. Does the experimental protocol include several hardware platforms? Most of the papers kept in this study chose a unique model of hardware to run their experiment. 17 papers proposed to either partly change the used hardware platform (*e.g.,* by changing the amount of

[1] List of papers at `https://github.com/llesoil/input_sensitivity/blob/master/results/RQS/RQ6/extended.md`

RAM or the L1 cache) or even the model of hardware used. We briefly discuss the state-of-the-art practice related to Q-D in Section 2.2.2.

Q-E. Does the paper train a performance model on compile-time options? 8 research works (12%) observe the compile-time variations of software systems. But the compilation of systems or programs has not caught so much attention in the state-of-the-art practice of performance models.

Q-F. Does the paper train a performance model on run-time options? And indeed, a wide majority (60 over 65 papers) of the research works are targeting to predict the run-time variations of software systems. None among them study the interaction between compile-time and run-time options. We briefly discuss the state-of-the-art practice related to Q-E and Q-F in Section 2.2.3.

Q-G. Does the paper propose a solution to generalise performance models across environments? We identified 15 papers [54, 68, 69, 93, 24, 26, 25, 23, 83, 86, 92, 44, 57, 84, 31] proposing contributions that may help in better managing the problem of generalising the performance prediction over different executing environments. However, most of them have been designed to operate over only parts of the computing environments [24, 26, 25, 23, 83, 86, 92] and should be extended. Other works [54, 68, 69, 93, 84] only apply it to a specific domain (database [54], compilation [68], cloud computing [69, 84, 31] or program analysis [93]) with open questions about applicability and effectiveness in other areas. We detail the content of these contributions in Section 2.3.

2.2 Impact of the Software Environment on Performance

Numerous research papers mention the effects of the executing environment of a software on its performance: hardware [94, 95, 96], operating systems [96, 97, 98, 99, 100], variants [101, 80], versions [101, 102, 103, 104], compilation options [105, 106] and input data [107]. As a consequence, stating that the software environment is modifying the scale of performance is not new. In contrast, we want to prove the existence of complex interactions between the software and the rest of the stack, study their impact on performance properties and acknowledge their consequences on the current practice of performance models.

To the best of our knowledge, there are two open research directions related to these works: (1) a systematic and comprehensive assessment of the influence of individual elements onto the distribution of software configurations: only a few hardware, input data, or software configurations are usually considered; (2) empirical knowledge about how these elements composed and interact each other, for example how hardware and input data both influence the performance properties of a configurable system.

In this section, we gather research works related to (1) and (2) on a per-element basis.

2.2.1 Input Sensitivity

Numerous research works have proposed to model performance of software configurations, with several use-cases in mind for developers and users of software systems: the maintenance and understanding of configuration options and their interactions [13], the selection of an optimal configuration (tuning) [41, 30, 43], the performance prediction of arbitrary configurations [75, 78, 21]

or the automated specialization of configurable systems [37, 38]. These works measure the performance of a sample of several configurations under specific settings to then build a performance model. Input data further challenges these use-cases, since both the software configuration and the input spaces should be handled. Inputs also question and can threaten the generalization of configuration knowledge *e.g.,* a performance prediction model for a given input may well be meaningless and inaccurate for another input.

On the one hand, some works have been addressing the performance analysis of software systems [108, 109, 110, 111, 112, 113] depending on different input data (also called workloads or benchmarks in the literature), but all of them only considered a rather limited set of configurations. To illustrate this, let us provide few examples coming from the video encoding community: Maxiaguine *et al.* [114] classify multimedia streams in groups depending on their characteristics and study their performance (*i.e.,* multimedia processing time and I/O rate) on MPSoC platforms. Netflix conducts a large-scale study for comparing the compression performance of `x264`, `x265`, and *libvpx* [115]. 5000 12-second clips from Netflix catalog were used covering a wide range of genres and signal characteristics. However, only two configurations were considered and the focus of the study was not on predicting performance. As in the video encoding area, the input sensitivity issue has also been identified in some other domains: SAT solvers [116, 117], compilation [118, 68], data compression [119], database management [54, 120], cloud computing [69, 84, 31], *etc.* These works purposely leverage the specifics of their domain. However, it is unclear how proposed techniques could be adapted to any domain and all systems of the configurable system community. We aim at generalising these works, the goal being to provide a domain-agnostic study of performance according to the input data fed by configurable systems whatever the underlying domain.

On the other hand, works and studies on configurable systems usually neglect input data *e.g.,* using a unique video for measuring the configurations of a video encoder in [13]. Valov *et al.* [83] proposed a method to transfer the Pareto frontiers (encoding time and size) of performance across heterogeneous hardware environments. The inputs (video) remain fix however, which is a threat to validity. Very recently, reinforcement learning has been applied for improving the rate control policy of the VP9 video encoder [121], using the YUGC dataset [122]. Their work focuses on a specific part of the encoder and thus parameter options. In addition, reinforcement learning was used to improve the policy on average of all inputs and not according to the individual specificity of inputs. Pereira *et al.* [75] study the effect of sampling on `x264` configuration performance models for 19 input videos on two performance properties.

We aim to combine both dimensions by performing an in-depth, controlled study of several configurable systems to make it vary in the large, both in terms of configurations and inputs. We thus favor generic and domain-agnostic approach as part of this thesis. Importantly, most of the domain-specific works pursue the objective of optimizing the performance of a software system according to a given input (workload) while the work of the configurable system community aim at predicting the performance of any configurations. We want to bridge the gap between these two areas of research with the second use case in mind. Our key goal is to investigate how configuration knowledge can generalize or be transferred among all inputs.

2.2.2 Hardware Platforms

The study of Valov *et al.* [26] suggests that changing the hardware has reasonable impacts since linear functions are highly accurate when reusing prediction models. There are related work showing hardware variability [123] but not in addition to the variability brought by configurations of software systems. Most of the configurable systems researchers have shown only linear interactions between the software system and the hardware platforms on which it is executed.

For instance, Jamshidi *et al.* [24] conducted an empirical study on four configurable systems, varying software configurations and environment conditions, such as hardware, input, and software versions. It is a pioneer work *w.r.t.* the content of this thesis. Moreover, without isolating the individual effect of hardware platforms or input data on software configurations, it is challenging to understand the existing interplay between these elements. Compared to these approach, we aim at only studying elements of the software environment one by one.

2.2.3 Compile-time Variability

The problem of choosing which optimizations and flags of a compiler to apply [124, 125] has a long tradition. Since the mid-1990s, machine-learning-based or evolutionary approaches have been investigated to explore the configuration space of compilers [126, 127, 128, 129, 130, 105, 131]. Such works usually consider a limited set of run-time configurations in favor of a broad consideration of input programs. The goal is to understand the cost-effectiveness of compiler optimizations or to find tuning techniques for specific inputs and architectures.

Another direction focuses on the features of software systems that can be chosen at compile-time rather than what could be optimised thanks to the compiler. Leveraging these compile-time options has been considered in the literature to find bugs [7], to specialize the resulting software system [6]. But as shown in Table 2.1, only few works are really targeting the compile-time options with performance goals and none of them actually study the interactions between compile- and run-time variability. 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.

2.3 Existing Solutions

Most of the studies support learning models restrictive to specific static settings (*e.g.,* inputs, hardware and version) such that a new prediction model has to be learned from scratch once the environment change [21]. Besides the solutions provided by the ML community, like fine-tuning the model [132] — updating the weights of the model for the new data — for each new environment, the literature provides solutions that are specifically designed for configurable systems.

2.3.1 Transfer Learning

The current state-of-the-art solution to handle environment changes of software systems is called Transfer Learning (TL) [133, 25, 26, 134, 135]. TL has been considered to reuse measure-

ments and ML models across different computing environments, the promise being to reduce the efforts and costs of measuring new configurations. TL exploits what have been learnt on a *source* environment to apply it a *target* environment, using either the similarities between environments [25, 26, 134] or selecting the best source environment [23]. TL has also been applied to SPL in many contexts, like reconfiguration [133] or multi-objective optimisation [133] or defect prediction [136]. However, in our case, we only consider the case of performance prediction.

In this context, Valov *et al.* [26] were the first to show that linear models are effective to transfer knowledge between couples of source and target hardware environments. They proposed Model Shift (MS), training two performance models, one predicting the performance of the source and one handling the differences of performance distributions between the source and the target. This last model was in their case, linear. However, as discussed in the rest of the manuscript, environments can significantly alter performance distributions in a non linear way. There is not necessarily a linear correlation and relationship, as for hardware changes. In the recent literature, few improvements have been made to MS. For instance, Jamshidi *et al.* define Learning to Sample (L2S) [25] that combines an exploitation of the source and an exploration of the target and samples a list of well-chosen configurations to transfer the knowledge from the source to the target. As many other transfer learning works, L2S is applied between different executing environments (*e.g.,* hardware changes), but not yet on input changes (from one source input to a target input). However, L2S is highly sensitive to the selection of a source for a given target. There is this open question on how to *a priori* select the pairs source-target. Krishna *et al.* provide an answer to this issue by proposing BEETLE [137]. The principle is to find one "bellwhether" environment *i.e.,* a source environment that lead to better transfer results whatever the target. BEETLE has been designed to find optimal configurations, not to predict performance of any configurations, as in our study. The idea of BEETLE is orthogonal to the rest of the techniques and could be combined with most of them. Once a bellwhether environment is identified, it is quite easy to apply the Model shift approach or even L2S.

Martin *et al.* develop TEAMs [138], a transfer learning approach predicting the performance distribution of the Linux kernel using the measurements of its previous releases. They were the first to use transfer learning by considering a source software system (here a former version of the same system) and a target software system instead of environments. To go further on this topic, heterogeneous Transfer Learning [139, 140, 141] is an extension of (homogeneous) transfer learning handling the differences between the source and the target feature spaces — configuration spaces when studying software variability or features when considering software systems. It creates a representation of the feature space, in between the source and the target, and finally transforms both feature spaces so they fit in this representation. Applied to the transfer across software systems, it would handle the changes of features between the configuration space of the source software and the configuration space of the target software. In many areas, related work compares the performance of different tools performing the same task [142, 143, 144, 145, 146, 147]. These empirical evidences could be relevant; if two software systems have similar performance distributions, they are good candidates for the transfer. Following the idea of TEAMs, we could apply transfer learning across different software systems, to reduce the cost of training a new performance model for each system.

The literature around TL evolves at a rapid pace. There are consequently recent improvements that are not included as part of the manuscript. For instance, we did not include the causalities in the performance model to improve the transfer [148, 149], but it is another promising research direction to explore in terms of interpretable (transfer) learning. Also, we did not show the results of TL techniques considering multiple source environments, with different weights applied to each of them [150].

Overall, the transfer learning is currently the best approach to handle environment changes. However, the main drawback of transfer learning is the need to measure few configurations on the target environment, which can be a threat to the deployment or the applicability of TL techniques. In the real-world, the computation of these measurements

2.3.2 Contextual Performance Models

The relation between the software layer and its environment is not new and has been already studied with other use cases in mind: including the context of systems in feature models [151], searching for context-aware requirements [152], context-oriented programming [153] or reconfiguration according to the context [154, 155]. In the reconfiguration case, there is even a wide literature related to self-adaptive systems [156] that we do not detail in this document because related papers do not train performance models. But to the best of our knowledge, the design of context-aware techniques has not caught so much attention in the configurable system community — transfer learning could be seen as a way to avoid the problem — and especially in the associated works proposing performance models.

However, Temple *et al.* propose to learn contextual performance models [38], including features of the context (here the executing environment) as part of the data processed by the model. Training contextual-like models (for instance [54, 68, 69, 93] of Table 2.1 in their respective domains, even if they do not explicitly use this terminology) could be a viable alternative to TL if some properties of the environment are added to the model. Instead of adapting the model for each new environment, the idea would be to train the model so it is by-design general enough to covers all the possible target environments.

The existence of a general solution applicable to all domains and software configurations remains an open question. For example, is may not be always possible to extract properties of the environment that are actually useful to train a contextual performance model. In addition of that, some properties may not be relevant for the performance prediction, and may not improve the performance model. We further explore this state-of-this art practice with input data and compare it to a transfer learning baseline in Chapter 8 (page 108).

Part II

Empirical Evidence of Deep Variability

In this part, we empirically prove that the executing environment of a software system matters when measuring and predicting its performance. In fact, few elements of the environment can interact with each other but also with the software systems, changing its performance distribution. As a result, software variability is deeper than expected. In Chapter 3 (page 42), we define and illustrate the concept of deep variability. We further explore two aspects of deep variability in Chapter 4 (page 50) and Chapter 5 (page 66), namely the input data and the compilation of software systems.

Chapter 3

Introducing Deep Variability

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

3.1 Definition

Software variability has caught the attention of various scientific communities, like the software product line community. Therefore, research work have been published to improve the practices of handling the maintenance of highly configurable systems with a consequent number of options, of selecting default values of software options or understanding their effect on performance [13, 31]. But often, these research works are focused on the software system and neglected the rest of the elements. For instance, they consider one model of hardware with one version of operating system in their protocol, which is completely understandable owing to the cost of running the experiment. However, it has been shown that the software environment, in a broad sense, can interact with its configuration, thus changing its validity [12] or its scale of performance [25, 95]. That is, only considering the software elements might hide important bugs or provide non-optimal values for configuration options of the software. Beyond making more complex any kind of performance prediction, it also raises the issue of reproducible results, particularly stringent in the performance evaluation community.

Environment properties can be organised into variability layers, as the four ones depicted on Figure 3.1: hardware, operating system, software and input data.

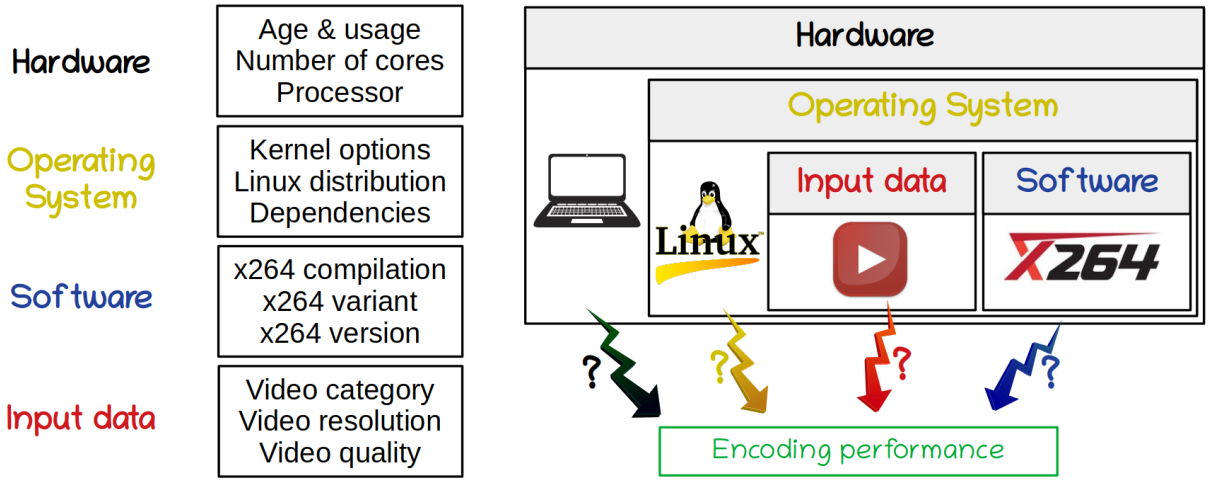


Figure 3.1 – Deep Variability of `x264`

Definition. We call Deep Variability the interactions of all variability layers that could modify the behavior or non-functional properties of a software. Deep variability could be seen as an extension of software variability encompassing all its executing environment, and the natural extension of software variability taking into account the differences of executing environments between all the users of the software system.

We propose to break the study of deep variability down to five steps:

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

We illustrate each of these steps with concrete examples in Section 3.2 and emphasize the related problems, challenges and opportunities in Section 3.3.

3.2 Motivational Example

Let us illustrate the concept of deep software variability with `x264`. In Figure 3.2, we show the duration of the video compression (in seconds) and the size of the encoded video (in *MB*), for two configurations [1] under two different environments.[2] Performance were measured with two different input videos [3] and the same version of `x264`.[4] The remainder of this section describes

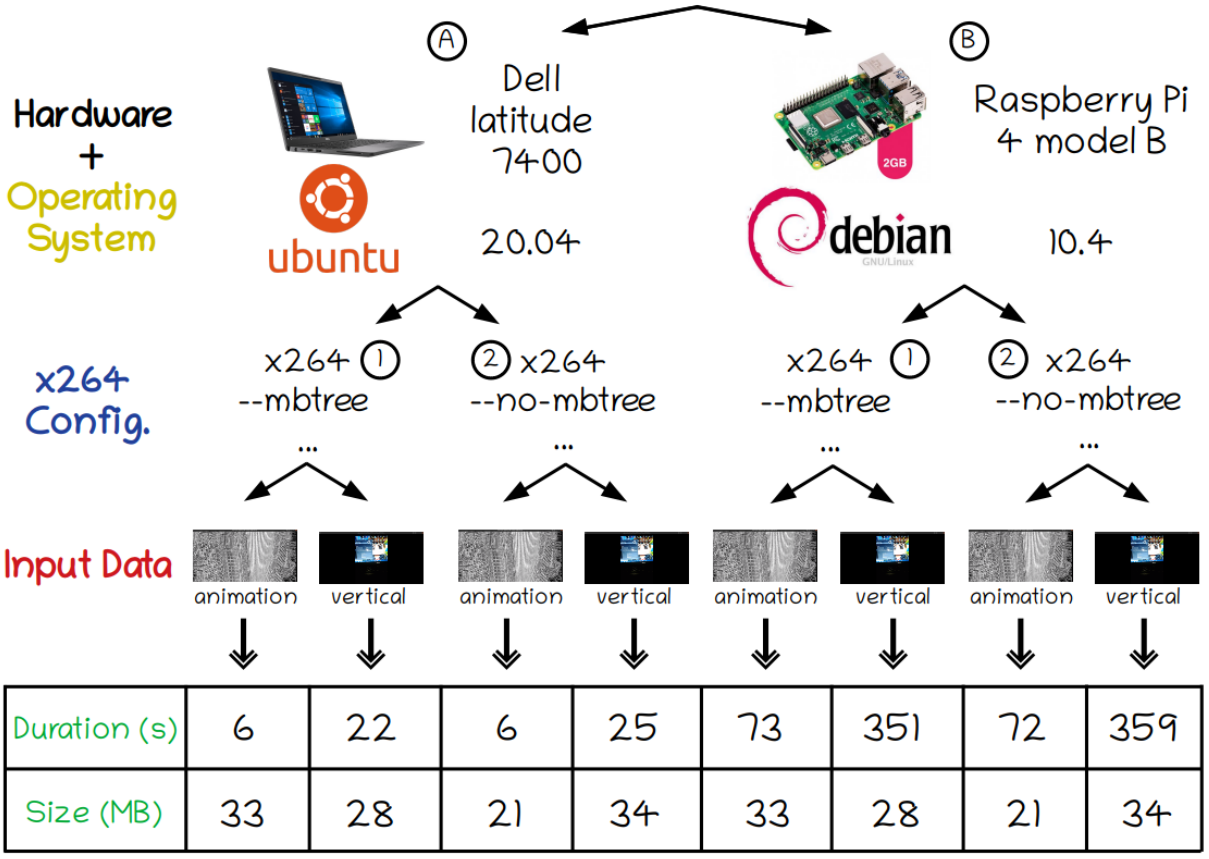
[1] ①: *--mbtree* activated, ②: *--mbtree* deactivated

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

[3] *animation* for the $Animation\_1080P-3d67.mkv$ video, *vertical* for the $VerticalVideo\_1080P-2195.mkv$ video, both extracted from the Youtube User General Content Dataset [122]

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

five challenges related to the deep variability of `x264`.



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

Figure 3.2 – Combinatorial explosion of cross-layer configurations for measuring the deep variability of `x264`

Impacts of variability layers. According to Figure 3.2, changing environment Ⓐ for environment Ⓑ would make the compression approximately 15 times longer. However, the compression durations of configurations ① and ② remain stable, and roughly proportional between Ⓐ and Ⓑ for both videos. There are more complicated cases, where the variability layers change the relative importance of configuration options. For an example, switching from configuration ① to configuration ② (deactivating the feature *--mbtree*) decreases the size of the encoded *animation* video ($33MB \searrow 21MB$), while it increases the size of the encoded *vertical* video ($28MB \nearrow 34MB$). It becomes difficult to determine the effect that *--mbtree* may have on the size of the output video. Similar results have been reported in the literature [50, 25, 75]: hardware, input data, software version (or a combination thereof) can impact performance properties of `x264`.

Testing and benchmarking. To go further that the example of Figure 3.2, we could envision a general `x264`'s performance model that has to work in environments Ⓐ and Ⓑ, but for 10 input videos and 20 boolean features of `x264`; assuming that each video is compressed in ten seconds, testing exhaustively the 21 millions of possible configurations would take about 8 months of computation. Regardless of the time needed to compute these measurements, it would require both a lot of human and computational resources to test multiple environments and a framework allowing to measure performance on a higher level than the software layer *e.g.*, automatically testing multiple workloads and input videos. In short, deep variability exacerbates the combinatorial explosion problem.

Improve documentation. For both environments and both input videos, switching from configuration 1 to configuration 2 increases the number of frames encoded per second; `x264`'s users could benefit from this information. Occasionally, requirements between layers can break the build of `x264`. For instance, manually compiling `x264` (version 0.152.2854, git commit e9a5903) on Ubuntu 18.04 LTS requires the upgrade of the Linux package *nasm* (version > 2.13),[5] unless one deactivates *--asm* during the compilation, specifying the configuration option *--disable-asm*. To the best of our knowledge, this information is neither centralised nor included in the official documentation; we have to search in forums and discussions, where developers gave the answer to this problem. Add software dependencies and the effects of a variability layer on software performance would certainly improve the documentation of `x264`.

Generalize the configuration knowledge. Between environments Ⓐ and Ⓑ, both the hardware and its operating system are changed. What would be `x264` performance on the environment Ⓒ, composed of the hardware platform used in Ⓐ and the operating system used in Ⓑ? Can we estimate them without measuring configurations in environment Ⓒ just with measurements of environments Ⓐ and Ⓑ? As long as we are unable to quantify the marginal effect of each layer, changing a single property of the environment makes us incapable of predicting `x264` performance.

Cross-layer tuning. By generalizing the measures of Figure 3.2 to other environments, a streaming website could then automatically choose an environment, and configure a server fully dedicated to the encoding of the *animation* video. In order to achieve this specialization, experts would have to remove the layers' variability, selecting the optimal configuration corresponding to this setting and hopefully increasing the overall performance of `x264`.

3.3 Challenges and Opportunities

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

Impacts of variability layers. Figure 3.1 shows an example of these layers and possible impacts on two performance (*encoding time* and *size*), for instance:

— Hardware: material part of the computer on which the software is executed. *Examples of properties*: computer or server model and brand, amount and type of Random-Access Memory available, Graphics Processing Unit frequency and memory, *etc.*

— Operating system: the operating system and dependencies that could be used by the software. *Examples of properties*: operating system distribution and version, activated Linux kernel modules, packages that are installed or embedded, *etc.*

— Software: how the software was configured and built. *Examples of properties:* any configuration option (*--no-cabac* for `x264`), how the software was compiled (compiler and version), the version of the software, *etc.*

— Input data: the raw data fed and processed by the software. *Examples of properties*: a video for `x264`, a word or a sentence for a translator, a C program for a compiler, *etc.*

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

Problem. Software practitioners may not be aware of the impacts of some layers over some performance properties. In fact, it is hard to know *a priori* which and to what extent configurations' layers do have an impact on software. For software variability researchers seeking to train predictive performance models, these variability layers can disrupt the model's training and weaken the results, preventing their generalization and making them useless to operate under different conditions. *Challenges.* Can we predict how much combination of variability layers will change non-functional properties of a software? Owing to the cost of executing cross-layer configurations', identifying the influential layers remains a challenge; their importance might vary depending on the subject system, the targeted non-functional property as well as the software configurations considered. While characterizing the impact of variability layers individually is necessary, it is not sufficient. As observed for software options, we conjecture that cross-layer configurations interact and can alter software performance and configuration option's effects. Chapters 4 and 5 (pages 50 and 66) are two existing cases empirically proving the existence of such interactions. *Opportunities.* Once their effects have been identified and quantified, the influential variability factors can be leveraged to improve software performance, while the others remain fixed and can be forgotten.

Testing and benchmarking.

Problem. Since each user applies the software on his input data, with his dependencies on his laptop, there are almost as many possible environments as users. Because of the combinatorial explosion of possible environments, it becomes time-consuming to test all possible environment performance. Moreover, testing all variability layers requires to test variability on the system level, which is not supported by existing tools. *Challenges.* 1. How to sample a good benchmark of environments? A good benchmark should be representative of the real usage of the software. If its allocated budget could be infinite, the benchmark would contain not only a wide variety of possibilities within a layer — like different brands and models of hardware platforms — but also cross-layer diversity, for instance several variants of software running in different versions of operating systems. But with a limited budget, it seems difficult to obtain a good coverage of all exiting environments without missing corner case environments. 2. Can we build a framework that tests configurable systems in multiple environments? In general, testing software configurations requires to automatically observe their properties, being functional or related to performance. Deep software variability further challenges the automation: (1) each layer may come with its own specific tooling; (2) integrating all layers together to test the software is not straightforward: for instance, instrumenting the tests of a software configuration in variants of hardware and operating systems. 3. Can we detect corner cases configurations of layers that eventually cause software's performance bugs? As exemplified in Figure 3.2, a change in the input data or hardware can cause performance differences. There are two hypotheses worth investigating. First, such differences are expected and can be explained. For example, it is normal that a given software option has more or less effect when processing a specific input. A second hypothesis is that performance differences among variability layers are in fact a manifestation of a software bug that causes an accidental and unexpected decrease in performance. For validating or refuting the "bug or feature" hypothesis, software developers and domain experts of the different layers should be involved *e.g.,* by reviewing performance results or formalizing expected

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

Improve documentation.

Problem. Assisting users in charge of configuring software is still an open issue *per se*; deep software variability amplifies this problem, since impacts of different layers are only partially reported in the documentation. Due to the lack of documentation on the relative effects of configuration options, practitioners have to directly ask experts in forums or open topics to make their software work, or just to configure it properly. *Challenges.* Can developers include deep software variability constraints and effects in the documentation? The challenge is to develop user-friendly documentation that alerts users to the effects of deep software variability, its requirements, and gives them an overview of how the software should be configured for their environment. This documentation could be used as a pretext to centralize measurements from volunteers, benefiting to the community members ; users could compare their usage of the software to others running it in similar environments. *Opportunities.* Building such a documentation would encourage the fine-grained, informed customization of configurable systems. When the effects of software options are captured, users may try to optimize and conveniently replace the default software configuration. Advanced users could even build their own variant of the software, working in their environment and fulfilling their own needs. Furthermore, different experts can participate to the customization process, each focusing on their domain of expertise with the help of a precise documentation.

Generalize the configuration knowledge.

Problem. Changing a simple property of its executing environment can alter the performance of a software. For instance, operating systems evolve frequently, introducing numerous changes in software environments; performance models have to evolve and adapt their prediction following the same rule. Unfortunately, practitioners cannot afford to train one new model per environment. *Challenges.* Is it possible to accurately predict software performance for any environment? The goal is to train a general model that aims at predicting software performance for any user's environment robust to major changes, like changing simultaneously the hardware and the input data. Another achievement is to identify the limits of the model, and include the tests as a part of the learning process when transferring performance leads to poor predictions. *Opportunities.* Being able to transfer performance from one environment to another avoids the need to build a performance model for each environment, thus saving time and energy resources. From the user's point of view, it allows to estimate the performance of a configuration without testing it, facilitating the configuration of the software for its environment.

Cross-layer tuning.

Problem. Some operating systems and packages may include options that may be over-adapted to the environment in which they run. It can lead to performance bugs when the same configuration is applied to another software environment. On the contrary, a software always configured with the default values will not benefit from the optimization coming from its environment (*e.g.,* parallelization if the default sticks to sequential tasks), wasting energy and computing time. *Challenges.* How can we tune the software for one specific execution environment? Instead of enduring deep software variability, the goal is to pre-select the right environment for the software, tuning each layer separately in such a way it improves the overall software performance. The tuning process may have a cost since it requires to know which variability layer has an effect on software, what will be its interactions with other layers, and how to configure it in symbiosis with the rest of the environment. This cost should be confronted to the underlying benefits. *Opportunities.* Since cross-layer tuning works on multiple variability layers, in comparison to a simple software tuning, we expect it to largely outperform default values configurations in terms of performance (energy consumption, execution time, *etc.*) while keeping the same level of service. Cross-layer tuning should be more efficient at accomplishing a unique task; however, it requires to restrict some flexibility and forget some variability of each layer.

3.4 Conclusion

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

In fact, deep software variability concerns all scientists that aim at publishing reproducible research works [157]; if your results are not consistent from one environment to another, your conclusions could be either weakened or invalidated. Many scientific domains are potentially impacted. Applied to ethical decisions [158], deep software variability could also have dramatic consequences; what would you do if a machine learning model deciding on the release of a prisoner changes its prediction when modifying its executing environment?

Remainder of this manuscript

The rest of this thesis addresses different aspects of the deep variability problem. Our results mainly contribute to the first, second and fourth steps proposed before, namely identifying impactful variability layers, testing and benchmarking environments and generalising performance models across environments. We did not much progresses related to the third step related to the documentation improvement, and only a little regarding the fifth, also known as cross-layer tuning.

To finish, we position the rest of the chapters *w.r.t.* to these different steps. Chapter 4 (page 50) proves that input data constitute an important variability layer that should be taken into account (first step). In the same vein, Chapter 5 (page 66) proves that compile-time options interact with run-time options, so that there exists interactions inside the software layer. We also scratch the surface of cross-layer tuning (fifth step), because we demonstrate in RQ_4 that

an inside-layer tuning is possible *i.e.,* tune both compile- and run-time to achieve a performance goal. Chapter 6 (page 84) relates to transfer of performance models across executing environments (fourth step). We also show similarities between executing environments that are directly reusable in the context of testing and benchmarking software systems (third step). Chapter 7 (page 95) relates to the transfer of performance models across software systems (fourth step). Finally, Chapter 8 (page 108) is a proof of concept of a performance model generalizing the knowledge (fourth step) over different layers (here both the software and the input data layers).

Chapter 4

Exploring the Input Sensitivity of Configurable Systems

This chapter is addressing the deep variability problem on the input data layer. It details, to the best of our knowledge, the first systematic empirical study that analyzes the interactions between input data and configuration options for different configurable systems. Through four research questions, we characterise the input sensitivity problem and explore how this can alter our understanding of software variability.

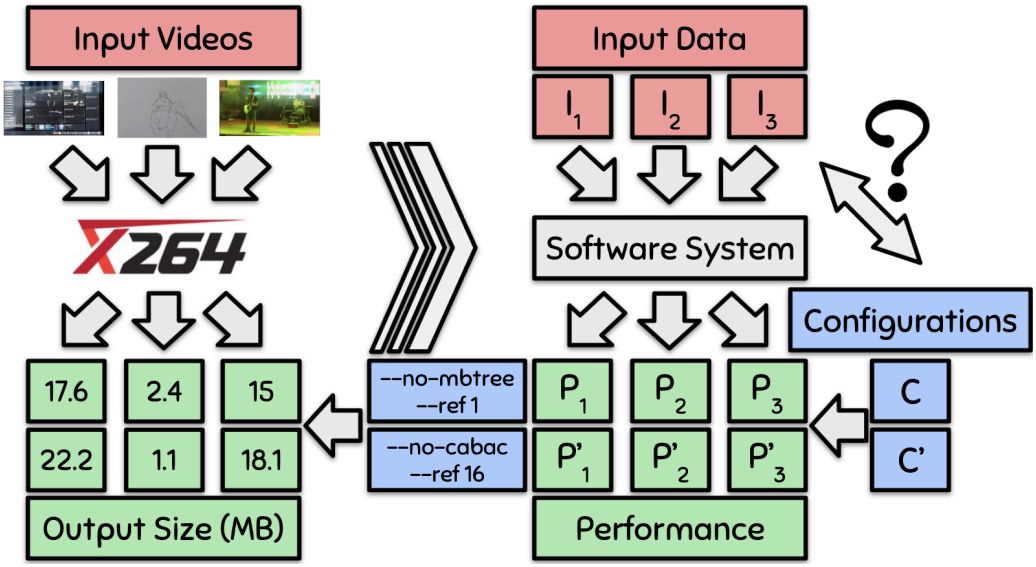


Figure 4.1 – How do inputs fed to software systems impact performance of configurations?

4.1 Problem Statement

4.1.1 Motivational Example

The `x264` encoder typifies the issue of input sensitivity. For example, Kate, an engineer working for a VOD company, wants `x264` to compress input videos to the smallest possible *size*. As illustrated in Figure 4.1, she executes `x264` with two configurations C (with options *--no-mbtree --ref* 1) and C' (with options *--no-cabac --ref* 16) on the input video I_1 and states that C is more appropriate than C' in this case. But when trying it on a second input video I_2, she draws opposite conclusions; for I_2, C' leads to a smaller output *size* than C. Now, Kate wonders what configuration to choose for other inputs, C or C'? More generally, do configuration options have the same effect on the output *size* despite a different input? Do options interact in the same

way no matter the inputs? These are crucial practical issues: the diversity of existing inputs can alter her knowledge of `x264`'s variability. If it does, Kate would have to configure `x264` as many times as there are inputs, making her work really tedious and difficult to automate for a field deployment.

4.1.2 Sensitivity to Inputs of Configurable Systems

Configuration options of software systems can have different effects on performance (*e.g.,* runtime), but so can the input data. For example, a configurable video encoder like `x264` can process many kinds of inputs (videos) in addition to offering options on how to encode. Our hypothesis is that there is an interplay between configuration options and input data: some (combinations of) options may have different effects on performance depending on input.

Researchers observed input sensitivity in multiple fields, such as SAT solvers [116, 117], compilation [118, 68], video encoding [114], data compression [119]. However, existing studies either consider only default or a limited set of configurations, a limited set of performance properties, or a limited set of inputs [115, 108, 109, 110, 111, 112, 113]. It limits some key insights about the input sensitivity of configurable systems. Valov *et al.* [26] studied the impact of hardware on software configurations, but fixed the input fed to software systems. Jamshidi *et al.* [24] explored how environment conditions (hardware, input, and software versions) impact performances of software configurations. Besides considering a limited set of inputs (*e.g.,* 3 input videos for `x264`), their study did not aim to isolate the individual effects of input data on software configurations. As a result, it is impossible to draw reliable conclusions about the specific variability factors - among hardware, inputs and versions.

4.2 The Input Dataset

We start by collecting measurements of 8 software systems processing input data and detail hereafter the methodology we follow to gather these measurements. Figure 4.2 depicts the step-by-step protocol we respect to measure performance of software systems. Each line of Table 4.1 should be read following Figure 4.2: System with Steps 1 and 2; Configurations #C with Step 3; the nature of inputs I and their number #I with Step 4; Performance P with Steps 5 and 6; Docker links a container for executing all the steps and Data the datasets containing the performance measurements. Figure 4.2 shows in beige an example with the `x264` encoder. Hereafter, we provide details for each step of the protocol.

Steps 1 & 2 - Software Systems. We consider 8 software systems. We choose them because they are open-source, well-known in various fields and already studied in the literature: `gcc` [118, 87], the compiler for gnu operating system; `ImageMagick` [159], a software system processing pictures and images; `lingeling` [160, 117], a SAT solver; `nodeJS` [161], a widely-used JavaScript execution environment; `poppler` [162, 163], a library designed to process *.pdf* files; `SQLite` [38, 24], a database manager system; `x264` [24, 75], a video encoder based on H264 specifications; `xz` [26, 80], a file system manager. We also choose these systems because they handle different types of input data, allowing us to draw conclusions as general as possible. For each software system, we use a unique private server with the same configuration running

Table 4.1 – Subject Systems. See Figure 4.2 for notations.

System	*Domain*	*Commit*	*Configs #C*	*Inputs I*	*#I*	*#M*	*Performance P*	*Docker*	*Dataset*
`gcc`	Compilation	ccb4e07	80	*.c* programs	30	2400	*size, ctime, exec*	Link	Link
`ImageMagick`	Image processing	5ee49d6	100	images	1000	100 000	*size, time*	Link	Link
`lingeling`	SAT solver	7d5db72	100	SAT formulae	351	35 100	*#confl.,#reduc.*	Link	Link
`nodeJS`	JS runtime env.	78343bb	50	*.js* scripts	1939	96 950	*#operations/s*	Link	Link
`poppler`	PDF rendering	42dde68	16	*.pdf* files	1480	23 680	*size, time*	Link	Link
`SQLite`	DBMS	53fa025	50	databases	150	7500	15 *query times q1-q15*	Link	Link
`x264`	Video encoding	e9a5903	201	videos	1397	280 797	*cpu, fps, kbs, size, time*	Link	Link
`xz`	Data compression	e7da44d	30	system files	48	1440	*size, time*	Link	Link

over the same operating system.[1] We download and compile a unique version of the system. All performance are measured with this version of the software.

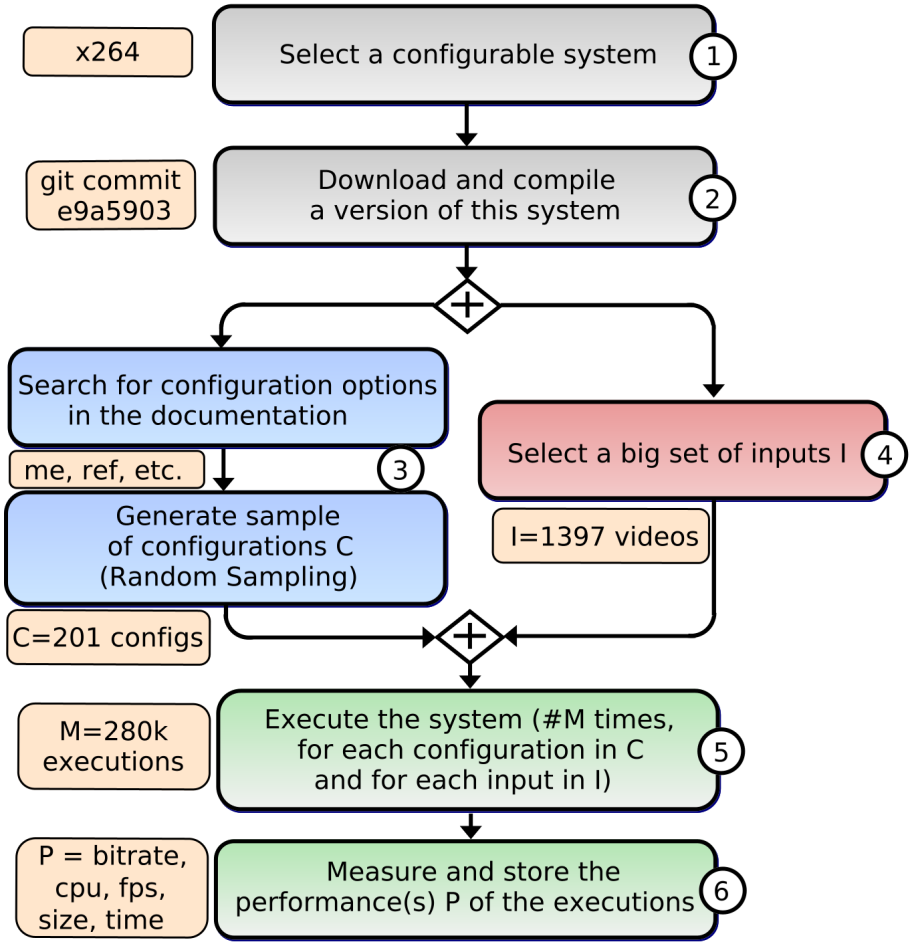


Figure 4.2 – Measuring performance - Protocol

Step 3 - Configuration options C. To select the configuration options, we read the documentation of each system and manually extract the options affecting the performance of the system. For instance, according to the documentation of `x264`, the option *--mbtree* "*can lead to large savings for very flat content*" and "*animated content should use stronger --deblock set-*

[1]The configurations of the running environments are available at: `https://github.com/llesoil/input_sensitivity/tree/master/replication/Environments.md`

tings".[2] Out of these configuration options, we then sample #C configurations by using random sampling [164]. In the previous example, after the selection of *--mbtree* and *--deblock*, the sampling step would generate multiple configurations with combinations of options' values: C_1, with *--mbtree* activated and *--deblock* set to "0:0"; C_2, with *--mbtree* deactivated and *--deblock* set to "-2:-2"; C_3, with *--mbtree* deactivated and *--deblock* set to "0:0". To ensure that each value of a software option is well represented in the final set of configurations, we statistically test the uniformity of its values. To do so, we apply a Kolmogorov-Smirnov test [165] to each option of our eight software systems.[3] In the previous example, for a boolean option like *--mbtree* that can be either activated or deactivated, a valid Kolmogorov-Smirnov test guarantees that *--mbtree* is activated in roughly 50% of the configurations. To mitigate the threat of only using random sampling, we also considered various informed configurations picked in the documentation. For instance, for `x264`, we considered the ten presets configurations recommended by the documentation.[4]

Step 4 - Inputs I. For each system, we select a different set of input data: for `gcc`, PolyBench v3.1 [166]; for `ImageMagick`, a sample of ImageNet [167] images (from 1.1 kB to 7.3*MB*); for `lingeling`, the 2018 SAT competition's benchmark [160]; for `nodeJS`, its test suite; for `poppler`, the Trent Nelson's PDF Collection [168]; for `SQLite`, a set of generated TPC-H [169] databases (from 10*MB* to 6 GB); for `x264`, the YouTube **U**ser **G**eneral **C**ontent dataset [122] of videos (from 2.7*MB* to 39.7 GB); for `xz`, the Silesia and the Canterbury corpus [170]. We choose them because these are large and freely available datasets of inputs, well-known in their field and already used by researchers and practitioners.

Steps 5 & 6 - Performance properties P. For each system, we systematically execute all the configurations of C on all the inputs of I. For the #M resulting executions, we measure as many performance properties as possible: for `gcc`, *ctime* and *exec* the times needed to compile and execute a program and the *size* of the binary; for `ImageMagick`, the *time* to apply a Gaussian blur [171] to an image and the *size* of the resulting image; for `lingeling`, the number of *conflicts* and *reductions* found in 10 seconds of execution; for `nodeJS`, the number of operations per second (*ops*) executed by the script; for `poppler`, the time needed to extract the images of the pdf, and the *size* of the images; for `SQLite`, the time needed to answer 15 different queries $q1 \rightarrow q15$; for `x264`, the *bitrate* (the average amount of data encoded per second), the *cpu* usage (percentage), the average number of frames encoded per second (*fps*), the *size* of the compressed video and the elapsed *time*; for `xz`, the *size* of the compressed file, and the *time* needed to compress it. It results in a set a tabular data, one for each input and each software system, consisting of a list of configurations with their performance property values.

Replication. To allow researchers to easily replicate the measurement process, we provide a docker container for each system (see the links in the *Docker* column of Table 4.1). We also publish the resulting datasets online (see the links in the *Data* column) and in the companion repository with replication details

[2] See the documentation of `x264` at `https://silentaperture.gitlab.io/mdbook-guide/encoding/x264.html`

[3] Options and tests results are available at: `https://github.com/llesoil/input_sensitivity/tree/master/results/others/configs/sampling.md`

[4] See `http://www.chaneru.com/Roku/HLS/X264_Settings.htm#preset`

4.3 Performance Correlations between Inputs (RQ_1)

When a developer provides a default configuration for its software system, one should ensure it will perform at best for a large panel of inputs. That is, this configuration will be near-optimal whatever the input. Hence, a hidden assumption is that two performance distributions over two different inputs are somehow related and close. In its simplest form, there could be a linear relationship between these two distributions: they simply increase or decrease with each other. **RQ_1. To what extent are the performance distributions of configurable systems changing with input data?** To answer this, we compute and compare performance distributions of different inputs. For software systems, unstable performance distributions across inputs induce that their optimal configuration change with their inputs. In particular, the default configuration should be adapted according to their input data.

Protocol

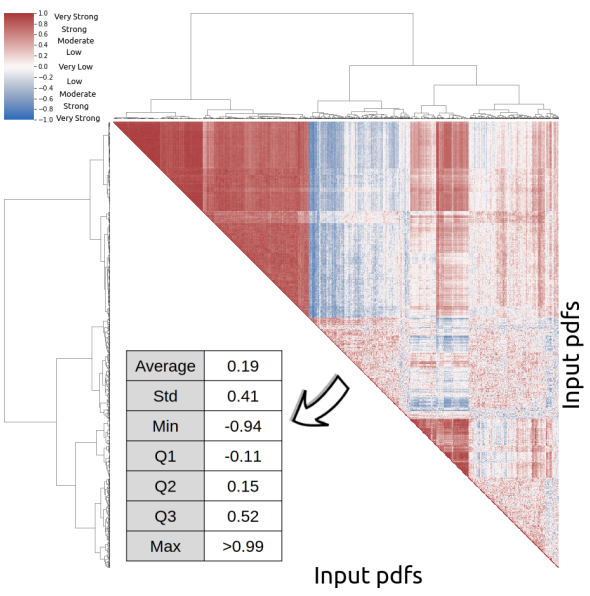
Based on the analysis of the Input Dataset (see Section 4.2), we can now answer the first research question. To check this hypothesis, we compute, analyze and compare the Spearman's rank-order correlation [172] of each couple of inputs for each system. It is appropriate in our case since all performance properties are quantitative variables measured on the same set of configurations. The correlations are considered as a measure of similarity between the configurations' performance over two inputs. We compute the related p-values: a correlation whose p-value is higher than the chosen threshold 0.05 is considered as null. We use the Evans rule [173] to interpret 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. A negative score tends to reverse the ranking of configurations. Very low or negative scores have practical implications: a good configuration for an input can very well exhibit bad performance for another input.

Evaluation

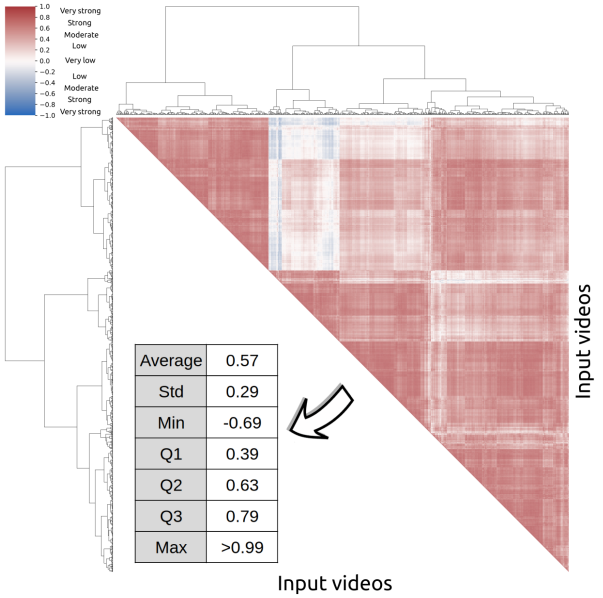
We first explain the results of RQ_1 and their consequences on the `poppler` use case *i.e.,* an extreme case of input sensitivity, and then generalize to our other software systems.

Extract images of input pdfs with `poppler`. The content of pdf files fed to `poppler` may vary; the input pdf can contain a 2-page extended abstract with plain text, a 10-page conference article with few figures or a 300-page book full of pictures. Depending on this content, extracting the images embedded in those files can be quicker or slower for the same configuration. Moreover, different configurations could be adapted for the conference paper but not for the book (or conversely), leading to different rankings of extraction *time* and thus different rank-based correlation values.

Figure 4.3a depicts the Spearman rank-order correlations of extraction *time* between pairs of input pdfs fed to `poppler`. Each square$_{(i,j)}$ represents the Spearman correlation between the *time* needed to extract the images of pdfs i and 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 their distribution. Results suggest a



(a) `poppler`, *time*.



(b) `x264`, *bitrate*.

Figure 4.3 – Spearman rank-based correlations

positive correlation (see dark red cells), though there are pairs of inputs with lower (see white cells) and even negative (see dark blue cells) correlations. More than a quarter of the correlations between input pdfs are positive and at least moderate - third quartile Q3 greater than 0.52.

Meta-analysis. Over the 8 systems,[5] we observe different cases. There exists software systems not sensitive at all to inputs. In our experiment, `gcc`, `imagemagick` and `xz` present almost exclusively high and positive correlations between inputs *e.g.,* Q1 = 0.82 for the compressed *size* and `xz`. For these, un- or negatively-correlated inputs are an exception more than a rule. In contrast, there are software systems, namely `lingeling`, `nodeJS`, `SQLite` and `poppler`, for which performance distributions completely change and depend on input data *e.g.,* Q2 = 0.09 for `nodeJS` and *ops*, Q3 = 0.12 for `lingeling` and *conflicts*. For these, we draw similar conclusions as in the `poppler` case. In between, `x264` is only input-sensitive *w.r.t.* a performance property; it is for *bitrate* and *size* (see Figure 4.3b but not for *cpu*, *fps* and *time* *e.g.,* 0.29 as deviation for *size* against 0.08 for *time*.

RQ_1. To what extent are the performance distributions of configurable systems changing with input data? We show that: (1) depending on the inputs, the rank-based correlations of performance distribution can be high, close to zero, or even negative; (2) since configuration rankings can change with input data, the best configuration for an input will not be the best configuration for another input. The consequence is that one cannot blindly reuse a configuration prediction model across inputs and that developers should not provide to end-users a unique default configuration whatever the input is.

[5]Detailed RQ_1 results for other systems are available at: `https://github.com/llesoil/input_sensitivity/tree/master/results/RQS/RQ1/RQ1.md`

4.4 Effects of Options (RQ_2)

But configuration options influence software performance, *e.g.,* the energy *consumption* [53]. An option is called influential for a performance when its values have a strong effect on this performance [25, 134]. For example, developers might wonder whether the option they add to a configurable system has an influence on its performance. However, is an option identified as influential for some inputs still influential for other inputs? If not, it would become both tedious and time-consuming to find influential options on a per-input basis. Besides, it is unclear whether activating an option is always worth it in terms of performance; an option could improve the overall performance while reducing it for few inputs. If so, users may wonder which options to enable to improve software performance based on their input data. **RQ_2 - To what extent the effects of configuration options are consistent with input data?** In this question, we quantify how the effects and importance of software options change with input data. If this change is significant, tuning these options to optimize performance should be adapted to the current input.

Protocol

To assess the relative significance and effect of options, we use two well-known statistical methods [19], also widely used in the context of interpretable machine learning and configurable systems [21, 19, 24]. For instance, Jamshidi *et al.* [24] used similar indicators to measure the sensitivity of configurations regarding computing environment conditions (hardware, input, and software versions).

Random forest importance. The tree structure provides insights about the most essential options for prediction, because such a tree first splits *w.r.t.* options that provide the highest information gain. We use random forests [19], a vote between multiple decision trees: we can derive, from the forests trained on the inputs, estimates of the options importance. The computation of option importance is realized through the observation of the effect on random forest accuracy when randomly shuffling each predictor variable [19]. For a random forest, we consider that an option is influential if the median (on all inputs) of its option importance is greater than $\frac{1}{n_{opt}}$, where n_{opt} is the number of options considered in the dataset. This threshold represents the theoretic importance of options for a software having equally important options.

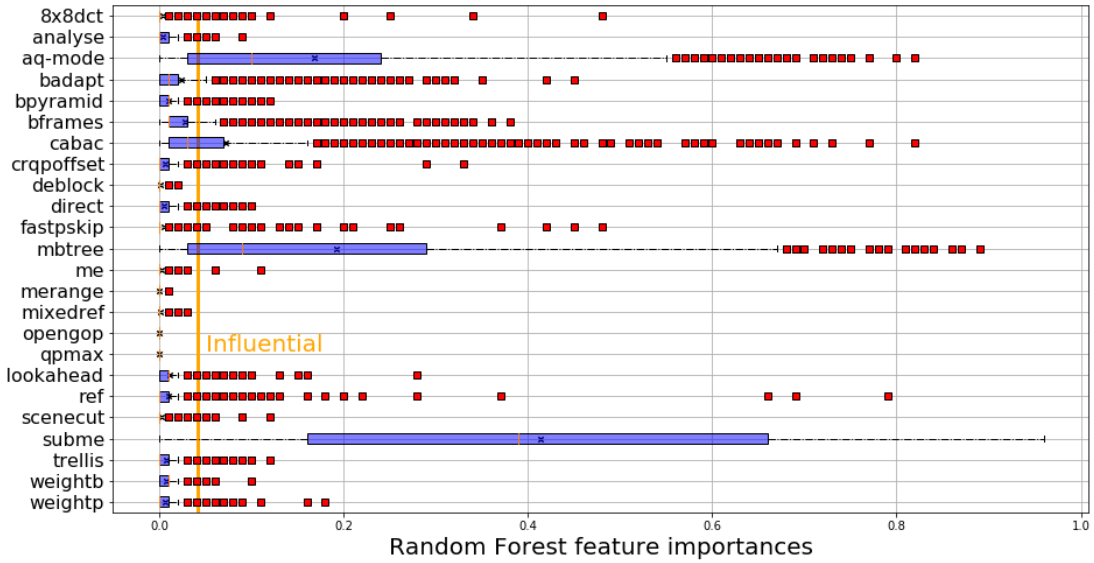
Linear regression coefficients. The coefficients of an ordinary least square regression weight the effect of configuration options. These coefficients can be positive (*resp.* negative) if a bigger (*resp.* lower) option value results in a bigger performance. Ideally, the sign of the coefficients of a given option should remain the same for all inputs: it would suggest that the effect of an option onto performance is stable. We also provide details about coefficients related to feature interactions [50, 20] in RQ_2 results.

Each algorithm is using 100% of the configurations in the training set. We also compute the Mean Absolute Percentage Error (MAPE) for all the systems, inputs and non functional properties when predicting the performance with random forests and linear regression. For random forest, we can ensure that our models are giving a good prediction, the median value of the MAPE across all inputs being systematically under 5% for each couple of software systems and

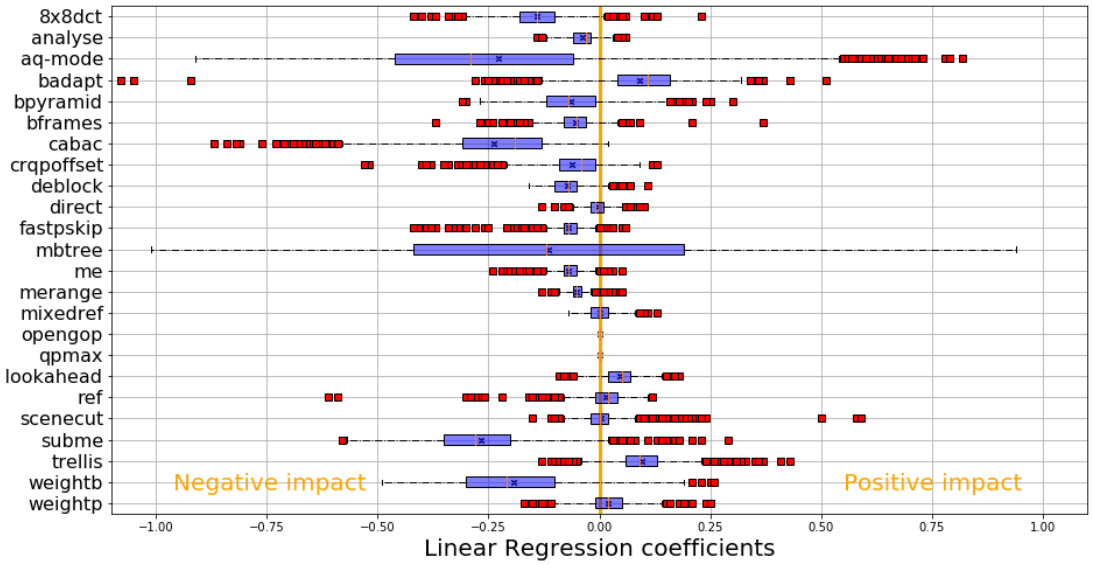
performance properties. For Linear Regression, results tend to show higher values of MAPE, suggesting that the configurations spaces are too hard to learn from for such simple models.

Evaluation

We first explain the results of RQ_2 and their concrete consequences on the *bitrate* of `x264` - an input-sensitive case, to then generalize to other software systems.



(a) Importance



(b) Effect

Figure 4.4 – Importance and effect of configuration options - `x264`, *bitrate*

Encoding input videos with ***x264.*** Figures 4.4a and 4.4b report on respectively the boxplots of configuration options' feature importance and effects when predicting `x264`'s *bitrate* for all input videos.[6] On the top graph, we displayed the boxplots of the distribution of importance

[6]Detailed RQ_2 results for other systems are available at: `https://github.com/llesoil/input_sensitivity/tree/master/results/RQS/RQ2/RQ2.md`

for each option (y-axis). On the bottom graph, we displayed the boxplots of the distribution of regression coefficients for each option (y-axis). Each red square is representing a model trained on one input, and all of them constitute the resulting distribution.

Other variants of feature importance and linear regression (permutation importance,[7] drop-column importance [8] and Shapley values [9]) have been computed to ensure the robustness of results in the companion repository. They reached similar results, which confirms our conclusions with the chosen indicators.

Three options are strongly influential for a majority of videos on Figure 4.4a: *--subme*, *--mbtree* and *--aq-mode*, but their importance can differ depending on input videos: for instance, the importance of *--subme* is 0.83 for video #1365 and only 0.01 for video #40. Because influential options vary with inputs, performance models and approaches based on *feature selection* [19] such as performance-influence model [13, 79] may not generalize well to all input videos.

Most of the options have positive and negative coefficients on Figure 4.4b; thus, the specific effects of options heavily depend on input videos. It is also true for influential options: *--mbtree* can have positive and negative (influential) effects on the *bitrate* *i.e.,* activating *--mbtree* may be worth only for few input videos. The consequence is that tuning the options of a software system should be adapted to the current input, and not done once for all the inputs.

Meta-analysis. For `gcc`, `imagemagick` and `xz`, the importance are quite stable. As an extreme case of stability, the importance of the compressed *size* for `xz` are exactly the same, except for two inputs. For these systems, the coefficients of linear regression mostly keep the same sign across inputs *i.e.,* the effects of options do not change with inputs. For input-sensitive software systems, we always observe high variations of options' effects (`lingeling`, `poppler` or `SQLite`), sometimes coupled to high variations of options' importance (`nodeJS`). For instance, the option *--format* for `poppler` can have an importance of 0 or 1 depending on the input. For all software systems, there exists at least one performance property whose effects are not stable for all inputs *e.g.,* one input with negative coefficient and another with a positive coefficient. For `x264`, it depends on the performance property; for *cpu*, *fps* and *time*, the effect of influential options are stable for all inputs, while for the *bitrate* and the *size*, we can draw the conclusions previously presented.

RQ$_2$. To what extent the effects of configuration options are consistent with input data? Two lessons learned: (1) the importance of software options changes with input data, implying that an option can be influential only for few input data, but not for the rest of the inputs; (2) the effect of software options on performance properties vary with input data. An option can have a positive influence for an input and at the same time a negative influence for another input. As a result, tuning the options of a software system should depend on its processed inputs.

[7] See results at `https://github.com/llesoil/input_sensitivity/blob/master/results/RQS/RQ2/RQ2_permutation.ipynb`

[8] See results at `https://github.com/llesoil/input_sensitivity/blob/master/results/RQS/RQ2/RQ2_drop.ipynb`

[9] See results at `https://github.com/llesoil/input_sensitivity/blob/master/results/RQS/RQ2/RQ2_shapley.ipynb`

4.5 Impact of Inputs on Performance (RQ_3)

RQ_1 and RQ_2 study how inputs affect (1) performance distributions and (2) the effects of different configuration options. However, the performance distributions could change in a negligible way, without affecting the software user's experience. Before concluding on the real impact of the input sensitivity, it is necessary to quantify how much this performance changes from one input to another. **RQ_3. How much performance is lost when reusing a configuration across inputs?** In particular, we estimate the loss in performance when configuring a software while ignoring the input sensitivity to inputs. To put it more positively, this loss is also the potential gain, in terms of performance, to tune a software system for its input data.

Protocol

To estimate how much we can lose, we first define two scenarios S_1 and S_2:

S_1 - *Baseline.* In this scenario, we value input sensitivity and just train a simple performance model on a target input. We choose the best configuration according to the model, configure the related software with it and execute it on the target input.

S_2 - *Ignoring input sensitivity.* In this scenario, let us pretend that we ignore the input sensitivity issue. We train a model related to a given input *i.e.,* the source input, and then predict the best configuration for this source input. If we ignore the issue of input sensitivity, we should be able to easily reuse this model for any other input, including the target input of S_1. Finally, we execute the software with the predicted configuration on the target input.

In this part, we systematically compare S_1 and S_2 in terms of performance for all inputs, all performance properties and all software systems. For S_1, we repeat the scenario ten times with different sources, uniformly chosen among other inputs and compute the average performance. For both scenarios, due to the imprecision of the learning procedure, the models can recommend sub-optimal configurations. Since this imprecision can alter the results, we consider an ideal case for both scenarios and assume that the performance models always recommend the best possible configuration.

Performance ratio. To compare S_1 and S_2, we use a performance ratio *i.e.,* the performance obtained in S_1 over the performance obtained in S_2. If the ratio is equal to 1, there is no difference between S_1 and S_2 and the input sensitivity does not exist. A ratio of 1.4 would suggest that the performance of S_1 is worth 1.4 times the performance of S_2; therefore, it is possible to gain up to $(1.4 - 1) * 100 = 40\%$ performance by choosing S_1 instead of S_2. We also report on the standard deviation of the performance ratio distribution. A standard deviation of 0 implies that we gain or lose the same proportion of performance when picking S_1 over S_2.

Evaluation

This section presents the evaluation of RQ_3.[10] Figure 4.5 presents the loss of performance (y-axis, in %) due to input sensitivity for the different software systems and their performance properties.

[10]Detailed RQ_3 results for other performance properties are available at: `https://github.com/llesoil/input_sensitivity/tree/master/results/RQS/RQ3/RQ3.md`

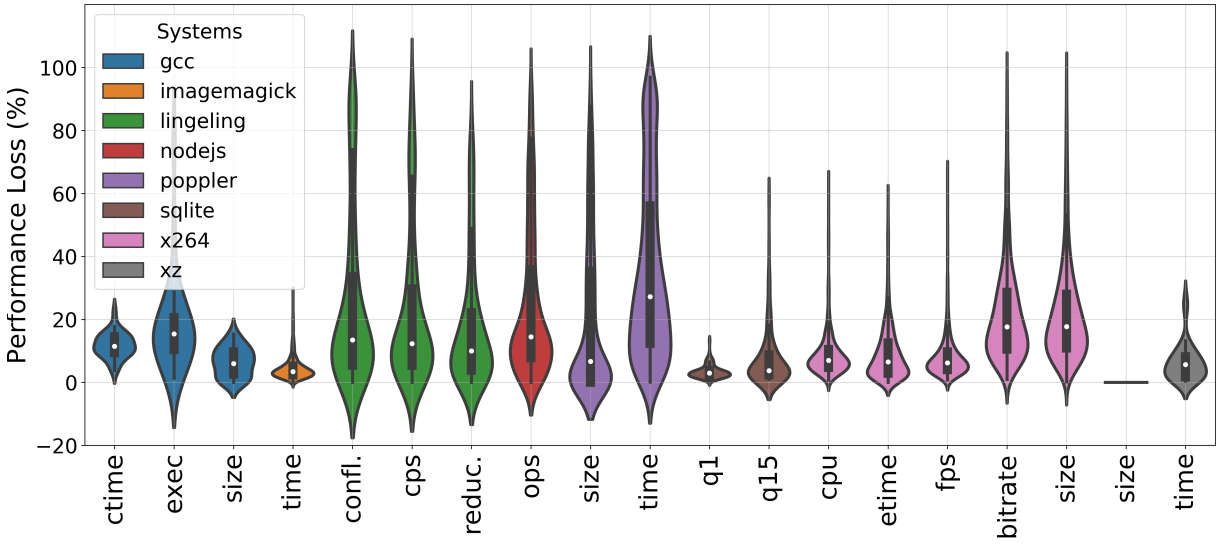


Figure 4.5 – Performance loss (%, y-axis) when ignoring input sensitivity per system and performance property (x-axis)

Key result. The average performance ratio across all the software systems is 1.38: we can expect an average drop of 38% in terms of performance when ignoring the input sensitivity.[11]

Meta-analysis. For software systems whose performance are stable across inputs (`gcc`, `xz` and `imagemagick`), there are few differences between inputs. For instance, for the output *size* of `xz`, there is no variation between scenarios S_1 (*i.e.,* using the best configuration) and S_2 (*i.e.,* reusing a the best configuration of a given input for another input): all performance ratios (*i.e.,* performance S_1 over performance S_2) are equals to 1 whatever the input.

For input-sensitive software systems (`lingeling`, `nodeJS`, `SQLite`, and `poppler`), changing the configuration can lead to a negligible change in a few cases. For instance, for the time to answer the first query *q1* with `SQLite`, the median is 1.03; in this case, `SQLite` is sensitive to inputs, but its variations of performance -less than 4%- do not justify the complexity of tuning the software. But it can also be a huge change; for `lingeling` and solved *conflicts*, the 95^{th} percentile ratio is equal to 8.05 *i.e.,* a factor of 8 between S_1 and S_2. It goes up to a ratio of 10.11 for `poppler`'s extraction *time*: there exists an input pdf for which extracting its images is ten times slower when reusing a configuration compared to the fastest.

In between, `x264` is a complex case. For its low input-sensitive performance (*e.g.,* *cpu* and *etime*), it moderately impacts the performance when reusing a configuration from one input to another - average ratios at *resp.* 1.42 and 1.43. In this case, the rankings of performance do not change a lot with inputs, but a small ranking change does make the difference in terms of performance.

On the contrary, for the input-sensitive performance (*e.g.,* the *bitrate*), there are few variations of performance: we can lose $1 - \frac{1}{1.11} \simeq 9\%$ of *bitrate* in average. In this case, it is up to the compression experts to decide; if losing up to $1 - \frac{1}{1.32} \simeq 24\%$ of *bitrate* is acceptable, then we can ignore input sensitivity. Otherwise, we should consider tuning `x264` for its input video.

[11] To compute this result, we removed `SQLite` biasing the results with its 15 performance properties

RQ$_3$. How much performance is lost when reusing a configuration across inputs? In average, ignoring the sensitivity to inputs leads to a performance drop of 38%, which suggests we cannot ignore input sensitivity. On the good side, performance can be multiplied up to a ratio of 10 if we tune other systems for their input data.

4.6 Threats to Validity

This section discusses the threats to validity related to our protocol.

Construct validity. Due to resource constraints, we did not include all the options of the configurable systems in the experimental protocol. We may have forgotten configuration options that matter when predicting the performance of our configurable systems. However, we consider features that impact the performance properties according to the documentation, which is sufficient to show the existence of the input sensitivity issue. The use of random sampling also represents a threat, in the sense that the measured configurations could not be representative of a real-world usage of the software systems. To mitigate this threat, we toke care of selecting documented and informed options, typically part of custom configurations and profiles, that are supposed to have an effect of performance. We mainly relied on documentation and guides associated to the projects. The validity of the conclusions can depend on the choice of systems under test. In the context of Chapter 7 (page 95), we conducted an additional experiment to ensure the robustness of our results for `x265`, an alternative software to `x264`. Results [12] show that the performance distributions are different from `x264` to `x265` (except for *size*) but the input sensitivity problem holds for `x265` when it is observed for `x264`.

Internal Validity. First, our results can be subject to measurement bias. We alleviated this threat by making sure only our experiment was running on the server we used to measure the performance of software systems. It has several benefits: we can guarantee we use similar hardware (both in terms of CPU and disk) for all measurements; we can control the workload of each machine (basically we force the machine to be used only by us); we can avoid networking and I/O issues by placing inputs on local folders. But it could also represent a threat: our experiments may depend on the hardware and operating system. To mitigate this, we conducted an additional experiment on `x264` over a subset of inputs to show the robustness of results whatever the hardware platforms.[13] The measurement process is launched via docker containers. If this aims at making this work reproducible, this can also alter the results of our experiment. Because of the amount of resources needed to compute all the measures, we did not repeat the process of Figure 4.2 several times per system. We consider that the large number of inputs under test overcomes this threat. Moreover, related work (*e.g.,* [75] for `x264`) has shown that inputs lead to stable performance measurements across different launches of the same configuration. Finally, the measurement process can also suffer from a lack of inputs. To limit this problem, we took relevant dataset of inputs produced and widely used in their field. For RQ_3, we consider oracles when predicting the best configurations for both scenarios, thus neglecting the imprecision of

[12] See at `https://github.com/llesoil/input_sensitivity/blob/master/results/others/x264_x265/x264_x265.ipynb`

[13] See the companion repository at `https://github.com/llesoil/input_sensitivity/blob/master/results/others/x264_hardware/x264_hardware.ipynb`

performance models: these results might change on a real-world case.

External Validity. A threat to external validity is related to the used case studies and the discussion of the results. Because we rely on specific systems and interesting performance properties, the results may be subject to these systems and properties. To reduce this bias, we selected multiple configurable systems, used for different purposes in different domains.

4.7 A Score to Quantify Input Sensitivity

In this section, we use the previous results to propose a score quantifying the level of input sensitivity of a software system.

Table 4.2 – Summary of the input sensitivity on our dataset

System	Perf.	Correlation $[C_{min}, C_{max}]$ (RQ_1)	Most infl. opt. Effect [min, max] (RQ_2)	Impact (%) $100 * (Q_2 - 1)$ (RQ_3)	**IS (0→1) Score**
gcc	*ctime*	[0.72, 0.97]	[-0.26, -0.18]	13	**0.32**
	exec	[-0.69, 1]	[-0.21, 0.64]	27	**0.92**
	size	[0.48, 1]	[-0.03, 0]	7	**0.27**
imagemagick	*time*	[-0.24, 1]	[-0.18, 0.95]	4	**0.39**
lingeling	*# conf*	[-0.9, 0.92]	[-0.79, 0.91]	15	**0.75**
	# reduc	[-0.99, 1]	[-0.79, 0.91]	10	**0.7**
nodejs	*ops*	[-0.87, 0.95]	[-1<, 0.93]	17	**0.79**
poppler	*size*	[-1, 1]	[-1<, >1]	7	**0.64**
	time	[-0.94, 1]	[-0.93, >1]	37	**0.98**
SQLite	*q1*	[-0.78, 0.87]	[-0.69, 0.58]	2	**0.45**
	q15	[-0.3, 0.94]	[-0.59, 0.6]	3	**0.37**
x264	*bitrate*	[-0.69, 1]	[-0.55, 0.28]	21	**0.84**
	cpu	[-0.31, 1]	[-0.1, 0.84]	7	**0.47**
	fps	[0.01, 1]	[-0.62, 0.23]	6	**0.37**
	size	[-0.69, 1]	[-0.58, 0.28]	21	**0.84**
	time	[0.02, 1]	[-0.17, 0.45]	7	**0.39**
xz	*size*	[0.14, 1]	[-0.02, 0.94]	0	**0.22**
	time	[-0.03, 0.97]	[-0.98, -0.15]	6	**0.37**

To support the discussions, we rely on a table that summarizes the major results of RQ_1, RQ_2 and RQ_3 by providing different indicators per software system and per performance property. Specifically, Table 4.2 reports the standard deviation of Spearman correlations (as in RQ_1), the minimal and maximal effects of the most influential option (as in RQ_2), the average relative difference of performance due to inputs (as in RQ_3). Out of the results of Table 4.2, we can make further observations. First, input sensitivity is specific to both a configurable system and a performance property. For instance, the sensitivity of `x264` configurations differs depending on whether *bitrate* or *cpu* are considered. Second, there are configurable systems for which inputs threaten the generalization of configuration knowledge, but the performance ratios remain affordable (*e.g.,* `SQLite` for *q1*). Intuitively, one needs a way to assess the level of input sensitivity per system and per performance property. We propose a metric that aggregates both indicators of RQ_1, RQ_2 and RQ_3. We define the score of *I*nput *S*ensitivity as follows:

$$IS = \frac{1}{4} * |C_{max} - C_{min}| + \frac{1}{2\alpha} * \min(Q_2 - 1, \alpha)$$

where C_{min} and C_{max} are the minimal and maximal Spearman correlations Q_2 is the median of the performance ratio distribution, and α a threshold representing the maximal proportion of variability due to inputs we can tolerate. The first part of the formula quantifies (in $[0, 0.5]$, as $|C_{max} - C_{min}|$ is in $[0, 2]$) how the input sensitivity changes the configuration knowledge (RQ_1 and RQ_2). For instance, a textbook case of software system with no input sensitivity would have only performance correlations of 1, leading to a first part equal to $\frac{1}{4} * |1 - 1| = 0$. But if the correlations are completely opposite between different inputs, this first part would be equal to $\frac{1}{4} * |1 - (-1)| = \frac{2}{4} = 0.5$. The second part quantifies (in $[0, 0.5]$) the impact of input sensitivity (RQ_3) in the actual performance. For instance, a software system with no impact of input sensitivity would have only performance ratios equals to 1, leading to $Q_2 = 1$ and $\frac{1}{2\alpha} * \min(Q_2 - 1, \alpha) = 0$. Conversely, for high performance ratios, $Q_2 - 1 >> \alpha$, $\min(Q_2 - 1, \alpha) = \alpha$ and $\frac{1}{2\alpha} * \min(Q_2 - 1, \alpha) = \frac{\alpha}{2\alpha} = 0.5$ IS thus varies between 0 (no input sensitivity) and 1 (high input sensitivity). We compute IS for each couple of systems and performance properties of our dataset, with α fixed at 25% (see Table 4.2). Empirical evidences show that IS values are robust and trustworthy when using the measurements of 15 inputs or more.[14]

IS scores are reported in Table 4.2 as follows: systems and performance properties with scores higher than 0.5 as input-sensitive (lightgray), and those with IS greater than 0.8 as highly input-sensitive (gray). IS scores highlight the input sensitive cases *e.g.,* 0.98 for the *time* of `poppler`, 0.84 for the *bitrate* and the *size* of `x264`. Systems like `xz` or `imagemagick` exhibit low IS scores that reflect their low sensitivity to inputs. As a small validation, we also compute the IS of `x264` for input videos used in [75]. We retrieve scores of 0.31 and 0.66 for the *time* and the *size* of `x264`.[15]

4.8 Implications, Insights and Open Challenges

Our study has several implications for different tasks related to the performance of software system configurations. For each task, we systematically discuss the key insights and open problems brought by our results and not addressed in the state of the art.

Tuning configurable systems. Numerous works aim to find optimal configurations of a configurable system. *Key insights.* Our empirical results show that the best configuration can be differently ranked (see RQ_1) depending on an input. The tuning cannot be reused as such, but should be redone or adapted whenever a system processes a new input. Another key result is that it is worth taking input into account when tuning: relatively high performance gains can be obtained (see RQ_3). *Open challenges.* The main challenge is thus to deliver algorithms and practical tools capable of tuning the performance of a system whatever the input. A related issue is to minimize the cost of tuning. For instance, tuning from scratch - each time a new input is fed to a software system - seems impractical since too costly.

Performance prediction of configurable systems. Numerous works aim to predict the performance of an arbitrary configuration. *Key insights.* Looking at indicators of RQ_1 and RQ_2,

[14] See `https://github.com/llesoil/input_sensitivity/tree/master/results/RQS/RQ5/RQ5-evolution.ipynb`
[15] See `https://github.com/llesoil/input_sensitivity/tree/master/results/RQS/RQ5/RQ5-other_ref.ipynb`

inputs can threaten the generalization of configuration knowledge. That is, a performance prediction model trained out of one input can be highly inaccurate for many other inputs. *Open challenges.* The ability to transfer configuration knowledge across inputs is a critical issue. Transfer learning techniques have been explored, but mostly for hardware or version changes [26, 138] and not for inputs' changes. Such techniques require measuring several configurations each time an input is targeted. It also requires training performance models that can be reused. We further explore this direction in Chapter 8 (page 108). Owing to the huge space of possible inputs, this computational cost can be a barrier if systematically applied.

Understanding of configurable systems. Understanding the effects of options and their interactions is hard for developers and users yet crucial for maintaining, debugging or configuring a software system. Some works (*e.g.,* [79]) have proposed to build performance models that are interpretable and capable of communicating the influence of individual options on performance. *Key insights.* Our empirical results show that performance models, options and their interactions are sensitive to inputs (see indicators of RQ_2). To concretely illustrate this, we present a minimal example using SPLConqueror [72] a tool to synthesize interpretable models. We trained two performance models predicting the encoding *sizes* of two different input videos fed to `x264`. Unfortunately, the two related models do not share any common (interaction of) option.[16] Let us be clear: the fault lies not with SPLConqueror, but with the fact that a model simply does not generalize to any input. *Open challenges.* Hence, a first open issue is to communicate when and how options interact with input data. The properties of the input can be exploited, but they must be understandable to developers and users. Another challenge is to identify a minimal set of representative inputs in such a way interpretable performance models can be learnt out of observations of configurable systems. We give insights to reduce the cost of measuring the joint space of inputs and configurations in Chapter 6 (page 84).

Effectiveness of sampling and learning strategies. Measuring a few configurations (a sample) to learn and predict the performance of any configurations has been subject to intensive research. The problem is to sample a small and representative set of configurations that leads to a good accuracy. *Key insights.* A key observation of RQ_2 is that the importance of options can vary across inputs. Therefore, sampling strategies that prioritize or neglect some options may miss important observations if the specifics of inputs are not considered. We thus warn researchers that the effectiveness of sampling strategies for a given configurable system can be biased by the inputs and the performance property used. *Open challenges.* Pereira *et al.* [75] showed that some sampling strategies are more or less effective depending on the 19 videos and 2 performance properties of `x264`. Kaltenecker *et al.* [78] empirically showed that there is no one-size-fits-all solution when choosing a sampling strategy together with a learning technique. We suspect that input sensitivity further exacerbates the phenomenon. Using our dataset, we are seeing two opportunities for researchers: (1) assessing state-of-the-art sampling strategies; (2) designing input-aware sampling strategies *i.e.,* cost-effective for any input.

Detecting input sensitivity. Practitioners and scientists should have the means to determine whether a software under study is input-sensitive *w.r.t.* the performance property of interest. *Key insights.* We propose several indicators (as part of RQ_1, RQ_2, and RQ_3) as well

[16] See the performance models for the first and the second input videos.

as IS a simple, aggregated score to quantify the level of input sensitivity. Such metrics can be leveraged to take inform decisions as part of the tasks previously discussed. *Open challenges.* Detecting input sensitivity has a computational cost. Selecting the right subset of configurations and input data is thus a key issue. Our empirical experiments suggest that a limited number of inputs can be used to quantify input sensitivity. To evaluate the input sensitivity score IS, considering around 15 inputs leads to a trustworthy (and green?) score, that can be reasonably cheap to compute. We do not need to spend as many resources as in this chapter in each research paper subject to input sensitivity. Other indicators and metrics can also be proposed to quantify sensitivity to inputs. Our study is the first to providence evidence of input sensitivity. We also share data with 1 976 025 measurements that can be analyzed and reused to consolidate configuration knowledge. However, further empirical knowledge is more than welcome to understand the significance of input sensitivity on other software systems and performance properties.

4.9 Conclusion

We conducted a large study over the inputs fed to 8 configurable systems that shows the significance of the input sensitivity problem on performance properties. We deliver one main message: inputs interact with configuration options in non-monotonous ways, thus making it difficult to (automatically) configure a system. By ignoring it, the input sensitivity could threaten the generalisation of results in the configurable system community. To address it, any research work related to configurable systems should at least cover few different inputs as part of the experimental protocol.

Our previous analysis of the literature (see Chapter 2) has shown that input sensitivity has been either overlooked or partially addressed on specific domains. We have pointed out several open problems to consider related to tuning, prediction, understanding, and testing of configurable systems. These results have been retrieved and confirmed by Mühlbauer *et al.* in a very recent publication [174]. In light of the results of this chapter, we encourage researchers to confront existing methods and explore future ideas with our dataset. It is an open challenge to solve the issue of input sensitivity when predicting, tuning, understanding, or testing configurable systems. In particular, a direct follow-up work aim at adapting the current practice of performance models to overcome input sensitivity and train models robust to the change of input data (see Chapter 8 in page 108).

Reproducible Science

Code and data has been made available in the companion repository at `https://github.com/llesoil/input_sensitivity`. A preprint of the related submission — still in major revision at JSS at the time being — can be consulted at `https://arxiv.org/abs/2112.07279`. For each subject system, we built a docker container to make it easy to reproduce the measurement process, kept the code of the container and put the dataset in Zenodo, see Table 4.1 (page 52).

Chapter 5

Interplay between compile-time and run-time variability

Many software projects are configurable through run-time options, but also through compile-time options. Related work has 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 using different compile-time options. For instance, is the best run-time configuration always the best *w.r.t.* the chosen compilation options? In this chapter, as a subset of the deep variability problem, we investigate the effect of compile-time options on the run-time performance distributions of software systems. In a way, this chapter details a replication of the previous chapter but with the compilation layer instead of the input data. Our results show there exists cases where the compiler layer effect is linear, which is an opportunity to generalize performance models or to tune and measure run-time 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. In terms of performance prediction, there exists cases where this kind of interactions will threaten the validity of performance models, only valid for the default compile-time option.

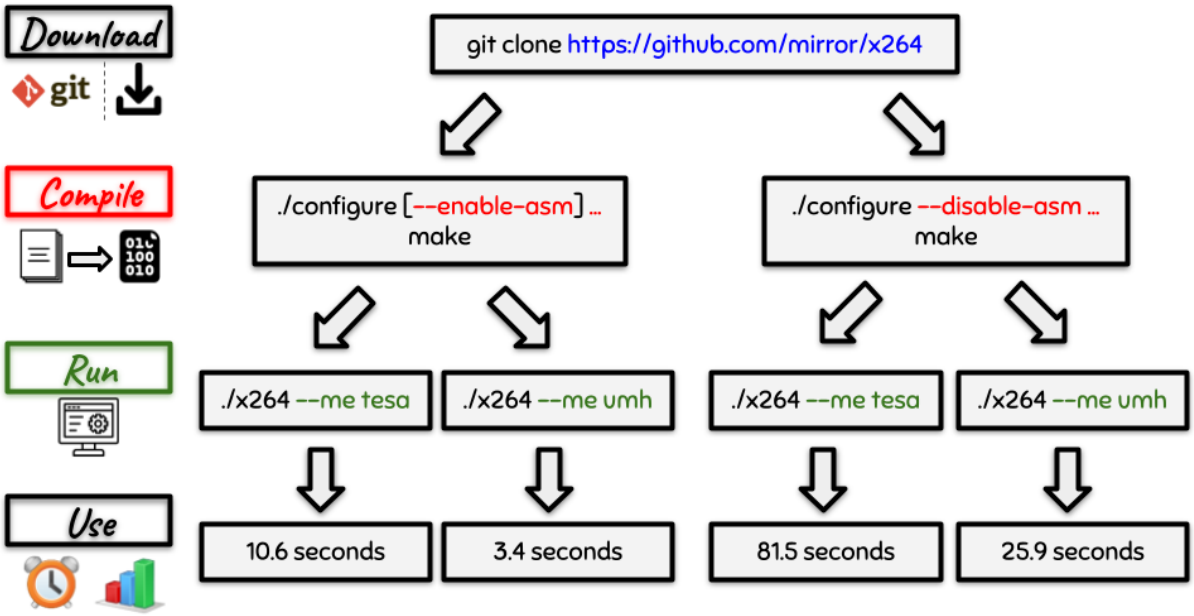
5.1 Problem Statement

5.1.1 Motivational Example

To build its own custom version of `x264`, a user has to follow a well-known protocol, as presented in Figure 5.1. The first step consists in downloading the last version of the software system sources, with the help of the git clone command — or simply by saving a zip file containing the source code with the file manager and unzipping it (see 'Download' level).

Then, to transform the source code of `x264` into a binary file that can actually be executed, one should compile the code. To do so, the user passes the command "./configure && make" in the root folder of the source code. Through this process, more parameters can be added to personalize the build of the binary file and thus configure the encoder at compile time (see 'Compile' level). Considering the `x264` encoder, all the "platform-specific assembly optimizations" are activated by default when compiling the software, there is a hidden *--enable-asm* after the ‘./configure‘ command. But a user can personalize it and disable these optimisations by explicitly adding the option *--disable-asm* as a parameter. Like these optimizations, many other configuration options can be leveraged at compile time[1]; there exists options to activate the needed functionalities

[1]See the list of `x264`'s compile-time options: `https://github.com/mstorsjo/x264/blob/master/configure`



This chapter investigates how compile-time options (red) can affect software performance and how compile-time options interact with run-time options (green).

Figure 5.1 – Cross-layer variability of `x264`

like for instance *--disable-avs*, to fit requirements related to the hardware or operating system, *etc.*

Once the binary executable of `x264` is built, it is possible to run it as a command-line ('Run' level). Once again, the execution of this command can be personalized by adding other command-line parameters changing the execution of the encoder at run time. For instance, it is also possible to configure `x264` at run time by using *--me*, a textual option specifying the type of motion estimator algorithm `x264` should select, Transformed Exhaustive (tesa), UMHexagonS (umh) or others.

The `x264` encoder is highly-configurable both at compile time and at run time. Some of these options have a suggested impact on performance ('Use' level). The idea defended in this chapter is that these two levels of configuration options can interact with each other, changing the performance values of the software system and thus make more complex the performance prediction of software systems. As an example, in the left part of Figure 5.1, when the option (at compile time) *--enable-asm* is combined to the option's value *--me* tesa (at run time), it leads to an execution of 10.6 seconds while it lasts more than a minute (81.4 seconds) for the same option at run time, but with the option *--disable-asm* at compile time.

5.1.2 Interplay between Compile-time and Run-time Options

Beyond `x264`, many projects propose to configure the software at two different levels, either at compile time or at run time. 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- and run-time options can be configured to reach specific functional and performance goals.

Compile-time options should be defined and dealt with like run-time options. When a software system is called, default run-time options' values are internally used and constitute a default run-time configuration. Similarly, it is possible to use "./configure" without setting explicit values. In this case, default values are assigned and form a default compile-time configuration. Just like run-time options, compile-time options can interact with each other and impact the performance of software systems [6]. From a terminology point of view, we consider that a compile-time configuration is an assignment of values to compile-time options.

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 performance 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 this chapter, we investigate the effect of compile-time options together with run-time 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 performance. We aim to address four research questions, each coming with their hypothesis. All the data, code and figures related to this chapter are freely available and can be found in this github repository: `https://github.com/llesoil/ctime_opt`.

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.

5.2 The Compile-time Dataset

We first create a dataset capturing the interactions between compile-time and run-time options.

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.

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

Table 5.1 – Table of considered configurable systems (see Algorithm 1 for the notations)

Measuring performance

Algorithm 1 - Measuring performance 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 5.1
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
```

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

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 $\mathcal{C}$, 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 performance 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 5.1. We keep the same version of the system for all our experiments. If needed, we ran the scripts (like `autogen.sh` for

[2]Consult the website at `openbenchmarking.org`

`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 $\mathcal{C}$: 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 performance 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 performance $\mathcal{P}$ (in .csv files). We then use these measurements to generate the results for answering the research questions.

Hardware. To avoid introducing a bias in the experiment, we measure all performance 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.

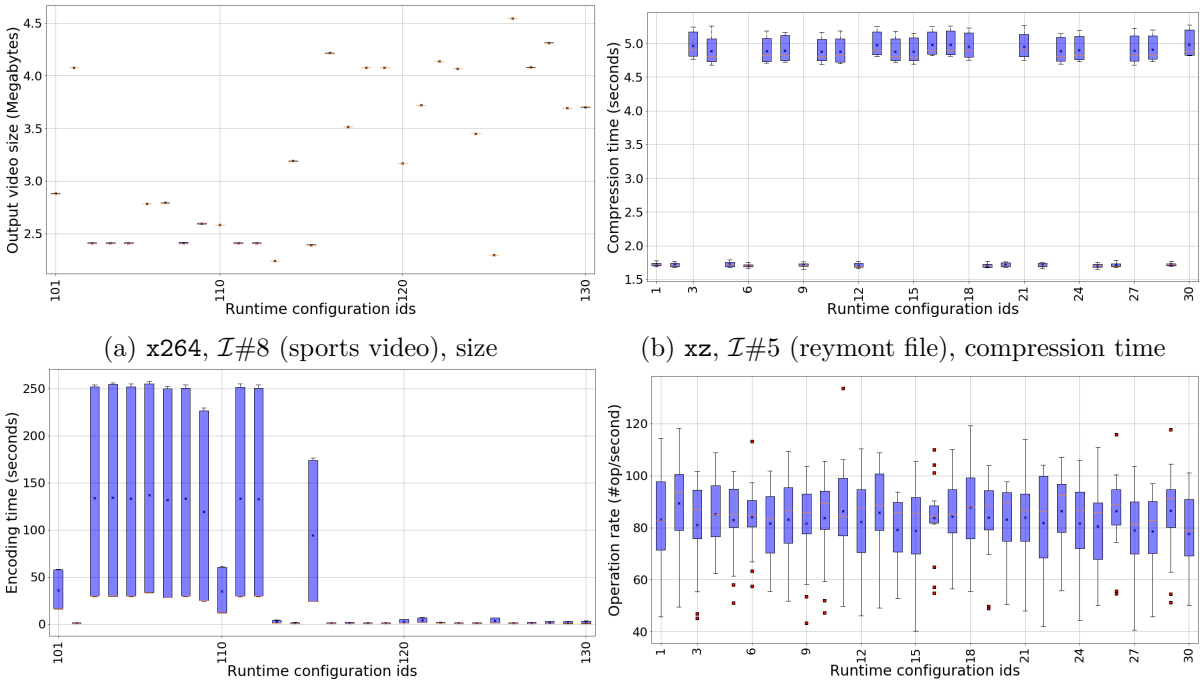
Replication. To allow researchers to easily reproduce our experiments, we provide docker containers for each configurable system. The links are listed in Table 5.1 in the "Docker" column.

5.3 Run-time Performance Distributions (RQ_1)

A hypothesis is that two performance models f_1 and f_2 over two compile-time configurations $c_1 \in \mathcal{C}$ (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, 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. **RQ_1. Do the run-time performance 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 performance of configurable systems.

Protocol

To do so, we compute and analyse the distributions of all different compile-time configurations for each run-time execution of the software system: given a input i and a run-time configuration r, we compute 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 performance. To visualize these variations, we first display the boxplots of several run-time performance and few systems in Figure 5.2. Note that for `x264` (Figures 5.2a and 5.2c), only an excerpt of 30 configurations is depicted. We then comment the values of the difference between the first and the third quartiles of a distribution, the InterQuartile Range (IQR) 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 [175] (significance level of 0.05) to distributions of run-time performance, and report on the performance leading to



(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 5.2 – 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 $\mathcal{P}$.

significant differences. This test is suited to our case since our performance distributions are quantitative, paired, not normally distributed and have unequal variances.

Evaluation

We distinguish according to different 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 5.2a (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 (3.02MB). This conclusion applies for all the sizes we measured over the 8; the size of the compressed file for `xz` (IQR=2.6B for $\mathcal{I}$#8 having an average size of 2.85MB) and the size of the image folder for `poppler` (IQR = 16B, avg = 2.36MB for $\mathcal{I}$#2). For the size of `x264`, 46% of the Wilcoxon tests do not compute because the run-time distributions were equals and all values are the same for all sizes.

The variation of the *time* depends on the considered system. Overall, for the execution time of `poppler`, it is stable - standard deviation of 29ms, for an execution of 2.6 s. For `xz`, and as depicted in Figure 5.2b, it seems to also depend on the run-time configurations. For instance, the distribution of the first run-time configuration $\mathcal{R}$#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 ($\mathcal{R}$#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`).

For the *number of operations* executed per second (`nodeJS`), the IQR values of performance distribution are high: as shown in Figure 5.2d for $\mathcal{I}\#3$, in average 19 per second, for 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 $\mathcal{C}\#1$ and $\mathcal{C}\#11$ ($p = 4.28 * 10^{-6}$) or $\mathcal{C}\#4$ and $\mathcal{C}\#7$ ($p = 1.24 * 10^{-5}$).

RQ_1. Do the run-time performance of configurable systems vary with compile-time options? 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 performance properties, like the operation *rate* for `nodeJS`, or the execution *time* for `x264`, that are sensitive to compile-time options.

5.4 Quantify Compile-time Performance Variations (RQ_2)

As an outcome of RQ_1, we found performance properties — like the operation *rate* for `nodeJS`— for which the way we compile the system significantly changes its run-time performance. 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? **RQ_2. How much performance can we gain/lose when changing the default compile-time configuration?** In other words, RQ_2 states whether it is worth changing the default compile-time configuration in terms of run-time performance. Moreover, RQ_2 tries to estimate the benefit of manually tuning the compile-time options to increase software performance.

Protocol

To quantify this gain or loss, we compute the ratios between the run-time performance of each compile-time configuration and the run-time performance of the default compile-time configuration. A ratio of 1 for a compile-time option suggests that the run-time performance of this compile-time option are always equals to the run-time performance 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 performance are worth twice the default compile-time option's performance in average. Section 5.4 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 RQ_1. We add the standard deviation of run-time performance distributions to estimate the overall variations of run-time performance 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 5.4. 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 (increasing of the number of operations executed per second for `nodeJS`) and the number of encoded fps (`x264`). As for Section 5.4, the best ratios are displayed for each input.

Evaluation

$\mathcal{S}$	nodeJS	poppler	x264		xz
$\mathcal{P}$	ops	time	fps	time	time
$\mathcal{I}$#1	0.8 ± 0.3	1.0 ± 0.0	0.6 ± 0.4	3.3 ± 2.4	1.0 ± 0.0
$\mathcal{I}$#2	0.8 ± 0.4	1.0 ± 0.0	0.6 ± 0.4	3.5 ± 2.5	1.2 ± 0.5
$\mathcal{I}$#3	0.9 ± 0.2	1.0 ± 0.0	0.6 ± 0.4	3.5 ± 2.6	1.1 ± 0.3
$\mathcal{I}$#4	1.0 ± 0.1	1.0 ±0.0	0.6 ± 0.4	3.3 ± 2.4	1.0 ± 0.0
$\mathcal{I}$#5	0.7 ± 0.4	1.0 ± 0.0	0.6 ± 0.4	3.5 ± 2.6	1.0 ± 0.0
$\mathcal{I}$#6	1.1 ± 0.2	1.0 ± 0.0	0.6 ± 0.4	3.4 ± 2.5	1.0 ± 0.0
$\mathcal{I}$#7	1.0 ± 0.0	1.0 ± 0.1	0.6 ± 0.4	3.8 ± 2.8	1.0 ± 0.0
$\mathcal{I}$#8	0.8 ± 0.4	1.0 ± 0.0	0.6 ± 0.4	3.3 ± 2.4	1.0 ± 0.0
$\mathcal{I}$#9	1.0 ± 0.0	1.0 ± 0.0			1.0 ± 0.0
$\mathcal{I}$#10	0.8 ± 0.3	1.0 ± 0.0			1.0 ± 0.1
$\mathcal{I}$#11					1.1 ± 0.2
$\mathcal{I}$#12					1.0 ± 0.0

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

As a follow-up of RQ_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 5.4. 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 performance 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 performance.

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 (like 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 performance, 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 ($*2.28$) 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`. Similarly, 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 performance. We can formulate a hypothesis with Figure 5.2c to explain these bad results: maybe few run-time configurations ($\mathcal{R}$#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.

RQ$_2$. How much performance can we gain/lose when changing the default compile-time configuration? Depending on the performance property 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 changing the default configuration. For `x264`, we can gain about 10% of execution time with the tuning of compile-time options. It is worth to tune the compile-time options to optimize these performance properties.

5.5 Interplay between Compile-time and Run-time Options (RQ_3)

RQ_1 and RQ_2 highlight few systems and performance properties for which we can increase the performance by tuning their compile-time options. Now, how to achieve this tuning process, and choose the right values to tune the performance of a software system is a problem to address. There are certainly compile-time options with negative or positive impacts on performance, typically debugging options. Hence an idea is to tune the right compile-time options to eventually select optimal run-time configurations. A hypothesis is that compile-time options interact with run-time options, which can further challenges the tuning. Depending on the relationship between performance distributions (RQ_1), the tuning strategy of compile-time options may differ. Then, **RQ$_3$. 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 RQ_1 states whether the compile-time options change performance, but the compilation could just change the scale of the distribution without really interacting with the run-time options.

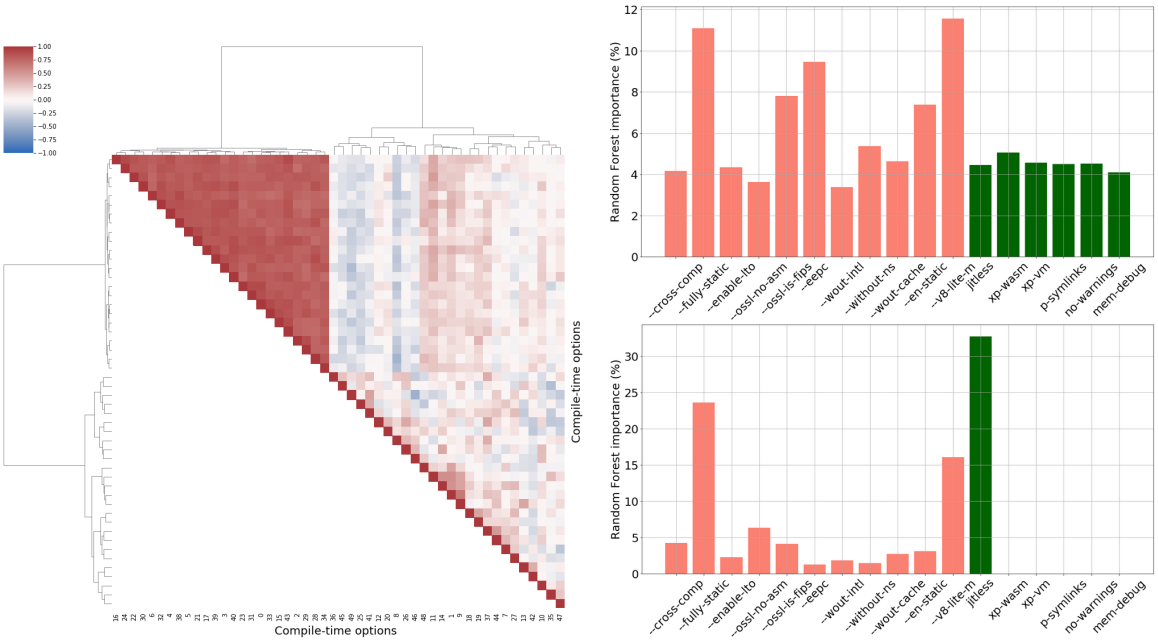
Protocol

To discover such interactions, we compute the Spearman correlations [172] 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 performance. Two uncorrelated run-time performance distributions with different compile-time options suggests that compile-time options change the rankings of run-time performance. It would and empirically show the interplay between compile-time and run-time options. We depict a correlogram in Figure 5.3a. Each square$_{(i,j)}$ represents the Spearman correlation between the run-time performance of the compile-time configurations $\mathcal{C}$#i and $\mathcal{C}$#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). As in the last chapter, we apply the Evans rule [173] 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 [176] 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 $\mathcal{C}$ and the run-time options

$\mathcal{R}$ related to the performance $\mathcal{P}$ are used as predicting variables in this model. We then report the feature importance [177, 19, 178] for the different options (run-time or compile-time) in Figure 5.3b. Intuitively, a feature is important if shuffling its values increases the prediction error. Note that each Random Forest only predicts the performance $\mathcal{P}$ for a given input $\mathcal{I}$. The idea of this graph is to show the relative importance of the compile-time options, compared to the run-time options.

Evaluation



(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 5.3 – Illustration of the interplay between the run-time and the compile-time configurations ($\mathcal{S}$ = `nodeJS`, $\mathcal{P}$=operation rate)

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 and so 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 5.3a. 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 performance are moderately, strongly or even very strongly correlated. For instance, the compile-time option $\mathcal{C}$#16 is very strongly (0.91) correlated with $\mathcal{C}$#24 in terms of run-time performance. Similarly, compile-time options $\mathcal{C}$#40 and $\mathcal{C}$#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, $\mathcal{C}$#27 and $\mathcal{C}$#13 are uncorrelated -very low correlation of 0.01. Worse, switching from compile-time option $\mathcal{C}$#8 to $\mathcal{C}$#29 changes the rankings to such an extent that their run-time performance

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 5.3b. The feature importance 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 (the one with 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*.

RQ$_3$. Do compile-time options interact with the run-time options? `xz`, `poppler` and `x264`'s performance rankings are not sensitive to the change of compile-time options. On the other hand, `nodeJS`'s performance changes at run time with different the compile-options, showing an instance of interaction between compile-time options and run-time options.

5.6 Cross-Layer Tuning (RQ_4)

RQ_3 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 performance. **RQ$_4$. How to use these interactions to find a set of good compile-time options and tune the configurable system?** Even though it is a part for the solution brought to the deep variability problem, we kept it in this part to be consistent with the rest of the compile-time study.

Protocol

As in RQ_3, we used a Random Forest Regressor [176] 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 performance. 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 5.6. A performance ratio of 1.5 suggests that we found a configuration increasing the performance of default configuration by $1.5 - 1 = 50\%$.

Evaluation

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 $\mathcal{C}$, 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`):

	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 5.3 – Performance ratios between the best predicted configuration and the default configuration for `nodeJS` and the operation rate, for different training sizes and inputs

However, for `nodeJS`, it requires additional measurements (as shown in Section 5.6). 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 performance. 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 performance 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 performance 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%.

RQ_4. How to use these interactions to find a set of good compile-time options and tune the configurable system? If the run-time options of a software system are not sensitive to compile-time options (`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 run-time performance 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 performance distributions. We prove we can overcome this problem with a performance model using these interactions to outperform the default configuration of `nodeJS`.

5.7 Threats

Construct validity. While constructing the experimental protocol and measuring the performance 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 performance 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 performance 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 RQ_4, 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.

5.8 Discussion

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 performance 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, RQ_1 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 RQ_3, 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.

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 RQ_4). This may be of particular interest to developers responsible for compiling software into packages (like `apt` or `dnf`). 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 run-time 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` [179, 15, 180] and we show that for this system the performance is remarkably stable. `xz` considered in [80] 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 [21]. 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 without measuring their performance 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.

What about the compiler flags? 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 (including *-02* and *-03* for `gcc`) relate to run-time configurations. More generally, the variability of interpreters and virtual machines [181, 182, 183] can be considered as yet another variability layer on which we encourage researchers to perform experiments.

Impact on performance prediction. 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 $\mathcal{C}$ and a run-time configuration space $\mathcal{R}$, a performance model is a black-box function $f : \mathcal{C} \times \mathcal{R} \to \mathbb{R}$ that maps each run-time configuration $r \in \mathcal{R}$ to the performance of the system compiled with a compile-time configuration $c \in \mathcal{C}$. The construction of a performance model consists in running the system in a fixed compile-time setting $c \in \mathcal{C}$ on various configurations $r \in \mathcal{R}$, and record the resulting performance values $p = f(c, r)$. 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. The training data for learning a performance model of a system compiled with a configuration $c \in \mathcal{C}$ 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, decision trees, or random forests, 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 5.1, 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?

5.9 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. 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 performance 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.

These interactions are part of the deep variability problem explored throughout this part. We show that a combination of layers — here the compilation and run-time, sometimes combined with the input data — can interact, thus changing the performance distributions of configurable systems. These kind of complex interactions should be taken care of when designing our performance models, to ensure a robust performance prediction.

Reproducible Science

Code and data are available in the companion repository at `https://github.com/llesoil/ctime_opt`. The related submission can be consulted at `https://hal.archives-ouvertes.fr/hal-03286127/`. For each subject system, we built a docker container to make it easy to reproduce the measurement process.

Part III

Exploit Deep Variability to Extend the Lifespan of Performance Models

---

Multiple research studies predict the performance of configurable software using machine learning techniques, thus requiring large amounts of data. Since measuring all those configurations is resource-consuming, we aim at recycling the previous performance models created in other conditions. In the state-of-the-art, transfer learning serves this purpose and has been successfully applied on different hardware platforms, software versions or variants. To avoid restarting from scratch each time we need to train a new model, we propose to exploit deep variability and use the similarity between different executing environments (Chapter 6 at page 84) and software systems (Chapter 7 at page 95) to optimise the transfer. If the two systems or environments share commonalities, the idea is that we should be able to reuse what has been learnt before. It is a positive aspect of deep variability.

Chapter 6

Reuse Performance Models across Environments

To extend the lifetime of performance models, a first direction to explore is to reuse them across distinct executing environments. To do so, we aim at forming clusters of similar executing environments before applying any learning process. If the software environments are alike, we are placing ourselves in the case of a simple case of transfer learning, with few differences to actually learn and embed in the models and it makes it easier to reuse performance models across environments. The purpose of this chapter is to show the existence of such clusters of environments, homogeneous in terms of performance. We want to use what has been done in the domain of deep variability to automatically generate them, in order to prioritise the transfer of performance between similar executing environments. Also, it is a way to reduce the dimensionality of the deep variability problem, and to extract example cases that will capture the essence of deep variability.

6.1 Motivational Example

Most of the time, a commit will optimize performance properties for a vast majority of users' environments. But maybe not for all of them. Software performance is sometimes sensitive to different variability layers of its execution environment, such as the hardware, the operating system or the input data processed by the software. Existing interactions between different variability layers can modify or disturb our understanding of software variability and thus change its underlying performance [184]. Since the performance decreases in a heterogeneous way across all tests, tasks and projects,[1] it sometimes makes more complex the developer life when reviewing code or pull requests.

In this example, we consider `MongoDB`, a well-known Data Base Management System.[2] Previous work on `MongoDB` has shown that thanks to testing infrastructures like Evergreen [185], it is yet possible to identify change points [186]. These change points mark out modifications of the code made by developers that significantly alter software performance and measure their impact on the daily usage of the software. This motivational example shows that (1) change points have various effects depending on the underlying software environment but, more importantly, that (2) there exists commonalities that can be exploited across the distinct environments, so that we can identify performance profiles of software environments.

[1] See also `https://github.com/mongodb/mongo/pull/23`
[2] See `https://www.mongodb.com`

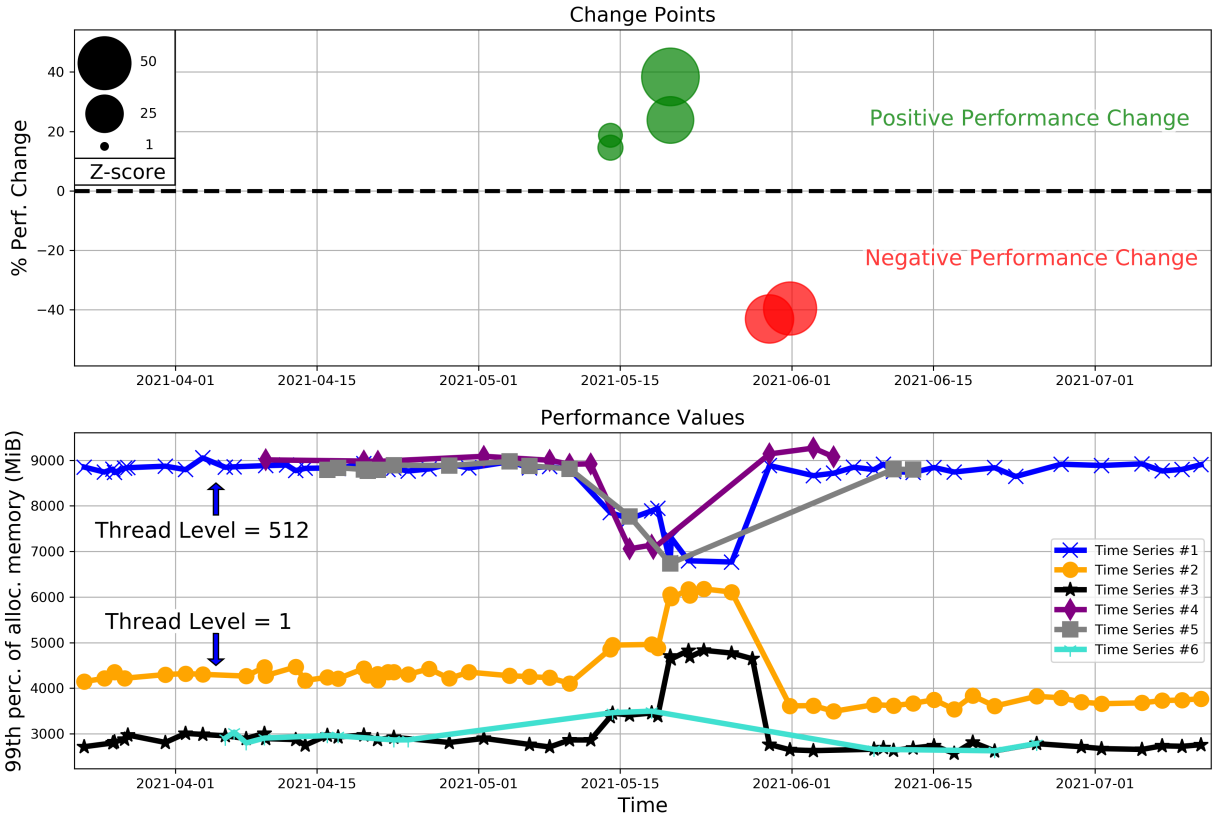


Figure 6.1 – Joint evolution of MongoDB change points (top) and performance values (bottom) - The same thread level conducts to the same performance evolution

On the top graph of Figure 6.1,[3] we display the change points detected by the test infrastructure over time, as well as their effect and impact on the performance (y-axis, percentage change). Bubble sizes represent the Z-scores [187] related to the change points. On the bottom, we plot the evolution of performance *i.e.,* Time Series (TS) for six different executing environments but considering the same project, task, variant of hardware and test for each. Like that, we are able to observe the effect of a change point on the different Time Series, each TS representing one possible environment having its own evolution. The interesting part starts in the middle of May and ends in June 2021 ; around the 15^{th} of may, the performance property suddenly drops for TS #1, TS #4 and TS #5 while it increases for TS #2, TS #3 and TS #6. This suggests that two groups of TS and the related software environments react differently to the same commit.[4] We explain this result by looking at the thread level set when executing `MongoDB`. With the thread level fixed at 1, the performance property [5] increases. With a thread level of 512, the performance drops. We would have to train at least two ML models to predict the performance of `MongoDB` in these cases. But it might be overkill to train six performance models — one per time series — since the evolution are the same for TS #1, TS #4 and TS #5. In other words, we want to gather environments related to TS #1, TS #4 and TS #5 together, environments related to TS #2, TS #3 and TS #6 together and train a performance model for each group.

[3]Dataset: *Expanded Metrics*, Project: *sys-perf*, Task: *industry benchmark wmajority*, Hardware: *Linux 3 node replSet*, Test: *csb 50 read 50 update w majority*

[4]Change point seems to refer to `https://github.com/mongodb/mongo/commit/72ed8227aa029afd554aa5809d36529ac145c3e8`

[5]Here, the performance property is the 99^{th} percentile of allocated memory in Megabytes

In this chapter, we form clusters of homogeneous executing environments sharing similar performance distributions (through space, stability across software configurations) or performance evolution (through time, stability across software versions, as in this example).

6.2 Groups Inputs across Space (RQ_1)

Following the input sensitivity problem presented in Chapter 4, this research question explores the limits of this problem and give insights on how to address it concretely. Though all inputs are different, the number of possible interactions between the software systems and the processed inputs is limited. Therefore, there might exist inputs interacting in the same way with the software, and thus having similar performance profiles. **RQ_1. What is the benefit of grouping the inputs in terms of variability?** For this question, we form, analyze and characterize different groups of inputs having similar performance distributions and show the benefits of these groups to address the input sensitivity issue.

Protocol

For mitigating input sensitivity, an idea is to group together inputs based on their performance distributions. The inputs belonging to the same group are supposed to share common properties and be processed in a similar manner by the software [68]. We perform hierarchical clustering [188] to gather inputs having similar performance profiles. This technique considers the correlations between performance distributions as a measure of similarity between inputs. Based on these values, it forms groups of inputs minimizing the intra-class variance (discrepancy of performance among a group) and maximizing the inter-class variance (discrepancy of performance between different groups of inputs). As linkage criteria, we choose the Ward method [188] since in our case, (1) single link (minimum of distance) leads to numerous tiny groups (2) centroid or average tend to split homogeneous groups of inputs and (3) complete link aggregates unbalanced groups. As a metric, we kept the Euclidian distance - used as default. We manually select the final number of groups. For each group, we then report on few key indicators summarizing the specifics of inputs' performance: the Spearman correlations between performance distributions of inputs (RQ_1), the importance and effects of options (RQ_2) as well as few properties characterizing the inputs *e.g.,* the spatial complexity of an input video or the number of lines of a *.c* program. We compare their average value in the different groups.

Evaluation

We illustrate the results of this section using the *bitrate* of `x264` when encoding input videos.[6] In Figure 6.2, we first compute the correlations between performance of all input videos, as in Figure 4.3b. Then, we perform hierarchical clustering on `x264` measurements to gather inputs having similar *bitrate* distributions and visually group correlated videos together. The resulting groups are delimited and numbered directly in the figure. For instance, the group ① is located in the top-left part of the correlogram by the triangle ①).

[6]The results for the rest of software systems can be consulted in the companion repository at `https://github.com/llesoil/input_sensitivity/blob/master/results/RQS/RQ4/groups.ipynb`

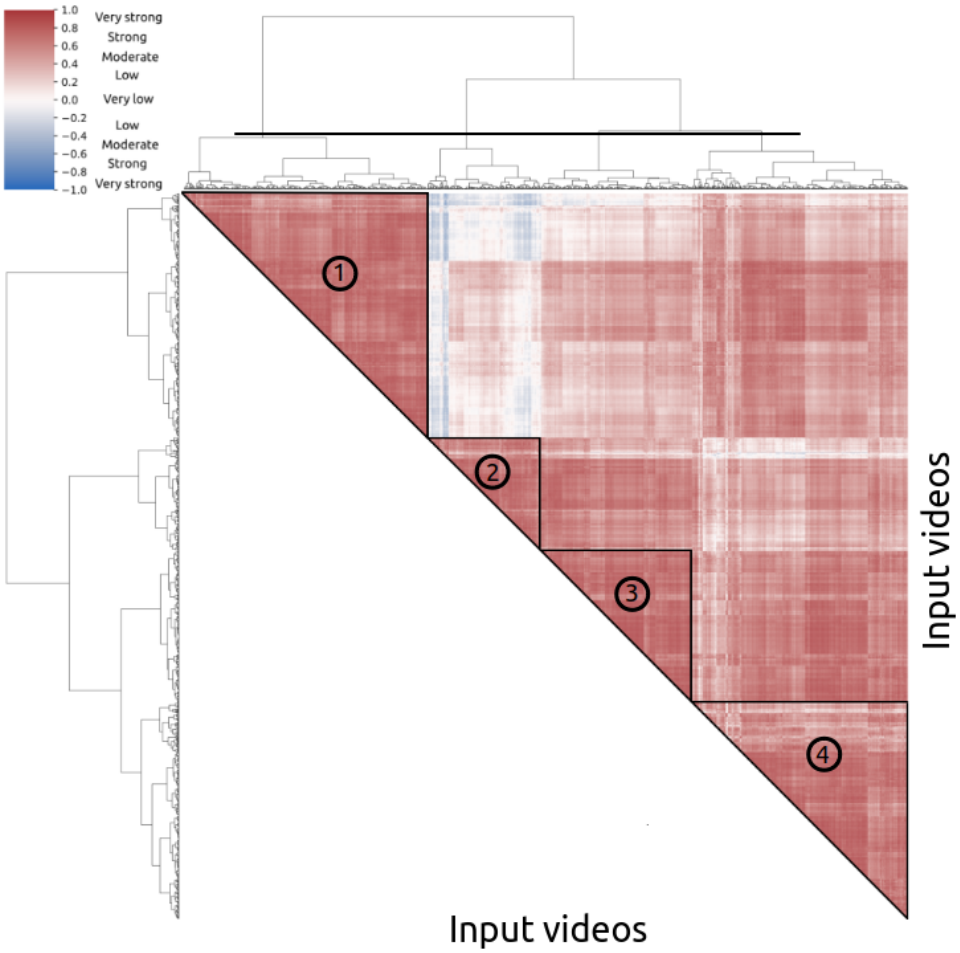


Figure 6.2 – Performance groups of input videos - `x264`, *bitrate*

Group description. In total, we isolate four groups of input videos. These groups are presented and described in Table 6.1:

— Group ① is mostly composed of moving or action videos, often picked in the sports or news categories and with high spatial and chunk complexities;

— Group ② gathers large input videos, with big resolution videos, taken for instance in the High Dynamic Range category. They typically have a low spatial complexity and a high temporal complexity;

— Group ③ is composed of "still image" videos *i.e.,* input videos with few changes of background, with low temporal and chunk complexities. A typical example of this kind of video would be a course with a fixed board, chosen in the Lecture or in the HowTo category;

— Group ④ is a group of average videos with average properties values and various contents.

Performance Correlations. In a group of inputs, performance distributions of inputs are highly correlated with each other - positively, strong or very strong. The input videos of the same group have similar *bitrate* rankings; their performance react the same way to the same configurations of `x264`. However, the group ① is uncorrelated (very low, low) or negatively correlated (moderate, strong and very strong) with the group ② - see the intersection area between triangles ① and ②. In this case, a single configuration of `x264` working for the group ① should not be reused directly on a video of the group ②. So, these groups are capturing the

Table 6.1 – Performance groups of input videos - x264, *bitrate*

Group		**① Action**	**② Big**	**③ Still image**	**④ Standard**
# Inputs		470	219	292	416
Input Properties		Spatial ++ Chunk ++	Spatial — Temporal ++ Width ++	Spatial — Temporal — Chunk —	Width - Height - Temporal -
Main Category		Sports News	HDR	Lecture HowTo	Music Vertical
Avg Correlation (*e.g.,* like in fig. 4.3)		**0.82** ± 0.11	**0.79** ± 0.14	**0.85** ± 0.09	**0.74** ± 0.17
Imp. (fig. 4.4a)	—*--mbtree*	**0.09** ± 0.09	**0.47** ± 0.2	**0.34** ± 0.22	**0.05** ± 0.07
	—*--aq-mode*	**0.27** ± 0.19	**0.13** ± 0.13	**0.04** ± 0.07	**0.15** ± 0.18
Effect (fig. 4.4b)	—*--mbtree*	**0.33** ± 0.19	**-0.68** ± 0.18	**-0.42** ± 0.15	**-0.11** ± 0.15
	—*--aq-mode*	**-0.5** ± 0.14	**0.36** ± 0.21	**-0.14** ± 0.14	**-0.29** ± 0.18

difference of performance between inputs; once in a group, input sensitivity does not represent a problem anymore.

Effect of options. Within a group, the effect and importance of options are stable and the inputs all react the same way to the same options, while they differ between the different groups. For instance, for the group ①, *--aq-mode* is influential (Imp = 0.27), while it is not for the group ③ (Imp = 0.04). Likewise, the effects of *--mbtree* vary with the group of inputs; for the group ①, activating *--mbtree* always increases the *bitrate* (Effect = +0.33), while for the groups ②, ③ and ④, it diminishes the *bitrate* (Effects = -0.68, -0.42, -0.11 respectively). Under these circumstances, configuring the software system once per group of inputs is probably a reasonable solution for tackling input sensitivity.

Reusing configurations. For the *bitrate* of x264, reusing a configuration from a source input to a target input generate a lower performance drop if the source and the target inputs are selected in the same group (*e.g.,* 13% for group 2) compared to a random selection (34% in general). If we are able to find the best configuration for one input video in a group, this configuration will be good-enough for the rest of the inputs in this group.

Classify inputs into groups. In short, grouping together inputs seems a right approach to reduce input sensitivity. However there is now a problem: we need to map a given input into a group *a priori*, without having access to all measurements. Since these four groups are consistent and share common properties, one domain expert or one machine learning model could classify these inputs *a priori* into a group without measuring their performance just by looking at the properties of the inputs.

Benefits for benchmarking. These groups allow to increase the representativeness of profiles of inputs used to test software systems, while greatly lowering the number of inputs of this set. In the companion repository, we operate on previous results to create a short but representative set of input videos dedicated to the benchmarking of x264: we reduce the dataset, initially composed of 1397 input videos [122], to a subset of 8 videos, selecting 2 cheap videos in each group of performance.[7]

Meta-analysis. We also computed the results for other systems — see Table 4.1 (page 52) — input-sensitive systems. Results tend to show similar results as for x264 and the *bitrate*. For

[7]See the resulting benchmark and its construction at: `https://github.com/llesoil/input_sensitivity/tree/master/results/RQS/RQ4/x264_bitrate.md`

instance, with `poppler`, grouping tend to gather inputs with the same influence and effect of options; for the *size*, the importance of format is influential in groups 2 and 4 but not in groups 1 and 3; for the execution *time*, in groups 1 and 2, *-jp2* has a positive effect overall while a negative effect for groups 3 and 4. When the groups discriminate inputs with different effects of options, reusing configurations lead to better results *e.g.,* the average performance loss of 49% vanishes when grouping the inputs (in the four groups, 1%-4%-12%-2% when reusing a configuration inside the group). The same applies for `Nodejs`: *--jitless* is influential for groups 2, 3 and 4 but not for group 1. Tthe average performance drop of 44% becomes 7.4%-6.7%-13.7%-9.0% in the four groups. For non input-sensitive systems, we do not observe such difference between the groups. Grouping seems to be ineffective; the same effect of options are observed; it does not change the performance loss, already low *e.g.,* for the execution `time` of `imagemagick`, 6% in general and 1%-1%-2%-6% in the groups.

RQ_1 - What is the benefit of grouping the inputs in terms of variability? Grouping inputs together is beneficial to apply in input-sensitive systems: the performance distributions, the influence of options, and the effect of options are alike between inputs of the same group. To classify inputs in one of those groups, the characteristics of inputs can be used without measuring any configuration. These groups can also be derived to create short but representative sets of inputs designed to benchmark software systems.

6.3 Group Hardware Platforms across Time (RQ_2)

According to the hardware platform of the final user, the performance of MongoDB may evolve differently. In this section, we quantify and study these differences of evolution, by answering the following research question: **RQ_2 - What is the benefit of grouping the hardware platforms in terms of evolution?**

6.3.1 Protocol

Dataset. In this question, we use the MongoDB Dataset. We refer to the MongoDB Dataset for the one used in the ICPE 2022 Data Challenge [8] for all our experiments.[9] Specifically, we use the *Legacy Performance Dataset* [186].

Metric. We first need a measure to quantify the similarities between time series. We rely on Dynamic Time Warping [189] (DTW). Unlike Euclidian distance - a point by point comparison, it is able to detect a pattern common to two time series even if this pattern does not appear at the same time for both series.

Time series pre-processing. We remove all the time series having less than two measurements. Since two time series do not necessarily have the same time stamps (*e.g.,* TS #3 and TS #6 in Figure 6.1), we only compare them during their common period of definition. For instance, to compare TS #3 and TS #6, we would remove the values of TS #3 before the starting time of TS #6 and after the last value of TS #6. When there exists a point in one time series that does

[8] See `https://icpe2022.spec.org/tracks-and-submissions/data-challenge-track/` and `https://www.daviddaly.me/2021/10/questions-on-icpe-2022-data-challenge.html` for additional explanations

[9] Download the dataset at `https://zenodo.org/record/5138516`

not have a corresponding point in the other one, we interpolate the value with a linear function based on the two values closest to the missing one. To avoid biasing the results with different scales, we standardise [190] the performance values.

Implementation. For each project, each test and each workload, we compute the DTW between the time series related to different hardware platforms. Then, we average the DTW values for each pair of variants of hardware. We consider the resulting value as a measure of similarity between the time series of two different hardware platforms. In Figure 6.3a, each column and each line represents a variant of hardware, on which MongoDB is executed; the intersection of a line and a column shows this resulting DTW value between the variant of the line and the variant of the column. If there is no time series with common project, test and workload between two variants, we leave it blank. Due to space issue, names of variants are cut down to 20 characters.[10]

Interpretation. When two times series have exactly the same evolution, their DTW is equal to zero. Then, the greater the differences between time series, the greater the DTW value. We show four pairs of time series with their measure of DTW values in Figures 6.3b to 6.3e to help interpret Figure 6.3a.[11]

6.3.2 Results

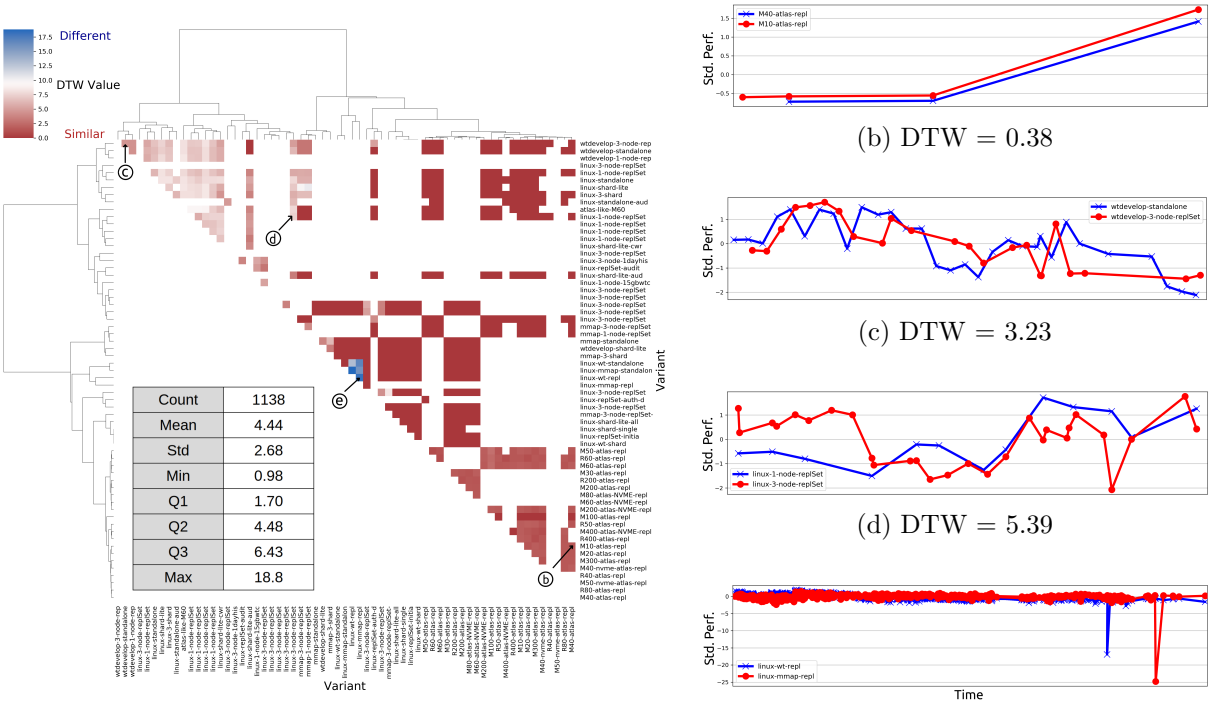
The first result is a good news *w.r.t.* stability: 25% of couples of hardware have a very low average DTW value *i.e.,* inferior to 1.70 and half of them have a low DTW value *i.e.,* inferior to 4.48. This is represented by clusters of dark red cells in Figure 6.3a. Knowing the evolution of MongoDB for one of these hardware platforms is enough to predict or estimate the evolution for all the other hardware platforms, as shown in Figure 6.3b or Figure 6.3c. In other words, we can reduce the benchmarking cost for all these variants of hardware, because they have similar performance evolution. As a MongoDB developer, it shows consistency between these hardware platforms, which is reassuring; optimizations made to MongoDB over time will generalize to most users' executing environments.

Then, we highlight of set of hardware platforms for which the evolution of performance differs over time. In Figure 6.3a, they are mostly located on the top-left corner. A good example that shows how their evolution differ is depicted in Figure 6.3d. It seems that there is no common pattern of evolution between the different variants of hardware. As a result, the hardware layer deserves to be carefully benchmarked in such a way we understand how they react to code changes.

Finally, we isolate few hardware platforms with higher DTW values. They are blue cells in Figure 6.3a and their differences of evolution are typified by Figure 6.3e. High DTW values can be explained by the presence of outliers in their performance distributions. We suspect two potential factors at the origin of these outliers: i) either we incidentally create outliers when standardising the performance or ii) this is a problem related to the performance infrastructure. Further experiments are needed to conclude about the root cause.

[10] See `https://github.com/llesoil/icpe2022/blob/main/results/fig2.png` for Figure 6.3a with full names

[11] Details about Figures 6.3b to 6.3e environments properties can be consulted at `https://github.com/llesoil/icpe2022/blob/main/Data%20Challenge.ipynb`



(a) Heatmap of average DTW between times series related to different variants of hardware

(b) DTW = 0.38

(c) DTW = 3.23

(d) DTW = 5.39

(e) DTW = 15.31

Figure 6.3 – Performance evolution of MongoDB according to hardware platforms

RQ$_2$ - What is the benefit of grouping the hardware platforms in terms of evolution? In general, there is no big evolution of performance distributions related to hardware platforms, which indicates performance models can easily be reused across different versions in the case of MongoDB. Grouping the hardware platforms allows to detect the few corner cases for which we observe unstable performance evolution.

6.4 Group Inputs across Time (RQ_3)

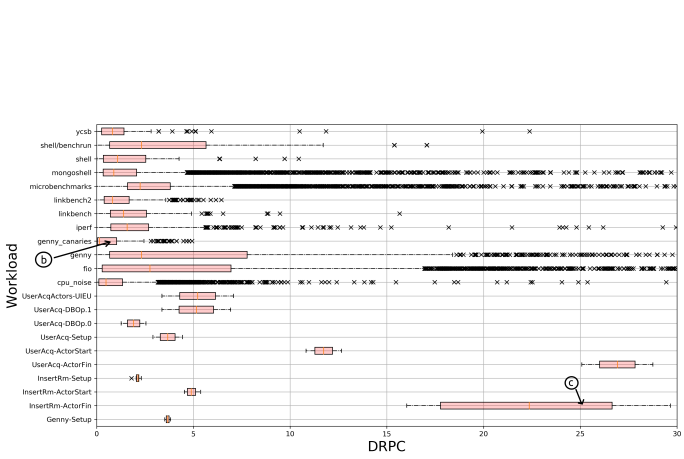
We now study the stability of performance evolution for different inputs (or workloads) fed to MongoDB. How many percents do we gain or lose between each commit? Does this value vary with the workloads? Do the workloads processed by MongoDB change its performance evolution? To address this, we answer the following research question: **RQ$_3$ - What is the benefit of grouping the inputs in terms of evolution?**

6.4.1 Protocol

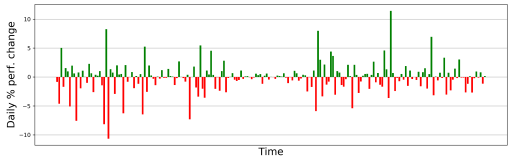
Dataset. As in the previous question, we use the MongoDB Dataset.

Metric. We define the Daily Relative Percentage Change (DRPC), a metric designed to measure the relative difference of performance (in percentage) per day:

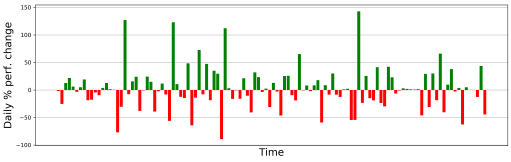
$DRPC(t) = \frac{100}{d(t,t+1)} * \frac{p(t+1)-p(t)}{p(t)}$ where $p(t)$ is the performance value at the time t and $d(t, t+1)$ is the number of days between t and $t+1$. We divide by the number of days between t and $t+1$ to avoid artificially increasing the results when there are few measurements separated by long time periods. We then average the results in absolute value for each date of measurement t, which provides the average DRPC related to a time series.



(a) DRPC distributions per workload



(b) DRPC = 1.61%, "genny_canaries" workload



(c) DRPC = 25.07%, "UserAcq-ActorFin" workload

Figure 6.4 – Performance evolution of MongoDB according to different workloads.

Time series pre-processing. We remove all the time series having less than two measurements, because the metric is impossible to compute in this case.

Implementation. As inputs, we consider the 22 workloads of the dataset; for each, we gather the time series including performance measurement on this workload and compute the average DRPC. In Figure 6.4a, we display the boxplots of DRPC distributions per workload. We also display some examples of evolution for two workloads with the detail of percentage change values in Figures 6.4b and 6.4c.[12]

Interpretation. The DRPC measures the stability of the evolution of MongoDB performance and quantifies the daily average percent change between two measurements. For a constant performance distribution, the DRPC is equal to zero. The greater the DRPC, the greater (and the more unstable) the evolution.

6.4.2 Results

Figure 6.4a illustrates how the variability of workloads affects the evolution of MongoDB. The daily percentage change of performance fluctuates with workloads; it can be very low or really high *e.g.,* median at 0.15% for the "genny_canary" workload or at 26.9% for the "UserAcq-ActorFin" workload. In Figures 6.4b and 6.4c, we plot the detail of percentage changes for these two workloads to show the difference of scale between their evolution of performance - up to a factor of 10 between their percentage changes.

Beyond the median values of the percentage changes, the results also show a difference in stability in the evolution of workload performance; while few stable workloads have a low IQR *e.g.,* "Genny-Setup" (IQR = 0.14%), others can have various ranges of percentage changes *e.g.,* "shell/benchrun" (IQR = 4.97%) or "UserAcq-ActorFin" (IQR = 8.87%). These stable workloads are reliable. For them, it is quite easy to detect an outlier, since their performance value rarely exceeds a given threshold. We guess they can be used as reference to detect a regression in the code. If such workloads observe a big decrease of performance, it is the sign that an error occurred in one of the last code modifications.

[12]Details about the environments used in Figures 6.4b and 6.4c can be consulted at `https://github.com/llesoil/icpe2022/blob/main/Data%20Challenge.ipynb`

RQ$_3$ - What is the benefit of grouping the inputs in terms of evolution? Depending on the input data, performance distribution can evolve differently over time. We can group inputs to find inputs with stable performance evolutions *i.e.,* for which performance variations are negligible across time. For these, performance models can easily be reused or adapted, without -too much- changes.

6.5 Discussion

Other Variability Layers. In this paper, we have considered two variability layers part of the deep variability problem, namely hardware and input data (or workload). Other variability layers can well be considered in the future, for example i) operating system: the Linux kernel is highly configurable and may have an impact on performance evolution of MongoDB; ii) compile-time options and flags: the way MongoDB has been built [13] can change the performance distributions. For instance, Chapter 5 (page 66) reports on preliminary evidence of complex interactions between run-time options (*e.g.,* command line parameters) and compile-time options (*e.g.,* using `./configure`) with different effects of non-functional properties of software, we could construct profile of similar compilations. Another limitation of our work is that we have not studied the interactions *between* variability layers. That is, we have focused on forming groups based on the individual effects of each variability layer. As future work, we could investigate whether hardware *together with* workload changes the performance of a software system. A hypothesis is that specific combinations of hardware and workload cause a shift in performance. The identifying of such interactions typically requires controlled measurements of pairs of workload–hardware and is arguably costly to instrument at scale.

Testing and benchmarking configurable systems. With limited budget, developers continuously test the performance of configurable systems for ensuring non-regression. Testing software configurations on a single, fixed input can hide several interesting insights related to software properties (*e.g.,* performance bugs). To reduce the cost of measurements, the ideal would be to select a set of input data, both representative of the usage of the system and cheap to measure. We believe our work can be helpful here. On the `x264` case study, for the *bitrate*, we isolate four encoding groups of input videos - see Table 6.1 in RQ_1. Within a group, the videos share common properties, and `x264` processes them in the same way *i.e.,* same performance distributions, same options' effects and a negligible impact of input sensitivity. In the companion repository, we propose to reduce the dataset of 1397 input videos [122] to a subset of 8 videos, selecting 2 cheap videos in each group of performance. Automating this grouping could drastically reduce the cost of testing. An approach applicable to any kind of input and configurable software is yet to be defined and assessed.

Transferring the knowledge. Besides testing or benchmarking, these groups of inputs or hardware platforms share commonalities. For instance, groups of input videos processed by `x264` formed in RQ_1 are consistent in terms of domain expertise, they are characterised by the same content, the same input properties, and their performance distributions are also alike. According to our results, the inputs of one group are promising candidates to apply transfer learning. In

[13] See `https://github.com/mongodb/mongo/blob/master/docs/building.md`

terms of performance prediction, it indicates that only a performance model could be trained per group of inputs, which would significantly reduce the problem of input sensitivity emphasized in Chapter 4 (page 50).

6.6 Conclusion

In this chapter, we identified clusters of software environments that behave the same way *w.r.t.* to performance. These clusters capture the essence of deep variability: the performance distributions are alike inside a cluster, but different when compared to other clusters. Identify these clusters have two major benefits. First, rather than testing and benchmarking random environments in the wild, we can use our results to pick a set of environments per cluster. By doing so, we select environments that are actually different and constitute a good sample of all existing performance profiles. In other words, we reduce the dimensionality of the deep variability problem. Second, in terms of performance prediction, identifying those clusters simplifies the problem of deep variability presented before: once in a cluster, it is easy to reuse a configuration or a performance model. Once in a group, the correlations of performance are alike, exhibiting stability across space or time in terms of performance. And if the correlations of software performance are the same, then the effects of options and the order of configurations are also similar, which implies we only have to adapt the scale of the performance distribution. So, between the different environments of a group, we are in a favorable condition for a simple transfer learning case (only linear interactions between the environment and the software stack) easy to handle [26]. It emphasizes the bright side of deep variability, allowing to reuse performance models across environments in a smart way.

Reproducible Science

Code and results for RQ_1 are available in `https://github.com/llesoil/input_sensitivity/tree/master/results/RQS/RQ4`. Code and results for the example, RQ_2 and RQ_3 are publicly available in `https://github.com/llesoil/icpe2022`, as well as the related publication at `https://hal.archives-ouvertes.fr/hal-03624309/`.

Chapter 7

Reuse Performance Models across Software Systems

To extend the lifetime of performance models, we also seek to reuse measurements of configurations and models as much as possible. This chapter is a proof of concept showing that we are able to transfer performance models from one software system to another. In Section 7.1, we form clusters of software systems sharing similarities. Then, we design a study involving two video encoders — namely `x264` and `x265` — coming from different code bases to prove the concept of transfer learning across *distinct* software systems in Section 7.2. Our results are encouraging since transfer learning outperforms traditional learning for two performance properties (out of three). We also discuss the open challenges to overcome for a more general application of transfer learning across software systems.

7.1 Identify Similar Software Systems (RQ_1)

With the multiplication of the number of concurrent organizations, standards or beliefs, we are overwhelmed by the number of concurrent systems designed for the same purpose and executing the same task. Should one use `Amazon Web Service` or `Google Cloud Platform`? `x264` or `x265`? `Tensorflow` or `PyTorch`? While each of these software systems is coming with its own original implementation, they share a common set of core features needed to configure the main aspect of the execution, which is encouraging to potentially reuse the performance models across systems. But if the intersection between two configuration spaces is too small, maybe one should not reuse the knowledge across the two related software systems. The same applies if their performance distribution are too far from each other.

This section shows that, based on open source data, we are capable of forming clusters of software systems, good candidates for the reuse of performance models. Moreover, we can derive these clusters to identify couples of software systems that are (1) from the same domain (2) share similar performance distributions and so are good candidates for a reuse of performance models. We answer the following research question: **RQ_1 - Can we identify couple of similar software systems?**

7.1.1 Protocol

In this part, we analyze the Phoronix Dataset to highlight groups of software systems sharing similarities. The construction of this dataset is detailed in Section 9.4 (page 136). In short, this dataset gathers the average performance of different hardware platforms on multiple software

systems.

We first compute a matrix of performance, with the models of hardware platform in rows and the software systems in columns. In this matrix, a value represents the average performance of the software system on this model of hardware platform. As the performance scales vary with each software system, we standardize [190] their values previous to the analysis to easily compare the different software system and avoid biasing the learning process. After this standardization, all the performance distributions of one software system are centered on 0, with a standard deviation at 1.

To compare the different performance distributions of each software systems, we then apply a Principal Component Analysis (PCA) [191] to this matrix. The purpose of this technique is to create a new representation of the data on which we can directly compare the different performance of software systems. Technically, the PCA projects the matrix in a space with a smaller dimension than the initial space representing the data. By doing so, we create common dimensions to all the software systems and make it easier to compare all the software systems together. We kept two components — two dimensions of this new space — based on the Kaiser rule [191] *i.e.,* keeping only the components with more than the average of explained variance. We then apply a K-Means algorithm on the projected data to form clusters of similar systems based on their performance values in this new space.

7.1.2 Evaluation

In Figure 7.1, we display the result of the principal component analysis of the *Phoronix* software systems, on the first two factorial planes capturing in total 78.2% of the information. We standardise the performance of the different systems so that their scale does not influence the construction of the axes. The closer two systems are to each other, the more similar are the performance distributions of the SKUs on these systems. Out of the model of K-means, we distinguish four clusters of systems.

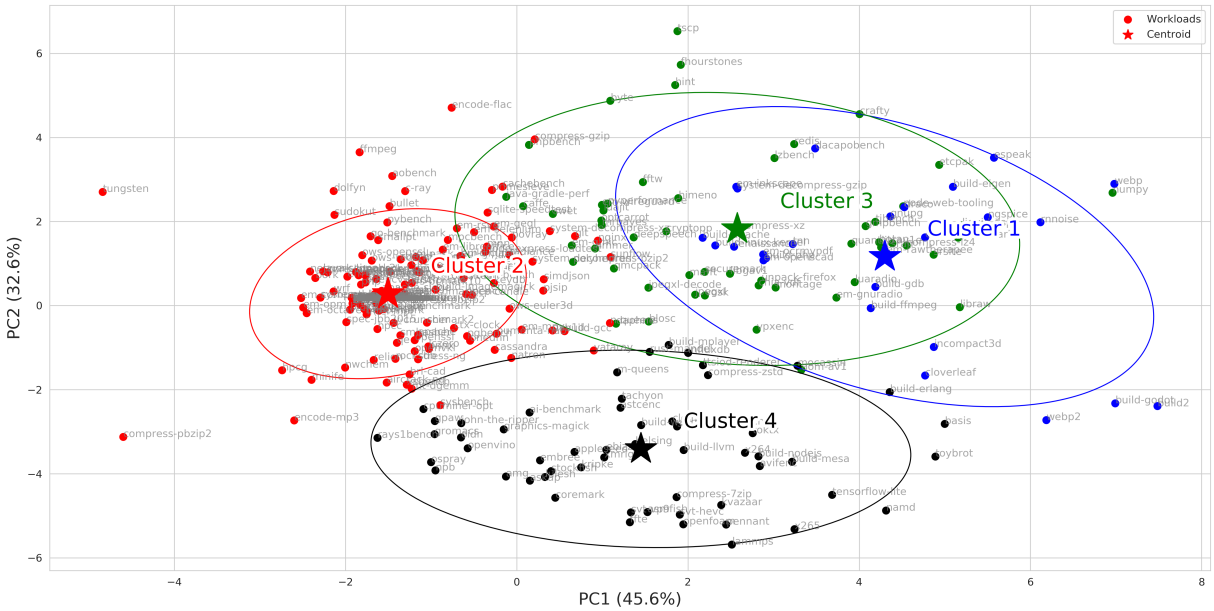


Figure 7.1 – Principal Component Analysis of 358 software systems based on the performance of 642 hardware platforms. The stars are the centroids of four clusters identified by K-means.

While it is sometimes difficult to find a logic behind each cluster, natural groupings of systems occur in this classification; for example, cluster 1 (blue) contains (among others) the systems `build-godot`, `build-apache`, `build2`, `build-gdb`, `build-ffmpeg`, `build-php`, `build-eigen` and `build-linux-kernel`, all suites involving the build of a software system. Similarly, within cluster 4 (black), we find (among others):

— `pts-oidn`, `graphics-magick`, `toybrot` and `toktx`, all processing images
— `x264`, `x265`, `vp9`, `avifenc`, `hevc` and `av1`, all video encoders
— `ttsiod-renderer`, `OSPray`, `lulesh`, `openfoam`, `lammps`, `tachyon`, `kripke`, `gromacs` and `gpaw` working on 3d simulation

These initial results suggest that measurements could be saved by testing the new processor models only on certain systems, representative of (part of) their cluster. Also, it justifies the natural choices we could have made when grouping systems together with objective indicators of performance. In particular, the couple of video encoders `x264` and `x265` used in the rest of this chapter are in the same clusters, and their two points are close to each other in the graph. So they validate the two conditions mentioned earlier, in the sense that they (1) target the same domain and (2) share similar performance distributions.

RQ_1 - Can we identify couple of similar software systems? Mining open data makes it possible to find couples of similar software systems, sharing common performance distribution across hardware platforms. It is a way to rationalise the (absence of) reuse of performance models across distinct software systems.

7.2 Transfer Learning across Distinct Software Systems: A Proof of Concept (RQ_2)

Then, we propose to apply transfer learning to *distinct* software systems performing the same task, such as compilers (*e.g.,* `gcc` and `llvm`), container managers (*e.g.,* `podman` and `docker`), *etc.* Intuitively, the model trained on one software could be used -at least partially- to train the other performance model, as depicted in Figure 7.2 for `x264` and `x265`. This section presents the first minimal example showing that under certain conditions, it is possible to transfer performance models across software systems. We also discuss the limitations of our work and highlight the open challenges to face when scaling to other software systems. Data [1] and code [2] are publicly available. We then address: **RQ_2 - Can we transfer performance across distinct software systems?** We aim at comparing the transfer learning to simple supervised learning and quantify how much we gain, both in terms of measurements and accuracy, when switching from simple learning to transfer learning. More specifically, we want to ensure that transfer learning does indeed outperform simple learning and avoid any instance of *negative transfer*.

7.2.1 Protocol

To answer the second research question, we design the following study.

[1] Dataset available at `https://zenodo.org/record/5662589`
[2] Code available at `https://github.com/llesoil/TL_cross_soft`

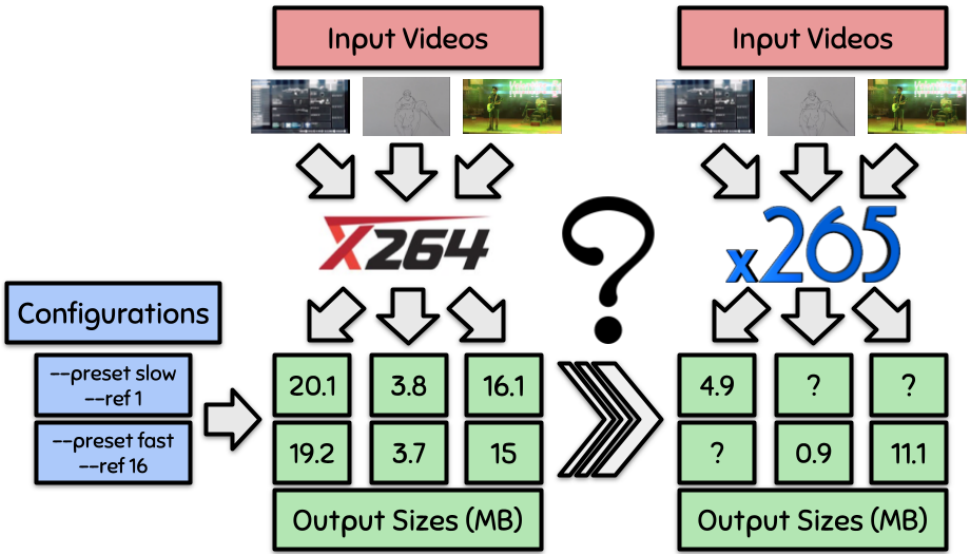


Figure 7.2 – Can `x264` be used to predict `x265` performance?

The VideoLAN Dataset

Prior to the training of performance models, we gather configurations of two video encoders, as well as their performance value. This dataset will be used in all the remainder of this section.

System	X264	x265
Language	C	C++
# Options	118	168
Average encoding Time (sec)	9	193

Figure 7.3 – Differences between `x264` and `x265`

Software Systems. We select `x264` and `x265`,[3] two video encoders, as depicted in Figure 7.3. `x264` and `x265` realize the same task, but with different compression standards (*resp.* H.264 and H.265). It has a profound impact on the visual quality algorithms internally implemented. Though `x265` has the ambition to borrow heavily from `x264`'s features, `x265` is not directly based on `x264` source code. `x265` is fully developed in C++ (and assembly) whereas `x264` is written in C. `x265` also implements novel algorithms such as CU-Tree the successor to `x264`'s macroblock-tree. Overall, `x264` and `x265` are two distinct software projects (*i.e.,* `x264` is not a version or a fork of `x265`). Importantly, both are developed by VideoLAN, which makes it easier to find similarities between them (same options, same conventions, *etc.*). From this respect, `x264` and `x265` can be seen as a favorable yet challenging case of transfer learning across systems.

Configuration Options. We search for common configuration options in their documentation.

[3]See `x264` and `x265` webpages: `https://www.videolan.org/developers/x264.html` and `https://www.videolan.org/developers/x265.html`

For instance, and according to their documentation, both `x264` and `x265` implement the features *–ref* and *–preset*. *–ref* could be set to 1, 8 or 16 while *–preset* can be fast or slow. Possible resulting configurations like (slow, 1) or (fast, 16) are accepted and valid for `x264` and `x265`. In the end, we keep 35 configuration options common to `x264` and `x265` (out of *resp.* 118 and 168).[4] We exploit these common features and make their values vary to generate a set of 3125 configurations working for both systems. We check the uniformity of the resulting distribution of options' values with a Kolmogorov-Smirnov test [165].[5]

Input Data. We select eight input videos extracted from the Youtube UGC Dataset [122], well-known in the community of video compression. For this selection, we vary the content (LiveMusic, Sports) and the resolution (360P, 480P) of videos.

Performance Properties. We then use `x264` and `x265` to transcode these eight videos from the mkv to the mp4 format. During each execution, we measure the percentage of *cpu* usage, *etime* the elapsed time in seconds and the file *size* of the resulting video in bytes.

Executing Environment. We measure all performance sequentially on a dedicated (and warmed-up) server - model Intel(R) Xeon(R) CPU D-1520 @ 2.20GHz, running Ubuntu 20.04 LTS.

Comparing performance

To get started, we first study the differences of performance between different software systems executing the same task. If they are alike, we can probably use similarities between their performance distributions when training the model. We analyze the differences between the distribution of performance properties of `x264` and `x265`. As a measure of (dis)similarity, we compute their Spearman rank-based correlation [172]. It is suited for our case since all performance properties are quantitative variables relative to the same configurations. If both encoders obtain the same rankings in terms of performance, the correlation is close to 1, and there is a good chance of getting good results with transfer learning. If they react differently to the same configurations, the correlation is close to 0 and the transfer might be challenging to achieve.

Transferring performance

As displayed in Figure 7.2, we try to transfer the performance from `x264` (*i.e.,* source software) to `x265` (*i.e.,* target software). We use Model Shift (MS), a simple and state-of-the-art transfer learning approach defined by Valov *et al.* [26].

The protocol should be read following Figure 7.4:
1. First, it trains a shifting function, mapping the performance distribution of the source on the target software's performance distribution, 2. Then, it trains a performance model on the source software, 3. Finally, it predicts the performance distribution of the source software and applies the shifting function to the predictions, in order to estimate the performance of the target software system.

We compare MS to a simple baseline acting as a control approach, training a performance model directly on the target software, without using any measurement of the source. We call this

[4]The list of selected configuration options can be consulted here: `https://anonymous.4open.science/r/TL_cross_soft-855B/replication/x26x/README.md`

[5]Results can be consulted at: `https://anonymous.4open.science/r/TL_cross_soft-855B/replication/x26x/x264_x265_options.ipynb`

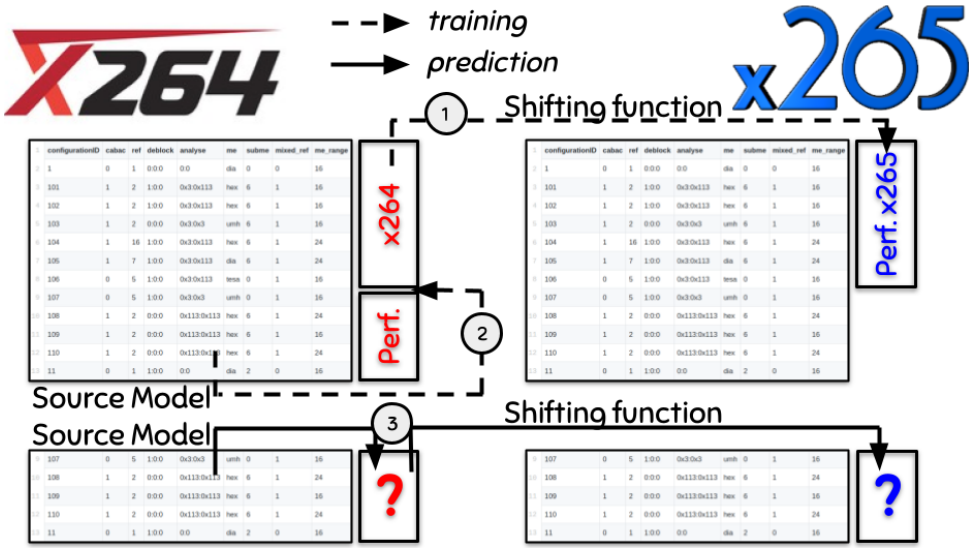


Figure 7.4 – Model Shift, a transfer learning approach

baseline *No Transfer*. For both, we used a Random Forest algorithm [192] to predict software performance, without tuning its hyperparameters. We separate the dataset of the target into training and test, varying the size of the training set. In the evaluation, we compare and display the MAPE [19] between the predicted values (*i.e.,* predicted by the approaches) and the real values (*i.e.,* measured on the test set of the target software). To reduce the variance induced by machine learning randomness, we repeated the process five times and display the average MAPE for the test sets. We rely on the python library scikit-learn [193]. [6]

7.2.2 Evaluation

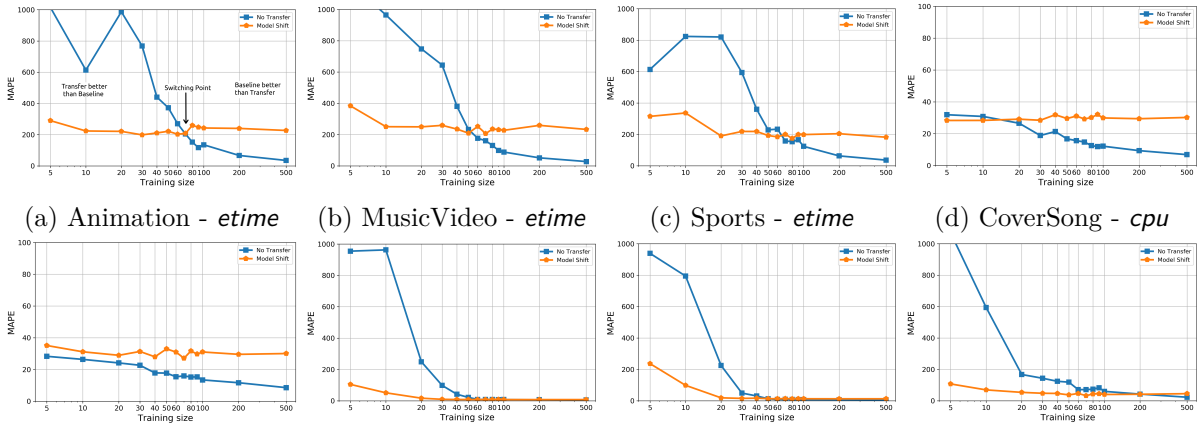
Table 7.1 – Correlations between the performance distributions of `x264` and `x265` for eight input videos

Video	*cpu*	*etime*	*size*
Animation	0.05	0.74	0.98
CoverSong	0.0	0.73	0.98
Gaming	-0.02	0.81	0.99
Lecture	0.07	0.75	0.98
LiveMusic	0.01	0.77	0.99
LyricVideo	0.14	0.73	0.96
MusicVideo	0.07	0.75	0.99
Sports	-0.0	0.78	0.98

Comparing performance

Among the three performance properties, we can distinguish three cases: between `x264` and `x265`, the *cpu* consumption has an average a correlation close to 0 (lower than 0.1), the elapsed

[6] A description of our python environment can be consulted at: `https://github.com/llesoil/TL_cross_soft/replication/requirements.txt`



(a) Animation - *etime* (b) MusicVideo - *etime* (c) Sports - *etime* (d) CoverSong - *cpu*

(e) Lecture - *cpu* (f) Gaming - *size* (g) LiveMusic - *size* (h) LyricVideo - *size*

Figure 7.5 – Mean Average Percentage Error (y-axis, lower is better) when transferring `x264` to `x265` performance depending on the training size (x-axis, log scale), for eight input videos and three performance properties

time *etime* is overall positively correlated (about 0.75), and there is almost no differences between the *sizes* of the resulting mp4 videos (correlations close to 1). Transferring *sizes* is likely to be easy and transferring *cpu* consumption will probably be too difficult. The elapsed time *etime* is in between. Out of three performance properties, two are highly correlated whatever the input video. `x264` and `x265` seems to be good candidates for transfer.

Transferring performance

In Figure 7.5, we depict the results varying for different inputs and performance properties:

— The transfer of *cpu* consumption is almost always negative. For example, after 15 configurations in the training set for the CoverSong video, in Figure 7.5d, the baseline is always more accurate than transfer learning. For the Lecture video, in Figure 7.5e, it is even always a negative transfer.

— For *etime*, the transfer is cost-effective at first, until a given training size *e.g.,* 65 for Figure 7.5c. After this threshold, the error of the baseline keeps dropping while the errors stabilise for the transfer. For the Animation video, in Figure 7.5a, and for a budget of 500 configurations, *No Transfer* decreases to a MAPE of $35 \pm 1.3\%$ and *Model Shift* stays at $220 \pm 21\%$.

— Finally, for the encoded *sizes* of videos, *Model Shift* is at least equivalent to the baseline whatever the number of configurations and the video. This can be explained by the high correlations observed in Table 7.1. At first, the transfer is really outperforming the baseline *e.g.,* in Figure 7.5h, for 5 configurations, the transfer has an error of about $106 \pm 64\%$ while the baseline amounts to $1071 \pm 556\%$.

RQ_2 - Can we transfer performance across distinct software systems? The effectiveness of the transfer varies with the performance property we consider. Overall, it is possible to outperform the *No Transfer* baseline for two performance properties (out of three), especially when the budget (*i.e.,* the number of configurations in the training set) is low. As noticed by [24], the greater the correlation between performance distributions, the more accurate the transfer. As a concrete piece of advice for developers, this correlation could be a cheap indicator to estimate *a priori* whether transfer techniques are adapted between two systems.

7.2.3 Threats to Validity

Due to the cost of measurements (58 days of system time), we did not measure performance more than once. Therefore, the performance distribution of `x264` and `x265` could change with new measurements. To address this threat, we check the results for eight different input videos; given their consistency, we are confident that similar conclusions could be drawn by reproducing the experiment. Another threat to validity is related to the use of machine learning algorithms, which leads to non-deterministic results. To mitigate this threat, we display the average result of five runs of the model. When benchmarking, we only select the configuration options common to `x264` and `x265`, ignoring a large majority of features. This represents a potential threat to validity when generalising the transfer to the whole configuration space.

7.3 Discussion

Although our experiment is encouraging to further explore this research direction, it does not cover all possible cases of transfer learning between software systems. In this section, we identify three lines of research and for each (1) we point out the limitations of our study; (2) we discuss the open challenges to overcome when transferring performance between *distinct* software systems; (3) we describe the potential approaches and solutions.

7.3.1 Find Transferable Software Systems.

Limitation. A threat to our study is the choice of `x264` and `x265`. There might be other pairs of systems for which non-functional properties are dissimilar *e.g.,* with a correlation of zero, the transfer learning might perform poorly. For `x264` and `x265` our experiments show that two non-functional properties out of three are positively and strongly correlated. *Open challenge.* We cannot guarantee the effectiveness of transfer learning for every pair of software systems. What is difficult is to know whether transfer will work for a given pair of software systems. *Possible Solution.* For now, the only reasonable assumption we can make is to choose software systems within the same domain *e.g.,* compilers like `gcc`, `llvm` or `clang`, container managers like `podman` or `docker`, learning libraries like `theano`, `pytorch` or `tensorflow`, text editors like `emacs`, `gedit` or `vim`, video encoders like `vp9`, `x264`, or `x265` Measures on how performance distributions differ (*e.g.,* with correlations) across systems can provide a first indicator on whether transferring is worth. RQ_1 also provide a way to pre-select software systems part of the same cluster, alleviating the threat of an arbitrary choice. However, the automation of the selection is far from being

obvious. We may want to filter software that has enough measurements available, that will share common options.

7.3.2 When and How to Transfer?

Limitation. The results of Section 7.2 show limits of transfer learning: after a given training size, the interest in adding noisy measures – such as source data – decreases. This learning size is a switching point, as shown by Figure 7.5a: before this point, it is preferable to use transfer learning and after this point, we should switch to simple learning. This point seems difficult to estimate *a priori i.e.,* without any measurement. This may depend on the complexity of the configuration space: the more complex the configuration space, the more configurations are needed to make an accurate prediction. And the more configurations needed to be accurate, the larger the switching point. *Open Challenge.* It is challenging to know when to apply transfer learning and when to switch to non-transfer learning. These results could also be better (or worse) with other transfer learning techniques. The challenge here is to determine the best learning approach to use for a pair of software systems. *Possible Solution.* Empirical studies comparing different approaches on representative pairs of systems would be of great help in addressing this challenge. Specialized learning algorithms, capable of handling distribution shifts across software systems, are also expected.

7.3.3 What to Transfer?

Limitation. Our choice of configuration options is another limitation of our study. We have deliberately considered a favorable case: since `x264` and `x265` have been developed by the same team, the two systems share a common set of configuration options with the same range of values.

Table 7.2 – The difficulty of aligning two configuration spaces

Problem	`x264`	`x265`	`vp9`
Features do not have the same name	*--level*	*--level-idc*	
One system's feature encapsulates the others'	*--fullrange*	*--range* 'full'	
A feature is not implemented	X	*--rc-grain*	
A feature value is not implemented	*--me* 'star'	X	
Features do not have the same default value	*--qpmax* [51]	*--qpmax* [69]	
Different requirements or feature interactions	.yuv $\Rightarrow$ *--input-res*	.yuv $\not\Rightarrow$ *--input-res*	
Feature ranges differ between source and target	*--crf* [0-51]	*--crf* [0-51]	*--crf* [0-69]

Open Challenge. In general, the configuration spaces will often be very different between the source and the target systems, making the sampling of configurations difficult in practice. The challenge is to map the configuration space of the source to the configuration space of the target. We illustrate these differences between configuration spaces with `vp9`, `x264`, and `x265` in Table 7.2[7]:

[7]One can verify our illustrations with the lists of features, available at:
`https://cinelerra-gg.org/download/CinelerraGG_Manual/VP9_parameters.html` for `vp9`,
`http://www.chaneru.com/Roku/HLS/X264_Settings.htm` for `x264`,
`https://x265.readthedocs.io/en/2.5/cli.html` for `x265`.

— The same feature is implemented in the source and the target with different names *e.g.,* *--level* for `x264` and *--level-idc* for `x265`;

— The same feature is implemented in the source and the target, but a value is only implemented in one software system *e.g.,* unlike for `x265`, the motion-estimator feature *--me* of `x264` does not implement the 'star' pattern search;

— The feature of one system encapsulates one feature (or more) of the other *e.g.,* activating *--fullrange* in `x264` is equivalent to choose *--range* full for `x265`;

— The feature is only implemented in one software system *e.g.,* the feature *--rc-grain* of `x265` does not exist for `x264`;

— The feature does not have the same default value for the source and the target *e.g., --qpmax* is set to 51 by default for `x264` and set to 69 by default for `x265`;

— Both software systems do not have the same requirements or feature interactions *e.g.,* for `x265`, passing a input video in the yuv format does not work unless you specify *--input-res*, while it does for `x264`;

— The same feature is implemented in the source and the target, but the scale of the values differ between the source and the target. For instance, the constant rate factor *--crf* goes from 0 to 63 for `vp9` but from to 0 to 51 for `x264` and `x265`, which is problematic when comparing a value (*e.g., --crf* = 35) that will not have the same meaning for all systems. We could even imagine a situation where the values of a configuration option are increasing for one system and decreasing for the other.

In addition, and as acknowledged by [194], the study of configuration options also requires domain knowledge, which is also true when mapping the configuration spaces of different software systems.

Possible Solutions. Mapping configuration spaces could be envisioned using recent advances in the variability community. We propose a very simple protocol: (1) Align features [195, 196] between systems; (2) Meticulously model both configuration spaces *e.g.,* with feature models [9, 34]; (3) Analyse [197] and instrument [198, 199] them in order to create a resulting feature model generating configurations accepted by both systems. If the objective is purely to predict performance, heterogeneous transfer learning may be a possible black-box alternative that infers the existing relationships between distinct configuration spaces.

7.4 Conclusion

This section shows it is challenging yet possible to transfer performance between *distinct* configurable software systems, using two video encoders (namely `x264` and `x265`). We also discuss the limitations of our work and highlight the open challenges to overcome when generalising the applicability of transfer learning across *distinct* systems.

The variability induced by the choice of the system under test can be seen as part of the deep variability problem. With expertise, one could tell what couple of software systems could be used. We show that we can retrieve it by mining open data initiatives like Phoronix. Reusing the combination of the two protocols could help reusing performance models, and thus sparing some energy for the same accuracy in terms of performance prediction.

Reproducible Science

All the code and results for the first part of this chapter can be found at `https://github.com/llesoil/poc_phoronix`. The dataset collected out of the *OpenBenchmarking.org* web scrapping is available at `https://zenodo.org/record/5535465`. For the proof of concept of transfer learning across distinct software systems, the data, code and results can be found at `https://github.com/llesoil/TL_cross_soft`, as well as the related publication at `https://hal.inria.fr/hal-03514984/`.

Part IV

Train Performance Models Resilient to Deep Variability

Deep Variability is an issue for performance prediction. There exists interactions between different elements of the software environment altering software performance. And if software variability is deeper than expected, so should be the models trained to predict software performance. In this part, we want to promote the training of extended performance models that would be robust to a change of software environment and encapsulate deep variability. In Chapter 8 (page 108), we prove the concept by training an input-aware performance model, that can generalise on two distinct layers, the input data and the software layers.

Chapter 8

Train Input-aware Performance Models

Most modern software are widely configurable, featuring many configuration options that users can set or modify according to their needs, for instance to maximise some performance metric. So, various research work has been invested to predict key performance of the software (speed, size, energy, etc.) depending on the user's choices for its configuration. However software performances also obviously depend from its input data *e.g.,* a *JavaScript* script embedded in a web page interpreted by `Node.js` or an input video encoded with `x264` by a streaming platform. Owing to the huge variety of existing inputs, it is yet challenging to automate the prediction of software performance whatever their configuration and input. To the best of our knowledge, this chapter is the first domain-agnostic empirical evaluation of learning techniques addressing this problem. We empirically prove that measuring performance of configurations on multiple inputs allows capitalising on this knowledge and to train performance models robust to the change of input data. It is a first proof of concept showing that embedding deep variability in performance models is yet possible. This example could (and should) be extended with other layers of the software environment.

8.1 Problem Statement

8.1.1 Input-Aware Performance Models

As presented in Chapter 4 (page 50), the performance of a given software also obviously depends on its input data [200, 201, 114, 202, 87], like a video compressed by a video encoder [114] such as `x264`, a program analysed by a compiler [87, 68] such as `gcc` or a database queried by a DBMS [202] such as `SQLite`. All these kinds of inputs might interact with the configuration space of the software [68]. For instance, an input video with fixed and high-resolution images fed to `x264` could reach a high compression ratio if configuration options like *--mbtree* are activated. The same option will not be suited for a low-resolution video depicting an action scene [114] with lots of changes among the different pictures, leading to different performance distributions, as for input videos in Figure 8.1.[1]

This chapter proposes to provide the end user with a performance model that could be both reusable in practice and robust to a change of inputs, taking into account both the configurations and the inputs Figure 8.1. This requires the performance model to learn the complexity of the configuration space, but also the interactions between its inputs and the software configuration

[1]Videos 1 and 2 are extracted from our dataset (Animation_1080P-5083 and Animation_1080P-646f)

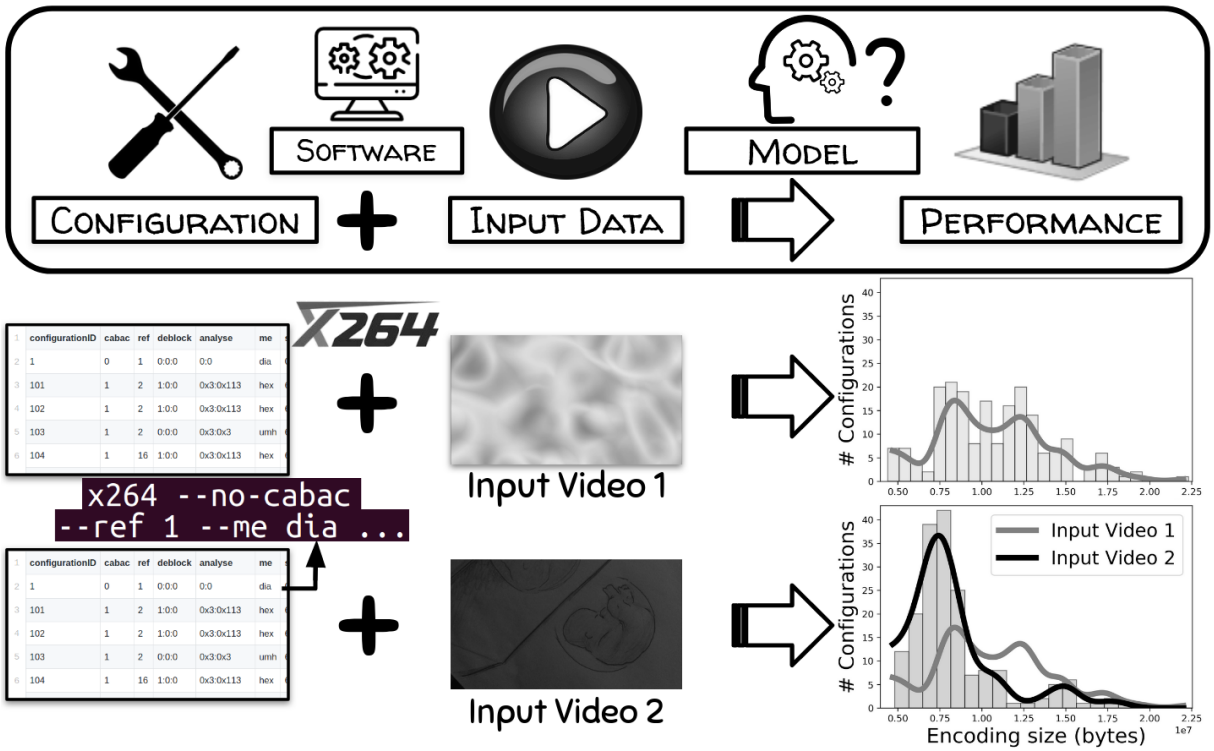


Figure 8.1 – The performance prediction problem: how to predict software performance considering both inputs and configurations?

space. In addition, and since these performance models typically need a substantial amount of data to accurately estimate software performance, we aim at reducing the cost of training the model by reducing the number of considered inputs and configurations, while ensuring a reliable prediction. How to select the input data and the configurations of the software to optimize the training of such a model? How to ensure a performance prediction robust to the change of input data?

To the best of our knowledge, this is the first domain-agnostic empirical evaluation of learning techniques applied on this performance prediction problem. Based on the requirements of the user, we propose to apply different learning approaches, different algorithms and different processing of data.

8.1.2 Offline and Online Costs

As illustrated above with the example of the *--mbtree* option, most software systems can be configured based on the input data they process. Since we cannot know *a priori* the input of the final user, tuning the software for this input has an online cost *e.g.,* the time needed to test different configurations and to understand which one would be the best, or at least good enough. We often want to keep this online cost as small as possible to avoid affecting the user experience. Fortunately, when designing the software, developers can train a model for predicting system configurations based on the input data fed to it. We call offline cost the cost related to the training of such a model, also including the computation of measurements. Figure 8.2 illustrates this difference between offline and online.

When researchers implement performance predictive models, these two aspects also have to be assessed: (1) prior to the use of the model, the offline budget is attributed to the organization

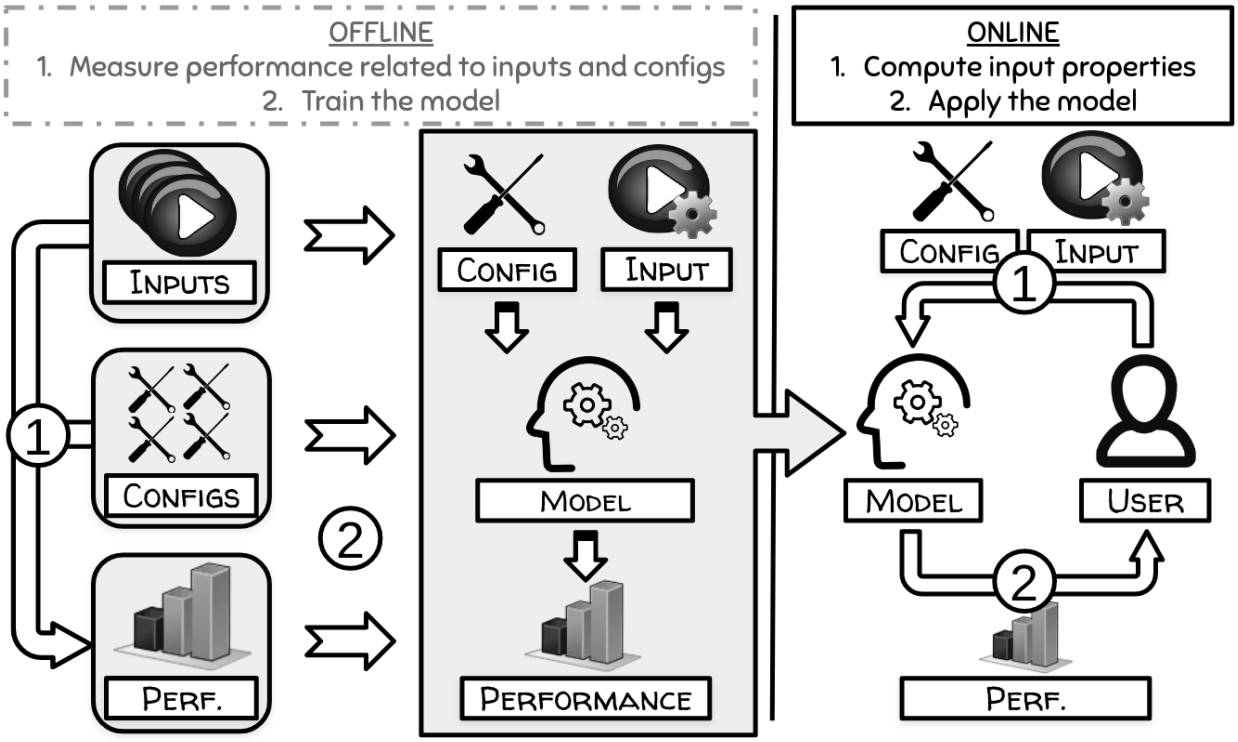


Figure 8.2 – Difference between Offline and Online

in charge of its training; (2) posterior to the training of the model, the online budget highly depends on the requirements of end users. As depicted in Figure 8.2, the offline budget includes the computation of measurements, but also the training of the model. In comparison, the online budget is mostly spent for the computation of input properties *e.g.,* the width, height or complexity of input videos - see Section 8.2. But the online budget sometimes includes additional performance measurements, here to adapt the model to the current input. The general goal is to ensure good performance predictions, while reducing the online budget and keeping a reasonable offline budget. These trade-offs should be adapted to the end-user needs. The distinction between offline and online learning has been given little consideration (most works only applying either "offline" or "online" learning), certainly because of the lack of consideration of input data that can, as empirically shown, alter predictive performance models of configurations.

8.1.3 User Stories

We present hereafter three different profiles of users that can face the performance prediction problem, each having its own constraints and objectives, as shown in Table 8.1:

— User $\mathcal{A}$ does not like to prepare much and cannot stand waiting. This user wants quick answers, even if it implies to sacrifice quality of service and end up with a *low accuracy* of the prediction model. It results in *low offline budget* and *no online budget*;

— In contrast, user $\mathcal{B}$ seeks to find a trade off between offline and online budgets. To obtain a *high accuracy*, this user is ready to spend time and attributes a *high offline budget* to train the model. However, this investment should be somehow profitable, the idea being to keep the *online budget as short as possible*;

— Finally, user $\mathcal{C}$ is committed to high quality standards. But $\mathcal{C}$ does not want specifically to capitalise on the inputs *i.e., no offline budget*. To reach a *high accuracy*, $\mathcal{C}$ is ready to wait and measure whatever is needed at prediction-time *i.e., high online budget*.

User	Offline budget	Online budget	Expected Accuracy
$\mathcal{A}$	Low	None	Low
$\mathcal{B}$	High	Low	High
$\mathcal{C}$	None	High	High

Table 8.1 – The users, their constraints, and their goal

From this partitioning of user profiles, we see that some critical questions arise. The performance prediction problem can be addressed through an empirical evaluation. Similarly, selecting the ideal learning approach for establishing the performance prediction model can also be solved through a dedicated empirical analysis.

8.2 Using Properties to Discriminate Inputs

We reuse the measurements of the Input Dataset, as defined in Section 4.2 (page 51). To differentiate the inputs directly in the learning process, we computed and added input properties [68] to the existing dataset: for the *.c* scripts compiled by `gcc`, the size of the file, the number of imports, methods, literals, *for* and *if* loops and the number of lines of code (LOCs); for the images fed to `imagemagick`, the image size, width and height, category and its average (*r, g, b*) pixel value; for SAT formulae processed by `lingeling`, the size of the *.cnf* file, the number of variables, or operators, and operators; for the test suite of `nodejs`, the size of the *.js* script, the LOCs, number of functions, variables, if conditions and for loops; for the *.pdf* files processed by `poppler`, the page height and width, the image and pdf sizes, the number of pages and images per input pdf; for databases queried by `SQLite`, the number lines for eight different tables of the database; for input videos encoded by `x264`, the spatial, temporal and chunk complexity, the resolution, the encoded frames per second, the CPU usage, the width and height were already computed; for the system files compressed by `xz`, the format and the size. We consider these properties as a proxy to understand the wide diversity of existing inputs.

8.3 Implementation of Performance Prediction Models

In this section, we define the different performance models used in the rest of this chapter.

Non-Learning Baseline

Average is a baseline returning the average value of the configurations in the training set. These few configurations are measured in an *online* setting.

Learning Algorithm

We consider multiple supervised machine learning algorithms, namely OLS Regression, Decision Tree, Random Forest and Gradient Boosting Tree. These are prevalent algorithms and commonly used

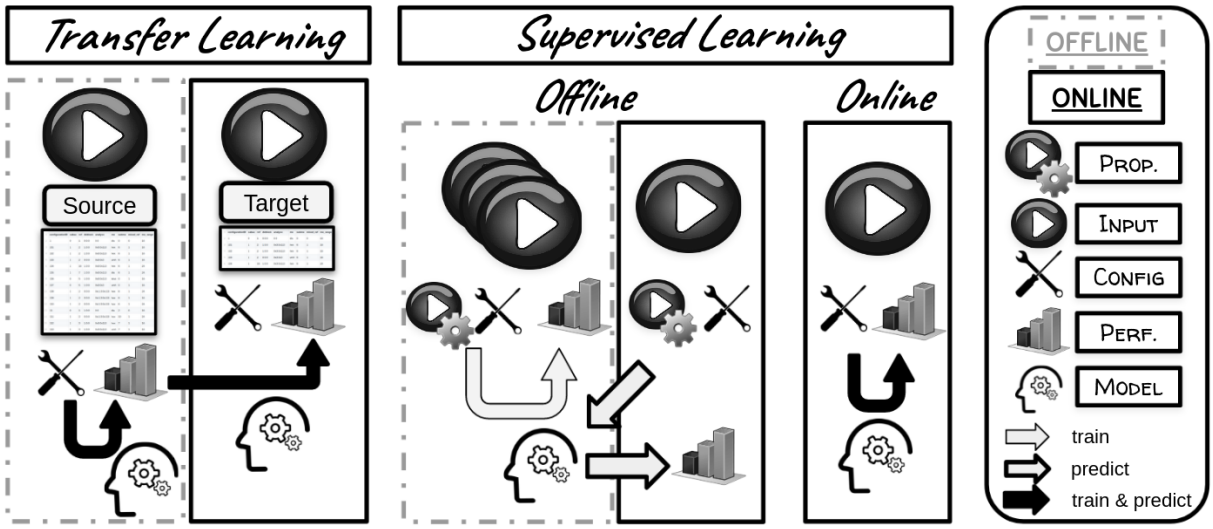


Figure 8.3 – The learning approaches to address the performance prediction problem

for tabular data. We do not consider deep learning or neural networks since we do not have lots of measurements in an online setting, which is usually a requirement, and since these methods are still commonly outperformed by random forests or gradient boosting [203, 204]. We train each machine learning algorithm with its default parameters, if not stated otherwise. The optimal parameters for an algorithm are dependent on the dataset, its size, and the performance property. As a result, they are necessary to be adjusted on a per-case basis.

Learning Approach

We define hereafter the learning approaches used in the evaluation. All are illustrated in Figure 8.3 and Table 8.2 reports a summary of their costs.

Transfer Learning. In performance modelling [15], Transfer Learning [25, 133, 138] is typically applied to capitalise on the data that are measured offline and apply it to improve the accuracy of the performance predictive model used online. In practice, it uses the measurements and the model obtained on a *source* input (offline) to reduce the budget needed to train a model on another input, the *target* input (online). In the rest of the evaluation, we implement the transfer approach Model Shift [26].

Supervised - Offline (& Online). Here, we consider supervised learning models trained in an input-aware manner [68, 38], *i.e.,* supervised offline leverages input properties as part of the dataset in addition to the classical measurements of configurations. Then there is no additional performance measurement of configurations at run time or prediction time, the "offline model" is simply reused for any input. It results in a negligible online cost.

Supervised - (Full) **Online**. Supervised online models require to compute and gather the performance measurements of configurations for each new input at run time or prediction time. It corresponds to the traditional baseline of the literature when a prediction model is learned from scratch for any change in the "environment" (new version, hardware, or input). Note that this approach can be very costly for end-users.

Separation Training-Test

We randomly split each set of configurations into a training and a test part for various training proportions – from 10% to 90% of the configurations used in the training set. The training set is available for data selection and training of the models, whereas the test set is purely dedicated to evaluate the trained models. To avoid biasing the results with different samplings of configurations, we fix the random seeds so the different techniques work with the same training and test sets.

Approach	Offline Cost	Online Cost	Accuracy
Supervised Online		***	***
Supervised Offline	***		*
Transfer Learning	***	*	***

Table 8.2 – The approaches and their costs - * = low, *** = high

8.4 Selecting Algorithm (RQ_1)

The production of a relevant performance prediction model involves the selection of the most appropriate algorithm for that task. Therefore, we first ask **RQ_1. How to compare different machine learning algorithms for establishing a relevant performance prediction model?** In order to address this question, we quantify the errors and the benefits of tuning hyperparameters of several relevant algorithms used in the literature. We also compare these results with a non-learning approach to the problem, used as a baseline for comparison. We decompose RQ_1 into three different questions.

Protocol

Why using machine learning?

Our first goal is to state whether machine learning is suited to address the performance prediction problem. To assess the benefit of using machine learning, we compare the *Average* baseline to different learning algorithms. We implement them in an online setting, *i.e.,* we use the supervised online approach; given the input of the user, we want to estimate its performance distribution. This is a prediction for one input at a time.

Which machine learning algorithm to use?

This evaluation is also the opportunity to compare these learning algorithms and search the one outperforming the others. We also study the evolution of their prediction errors with increasing training sizes. We consider those listed in Section 8.3. After training them on the training set, we predict the performance distribution of the test set and compute the prediction error. We repeat it for all combinations of system, input, and performance property. As prediction error, we rely on the Mean Absolute Percentage Error [19]. In Figure 8.4, we display the average MAPE values (y-axis) for various training sizes (x-axis).

What is the benefit of hyperparameter tuning?

Finally, we want to estimate how much accuracy we could expect to gain when we tune the hyperparameters of learning algorithms. To do so, we rely on a grid search [205] for hyperparameter tuning. We compute the training durations and prediction errors of these learning algorithms, with and without tuning their hyperparameters, and report the average difference for both.

Evaluation

Why using machine learning?

Figure 8.4 shows the benefits of using machine learning compared to the *Average* baseline. It appears that machine learning techniques clearly outperform the average performance value predicted by the baseline for all training sizes. The key indicator to study is the evolution of errors with increasing training proportions; while the baseline's accuracy does not progress with additional measurements, the learning algorithm improve their prediction, from 12% to 3% error.

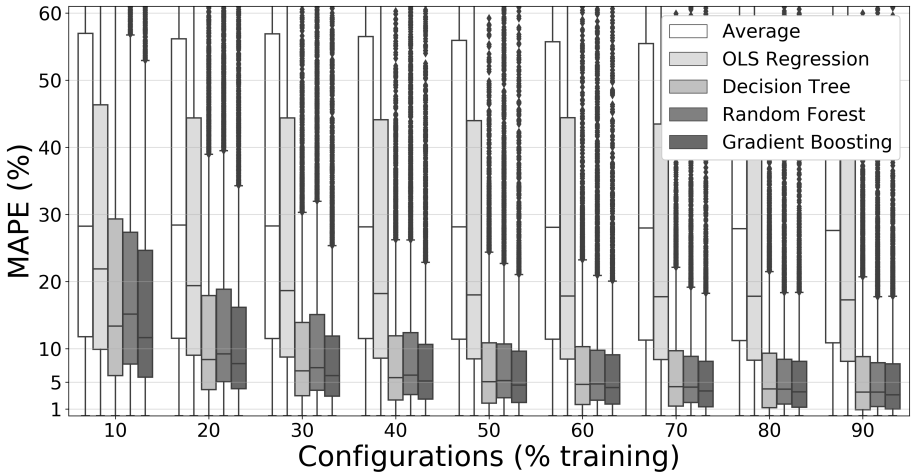


Figure 8.4 – Which (learning) algorithm to use?

Which machine learning algorithm to use?

While machine learning is generally beneficial, the prediction quality varies between the different algorithms. For instance, OLS Regression generally leads to bad results, *e.g.,* 28% of error with 10% of the configurations. Unlike the OLS regression, tree-based learning algorithms take advantage of the addition of new measurements. For a budget of 50% of the configurations, their median prediction goes under the 5% of error, which is encouraging. We want to emphasize that this result of 5% is only valid in average; the prediction will be better for few software systems, *e.g.,* `imagemagick` or `x264`, but will not stand for others, *e.g.,* `lingeling` or `poppler`. Though there is no big difference between these three learning algorithms, we observe slightly better predictions for Random Forest compared to Decision Trees, and for Gradient Boosting compared to Random Forest.

What is the benefit of hyperparameter tuning?

Our results found that hyperparameter search improves the MAPE result on average by $8 \pm 16\%$ within a range of $3 \rightarrow 37\%$ but also requires on average 120 times more training time when doing a grid search of estimator parameters. It is to be noted that is an improvement in percent, not percentage points. While the exact overhead of hyperparameter search is dependent on the number of configuration options of the model and the search method, we note that the overall benefit on the datasets is limited. This is especially confirmed by the observation that the best found hyperparameters were changing depending on both the system and the size of the training dataset. For simplicity of the setup, the rest of the evaluation uses the default parameters of each model, if not otherwise noted.

RQ_1. How to compare different machine learning algorithms for establishing a relevant performance prediction model? Machine learning is suited to address the performance prediction problem. We recommend tree-based learning over OLS Regression. These tree-based algorithms reach decent levels of errors with reasonable online budgets, between 5% and 10% relative error for most of the cases, and potentially improved by 8% when tuning their hyperparameters.

8.5 Selecting Inputs (RQ_2)

RQ_1 studies the effectiveness of machine learning. However, different ways of selecting inputs during the data collection or different number of inputs could alter the accuracy of the final performance model. Then, we address **RQ_2. How to create an appropriate dataset for training a performance prediction model?** Prior to the prediction, we have to select a list of inputs whose measurements will constitute the training dataset. We call this process *input selection*. RQ_2 investigates what choice of input selection technique (*e.g.,* all inputs, random selection of inputs, most diverse inputs, *etc.*) leads to the best results in terms of performance prediction. Depending on the offline budget of our users, we propose and compare various techniques of input selections. We separate RQ_2 into three different questions.

Protocol

How many inputs do we need to learn an accurate performance prediction model?

From the perspective of a user in charge of the training, adding an input to measure incurs a computational cost and should be justified by an improvement of the model. So, what is the effect of adding new inputs on the accuracy of performance prediction models? How many inputs do we need to reach a decent level of accuracy? We aim at minimising the number of inputs used in the training while maximising the accuracy of the obtained model. To do so, in this part of the evaluation, we train different performance models using various numbers of inputs and compare their prediction errors.

Then, once the number of inputs is fixed, we search if there is any benefit in precisely and methodologically selecting the inputs for data collection and model training. Does it bring any improvement over random selection of inputs? Do the different inputs used in the training set

change the final performance prediction accuracy? Depending on the offline budget of the user, we have different objectives.

How to select the input data for an offline setting?

If the offline budget is consequent a.k.a. the *offline setting*, the goal is to constitute a representative set of inputs to learn from, in order to build a performance model that will generalise as much as possible. We care about selecting diverse and representative inputs, in order to predict accurate results whatever the input data. For this setting, we compare the following input selections:

1. Random - Using a uniform distribution to decide which inputs should be included in the training;
2. K-means - Based on the properties of the different inputs, we apply K-means clustering to differentiate clusters of inputs with distinct characteristics. To increase the diversity in the selection, we then pick the inputs closest to the centre of the clusters;
3. HDBScan - Similar to the previous technique but with another clustering algorithm, namely the HDBScan, using density-based instead of means-based [2];
4. Submodular (Selection) - This technique computes a similarity matrix between the different inputs of a software system and optimises facility location functions to choose a representative set of inputs.[3]

For this *offline setting*, we implement the supervised offline approach with a Gradient Boosting (best in Section 8.4). Once the input selection technique chose the input, we include all related measurements in the training set. The test set is then composed of the measurements of all other inputs, not selected. To avoid biasing the machine learning model with different scales of performance distribution, we choose to standardise [190] all performance properties. But it has a drawback: since their values are close to zero, it artificially increases the MAPE values. To overcome this, we switch to the Mean Absolute Error (MAE) [19]. Since the performance property is standardised, we consider that models with MAE values inferior to 0.2 are good – way better than the expected average distance $\frac{2}{\pi} \simeq 1.13$ [206] between two points selected uniformly. We repeat the prediction 20 times and depict the average MAE (y-axis, left) in Figure 8.5 for different input selection techniques (lines) and number of inputs (x-axis) on a per-system basis. We added the number of training samples (y-axis, right). Except for `gcc` and `xz`, x-axis are in log scale.

How to select the input data for an online setting?

If the offline budget is low, a.k.a. the *online setting*, then we must be efficient and focus on predicting the performance distribution for the current input of the user. For this setting, we implement a transfer learning approach: the input of the user becomes the target input, and the candidate input to select the source input. In this online setting, the goal of input selection becomes to find one good source input that is as close as possible to the current input of the user - in terms of characteristics and performance. The choice of a good source should improve

[2] We rely on this implementation: `https://hdbscan.readthedocs.io/`
[3] We rely on this implementation: `https://apricot-select.readthedocs.io/`

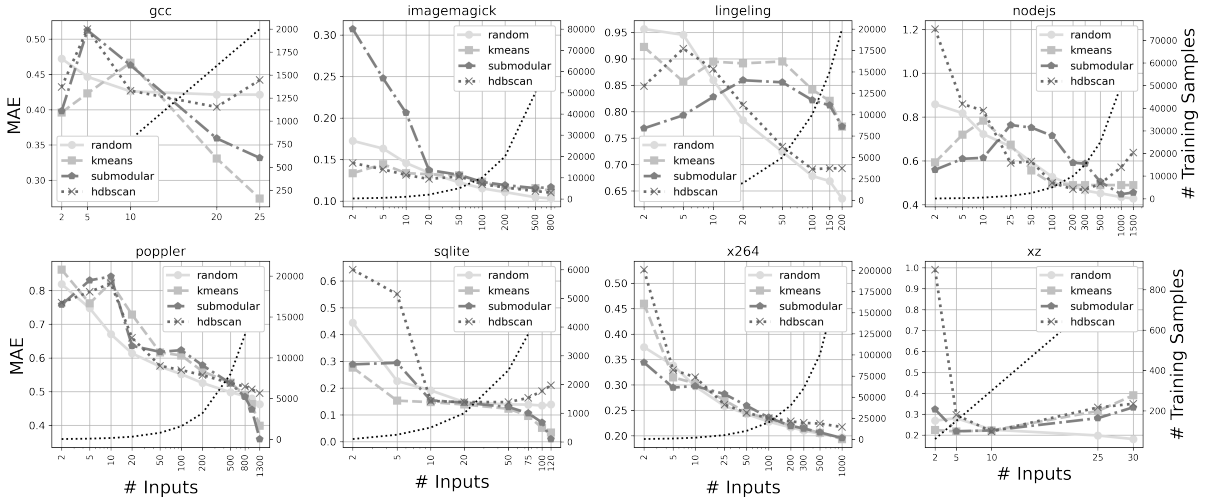


Figure 8.5 – Offline Setting - Influence of input selection and the number of inputs.

the performance prediction of the transfer learning approach [137]. We propose the following input selections:

1. Random - Same baseline as in the offline setting;
2. Closest (Input) Properties - Two inputs sharing common characteristics might also share common performance distributions. Following this reasoning, to improve the performance prediction, we have to select an input whose properties are similar to the current input's properties. To do so, we compute the MAE between the properties of the current input and the properties of all the candidate inputs. We pick the input obtaining the smallest MAE value;
3. Closest Performance - We use the few measurements already measured on the current input. For these, we compute the Spearman correlation between the performance distribution of the current input and all the candidate inputs. Finally, we select the candidate input with the highest performance correlations;
4. Input Clustering (& Random) - With the help of a K-Means algorithm, we form different clusters of inputs based on their properties. We randomly pick a candidate input in the cluster of the current input.

For this question, we use Gradient Boosting. We display the median MAPE results over 10 predictions for all software systems, performance properties and number of inputs in Table 8.3.

Evaluation

How many inputs do we need to learn an accurate performance predictive model?

Measuring inputs differs across software systems: measuring 5 inputs represents $5*201 = 1005$ configurations for `x264` but only $5 * 30 = 150$ for `xz`. Figure 8.5 shows that the performance model reaches its lower error threshold when considering about 20 inputs. Results are thus easier to interpret on systems with more inputs compared to `gcc` and `xz`. There exists however more difficult cases, *e.g.,* `Node.js` and `poppler`, in our experiment that might require more inputs to improve the accuracy of the prediction. To get a consistent prediction, we recommend the user to measure at least 25 inputs with a sufficient number of configurations per input, see Section 8.5.

How to select the input data for an offline setting?

As a reminder, in the offline setting, we seek to train a generalized model for all inputs; the selected inputs are supposed to be representative of the diverse set of inputs to be expected during deployment. Figure 8.5 presents our results. As expected, with an increased number of selected inputs, the influence of the input selection decreases. The input selection is especially important for small numbers of inputs. The evaluation shows that our techniques kmeans, submodular and hdbscan fail to beat the random baseline when selecting the inputs prior to the training of the model. There is no clear outperforming technique of input selection. These results could be explained by multiple factors: (1) the input properties processed by the input selections are not sufficient to differentiate the inputs (2) our baseline focuses on selecting different profiles of inputs, while it may be more efficient to select a set of average-like inputs. Out of this result, we advise keeping it simple and to adopt the random baseline. We challenge researchers to beat random.

How to select the input data for an online setting?

Input Selection	MAPE (%)	Training Time (sec)
Random	5.22	0.02
Closest Properties	4.17	0.05
Closest Performance	3.82	0.07
Input Clustering	4.70	0.02

Table 8.3 – Online Setting - Influence of input selection

As a reminder, in the online setting, we specifically build a model for the current input and select a similar input to transfer the knowledge from. Table 8.3 details the results of the different input selection. Unlike the offline setting, our input selection techniques were able to beat the random baseline, the best input selection technique being the Closest Performance with an average MAPE around 3.8, followed by the Closest Properties (4.2) and the Input Clustering (4.7). Wilcoxon signed-rank tests [175] (with significance levels at 0.05) confirm that predictions related to different input selections are significantly different from those using the random baseline: $p = 0.0$ for Closest Performance, $p = 1 * 10^{-184}$ for Closest Properties and $p = 1 * 10^{-27}$ for the Input Clustering. Therefore, and to continue to provide guidance for users, we advise using the Closest Performance to select the input in an online setting. But beyond the raw comparison of error values, beating the Random baseline with the Closest Property technique is a strong result. Empirically, it validates that these input properties are valuable to compute and should be included in the models to improve the machine learning prediction. The evaluation shows that training times are negligible.

RQ₂ - How to create an appropriate dataset for training a performance prediction model? In an offline setting, we recommend to select at least 25 inputs using a uniform distribution to pick the inputs, *i.e.,* the random baseline. In an online setting, we recommend to use the performance correlations technique, gaining about 1.4 point of error compared to a random selection of inputs. Our results empirically validate the use of input properties when predicting performance of software systems.

8.6 Selecting Configurations (RQ_3)

Since inputs and configurations interact with each other to change software performance, input selection is sensitive to the sampling of configurations *i.e.,* in the way we select the configurations used to feed the model. In this section, we vary the budgets of (1) inputs and (2) configurations used to constitute the training set fed to the model. The red thread of this section is the following question: **RQ₃. How does the number of measured configurations affect the performance prediction model?** To answer RQ_3, we train performance models fed with different numbers of inputs and budgets of configurations.

In this research question, we compare the accuracy of models according to the numbers of inputs and configurations used during their training, answering these questions:

Protocol

What is the best tradeoff between selecting inputs and sampling configurations?

For a fixed number of configurations, we study the evolution of the accuracy with the number of considered inputs. Is it better to measure lots of configurations or numerous inputs? Since the evaluation is designed to improve the generalization of the model, it mostly relates to models trained in an offline setting. Therefore, we implement the supervised offline approach, fix the algorithm to Gradient Boosting and use the random baseline as input selection technique.We repeat the experiment 20 times. In Figure 8.6, we depict the MAE (colour) for various numbers of inputs (x-axis) and configurations (y-axis).

How many configurations should we select to accurately predict performance?

Rather than just selecting the inputs, we want to understand the effect of the existing interactions between the inputs and the configurations on the performance model. How does the budget of configuration affect the prediction of the model given a fixed number of inputs?

Evaluation

What is the best tradeoff between selecting inputs and sampling configurations?

According to Figure 8.6 results, diversifying the inputs seems to be more effective than selecting different configurations to train an input-aware performance model. But for a fixed budget of inputs, there is a slight improvement of accuracy when increasing the number of configurations. As a result, both should be combined to obtain the best possible model. Overall,

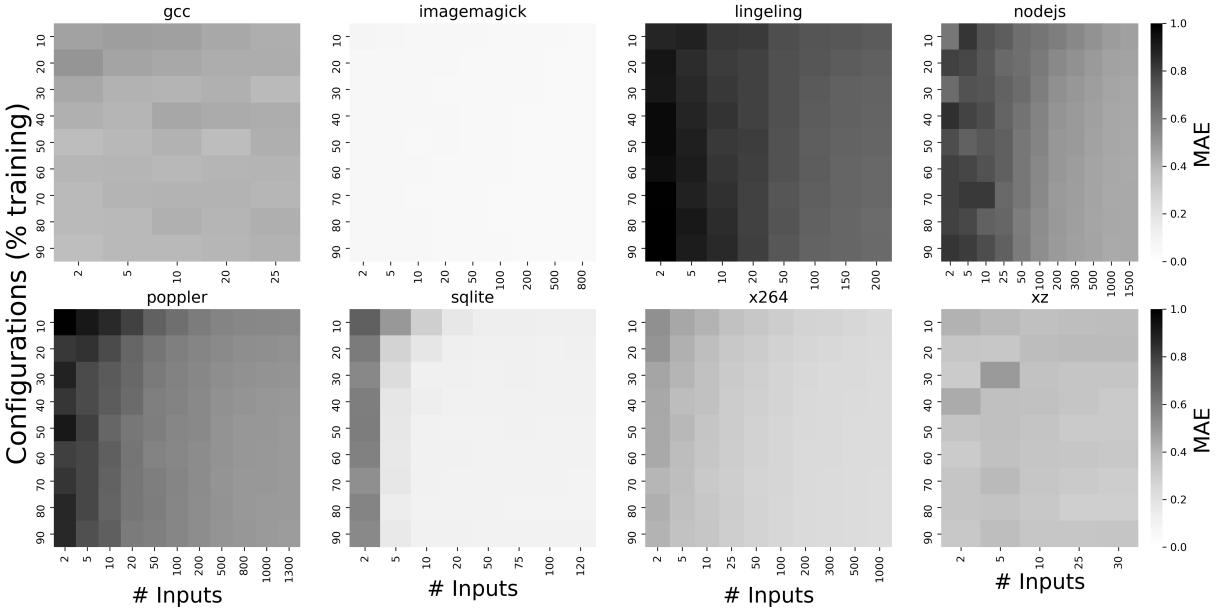


Figure 8.6 – Error of performance models (low MAE = better) according to different budgets (inputs & configurations).

the ideal budget – both in terms of inputs and configurations – highly depends on the expected level of errors combined with the difficulty of learning predictive performance model for the software under test. For instance, predicting performance of `imagemagick` is relatively easy, with an average MAE value at 0.06. For this software system, picking 25 inputs is almost already a waste of resources, we do not need that much data - 5 inputs is already enough with 30% of the configurations. Other systems are harder to learn from *e.g.,* `lingeling` with an average MAE of 0.76. For these, we recommend increasing the number of inputs and configurations. The MAE obtained on the training set should be used as a proxy to estimate the difficulty of predicting the performance of the software under test. The greater its value, the greater the budget needed to learn an accurate model.

How many configurations should we select to accurately predict performance?

We advise to use in average 50 configurations per input - about 30% of configurations for most of systems. After this amount of configurations, the progression of the error level when increasing the number of inputs is stabilising: the model has enough information to be generalized to unseen configurations. To give concrete numbers illustrating the cost of measuring this budget of configurations for one input, 50 configurations are measured in about eight minutes for `x264`, in about six minutes for `imagemagick` or in about ten seconds for `gcc`.

RQ_3 - How does the number of measured configurations affect the performance prediction model? To learn an accurate performance model on our software systems, we can advise to include at least 50 configurations per input. Users should adapt their budget according to the prediction error obtained on their configurations.

8.7 Selecting Approaches (RQ_4)

Depending on the user profile, different trade-offs between offline and online budgets may be sought. It is critical to determine which learning algorithm to select and with which expected accuracy for each type of user profile. To this end, we ask the fourth and last question of this chapter, **RQ_4. Which approach to recommend based on the user profile and need?** To concretely address it, we compare the state-of-the-art approaches listed in Section 8.3. The final outcome of this research question is a set of rules deciding which learning approach we should use to overcome the performance prediction problem depending on our constraints.

Protocol

Is it better to use the transfer learning or the supervised online approach?

Depending on the online budget of the user, it can be worth (or not) to transfer the knowledge from one input to another. If the online budget is high, we guess there is no need to transfer, *i.e.,* we do not use transfer learning. If this budget is low, we can benefit from the transfer learning approach. How much online budget justifies the decision for transfer learning? To answer this, we implement both approaches with different numbers of configurations on the target input. We use Gradient Boosting and predict the performance spanning all systems and performance properties. Due to outliers drastically increasing the average value, we compute the median MAPE instead of the average. In Figure 8.7a, we display the MAPE value (y-axis) for different budgets of configurations (x-axis).

Should we train performance models offline or online?

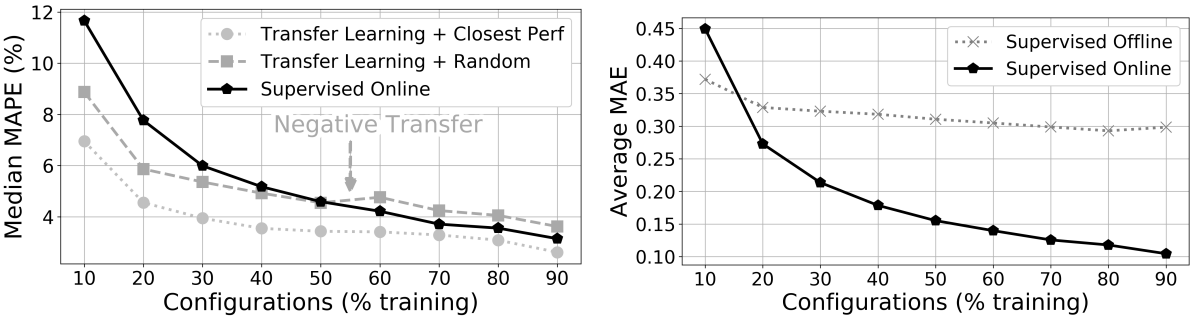
To answer this question, we compare the supervised offline approach to the supervised online approach. We use Gradient Boosting as learning algorithms. To be able to compare them, we rely on the MAE. We predict performance implementing both approaches and compute the MAE value for different training size - varying from 10% to 90% of the configurations available per input. In Figure 8.7b, we display the results for both approaches, *i.e.,* the average MAE on all software systems and performance properties.

Evaluation

Is it better to use the transfer learning or the supervised online approach?

With the input selection technique set to Closest Performance, the transfer approach always outperforms the supervised online approach, as shown in Figure 8.7a. This strong result demonstrates the importance of capitalising on the existing measurements, measured in an offline setting. If we are able to create a representative sets of inputs for each software system, then the transfer learning becomes the best approach to use. But since we pick the source input among all the inputs of our dataset, we add the comparison of transfer learning with a random input selection baseline, thus putting ourselves in the real situation of a user with a low offline budget *i.e.,* not able to select a good source input. Even in this case, we still outperform the supervised approach for less than 55% of the configurations. But this transfer with random selection has an

expiration date; after a training proportion of 55% - represented by the arrow on the graph, it leads to negative transfer: the added measurements (of the source) become noisy data interfering with the training of the target model. If the online budget is low, transfer learning should be used. But with a huge online budget, it might be better to use the supervised online approach.



(a) Transfer Learning VS Supervised Online.

(b) Supervised Offline VS Supervised Online.

Figure 8.7 – Errors of the different techniques when training input-aware models

Should we train performance models offline or online?

Figure 8.7b shows the evolution of the two supervised approaches, offline and online. The first point we notice is the slow progression of the supervised offline approach, only from 0.37 to 0.30 between 10% and 90% of configurations. A possible explanation is that it is so hard to generalise over the input dimension that it hides the benefit of adding configurations. While when considering only one input at a time, this difference of performance distributions does not bother the training of machine learning model. Nevertheless, comparing the raw numbers provides a straight answer to the initial question: unless the online budget is really low, if the choice between a supervised offline and a supervised online approach occurs, one should definitely prefer the online approach. But this finding has to be contextualised. From the point of view of the final user, an online prediction computing measurements will always last longer than an offline prediction, using an already-trained model. As a consequence, users should always prefer the online approach compared to the offline approach when they have an online budget (even a small one). The supervised offline approach should be adopted for lack of a better solution, as the last approach to implement when users cannot afford to measure configurations in an online setting.

RQ_4 - Which approach to recommend based on the user profile and need? We are able to (1) train input-aware performance models and (2) recommend users learning approaches adapted to their offline and online budgets.

8.8 Discussion

Each part of the evaluation provides a recommendation for our three user profiles defined in Section 8.1.3, depending on the trade-off between their offline and their online budgets in Table 8.1. This discussion summarizes our findings while answering $RQ_1 \rightarrow RQ_4$ and turns these

findings into recommendations and actionable rules, thus guiding the user to solve the performance prediction problem. **How to help users predicting their software performance, whatever be the input data and their configuration?**

Depending on the available online budget, we distinguish the following cases:

— If the user has a **high online budget** (*e.g.,* User $\mathcal{C}$) we recommend using the supervised online approach (Section 8.7) with a Gradient Boosting Tree implementation (Section 8.4) and tuned hyperparameters (Section 8.4). In that case, users can expect low prediction errors - under 5% most of the time;

— If the user has a **low online budget**, we can also recommend the supervised online approach but cannot guarantee outstanding performance estimations *e.g.,* 12% of errors with 10% of the configurations. But in this case, if a representative set of inputs has already been measured *i.e.,* with a big offline budget (as for User $\mathcal{B}$) we recommend using the transfer learning approach (Section 8.7) with a Gradient Boosting algorithm and using the closest performance input selection technique (Section 8.5). Our experiments show that the performance predictions of User $\mathcal{B}$, whom is counting on inputs in an offline setting, will outperform the online predictions of User $\mathcal{C}$ whatever be the budget of configurations - under 3% of error (Section 8.7) for reasonable budgets of configurations;

— If the user has **no online budget** (*e.g.,* the User $\mathcal{A}$), then the available offline budget is key. If the offline budget is low, there is no silver bullet: since we cannot guarantee low errors with our models, it is probably better to avoid predicting than providing a poor estimation of software performance. If the user can benefit from numerous inputs *e.g.,* more than 25 (Section 8.5), then we can advise to use the supervised online approach (Section 8.7) implementing a Gradient Boosting algorithm and with a random selection of inputs (see Section 8.5). We expect an error level around 8% and 10% depending on the number of measured configurations per input.

Figure 8.8 summarises these rules of thumb into a flow diagram. We also depict likely locations of users $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ at the end of the decision process, based on our previous recommendations, as well as the expected relative errors.

As a limitation of our work, we highlight that it is difficult to learn the performance distribution for a few software systems *e.g.,* `lingeling`, `poppler`, and even `Node.js` For these systems, the prediction errors are above 20% when implementing a supervised offline approach with tight budgets of configurations or inputs. Though, note that the average results may not exhibit this issue. Note also that if these rules are applied on a concrete case, a consequent effort in terms of measurements is then requested *i.e.,* to measure more than the general and averaged threshold of 25 inputs. The requested amount of inputs on a per-system basis is documented in the companion repository of this chapter.[4]

8.9 Threats to Validity

A first threat to validity relates to the input properties computed in Section 8.2. Since we are not domain experts of each of the eight software systems considered in this experiment, we cannot validate the construction of such properties, *i.e.,* it is likely that there is an opportunity

[4]See `https://github.com/HelgeS/variability/tree/data/src/when_to_stop_measuring_inputs.ipynb`

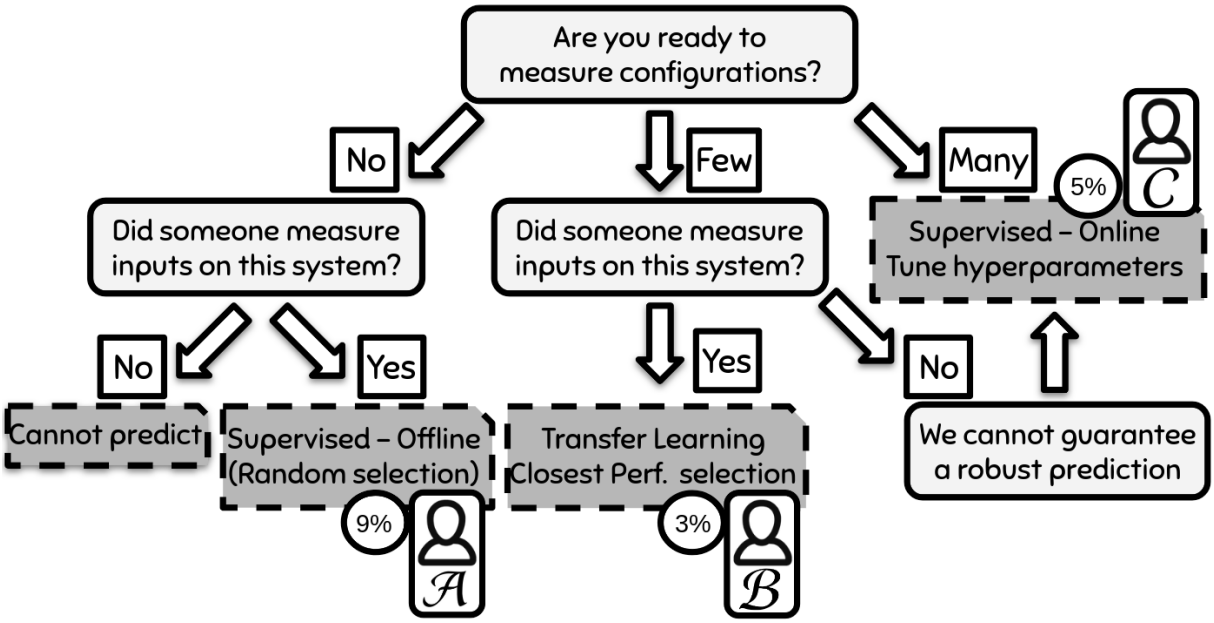


Figure 8.8 – Lessons Learned - A flow diagram for users

to craft more expressive input properties. To the best of our knowledge, which input properties to use in order to improve the performance prediction remains an open and unanswered question. Furthermore, we neglect their computational cost; this can threaten the results of Section 8.5 or overestimate the benefit of the supervised offline approach. A second threat to validity is related to the randomness in machine learning methods, subject to modifications in their predictions. To reduce these stochastic effects, we (1) fix the random seed to feed the same training and test sets to all models and have comparable results and (2) repeat the experiments 20 times.

8.10 Conclusion

Due to the interactions between inputs and configurations, predicting the performance property of a software system whatever the input data is non-trivial. In this chapter, we propose to train input-aware performance models *i.e.,* performance models that work whatever the input data. We empirically evaluated the effectiveness of different machine learning techniques compared to these techniques. In terms of deep variability, it is a first step towards embedding deep variability into performance models, it pushes a bit the limit of performance modelling, including the input data as part of the model (in addition to the configuration options).

It is also the occasion to propose to evaluate the cost of performance models differently according to their offline and online costs. The distinction between offline and online in our context relates to the time when measurements are made and used for training. The offline setting resembles the standard supervised ML approach: an initial dataset of representative measurements is collected, and the prediction model is trained. It is then used for predictions on any input without changes. This is useful when the user has only a small budget and cannot make additional measurements at prediction-time. This is different in the online setting. Here we can start from the previously trained model, but can also sample additional measurements for the specific input of the user and use those to fine tune the model. The number of measurements

depends on the budget. That is, in the online context we can perform additional learning after deployment and during the model's use, similar to the online learning setting in ML.

This chapter empirically proved that measuring performance of configurations on multiple inputs allows capitalising on this knowledge and to train performance models robust to the change of input data. Offline learning can build configuration knowledge that pays off and benefits to online learning when a new input needs to be processed. We emphasized the need to compute relevant input properties to discriminate the different inputs fed to software.

We gave practical and concrete recommendations to users based on their constraints and resources. According to their *offline* and *online* budgets, we recommend them to use different performance models: (1) transfer learning if they have a high offline budget and a low online budget, (2) supervised learning with measurements computed on the current input if they only have high online budget or (3) supervised learning on a representative set of inputs and their properties if they have only high offline budget.

Reproducible Science

All the scripts, procedures and containers are available in `https://github.com/HelgeS/variability/tree/data`, working on the dataset collected in Chapter 4 (page 50). We also pushed a container in dockerhub (see `https://hub.docker.com/r/anonymicse2021/x264_monoconfig`) and store the related source code to allow modifications.

Chapter 9

Conclusion and Perspectives

9.1 Conclusion

In this thesis, we introduce the concept of deep variability, defined as the numerous existing interactions between the software layer and the rest of the software stack altering the resulting software performance distribution. We claim that deep variability represents a major threat to the applicability of performance models.

To support this (strong) claim, we demonstrate that those interactions (1) exist and (2) are not always trivial to handle *i.e.,* not linear, as it is often supposed and presented in the state of the art. Besides the motivational examples proposed throughout the document, we empirically and systematically demonstrate the existence of deep variability by studying two factors of the software environment. In Chapter 4 (page 50), we first show the complexity of some interactions between input data and (configuration) options fed to software systems. We prove that depending on the variety of inputs fed to software systems, the resulting performance distribution will not be the same. In Chapter 5 (page 66), we then apply a similar protocol to exhibit other existing interactions between compile-time and run-time configurations.

All cases are not always problematic to handle. To present our results in a transparent manner, we depicted the measurements made in the context of this thesis in Table 9.1. Each line corresponds to a couple of software system and a performance property. Each column represent a factor or a layer of the software environment. At the intersection, we apply the following code: the gray areas represent what we did not measure, we did not vary the environment factor for this software system and this performance property; ✓ indicate simple cases of interactions, often just a difference of scale, but with high and positive correlations between the performance distributions and a similar effect of options; x indicate that it is generally simple to handle this kind of environment change, as the ✓ signs, but we identified corner cases that could occur; ❦ indicate that complex interactions are often observed for this factor, making performance distributions unstable and difficult to predict with weak performance models.

In terms of results, we retrieve only simple cases of interactions for the hardware layer, which is consistent with conclusions of other papers [26], but this should be cross-checked and confirmed with more measurements. The Operating System (OS) layer has not been studied (yet) and left as future work. Results on `MongoDB` tend to confirm that software evolution makes the performance distributions unstable and hard to predict [138, 207]. The compile-time layer is generally simple to handle, except for `nodeJS` and the *size* of `poppler` for different inputs. The input data is generally unstable and should deserve to be carefully benchmarked, because most of the software system are sensitive to inputs *i.e.,* their performance distribution vary with the inputs they process.

System	Performance	Hardware	OS	Evolution	Compile time	Run time	Input Data
gcc	*ctime*					❦	✓
gcc	*exec*					❦	❦
gcc	*size*					❦	✓
ImageMagick	*size*					❦	✓
ImageMagick	*time*					❦	✓
lingeling	*#confl.*					❦	x
lingeling	*#reduc.*					❦	x
MongoDB	*various*	x		❦			x
nodeJS	*#operations/s*				❦	❦	x
poppler	*size*				x	❦	x
poppler	*time*				✓	❦	❦
SQLite	*q1*					❦	x
SQLite	*q15*					❦	x
x264	*cpu*	✓			✓	❦	✓
x264	*fps*	✓			✓	❦	✓
x264	*kbs*	✓			✓	❦	❦
x264	*size*	✓			✓	❦	❦
x264	*time*	✓			✓	❦	✓
x265	*cpu*					❦	✓
x265	*size*					❦	❦
x265	*time*					❦	✓
xz	*size*				✓	❦	✓
xz	*time*				✓	❦	✓

Table 9.1 – Performance Evaluation of Deep Variability (❦ complex cases, x only few interactions, ✓ linear or no interaction)

Complex cases of interactions between the software layer and the rest of the stack increase the — already — difficult task of configuring software systems. With different software environments, the effect of configuration options can change. As a consequence, it can be beneficial to activate an option for an environment, but not for another one. With different software environments, the influence of configuration options can vary. As a consequence, an option can be critical to leverage for an environment but irrelevant to leverage for another one. With different software environments, the order or ranking of configurations to obtain a performance goal can be different. As a consequence, the best configuration for an environment will not be the same for another environment. Also, the default configuration can not be estimated just once for all environments. As each user possesses its own software environment, configuring the software layer cannot be achieved by only looking at the performance of configurations in one environment, but rather with their performance distributions in different environments. Alternatively, the chosen configuration could be adapted to the final user.

Depending on the nature of interactions, different solutions can be suited to configure software systems. If the interactions are only linear (✓), the effect and influence of options remain the same, as well as their rankings. As a result, the configurations can be directly reused across environments and there is no need to reconfigure anything. If there exists few interactions (x), the effect and influence of options are generally stable, but we exhibit few examples of interac-

tions. In this scenario, we advise to check the stability of results with few measurements before blindly applying the default configuration. If complex interactions occur (❦), we advise to use more advanced techniques (*e.g.,* self-adaptive systems) to find the best configuration for the current environment.

Deep variability highlights the limitations of the current practice of performance models. For instance, in the case of the input layer, performance models trained with one input can not always be reused on another input, because the effects of options captured by the machine learning may vary across inputs. If we can not predict the input chosen by the end-user, we will not be able to propose him a performance model suited to her case. Through a literature review, we also identified multiple research papers affected by this issue. The same can apply with the compilation layer; a performance model trained with a software system using the default compile-time can not generalise over the same system compiled with fancy compile-time options. To generalise what has been learnt on these two cases, if the effects and importance of options of a system are changing with their environment, the performance models should be either adapted or re-trained according to the software environment. For developers and researchers, it implies that we should not provide a unique performance model per system and performance property. It even implies that we cannot document or interpret the effects of options without specifying the used environment.

However, we do believe that researchers should not give up on performance models and rather improve them with deep variability in mind.

To mitigate the previous limitations, Chapter 6 (page 84) proposes techniques to group together environments that share similar performance distributions. This permits the reduction of the number of performance models needed to cover all the environments and thus the cost of benchmarking environments when studying the deep variability of a software. Chapter 7 (page 95) also shows that measurements and performance models can be recycled across different software systems. Giving a second life to these measurements allows to counterbalance the drawbacks brought by deep variability.

Related work has proposed transfer learning techniques to handle environment changes. Indeed, those techniques are suited if the end-user has some time to compute additional measurements *i.e.,* in an online setting. However, if the user is impatient or just not able to wait for additional measurements *i.e.,* in an offline setting, we propose to revisit contextual performance models, including the properties of the environment (like the properties of inputs as in Chapter 8 (page 108) or the return of a `lscpu` command) as part of the data processed by the performance model. In addition, these models are trained once and for all, and valid for all the possible environments (in our case all the possible inputs), which shows we are able to push a bit the limits of performance models. We believe that training contextual performance models instead of performance models is a cheap way to embed deep variability directly in performance models and hope it will soon be adopted by the community. We present encouraging results on the input data layer, showing that contextual performance models are competitive with transfer learning when the online budget allocated to the target environment is very low *i.e.,* if the end-user is not patient enough to measure new configurations.

Depending on the nature of interactions, various solutions can be suited to train or adapt

performance models when the software environment is changed. If the interactions are linear (✓), only the scale of the new performance distribution has to be estimated, which can be performed with only few additional measurements (*i.e.,* a calibration phase) or even with the properties of the environment. Once it is done, the initial model can be reused, just by changing the scale of estimation. It is an encouraging result, because most of the cases do not require any additional training. If there exists few interactions (x), transfer learning techniques might be adapted to update the performance model based on the observed differences with an existing model. If complex interactions are acknowledged (☯), different approaches can be applied. If there is no (or a very low) budget to measure additional configurations, contextual performance models might be the best solution to apply. Though they are not as accurate as transfer learning in our experiments for consequent budgets, it is, to the best of our knowledge, the only attempt to build a general model working whatever the configuration and the environment. If there is a consequent budget to measure additional configurations, transfer learning techniques might work, up to the point where there is nothing left in common between the source and the target environment. Grouping together the environments with similar performance profiles can be used to choose the best source environment and thus extend the scope of transfer learning.

9.2 Perspectives

Then, we detail few other research directions related to deep variability.

9.2.1 Estimate the Uncertainty of Scientific Results

In the configurable system area, we define this deep variability issue as an extension of the software variability, the software being our core of study. But software systems are also used as tools in other domains (biology, physics, chemistry) by other researchers, already struggling with software variability. Typically, they want to ensure the robustness of scientific results. To this matter, if changing a configuration of the software system in their protocol is also changing the results, it might soon — if not yet — become a recurrent threat to validity for their experiment. We believe deep variability further exacerbates the problem. Identifying the influential layers of deep variability is especially relevant for them, to understand what are the factors they should change, as opposed to the ones they can set once and for all.

If there is no interaction in the remaining variability layers that are not changed in the paper, it does not matter, because the results will hold whatever the executing environment: modulo the scale of the measurements, the general trend and main outcomes should not change. But if interactions happens, different conclusions can be drawn out of the same protocol, simply applied in different research teams with different working conditions [208]: versions [207], compiler [209], operating systems [210], *etc.*

For this reason, we think that increasing our knowledge of deep variability will strengthen the confidence we can put in research results. Moreover, we could quantify the uncertainty associated to a scientific conclusion, based on the software, the environments and the performance property they are considering.

For instance, we could address the following research questions, trying to estimate the general

impact of deep variability on other fields of science using software systems:

RQ_1 - Can we estimate the proportion of scientific results are affected by the choice of software environment?

RQ_2 - To what extent are scientific protocols sensitive to deep variability? Is it possible to estimate how much of the variability of the procotol is due to deep variability?

9.2.2 Towards Collectively Defining a Dataset for Deep Variability

In the current practice of research, each paper defines and implements its own measurement process. For instance, in the performance modeling community, before actually training the performance model, the vast majority of papers propose another dataset designed to address their current problem (see the explanations related to Table 2.1 on page 33). In many aspects, it contributes to open science and it is truly a valuable contribution to improve, dataset after dataset, our understanding of the software.

Because testing all possible combinations of hardware platforms and operating systems has a huge cost, researchers — including us — often only compute measurements on a simple hardware model with one operating system, which is methodologically sound, convenient to produce fast and reliable results and ease the comparison of the different results obtained during the protocol evaluation. It would not be a problem if the datasets related to different papers were comparable. But since we all start from scratch, we often select (1) our subject systems, even if some common software systems are reused from papers to others (2) our configuration options and arbitrary associated values, as well as (3) different sets of inputs fed to the configurable systems.

This perspective addresses a message to configurable system researchers, but could also resonate in other community. To avoid making more complex the (already consequent) deep variability problem, it might be beneficial to define what we would call the "Hello World" dataset of deep variability. This dataset could be the result of discussions between researchers, agreeing on (1) the subject systems to include in the protocol (2) the configuration options and the related configurations to measure (3) the set of inputs to benchmark. If we construct such a dataset, it would be simpler to compare the effects of the other variability layers, because we control the inner variability of "Hello World" dataset *i.e.,* the software variability related to each system and its options, as well as its input sensitivity if any.

Historically, we believe that the SPLConqueror [72, 13] initiative was targeting the same objective. To recall, Siegmund *et al.* provided a dataset of measurements — likely the mostly reused in the community — as well as a framework simplifying the way to sample configurations and to train performance models. In other words, this perspective could be seen as a revisit of the SPLConqueror initiative with the deep variability problem in mind. First, we want to define this "Hello World" dataset, with the same systems, configurations and inputs to test software variability in a known context, as they did it ten years ago. If the principle is applied, it allows to collectively test different environments using this "Hello World" structure. After that, we can also extend the SPLConqueror tooling to add the possibility to benchmark this "Hello world" dataset in different operating systems and in different environments. Paper after paper, it would allow to collectively tackle deep variability, without putting the cost of measuring all the layers on one or two groups of researchers [25, 26].

This perspective is mostly a human issue, we can not propose it alone. To ensure this is a relevant issue and that the resulting guidelines should be followed by a majority of researchers in our domain, this problematic should probably be discussed during several workshop sessions. However, there are also underlying scientific problematic to address:

RQ_1 - How to choose the different software systems to consider as part of this collaborative dataset? In an ideal situation, they would be cheap to compute *e.g.,* selecting `x264` rather than `x265` according to the results of Figure 7.3 (page 98) and already known and used in the community.

RQ_2 - Once the software systems are chosen, how to select the configuration options and to sample the configurations? Between the different papers, this choice can vary, maybe resulting in different results.

RQ_3 - How to select a good benchmark of inputs for these systems and these configurations? This thesis might be helpful here, especially the results of Chapter 4 (page 50) and the procedures of Chapter 6 (page 84).

9.2.3 Mining Open Data to Infer Information related to Deep Variability

To complete what can be inferred out of measurements, another idea is to mine the web to find additional information helping us in increasing our understanding of deep variability.

For instance, as part of our experiments on `x264`, we mined its commit messages out of the Git repository. Some of these messages show that developers have a fine-grained domain expertise of input sensitivity.[1] These commits contains information that we could use, sometimes linking the effect of configurations options to few inputs or giving information complementary to the documentation regarding the effect of options on performance.[2]

Also, constraints or bugs related to deep variability can be found in forums, where users try to find solutions and actually connect with domain and software experts. Even if we can not assess of their validity, it can provide insights on the influential layers altering some performance properties of software systems.[3] It represents an alternative to the (costly) measurements to list the existing interactions between the different layers of the software environment.

Another source of information are the initiatives collecting and publishing data online. To give an example, by web scrapping the website *openbenchmarking.org*, we were able to train a ML model predicting the performance of the default configuration of `x264` based on the hardware properties (amount of L1 cache, RAM, *etc.*) of the hardware platform executing it. More details about this experiment can be found in Section 9.4 (page 136). This accessible contextual information allows us to take shortcuts when combined with a knowledge of deep variability. For instance, if we know for sure that the hardware layer is only interfering with the software in a linear way, when transferring the performance model from one hardware platform

[1] See the resulting commits at `https://github.com/llesoil/input_sensitivity/blob/master/results/systems/x264/commits_x264_IS.png`

[2] Respectively two examples at `https://github.com/mirror/x264/commit/f30aed6d810ef408cbf19cc6760605b0b87cbfde` or `https://github.com/mirror/x264/commit/76a8276f19ca5b01b3d54858cfc95ddc20fb2a71`

[3] See for instance the post at `https://forums.tomshardware.com/threads/problem-with-rx-580-encoding-x264.3590209/` or in the same vein this one `https://www.reddit.com/r/Twitch/comments/t4s9eo/hardware_amd_vs_software_x264/`

to another, the only thing we have to do is to estimate the scale of the distribution. This could be done with the information extracted out of this website instead of having to compute additional measurements, thus sparing time and resources.

For some of them, it is yet unclear how to include these rules, constraints or estimations as part of the performance models in a systemic manner, but at least it represents an (almost) costless source of exploitable information. We could address the following questions:

RQ_1 - To what extent is the web informative to understand deep variability? Among the documentation of software systems, the github issues and commit messages, the forums and the open data initiatives, it is unclear what (1) is worth to extract and (2) should be trusted.

RQ_2 - How difficult is it to automate this extraction and is it worth the cost of adding these information in performance models?

RQ_3 - How to build performance models capable of handling these various forms of data? While it is clear for contextual-performance models working directly with environment properties, the current practice does not allow the integration of rules or insights directly in the model.

9.2.4 Deep Variability-Aware Hacking

Security breaches can sometimes occur as a result of a very small vulnerability in the system, like using a specific version of an operating system [211] or the use of a firmware not up-to-date [212]. From the attacker's point of view, it could be useful to detect the properties of the environment to better identify potential vulnerabilities in the system [213]. On the contrary, for a service provider it is preferable to hide to a potential attacker how the system is configured. The less information the user has access to about how the system works, the less opportunity they have to misuse that information, and the better the overall security of the system.

When the provider offers a service including a software system to users (potential attackers), the latter might decide to use the observed performance of the service to detect properties of the system, such as the software configuration used by the service or the computing power of the server running the calculation. We believe that understanding software variability and more generally deep variability could give the attackers insight about the overall configuration of the environment executing the service, thus allowing him to exploit these pieces of knowledge for more effective attacks. For instance, if a configuration is particularly resource-consuming, the attacker could possibly shutdown a server with a minimal amount of requests.

This perspective proposes to study the feasibility of such an attack. Given a performance distribution associated with software, configurations and an execution environment, the goal would be to find (1) the configuration used by the software (2) indications on the execution environment (estimate of the amount of RAM, the operating system used, *etc.*) (3) how many requests or usage of the service the attacker needs to retrieve or estimate the properties of the environment. We would address the following research questions:

RQ_1 - What kind of useful information can we hack from the software stack? (e.g. the computing power of the server executing the software system, the configuration of the software system, *etc.*) We want to ensure that the understanding of deep variability can not allow to retrieve a sensible information that can be exploited to find a security breach.

RQ_2 - How many calls do you need to have a good estimation of this information? As a hacker, how to efficiently request the system so you retrieve its properties in as few calls as possible?

List of publications

[1] L. Lesoil, M. Acher, A. Blouin, and J.-M. Jézéquel, "Deep software variability: Towards handling cross-layer configuration," in *VaMoS*, ser. VaMoS'21. New York, NY, USA: ACM, 2021. [Online]. Available: https://doi.org/10.1145/3442391.3442402

[2] L. Lesoil, M. Acher, X. Ternava, A. Blouin, and J.-M. Jézéquel, "The interplay of compile-time and run-time options for performance prediction," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, ser. SPLC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 100–111. [Online]. Available: https://doi.org/10.1145/3461001.3471149

[3] L. Lesoil, H. Martin, M. Acher, A. Blouin, and J.-M. Jezequel, "Transferring performance between distinct configurable systems : A case study," in *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3510466.3510486

[4] L. Lesoil, M. Acher, A. Blouin, and J.-M. Jézéquel, "Beware of the interactions of variability layers when reasoning about evolution of mongodb," in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 39–43. [Online]. Available: https://doi.org/10.1145/3491204.3527489

[5] L. Lesoil, M. Acher, A. Blouin, and J.-M. Jézéquel, "Input sensitivity on the performance of configurable systems : An Empirical Study," in *Journal of Software and Systems*, ser. JSS '23., 2023, p. 39–43. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0164121223000663?dgcid=rss_sd_all

[6] M. Acher, H. Martin, J. A. Pereira, L. Lesoil, A. Blouin, J.-M. Jézéquel, D. E. Khelladi, and O. Barais, "Feature Subset Selection for Learning Huge Configuration Spaces: The case of Linux Kernel Size," in *SPLC 2022 - 26th ACM International Systems and Software Product Line Conference*, Graz, Austria, Sep. 2022, pp. 1–12. [Online]. Available: https://hal.inria.fr/hal-03720273

[7] H. Martin, M. Acher, L. Lesoil, J. M. Jezequel, D. E. Khelladi, and J. A. Pereira, "Transfer learning across variants and versions : The case of linux kernel size," *IEEE Transactions on Software Engineering*, vol. 1, pp. 1–1, 2021.

[7] X. Tërnava, M. Acher, L. Lesoil, A. Blouin, and J.-M. Jézéquel, "Scratching the surface of ./configure: Learning the effects of compile-time options on binary size and gadgets," in *Reuse and Software Quality: 20th International Conference on Software and Systems Reuse, ICSR 2022, Montpellier, France, June 15–17, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 41–58. [Online]. Available: https://doi.org/10.1007/978-3-031-08129-3_3

[8] X. Tërnava, L. Lesoil, G. A. Randrianaina, D. E. Khelladi, and M. Acher, "On the interaction of feature toggles," in *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3510466.3510485

APPENDIX

9.3 The Phoronix Dataset

We present hereafter how we constitute the *Phoronix* dataset, extracted from the eponymous test suite. This dataset gathers performance of Stock Keeping Units (SKU) *i.e.,* hardware models on different software systems, coming along with a detailed description of their characteristics.

About the Phoronix Test Suite.

Table 9.2 – The different benchmarks currently available to compare performance of SKUs. Name the name of the benchmark. # SKUs the number of processors in the benchmark. # Suites the number of software systems in the benchmark. Raw Perf is equal to Yes if and only if the public benchmark provides raw data about SKU performance.

Name	# SKUs	# Suites	Raw Perf.	How do we use it?
AI-Benchmark	885	45	Yes	Perf. prediction
GeekBench	178	27	Yes	Perf. prediction
Phoronix	642	384	Yes	Perf. prediction
SPEC CPU	638	28	Yes	Perf. prediction
AMD Specifications	538	-	No	Fill missing data
CPU World	2500+	-	No	Add new features
Intel Specifications	535	-	No	Fill missing data
PassMark	3656	-	No	Add new features
Technical City	3516	-	No	Add new features
TechPowerUp	2434	-	No	Add new features
WikiChip	8000+	-	No	Add new features

The Phoronix Test Suite was created in 2008 by Michael Larabel. Its original purpose was to compare the difference of performance between SKUs. The *Phoronix* test suite offers a large choice of 384 suites: among them, `git`, `php`, `x264`, `numpy`, *etc.* Each suite executes an algorithm or a software system. During this execution, it measures a given performance property and stores its value. This suite is then executed on different hardware models, which results in a performance value for each SKU. All these values are available online on the OpenBenchmarking.org website, that we requested to create this dataset.

Dataset Construction

We build this dataset by cross-referencing several sources available online: the *Phoronix* database,[4] *Intel and AMD specifications*[5] and the *TechPowerUp* database.[6] Most features of the final dataset directly come from the Openbenchmarking website; to get them, we use web scraping on `https://openbenchmarking.org/` pages. For each platform, we store the returns of the `lscpu` and `cat /proc/cpuinfo` command lines informing *e.g.,* # Cores the number of cores, # Threads the number of threads per core, the clock frequency, *etc.* For each suite, if users have tested it on their processor, we retrieve the average performance of this suite on this SKU.[7]

[4] See the Openbenchmarking initiative at `https://openbenchmarking.org/`

[5] See *Intel* `https://ark.intel.com/` and *AMD* `https://www.amd.com/en/products/specifications/processors` *specifications*

[6] See the *TechPowerUp* database at `https://www.techpowerup.com/cpu-specs/`

[7] See `https://openbenchmarking.org/test/pts/x264-2.6.1` for an example of suite (here pts-x264) performance distribution, each line is a different SKU

Table 9.3 – The Phoronix Dataset Features. Name of the feature in the database. Description of the feature. Data Type how the feature is encoded in our database. A concrete Example of a feature set for a real SKU.

Name	Description	Data Type	Example
Model	Name of the Model	One hot encode	Intel Core i5-6500 CPU
Vendor id	SKU Producer	One hot encode	Intel
Family	Family of Processor	One hot encode	Intel Core-i5
Architecture	CPU Architecure	One hot encode	x86_64
CPU id Level	Code identifying CPU Level	Integer encode	1
CPU	CPU number	Numerical	4
# Threads	Number of threads per core	Numerical	1
# Cores	Number of cores	Numerical	4
# Siblings	Number of siblings	Numerical	4
Core per Socket	Core per Socket	Numerical	6
Socket	Socket Number	Numerical	1
Numanode	Non-Uniform Memory Access nodes	Numerical	1
CPU MHz	Clock Frequency (basis/min/max)	Numerical	3200/800/3200
BogoMIPS	Indicator of Processor Speed	Numerical	6384
Caches	Cache Size (L1d/L1i/L2/L3)	Numerical	32/32/256/6144
Price	Street Price ($)	Numerical	129
TDP	Thermal Design Power (Watt)	Numerical	65
Process	Lithography Process (nm)	Numerical	14
Year	Year of Release (YYYY)	Numerical	2015
Performance	384 workload performance	Numerical	run time of 8 secs - *ffmpeg*

We complete this data with *TechPowerUp* by joining the two databases and manually mapping the identifiers of the first to the second. Out of the *TechPowerUp* database, we extract the lithography process (nm), the Thermal Design Power (TDP) and the release date (YYYY) of our processor models. For the hardware present in *Phoronix* but not in *TechPowerUp*, we manually complete our search on the web.[8] Then, we look for the market price for each processor model and add it to the table with the help of online databases [9] or by manually searching for the missing prices on online shopping sites. All features of the *Phoronix* dataset are summarised in Table 9.3 : last column provides an example of such values for a real processor model.

Strengths and Limitation

The strength of the *Phoronix* dataset lies in the diversity of the software systems tested; 384 in total, which is unmatched in the state of the art with respectively 27 and 28 software systems for *GeekBench* and *SPEC*. The percentage of missing data is its biggest weakness: on average, only 10.3% of the 642 SKUs are measured per workload, which makes the performance prediction more challenging. All the datasets used in this paper are depicted in Table 9.2, as well as how we use them. Most of them are used to fill missing data and add new features to the existing characteristics of SKUs.

9.4 Mine Open Data to Predict Hardware Performance

This subsection is a replication of the first part of [214]. There exists many websites comparing processor properties. As a customer, we have access to plenty of technical details (*e.g.,* cpu model, number of cores, *etc.*) summarizing SKU properties, as those displayed in Figure 9.1. But this piece of knowledge does not provide us

[8] With the help of *Technical City* `https://technical.city/en/cpu`, *WikiChip* `https://en.wikichip.org/wiki/` and *CPU World* databases `https://www.cpu-world.com/CPUs/`

[9] See the *PassMark* price database `https://www.cpubenchmark.net/cpu_list.php` and the recommended prices of *Intel specifications* `https://www.intel.com/content/www/us/en/products/overview.html`

any number about the concrete usage of the processor executing a software. Instead, we want to transform these properties into some valuable insights helping when making your choice of hardware. As depicted in Figure 9.1, we address the following research question : **Is it yet possible to predict hardware performance based on its characteristics?**

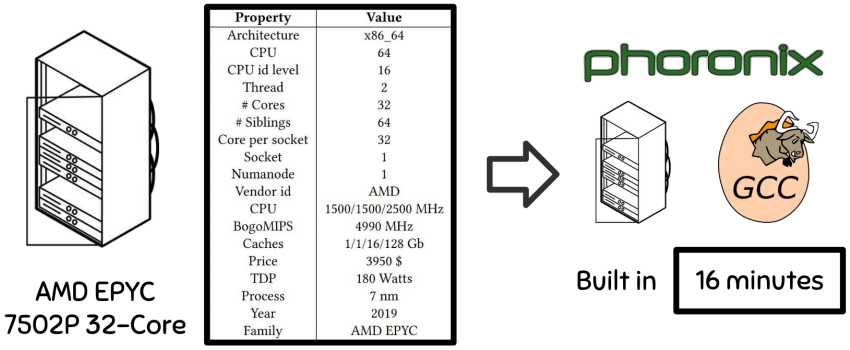


Figure 9.1 – Can we use SKU properties to predict their performance?

Protocol

To answer the previous questions, we train a supervised regression model on the Phoronix Dataset, taking as input the specifications of a SKU and predicting the performance value of a software system executed on this SKU.

Algorithms. To achieve this performance prediction, we rely on supervised learning techniques. Like the original paper [214], we consider **L**inear **R**egression and (deep) **N**eural **N**etwork. We add three other algorithms; **D**ecision **T**ree, **R**andom **F**orest and **G**radient **B**oosting tree [19]. We use the implementations of two python libraries : (1) Scikit-learn [215] for linear regression, decision tree, random forest and gradient boosting trees (2) Keras with Tensorflow [216] as backend for neural networks.

Benchmarks. We systematically test the previous algorithms on the systems of four benchmarks: *AI-Benchmark*,[10] *GeekBench*, *Phoronix* and *SPEC CPU 2006*.[11] In order to avoid biasing the results with software having not enough measurements, we do not include those with less than 150 different SKUs.

Metrics. To evaluate and quantify the effectiveness of performance models, we choose the Mean Absolute Error (MAE) [19] as metric.

Procedure & Code. For each system, we train the model with 90% of the SKUs and evaluate it on the remaining 10% *i.e.,* the test set. We repeat this process 20 times, using a repeated random subsampling validation [218] and report the average MAE value of the test set for this system. At this step, we obtain a prediction error for numerous triplets of algorithms, benchmarks and software systems; finally, we compute the average error per algorithm and benchmark. Procedures are freely available at `https://github.com/llesoil/poc_phoronix`

Evaluation

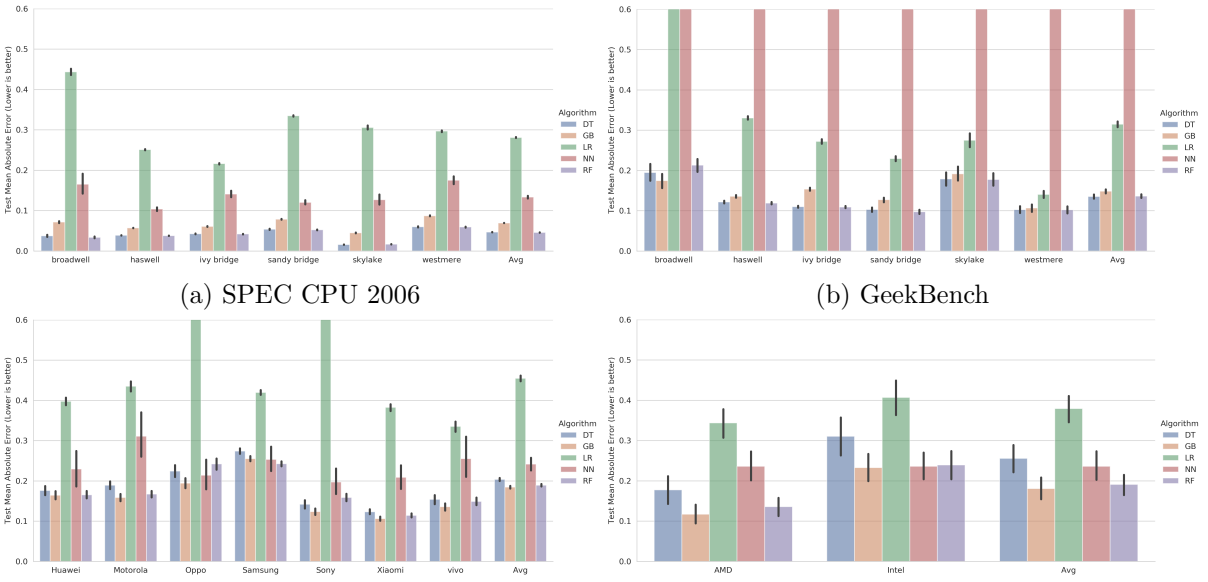
In Figure 9.2, we depict the resulting error when achieving this prediction with multiple algorithms. We also show error differences when predicting the performances on Intel and AMD's hardware platforms.

We were able to retrieve the results obtained by the original paper [214] for *Geekbench* and *SPEC CPU*; they are displayed in Figures 9.2a and 9.2b. For *GeekBench*, the neural network models do not converge with the architecture proposed in the original paper. We also test other variants of neural network architectures without success; since to the best of our knowledge, the code is not available and we got similar results with tree-based approaches, we do not fix this bug.

Compared to the original experiment, we get a slightly larger error in average for *Phoronix* (MAE = 0.24 in Figure 9.2d) and *AI-Benchmark* (MAE = 0.26 in Figure 9.2c) while it was at 0.11 and 0.19 for *SPEC* and

[10] See more about the *AI-Benchmark* at `https://ai-benchmark.com/` and in the related publication [217]

[11] For *GeekBench* and *SPEC*, we rely on the datasets used in the original study, see `https://github.com/Emma926/cpu_selection`



(a) SPEC CPU 2006

(b) GeekBench

(c) AI-Benchmark

(d) Phoronix

Figure 9.2 – Average error (y-axis) when predicting the performances of Phoronix workloads on new hardware platforms (x-axis) with different algorithms: Decision Tree (DT), Gradient Boosting tree (GB), Linear Regression (LR), Neural Network (NN) and Random Forest (RF).

Geekbench. But overall, these are still good results. We conjecture our results are sensible to (1) the choice and number of hardware properties (2) the number of measurements per workload and (3) the reliability of data *e.g.,* accuracy of the street price.

Neural networks perform worse in average for *Phoronix*, as shown in Figure 9.2d; with an MAE of 0.24, they do outperform linear regression (MAE = 0.38) but neither random forest (MAE = 0.19) nor gradient boosting tree (MAE = 0.18). Unlike the original paper, we do not apply the best of 20 neural network models on the test set but just apply a random model, which can partly explain the previous conclusions. However, we think this is the right methodological choice to make in order to fairly compare the different approaches. This may also be due to the number of SKUs in the training set; n = 206 and n = 617 unique performance measurements on average for *Phoronix* and *AI-Benchmark* while there are n = 37 158 and n = 6208 for *GeekBench* and *SPEC CPU*. This might be too small for the neural networks to really learn and detect the relationships between hardware typologies and their performance.

The tree-based approaches give the best results; not only do they have low errors in general with a MAE = 0.24 for Decision Tree, a MAE = 0.18 for Gradient Boosting Tree and 0.19 for Random Forest, but they achieve an accurate prediction in less time than deep neural networks, which is a good point to add to the original results. The performance prediction of AMD hardware platforms are more accurate than Intel's for each approach. This is encouraging and shows that the original work can be applied to AMD processors.

What matters when predicting SKU performance. Thanks to the feature importance of random forests, we extract the average percentage of information explained by each feature. Our results differ from those obtained by Lee *et al.* [14]: what matters the most to predict performance of hardware platforms is the L3 cache (14%), the number of siblings (9%), the number of CPUs (8%), the number of cores (8%), the year of release (8%), the number of core per socket (7%), the TDP (7%), the BogoMips (6%), the clock frequency (6%), the price (5%), the lithography process (5%) and the CPU level (4%). Other features have less than 2% importance.

9.5 What is my Hardware Model Worth?

In this section, we compare the performance of the different hardware platforms (or SKUs) of the Phoronix test suite. As the performance scales vary with workloads, we standardise [190] the performance in order to easily compare them between workloads. For 153 workloads, it is preferable to obtain a lower performance *e.g.,*

the workloads measuring an execution time: for them, we consider the opposite of the performance. Figure 9.3 represents a boxplot of the distribution of standardised performance values over all workloads. We discriminate according to processor range, as in Figure 9.1. If the standardised value is positive, then the corresponding performance value is above the average performance of SKUs and therefore the performance is considered good. Conversely, if the standardised value is negative, then the corresponding performance value is below average, and the performance is considered poor.

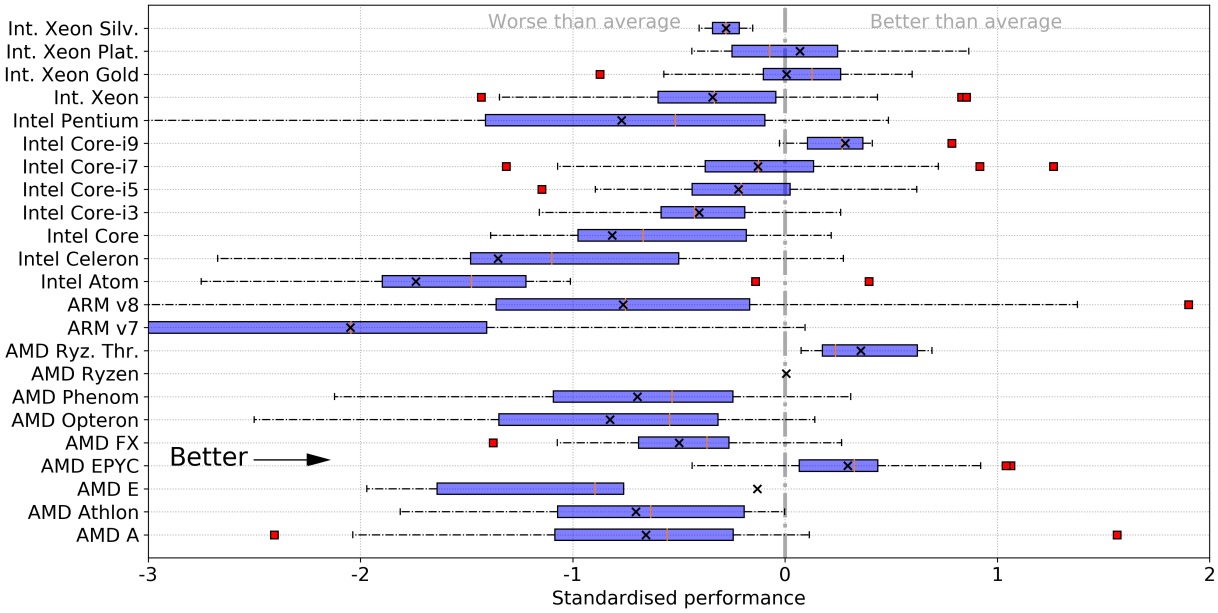


Figure 9.3 – Boxplot of standardised performances (x-axis) per family of processor (y-axis). Each boxplot represents the distribution over all Phoronix workloads. The greater the performance, the better. *"AMD Ryz. Thr."* stands for AMD Ryzen Threadripper.

In terms of pure performance, certain ranges of processors outperform the average by quite a margin, whatever the workload considered; among them, `AMD Ryzen Threapripper` (average standardised performance = 0.36), `AMD EPYC` (0.30) and `Intel Core i9` (0.28).

However, comparing raw performance is not always relevant, as each product has its own specificities and is intended for a very particular use case. For example, an `Intel Core-i3` (-0.40) is grafted onto an inexpensive entry-level computer, probably intended to perform fairly simple office tasks. It therefore hardly compares with the `Intel Xeon Platinum` (0.07) processors of the servers used in data centers or cloud infrastructures. Similarly, and not surprisingly, processors based on an `ARM v7` architecture obtain rather poor performance, with an average score of -2.04. This range includes in particular the `Cortex-M` sub-family, processors intended for embedded systems or light devices *e.g.,* `Arduinos`, designed to provide minimal service at low cost while ensuring low energy consumption.

Université de Rennes

DiverSE
Diversity-Centric Software Engineering

Titre : Variabilité Logicielle Profonde pour des Modèles de Performance Résilients

Mots clés : Variabilité Logicielle, Prédiction de Performance, Apprentissage Automatique

Résumé : Contexte. Les systèmes logiciels sont fortement configurables, au sens où les utilisateurs peuvent adapter leur compilation et leur exécution grâce à des configurations. Mais toutes ces configurations ne se valent pas, et certaines d'entre elles seront nettement plus efficaces que d'autres en terme de performance. Pour l'être humain, il est complexe d'appréhender et de comparer les différentes possibilités de configuration, et donc de choisir laquelle sera adaptée pour atteindre un objectif de performance. De récents travaux de recherche ont montré que l'apprentissage automatique pouvait pallier à ce manque et prédire la valeur des performances d'un système logiciel à partir de ses configurations.

Problème. Mais ces techniques n'incluent pas directement l'environnement d'exécution dans les données d'apprentissage, alors que les différents éléments de la pile logicielle (matériel, système d'exploitation, etc.) peuvent interagir avec les différentes options de configuration et modifier les distributions de performance du logiciel. En bref, nos modèles prédictifs de performance sont trop simplistes et ne seront pas utiles ou applicables pour les utilisateurs finaux des logiciels configurables.

Contributions. Dans cette thèse, nous proposons d'abord de définir le terme de variabilité profonde pour désigner les interactions existant entre l'environnement et les configurations d'un logiciel, modifiant ses valeurs de performance. Nous démontrons empiriquement l'existence de cette variabilité profonde et apportons quelques solutions pour adresser les problèmes soulevés par la variabilité profonde. Enfin, nous prouvons que les modèles d'apprentissage automatique peuvent être adaptés pour être par conception robustes à la variabilité profonde.

Title: Deep Software Variability for Resilient Performance Models of Configurable Systems

Keywords : Software Variability, Performance Prediction, Machine Learning, Deep Variability

Abstract : Context. Software systems are heavily configurable, in the sense that users can adapt them according to their needs thanks to configurations. But not all configurations are equals, and some of them will clearly be more efficient than others in terms of performance. For human beings, it is quite complex to handle all the possible configurations of a system and to choose among one of them to reach a performance goal. Research work has shown that machine learning can bridge this gap and predict the performance value of a software system based on its configurations.

Problem. These techniques do not include the executing environment as part of the training data, while it could interact with the different configuration options and change their related performance distribution. In short, our machine learning models are too simple and will not be useful or applicable for end-users.

Contributions. In this thesis, we first propose the term deep variability to refer to the existing interactions between the environment and the configurations of a software system, altering its performance distribution. We then empirically demonstrate the existence of deep variability and propose few solutions to tame the related issues. Finally, we prove that machine learning models can be adapted to be by-design robust to deep variability.

BIBLIOGRAPHY

[1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, 2005.

[2] F. J. Van der Linden, K. Schmid, and E. Rommes, *Software product lines in action: the best industrial practice in product line engineering.* Springer Science & Business Media, 2007.

[3] R. E. Lopez-Herrejon, J. Martinez, W. K. G. Assunção, T. Ziadi, M. Acher, and S. Vergilio, Eds., *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines.* Springer International Publishing, 2023. [Online]. Available: https://doi.org/10.1007/978-3-031-11686-5

[4] P. Clements and L. M. Northrop, *Software Product Lines : Practices and Patterns.* Addison-Wesley Professional, 2001.

[5] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques: Research articles," *Softw. Pract. Exper.*, vol. 35, no. 8, pp. 705–754, 2005.

[6] X. Tërnava, M. Acher, and B. Combemale, "Specialization of run-time configuration space at compile-time: An exploratory study," 2022. [Online]. Available: https://arxiv.org/abs/2210.14082

[7] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 421–432. [Online]. Available: https://doi.org/10.1145/2642937.2642990

[8] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 307–319.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," *Technical Report*, vol. 1, 01 1990.

[10] T. Thüm, C. Kstner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "Featureide: An extensible framework for feature-oriented software development," *Science of Computer Programming*, 2012.

[11] H. Martin, "Machine learning for performance modelling on colossal software configuration spaces," Ph.D. dissertation, Université de Rennes 1, 2021.

[12] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 129–139. [Online]. Available: https://doi.org/10.1145/1273463.1273482

[13] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proc. of ESEC/FSE'15*, 2015, p. 284–294. [Online]. Available: https://doi.org/10.1145/2786805.2786845

[14] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 185–194. [Online]. Available: https://doi.org/10.1145/1168857.1168881

[15] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Silicon Valley, CA, USA: IEEE Press, 2013, p. 301–311. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693089

[16] A. Samuel, "July 1959," *Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–29.

[17] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[18] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. Huang, "A tutorial on energy-based learning," *Predicting structured data*, vol. 1, no. 0, 2006.

[19] C. Molnar, *Interpretable Machine Learning*. Munich: Lulu. com, 2020.

[20] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. Lincoln, Nebraska: IEEE Press, 2015, p. 342–352. [Online]. Available: https://doi.org/10.1109/ASE.2015.45

[21] J. A. Pereira, H. Martin, M. Acher, J.-M. Jézéquel, G. Botterweck, and A. Ventresque, "Learning software configuration spaces: A systematic literature review," *ArXiv*, vol. abs/1906.03018, pp. 1–44, 2019. [Online]. Available: https://arxiv.org/abs/1906.03018

[22] 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.

[23] V. Nair, R. Krishna, T. Menzies, and P. Jamshidi, "Transfer learning with bellwethers to find good configurations," *CoRR*, vol. abs/1803.03900, pp. 1–11, 2018. [Online]. Available: http://arxiv.org/abs/1803.03900

[24] 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 *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, p. 497–508.

[25] 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*, ACM. New York: Association for Computing Machinery, 2018, pp. 71–82. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3236024.3236074

[26] P. Valov, J.-C. Petkovich, J. Guo, S. Fischmeister, and K. Czarnecki, "Transferring performance prediction models across different hardware platforms," in *Proc. of ICPE'17*, 2017, p. 39–50.

[27] 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. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106238

[28] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *IEEE Transactions on Software Engineering*, 2018.

[29] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 61–71. [Online]. Available: https://doi.org/10.1145/3106237.3106273

[30] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 49–60. [Online]. Available: https://doi.org/10.1145/3106237.3106256

[31] Y. Ding, A. Pervaiz, M. Carbin, and H. Hoffmann, "Generalizable and interpretable learning for configuration extrapolation," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 728–740. [Online]. Available: https://doi.org/10.1145/3468264.3468603

[32] K. Eggensperger, M. Lindauer, and F. Hutter, "Neural networks for predicting algorithm runtime distributions," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI'18. Stockholm, Sweden: AAAI Press, Jul. 2018, pp. 1442–1448.

[33] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.

[34] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated reasoning on feature models," in *Advanced Information Systems Engineering*, O. Pastor and J. Falcão e Cunha, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 491–503.

[35] 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," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1826–1867, Nov. 2017.

[36] J. Brown, "Some tests of the decay theory of immediate memory," *Quarterly Journal of Experimental Psychology*, vol. 10, no. 1, pp. 12–21, 1958. [Online]. Available: https://doi.org/10.1080/17470215808416249

[37] P. Temple, M. Acher, J.-M. A. Jézéquel, L. A. Noel-Baron, and J. A. Galindo, "Learning-based performance specialization of configurable systems," IRISA, Research Report, 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01467299

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

[39] 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. [Online]. Available: https://arxiv.org/abs/1704.00234

[40] ——, "Transfer learning for improving model predictions in highly configurable software," in *Proc. of SEAMS'17*, 2017, pp. 31–41. [Online]. Available: https://arxiv.org/abs/1704.00234

[41] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proc. of ESEC/FSE'17*, 2017, p. 61–71. [Online]. Available: https://doi.org/10.1145/3106237.3106273

[42] S. Kolesnikov, N. Siegmund, C. Kästner, A. Grebhahn, and S. Apel, "Tradeoffs in modeling performance of highly configurable software systems," *Software & Systems Modeling*, vol. 18, no. 3, pp. 2265–2283, Feb. 2018. [Online]. Available: https://doi.org/10.1007/s10270-018-0662-9

[43] 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.

[44] I. M. Murwantara, B. Bordbar, and L. L. Minku, "Measuring energy consumption for web service product configuration," in *Proc. of iiWAS'14*, 2014, p. 224–228. [Online]. Available: https://doi.org/10.1145/2684200.2684314

[45] P. Temple, J. A. Galindo, M. Acher, and J.-M. Jézéquel, "Using machine learning to infer constraints for product lines," in *Proc. of SPLC'16*, 2016.

[46] M. Weckesser, R. Kluge, M. Pfannemüller, M. Matthé, A. Schürr, and C. Becker, "Optimal reconfiguration of dynamic software product lines based on performance-influence models," in *Proc. of SPLC'18*, 2018.

[47] M. Acher, P. Temple, J.-M. Jézéquel, J. A. Galindo, J. Martinez, and T. Ziadi, "Varylatex: Learning paper variants that meet constraints," in *Proc. of VAMOS'18*, 2018, p. 83–88.

[48] P. Temple, M. Acher, B. Biggio, J.-M. Jézéquel, and F. Roli, "Towards adversarial configurations for software product lines," 2018.

[49] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Faster discovery of faster system configurations with spectral learning," *Automated Software Engg.*, vol. 25, no. 2, p. 247–277, Jun. 2018. [Online]. Available: https://doi.org/10.1007/s10515-017-0225-2

[50] P. Valov, J. Guo, and K. Czarnecki, "Empirical comparison of regression methods for variability-aware performance prediction," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 186–190. [Online]. Available: https://doi.org/10.1145/2791060.2791069

[51] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki, "Performance prediction of configurable software systems by fourier learning (t)," in *Proc. of ASE'15*, 2015, pp. 365–373.

[52] S. Kolesnikov, N. Siegmund, C. Kästner, and S. Apel, "On the relation of external and internal feature interactions: A case study," 2018.

[53] M. Couto, P. Borba, J. Cunha, J. a. P. Fernandes, R. Pereira, and J. a. Saraiva, "Products go green: Worst-case energy consumption in software product lines," in *Proc. of SPLC'17*, 2017, p. 84–93. [Online]. Available: https://doi.org/10.1145/3106195.3106214

[54] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proc. of ICMD'17*, 2017, p. 1009–1024.

[55] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based sampling of software configuration spaces," in *Proc. of ICSE'19*, 2019, p. 1084–1094. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00112

[56] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," *CoRR*, vol. abs/1606.06543, 2016. [Online]. Available: http://arxiv.org/abs/1606.06543

[57] M. Lillack, J. Müller, and U. W. Eisenecker, "Improved prediction of non-functional properties in software product lines with domain context," *Software Engineering 2013*, vol. 1, no. 1, pp. 1–14, 2013.

[58] M. Zuluaga, A. Krause, and M. Püschel, "e-pal: An active learning approach to the multi-objective optimization problem," *Journal of Machine Learning Research*, vol. 17, no. 104, pp. 1–32, 2016.

[59] B. Amand, M. Cordy, P. Heymans, M. Acher, P. Temple, and J.-M. Jézéquel, "Towards learning-aided configuration in 3d printing: Feasibility study and application to defect prediction," in *Proc. of VAMOS'19*, 2019. [Online]. Available: https://doi.org/10.1145/3302333.3302338

[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 *Proc. of NSDI'17*, 2017, p. 469–482.

[61] M. S. Saleem, C. Ding, X. Liu, and C.-H. Chi, "Personalized decision-strategy based web service selection using a learning-to-rank algorithm," *IEEE TSC*, vol. 8, no. 5, pp. 727–739, 2015.

[62] Y. Zhang, J. Guo, E. Blais, K. Czarnecki, and H. Yu, "A mathematical model of performance-relevant feature interactions," in *Proc. of SPLC'16*, 2016, p. 25–34. [Online]. Available: https://doi.org/10.1145/2934466.2934469

[63] S. Ghamizi, M. Cordy, M. Papadakis, and Y. L. Traon, "Automated search for configurations of convolutional neural network architectures," in *Proc. of SPLC'19*, 2019, p. 119–130. [Online]. Available: https://doi.org/10.1145/3336294.3336306

[64] A. Grebhahn, C. Rodrigo, N. Siegmund, F. Gaspar, and S. Apel, "Performance-influence models of multigrid methods: A case study on triangular grids," *Concurrency and Computation: Practice and Experience*, vol. 29, 2017.

[65] L. Bao, X. Liu, Z. Xu, and B. Fang, "Autoconfig: Automatic configuration tuning for distributed message systems," in *Proc. of ASE'18*, 2018, p. 29–40. [Online]. Available: https://doi.org/10.1145/3238147.3238175

[66] I. Švogor, I. Crnković, and N. Vrček, "An extensible framework for software configuration optimization on heterogeneous computing systems: Time and energy case study," *Information and Software Technology*, vol. 105, 2019.

[67] A. E. Afia and M. Sarhani, "Performance prediction using support vector machine for the configuration of optimization algorithms," in *Proc. of CloudTech'17*, 2017, pp. 1–7.

[68] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *ACM SIGPLAN Notices*, vol. 50. ACM, 2015, pp. 379–390.

[69] F. Duarte, R. Gil, P. Romano, A. Lopes, and L. Rodrigues, "Learning non-deterministic impact models for adaptation," in *Proc. of SEAMS'18*, 2018, p. 196–205. [Online]. Available: https://doi.org/10.1145/3194133.3194138

[70] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," in *Proc. of KDD'13*, 2013, p. 847–855.

[71] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proc. of ICSE'12*, 2012, pp. 167–177.

[72] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "Spl conqueror: Toward optimization of non-functional properties in software product lines," *Software Quality Journal*, vol. 20, no. 3–4, p. 487–517, Sep. 2012. [Online]. Available: https://doi.org/10.1007/s11219-011-9152-9

[73] D. Westermann, J. Happe, R. Krebs, and R. Farahbod, "Automated inference of goal-oriented performance prediction functions," in *Proc. of ASE'12*, 2012, p. 190–199. [Online]. Available: https://doi.org/10.1145/2351676.2351703

[74] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "White-box analysis over machine learning: Modeling performance of configurable systems," 2021.

[75] J. Alves Pereira, M. Acher, H. Martin, and J.-M. Jézéquel, "Sampling effect on performance prediction of configurable systems: A case study," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 277–288. [Online]. Available: https://doi.org/10.1145/3358960.3379137

[76] Y. Shu, Y. Sui, H. Zhang, and G. Xu, "Perf-al: Performance prediction for configurable software through adversarial learning," in *Proc. of ESEM'20*, 2020. [Online]. Available: https://doi.org/10.1145/3382494.3410677

[77] J. Dorn, S. Apel, and N. Siegmund, "Mastering uncertainty in performance estimations of configurable software systems," in *Proc. of ASE'20*, 2020, p. 684–696. [Online]. Available: https://doi.org/10.1145/3324884.3416620

[78] C. Kaltenecker, A. Grebhahn, N. Siegmund, and S. Apel, "The interplay of sampling and machine learning for software performance prediction," *IEEE Softw.*, vol. 37, no. 4, pp. 58–66, 2020.

[79] M. Weber, S. Apel, and N. Siegmund, "White-box performance-influence models: A profiling and learning approach," 2021.

[80] S. M. Muehlbauer, "Identifying software performance changes across variants and versions," in *Identifying Software Performance Changes Across Variants and Versions*. Passau: University of Passau, 2020. [Online]. Available: https://www.se.cs.uni-saarland.de/publications/docs/MAS+20.pdf

[81] X. Han and T. Yu, "Automated performance tuning for highly-configurable software systems," *arXiv preprint arXiv:2010.01397*, 2020.

[82] X. Han, T. Yu, and M. Pradel, "Confprof: White-box performance profiling of configuration options," in *Proc. of ICPE'12*, 2021, p. 1–8.

[83] P. Valov, J. Guo, and K. Czarnecki, "Transferring pareto frontiers across heterogeneous hardware environments," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 12–23. [Online]. Available: https://doi.org/10.1145/3358960.3379127

[84] F. Liu, N. R. Miniskar, D. Chakraborty, and J. S. Vetter, "Deffe: A data-efficient framework for performance characterization in domain-specific computing," in *Proc. of CF'20*, 2020, p. 182–191. [Online]. Available: https://doi.org/10.1145/3387902.3392633

[85] S. Fu, S. Gupta, R. Mittal, and S. Ratnasamy, "On the use of ML for blackbox system performance prediction," in *Proc. of NSDI'21*, 2021, pp. 763–784. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/fu

[86] H. Larsson, J. Taghia, F. Moradi, and A. Johnsson, "Source selection in transfer learning for improved service performance predictions," in *Proc. of Networking'21*, 2021, pp. 1–9.

[87] J. Chen, N. Xu, P. Chen, and H. Zhang, "Efficient compiler autotuning via bayesian optimization," in *Proc. of ICSE'21*, 2021, pp. 1198–1209.

[88] T. Chen, "All versus one: An empirical comparison on retrained and incremental machine learning for modeling performance of adaptable software," in *Proc. of SEAMS'19*, 2019, pp. 157–168.

[89] H. Ha and H. Zhang, "Deepperf: Performance prediction for configurable software with deep sparse neural network," in *Proc. of ICSE'19*, 2019, pp. 1095–1106.

[90] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," *Commun. ACM*, vol. 62, no. 11, p. 137–145, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3361566

[91] H. Ha and H. Zhang, "Performance-influence model for highly configurable software with fourier learning and lasso regression," in *Proc. of ICSME'19*, 2019, pp. 470–480.

[92] F. Iorio, A. B. Hashemi, M. Tao, and C. Amza, "Transfer learning for cross-model regression in performance modeling for the cloud," in *Proc. of CloudCom'19*, 2019, pp. 9–18.

[93] U. K. A. M. S. W. J. S. F. A. Porter, "Satune: A study-driven auto-tuning approach for configurable software verification tools," 2021.

[94] X. Teng, H. Pham, and D. R. Jeske, "Reliability modeling of hardware and software interactions, and its applications," *IEEE Transactions on Reliability*, vol. 55, no. 4, pp. 571–577, 2006.

[95] J. K. Fichte, M. Hecher, and S. Szeider, "A time leap challenge for sat-solving," in *Principles and Practice of Constraint Programming*, H. Simonis, Ed. Cham: Springer International Publishing, 2020, pp. 267–285.

[96] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," *SIGPLAN Not.*, vol. 51, no. 4, p. 545–559, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2954679.2872375

[97] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 59–73. [Online]. Available: https://doi.org/10.1145/3313808.3313817

[98] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici, "Unikernels everywhere: The case for elastic cdns," *SIGPLAN Not.*, vol. 52, no. 7, p. 15–29, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3140607.3050757

[99] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, p. 461–472, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2490301.2451167

[100] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, p. 255–265. [Online]. Available: https://doi.org/10.1109/ICSE.2017.31

[101] T. Thüm, A. v. Hoorn, S. Apel, J. Bürdek, S. Getir, R. Heinrich, R. Jung, M. Kowal, M. Lochau, I. Schaefer, and J. Walter, *Performance Analysis Strategies for Software Variants and Versions*. Cham: Springer International Publishing, 2019, pp. 175–206. [Online]. Available: https://doi.org/10.1007/978-3-030-13499-0_8

[102] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker, "Performance evolution blueprint: Understanding the impact of software evolution on performance," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. Eindhoven, NL: IEEE, 2013, pp. 1–9.

[103] J. Hasreiter, "Evolution of performance influences in configurable systems," in *Evolution of Performance Influences in Configurable Systems*. Passau: University of Passau, 2019. [Online]. Available: https://www.se.cs.uni-saarland.de/theses/JohannesHasreiterMA.pdf

[104] N. Werner, S. Apel, and C. Kaltenecker, "Energy and performance evolution of configurable systems: Case studies and experiments," in *Energy and Performance Evolution of Configurable Systems: Case Studies and Experiments*. Passau: University of Passau, 2019. [Online]. Available: https://www.se.cs.uni-saarland.de/theses/NiklasWernerMA.pdf

[105] K. Hoste and L. Eeckhout, "Cole: Compiler optimization level exploration," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 165–174. [Online]. Available: https://doi.org/10.1145/1356058.1356080

[106] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, and J.-H. Lee, "Automatic tuning of compiler optimizations and analysis of their impact," *Procedia Computer Science*, vol. 18, pp. 1312–1321, 2013. [Online]. Available: https://doi.org/10.1016/j.procs.2013.05.298

[107] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," *SIGPLAN Not.*, vol. 50, no. 6, p. 379–390, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2813885.2737969

[108] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong, "Performance analysis of private blockchain platforms in varying workloads," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, Jul. 2017. [Online]. Available: https://doi.org/10.1109/icccn.2017.8038517

[109] E. Coppa, C. Demetrescu, I. Finocchi, and R. Marotta, "Estimating the Empirical Cost Function of Routines with Dynamic Workloads," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 230:230–230:239. [Online]. Available: http://doi.acm.org/10.1145/2581122.2544143

[110] H. FathyAtlam, G. Attiya, and N. El-Fishawy, "Comparative study on CBIR based on color feature," *International Journal of Computer Applications*, vol. 78, no. 16, pp. 9–15, Sep. 2013. [Online]. Available: https://doi.org/10.5120/13605-1387

[111] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "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*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 395–404. [Online]. Available: https://doi.org/10.1145/1287624.1287681

[112] P. Leitner and J. Cito, "Patterns in the chaos—a study of performance variation and predictability in public iaas clouds," *ACM Trans. Internet Technol.*, vol. 16, no. 3, Apr. 2016. [Online]. Available: https://doi.org/10.1145/2885497

[113] U. Sinha, M. Cashman, and M. B. Cohen, "Using a genetic algorithm to optimize configurations in a data-driven application," in *Proc. of SSBSE'20*, 2020, pp. 137–152. [Online]. Available: https://doi.org/10.1007/978-3-030-59762-7_10

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

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

[116] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat," *Journal of artificial intelligence research*, vol. 32, pp. 565–606, 2008. [Online]. Available: https://www.aaai.org/Papers/JAIR/Vol32/JAIR-3214.pdf

[117] S. Falkner, M. Lindauer, and F. Hutter, "Spysmac: Automated configuration and performance analysis of sat solvers," in *Proc. of SAT'15*, 2015, pp. 215–222.

[118] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, and J.-H. Lee, "Automatic tuning of compiler optimizations and analysis of their impact," *Procedia Computer Science*, vol. 18, pp. 1312–1321, 2013. [Online]. Available: https://doi.org/10.1016/j.procs.2013.05.298

[119] M. Khavari Tavana, Y. Sun, N. Bohm Agostini, and D. Kaeli, "Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-gpu systems," in *Proc. of IPDPS'19*, 2019, pp. 664–674.

[120] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proc. VLDB Endow.*, vol. 2, no. 1, p. 1246–1257, aug 2009. [Online]. Available: https://doi.org/10.14778/1687627.1687767

[121] A. Mandhane, A. Zhernov, M. Rauh, C. Gu, M. Wang, F. Xue, W. Shang, D. Pang, R. Claus, C.-H. Chiang, C. Chen, J. Han, A. Chen, D. J. Mankowitz, J. Broshear, J. Schrittwieser, T. Hubert, O. Vinyals, and T. Mann, "MuZero with Self-competition for Rate Control in VP9 Video Compression," *arXiv:2202.06626 [cs, eess]*, Feb. 2022.

[122] Y. Wang, S. Inguva, and B. Adsumilli, "YouTube UGC dataset for video compression research," in *2019 IEEE 21st International Workshop on Multimedia Signal Processing (MMSP)*. United States: IEEE, Sep. 2019, pp. 1–5. [Online]. Available: https://doi.org/10.1109/mmsp.2019.8901772

[123] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, "Mitigating interference in cloud services by middleware reconfiguration," in *Proceedings of the 15th International Middleware Conference*, ser. Middleware '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 277–288. [Online]. Available: https://doi.org/10.1145/2663165.2663330

[124] P. B. Schneck, "A survey of compiler optimization techniques," in *Proceedings of the ACM Annual Conference*, ser. ACM '73. New York, NY, USA: Association for Computing Machinery, 1973, p. 106–113. [Online]. Available: https://doi.org/10.1145/800192.805690

[125] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization level in binary code from multiple architectures," *IEEE Access*, vol. 9, pp. 163 461–163 475, 2021.

[126] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," *ACM sigplan notices*, vol. 38, no. 5, pp. 77–90, 2003.

[127] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August, "Compiler optimization-space exploration," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* San Francisco California USA: IEEE, 2003, pp. 204–215.

[128] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS '09. USA: IEEE Computer Society, 2009, p. 1–12. [Online]. Available: https://doi.org/10.1109/IPDPS.2009.5161054

[129] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Comput. Surv.*, vol. 51, no. 5, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3197978

[130] Suprapto and R. Wardoyo, "Algorithms of the combination of compiler optimization options for automatic performance tuning," in *Information and Communication Technology*, K. Mustofa, E. J. Neuhold, A. M. Tjoa, E. Weippl, and I. You, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 91–100.

[131] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, and J.-H. Lee, "Automatic tuning of compiler optimizations and analysis of their impact," *Procedia Computer Science*, vol. 18, pp. 1312–1321, 2013. [Online]. Available: https://doi.org/10.1016/j.procs.2013.05.298

[132] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[133] J. Ballesteros and L. Fuentes, *Transfer Learning for Multiobjective Optimization Algorithms Supporting Dynamic Software Product Lines*. New York, NY, USA: Association for Computing Machinery, 2021, p. 51–59. [Online]. Available: https://doi.org/10.1145/3461002.3473944

[134] J. Dorn, S. Apel, and N. Siegmund, "Generating attributed variability models for transfer learning," in *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, ser. VAMOS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3377024.3377040

[135] F. Moradi, R. Stadler, and A. Johnsson, "Performance prediction in dynamic clouds using transfer learning," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. Arlington, VA, USA: IEEE, 2019, pp. 242–250.

[136] 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.

[137] R. Krishna, V. Nair, P. Jamshidi, and T. Menzies, "Whence to learn? transferring knowledge in configurable systems using beetle," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2956–2972, 2021.

[138] H. Martin, M. Acher, L. Lesoil, J. M. Jezequel, D. E. Khelladi, and J. A. Pereira, "Transfer learning across variants and versions : The case of linux kernel size," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[139] O. Day and T. M. Khoshgoftaar, "A survey on heterogeneous transfer learning," *Journal of Big Data*, vol. 4, no. 1, pp. 1–42, 2017.

[140] Q. Wu, H. Wu, X. Zhou, M. Tan, Y. Xu, Y. Yan, and T. Hao, "Online transfer learning with multiple homogeneous or heterogeneous sources," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 7, pp. 1494–1507, 2017.

[141] Y. He, X. Jin, G. Ding, Y. Guo, J. Han, J. Zhang, and S. Zhao, "Heterogeneous transfer learning with weighted instance-correspondence data," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34. Hilton New York Midtown, New York: AAAI, 2020, pp. 4099–4106.

[142] C. Park, M. Han, H. Lee, and S. W. Kim, "Performance comparison of gcc and llvm on the eisc processor," in *2014 International Conference on Electronics, Information and Communications (ICEIC)*. Kota Kinabalu, Malaysia: IEEE, 2014, pp. 1–2.

[143] Y. Li, Z. Zhang, F. Liu, W. Vongsangnak, Q. Jing, and B. Shen, "Performance comparison and evaluation of software tools for microRNA deep-sequencing data analysis," *Nucleic Acids Research*, vol. 40, no. 10, pp. 4298–4305, 01 2012. [Online]. Available: https://doi.org/10.1093/nar/gks043

[144] N. Boyko, O. Basystiuk, and N. Shakhovska, "Performance evaluation and comparison of software for face recognition, based on dlib and opencv library," in *2018 IEEE Second International Conference on Data Stream Mining Processing (DSMP)*. Lviv, Ukraine: IEEE, 2018, pp. 478–482.

[145] R. Emilsson, "Container performance benchmark between docker, lxd, podman & buildah," p. 23, 2020.

[146] J. A. Pereira, C. Souza, E. Figueiredo, R. Abilio, G. Vale, and H. A. X. Costa, "Software variability management: An exploratory study with two feature modeling tools," in *2013 VII Brazilian Symposium on Software Components, Architectures and Reuse*. Brasília, DF, Brazil: IEEE, 2013, pp. 20–29.

[147] R. Pohl, K. Lauenroth, and K. Pohl, "A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. USA: IEEE Computer Society, 2011, p. 313–322. [Online]. Available: https://doi.org/10.1109/ASE.2011.6100068

[148] M. A. Javidian, P. Jamshidi, and M. Valtorta, "Transfer learning for performance modeling of configurable systems: A causal analysis," 2019.

[149] R. Krishna, M. S. Iqbal, M. A. Javidian, B. Ray, and P. Jamshidi, "Cadet: A systematic method for debugging misconfigurations using counterfactual reasoning," 2020.

[150] J. Antunes, A. Bernardino, A. Smailagic, and D. Siewiorek, "Weighted multisource tradaboost," in *Pattern Recognition and Image Analysis*, A. Morales, J. Fierrez, J. S. Sánchez, and B. Ribeiro, Eds. Cham: Springer International Publishing, 2019, pp. 194–205.

[151] D. Wagelaar, "Towards context-aware feature modelling using ontologies," in *MoDELS 2005 workshop on MDD for Software Product Lines: Fact or Fiction?*, Oct. 2005, position paper.

[152] B. Desmet, J. Vallejos, P. Costanza, W. De Meuter, and T. D'Hondt, "Context-oriented domain analysis," *Modeling and Using Context*, pp. 178–191, 2007. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74255-5_14

[153] R. Hirschfeld, P. Costanza, and O. M. Nierstrasz, "Context-oriented programming," *Journal of Object technology*, vol. 7, no. 3, pp. 125–151, 2008.

[154] M. Raab and G. Barany, "Introducing Context Awareness in Unmodified, Context-unaware Software," in *ENASE 2017 - 12th International Conference on Evaluation of Novel Approaches to Software Engineering*, Porto, Portugal, Apr. 2017, pp. 1–8. [Online]. Available: https://hal.inria.fr/hal-01658620

[155] N. Gámez, J. Cubo, L. Fuentes, and E. Pimentel, "Configuring a context-aware middleware for wireless sensor networks," *Sensors*, vol. 12, no. 7, pp. 8544–8570, 2012. [Online]. Available: https://www.mdpi.com/1424-8220/12/7/8544

[156] Y. Brun, G. D. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software engineering for self-adaptive systems*. Springer, 2009, pp. 48–70.

[157] J. Pineau, "Building reproducible, reusable, and robust machine learning software," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2. [Online]. Available: https://doi.org/10.1145/3401025.3407941

[158] P. Delobelle, P. Temple, G. Perrouin, B. Frénay, P. Heymans, and B. Berendt, "Ethical adversaries: Towards mitigating unfairness with adversarial machine learning," *SIGKDD Explor. Newsl.*, vol. 23, no. 1, p. 32–41, may 2021. [Online]. Available: https://doi.org/10.1145/3468507.3468513

[159] M. Still, "Imagemagick," 2006.

[160] M. Heule, M. Järvisalo, and M. Suda, Eds., *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B. Finland: University of Helsinki, 2018, vol. B-2018-1.

[161] E. Incerto, M. Tribastone, and C. Trubiani, "Software performance self-adaptation through efficient model predictive control," in *Proc. of ASE'17*, 2017, pp. 485–496.

[162] M. J. Kilgard, "Anecdotal survey of variations in path stroking among real-world implementations," 2020.

[163] D. Maiorca and B. Biggio, "Digital investigation of pdf files: Unveiling traces of embedded malware," *IEEE Security Privacy*, vol. 17, no. 1, pp. 63–71, 2019.

[164] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform sampling of sat solutions for configurable systems: Are we there yet?" in *Proc. of ICST'19*, 2019, pp. 240–251.

[165] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[166] L.-N. Pouchet *et al.*, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, vol. 437, pp. 1–1, 2012.

[167] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[168] T. Nelson, "Technically-oriented pdf collection," 2014. [Online]. Available: https://github.com/tpn/pdfs

[169] M. Poess and C. Floyd, "New tpc benchmarks for decision support and web commerce," *SIGMOD Rec.*, vol. 29, no. 4, p. 64–71, Dec. 2000.

[170] S. Deorowicz, "Silesia corpus." 2003. [Online]. Available: http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

[171] R. A. Hummel, B. Kimia, and S. W. Zucker, "Deblurring gaussian blur," *Computer Vision, Graphics, and Image Processing*, vol. 38, no. 1, pp. 66–80, 1987.

[172] M. G. Kendall, *Rank correlation methods*. Griffin, 1948.

[173] J. D. Evans, *Straightforward statistics for the behavioral sciences.* Thomson Brooks/Cole Publishing Co, 1996.

[174] C. K. J. D. S. A. Stefan Mühlbauer, Florian Sattler and N. Siegmund, "Analyzing the impact of workloads on modeling the performance of configurable software systems," p. 255–265, 2023.

[175] S. Siegel, "Nonparametric statistics for the behavioral sciences." *The Journal of Nervous and Mental Disease*, vol. 125, p. 497, 1956.

[176] T. M. Oshiro, P. S. Perez, and J. A. Baranauskas, "How many trees in a random forest?" in *Machine Learning and Data Mining in Pattern Recognition*, P. Perner, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 154–168.

[177] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[178] T. Parr, K. Turgutlu, C. Csiszar, and J. Howard, "Beware Default Random Forest Importances," 2018, last access: july 2019.

[179] 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. New York, NY, USA: Association for Computing Machinery, 2015, p. 284–294. [Online]. Available: https://doi.org/10.1145/2786805.2786845

[180] J. Alves Pereira, H. Martin, M. Acher, J.-M. Jézéquel, G. Botterweck, and A. Ventresque, "Learning Software Configuration Spaces: A Systematic Literature Review (submitted)," Univ Rennes, Inria, CNRS, IRISA, Research Report, Jun. 2019. [Online]. Available: https://hal.inria.fr/hal-02148791

[181] J. Singer, G. Brown, I. Watson, and J. Cavazos, "Intelligent selection of application-specific garbage collectors," in *Proceedings of the 6th International Symposium on Memory Management*, ser. ISMM '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 91–102. [Online]. Available: https://doi.org/10.1145/1296907.1296920

[182] P. Lengauer and H. Mössenböck, "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*, ser. ICPE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 111–122. [Online]. Available: https://doi.org/10.1145/2568088.2568091

[183] S. Jayasena, M. Fernando, T. Rusira, C. Perera, and C. Philips, "Auto-tuning the java virtual machine," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. Hyderabad,India: IEEE, 2015, pp. 1261–1270.

[184] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 255–265. [Online]. Available: https://doi.org/10.1109/ICSE.2017.31

[185] D. Daly, *Creating a Virtuous Cycle in Performance Testing at MongoDB.* New York, NY, USA: Association for Computing Machinery, 2021, p. 33–41. [Online]. Available: https://doi.org/10.1145/3427921.3450234

[186] D. Daly, W. Brown, H. Ingo, J. O'Leary, and D. Bradford, "The use of change point detection to identify software performance regressions in a continuous integration system," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 67–75. [Online]. Available: https://doi.org/10.1145/3358960.3375791

[187] H. Abdi, "Z-scores," *Encyclopedia of measurement and statistics*, vol. 3, pp. 1055–1058, 2007.

[188] F. Nielsen, "Hierarchical clustering," in *Introduction to HPC with MPI for Data Science.* Springer, 2016, pp. 195–211.

[189] M. Müller, "Dynamic time warping," *Information retrieval for music and motion*, pp. 69–84, 2007.

[190] S. M. Dowdy and S. Wearden, *Statistics for research*, 1983.

[191] W. R. Zwick and W. F. Velicer, "Factors influencing four rules for determining the number of components to retain," *Multivariate Behavioral Research*, vol. 17, no. 2, pp. 253–269, 1982, pMID: 26810950. [Online]. Available: https://doi.org/10.1207/s15327906mbr1702_5

[192] T. M. Oshiro, P. S. Perez, and J. A. Baranauskas, "How many trees in a random forest?" in *International workshop on machine learning and data mining in pattern recognition*, Springer. Berlin, Germany: Springer, 2012, pp. 154–168.

[193] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[194] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, and T. Berger, "Towards a better understanding of software features and their characteristics: A case study of marlin," in *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VAMOS 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 105–112. [Online]. Available: https://doi.org/10.1145/3168365.3168371

[195] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Variability extraction and modeling for product variants," *Software & Systems Modeling*, vol. 16, no. 4, pp. 1179–1199, Jan. 2016. [Online]. Available: https://doi.org/10.1007/s10270-015-0512-y

[196] S. Schulze, M. Schulze, U. Ryssel, and C. Seidl, "Aligning coevolving artifacts between software product lines and products," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 9–16. [Online]. Available: https://doi.org/10.1145/2866614.2866616

[197] R. Di Cosmo and S. Zacchiroli, "Feature diagrams as package dependencies," in *Software Product Lines: Going Beyond*, J. Bosch and J. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 476–480.

[198] M. Acher, P. Collet, P. Lahire, and R. B. France, "Slicing feature models," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. USA: IEEE Computer Society, 2011, p. 424–427. [Online]. Available: https://doi.org/10.1109/ASE.2011.6100089

[199] M. Acher, P. Collet, P. Lahire, and R. France, "Composing feature models," in *Software Language Engineering*, M. van den Brand, D. Gašević, and J. Gray, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 62–81.

[200] A. Alourani, M. Bikas, and M. Grechanik, "Input-sensitive profiling," in *Advances in Computers*. Elsevier, 2016, pp. 31–52. [Online]. Available: https://doi.org/10.1016/bs.adcom.2016.04.002

[201] H. Wei, S. Zhou, T. Yang, R. Zhang, and Q. Wang, "Elastic resource management for heterogeneous applications on paas," in *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, ser. Internetware '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2532443.2532451

[202] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 415–432. [Online]. Available: https://doi.org/10.1145/3299869.3300085

[203] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *Information Fusion*, vol. 81, pp. 84–90, May 2022.

[204] Y. Gorishniy, I. Rubachev, V. Khrulkov, and A. Babenko, "Revisiting deep learning models for tabular data," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=i_Q1yrOegLY

[205] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[206] B. Thirey and R. Hickman, "Distribution of euclidean distances between randomly distributed gaussian points in n-space," *arXiv preprint arXiv:1508.02238*, 2015.

[207] P. Filip, P. Bednarik, L. E. Eberly, A. Moheet, A. Svatkova, H. Grohn, A. F. Kumar, E. R. Seaquist, and S. Mangia, "Different freesurfer versions might generate different statistical outcomes in case–control comparison studies," *Neuroradiology*, vol. 64, no. 4, pp. 765–773, 2022.

[208] M. Acher, "Reproducible science and deep software variability," in *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3510466.3510481

[209] F. Massonnet, M. Ménégoz, M. Acosta, X. Yepes-Arbós, E. Exarchou, and F. J. Doblas-Reyes, "Replicability of the ec-earth3 earth system model under a change in computing environment," *Geoscientific Model Development*, vol. 13, no. 3, pp. 1165–1178, 2020.

[210] T. Glatard, L. B. Lewis, R. Ferreira da Silva, R. Adalat, N. Beck, C. Lepage, P. Rioux, M.-E. Rousseau, T. Sherif, E. Deelman *et al.*, "Reproducibility of neuroimaging analyses across operating systems," *Frontiers in neuroinformatics*, vol. 9, p. 12, 2015.

[211] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[212] J.-b. Gao, B. Zhang, X.-h. Chen, and Z. Luo, "Ontology-based model of network and computer attacks for security assessment," *Journal of Shanghai Jiaotong University (Science)*, vol. 18, pp. 554–562, 10 2013.

[213] T. Laor, N. Mehanna, A. Durey, V. Dyadyuk, P. Laperdrix, C. Maurice, Y. Oren, R. Rouvoy, W. Rudametkin, and Y. Yarom, "Drawnapart: A device identification technique based on remote gpu fingerprinting," *arXiv preprint arXiv:2201.09956*, 2022.

[214] Y. Wang, V. Lee, G.-Y. Wei, and D. Brooks, "Predicting new workload or CPU performance by analyzing public datasets," *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 4, pp. 1–21, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3284127

[215] 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.

[216] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, and et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[217] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool, "Ai benchmark: Running deep neural networks on android smartphones," in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, 2018, pp. 0–0.

[218] W. Dubitzky, M. Granzow, and D. P. Berrar, *Fundamentals of data mining in genomics and proteomics*. Springer Science & Business Media, 2007.