Configuration-driven Software Optimization

Edouard Guégain

Université de Lille

École doctorale MADIS

Optimisation de logiciels par leur configuration

Configuration-driven Software Optimization

THÈSE

présentée et soutenue publiquement le 29 septembre 2023

pour obtenir le grade de

Docteur de l'Université de Lille

Mention Informatique

par

Edouard Guégain

Composition du jury

Président :	Anthony Cleve
Rapportrices :	Mireille Blay-Fornarino - Professeur, Université de Nice Anne-Cécile Orgerie - Directrice de recherche, CNRS
Examinateurs :	Hélène Coullon - Maître de conférence, IMT Atlantique Anthony Cleve - Professeur, Université de Namur
Directeur de thèse :	Clément Quinton - Maître de conférence, Université de Lille

Centre Inria de l'Université de Lille
CRIStAL — Spirals Research Team

Mis en page avec la classe thesul.

Acknowledgements

I would like to express my gratitude to all the people who helped me during my thesis and the writing of this manuscript.

First, I would like to thank my supervisor Clément Quinton for his support during this thesis. I would also like to thank Laurence Duchien for the inception of my thesis, and for directing me to Clément. I would also like to thank the persons I collaborated with during my thesis: Romain Rouvoy, Amir Taherkordi, Mathieu Acher, and Alexandre Bonvoisin. I would not have been able to publish without their technical and editorial help.

Finally, I thank the reviewers, Mireille Blay-Fornarino and Anne-Cécile Orgerie for agreeing to review this thesis.

Résumé

Le domaine du génie logiciel évolue rapidement, exposant les développeurs à une collection d'outils, langages, framework et paradigmes en constante croissance. Ainsi, concevoir un nouveau logiciel consiste à sélectionner des composants de cette collection, processus qui peut être assimilé à la création d'une configuration. Le critère pour configurer de tels systèmes est souvent la facilité de développement, ce qui peut conduire à des obésiciels sur-dimensionnés et consommateurs en énergie, loin des considérations frugales et environnementales. Cette dissertation s'intéresse donc à la possibilité d'utiliser la configuration des logiciels pour optimiser leur performance. Une attention spécifique est portée à la consommation énergétique et la taille de ces logiciels.

Un prérequis pour optimiser un système est de comprendre sa performance actuelle. Pour mieux appréhender ce sujet, cette dissertation commence par une analyse approfondie d'un logiciel configurable, au travers de plusieurs indicateurs de performance. Cette analyse a montré dans quelle mesure la configuration d'un système affecte ses performances, et quel impact chaque option a sur ses performances. Cette compréhension de la performance a permis de créer des configurations très performantes pour ce logiciel. Enfin, différents indicateurs de performance se sont avérés corrélés. Par conséquent, les objectifs d'optimisation peuvent être simplifiés en ignorant certains indicateurs redondants.

Ces configurations à haute performance ont été créées manuellement, ce qui n'est possible qu'avec un petit espace de configuration. Un algorithme a donc été créé pour automatiser ce processus, et ainsi l'adapter aux plus grands espaces de configuration. Cependant, optimiser une configuration en sélectionnant des options à haute performance n'est pas suffisant, car les options interagissent entre elles: par exemple, dans certaines situation, deux options à haute performance peuvent sous-performer quand elles sont associées. L'algorithme d'optimisation a donc été adapté pour prendre en compte ces comportements et en tirer profit. L'application de cet algorithme a un large ensemble de configurations a montré que la plupart d'entre elles atteignent une performance presque optimale, avec seulement un nombre limité de modifications.

Cependant, les contraintes de performance ne se limitent pas à un seul indicateur. La consommation énergétique d'un logiciel n'est pas toujours l'indicateur le plus impactant à optimiser. L'algorithme d'optimisation a donc été généralisé pour supporter l'optimisation de plusieurs indicateurs simultanément. Cet algorithme généralisé a été validé sur un couple d'indicateurs de performance: le temps d'exécution et la taille d'un logiciel. Cette validation a montré que la moitié des configurations analysées peut atteindre leur optimum local respectif en ajustant une seule option. Dans son état actuel, l'algorithme a été publié sous la forme d'un outil open-source nommé ICO.

Abstract

The field of software engineering is rapidly evolving, exposing practitioners to a growing collection of tools, languages, frameworks, and paradigms. Thus, designing a new software system consists in selecting components from this collection, which is akin to creating a configuration. Currently, the criterion to configure such systems often revolves around ease of development, which leads to oversized, power-hungry bloatware. This paradigm is not aligned with frugal or environmental concerns. Thus, this dissertation looks into the ability to leverage the configuration of software systems to optimize their performance. In particular, a specific focus is made on the energy consumption and the size of software systems.

A prerequisite to optimizing a system is to understand its current performance. To gain a comprehensive understanding, this dissertation begins with a thorough analysis of a configurable software system, considering multiple performance indicators. This analysis reveals that different configurations indeed yield varying performances, and these variations can be attributed to specific options. Leveraging insights gained from performance analysis enables the creation of high-performance configurations for this system. Furthermore, certain performance indicators proved correlated across configurations. Consequently, the optimization objective can be simplified by ignoring redundant performance indicators.

The creation of optimized configurations of this system was performed manually, which is only possible for small configuration spaces. To address larger configuration spaces, we propose an algorithm that outlines how to evaluate the performance of each option and subsequently improves a given configuration based on this performance data. However, optimizing a configuration by selecting high-performance options brought out limitations, as options can interact with each other: in some situations, pairing high-performance options may result in subpar performances. Similarly, low-performance options can prove unexpectedly efficient when paired together. We thus designed the algorithm to leverage such interactions. Applying this algorithm to a large set of configurations showed that most of them can reach near-optimal performances, with only a limited set of modifications.

However, performance constraints are not limited to a single performance indicator. Depending on the context, the energy consumption of a system may not be the single most impactful indicator to optimize for environmental concerns. Thus, the optimization algorithm must be generalized to accommodate multiple performance indicators. This improved algorithm is validated on a pair of performance indicators: the execution time and the software size. The key finding from this validation is that half of the assessed configurations can reach a local optimum by adjusting just one setting. To automate configuration optimization, this algorithm is implemented as an open-source tool called the ICO tool suite.

Contents

List of Figures

Part I

Introducing Notes

Chapter 1

Introduction

The field of software engineering has the specificity that anyone can quickly prototype, develop and publish new tools. Thus, the number of software components is rising, and as a matter of fact, it grows at an increasing rate. This trend can be observed through several software collections. GitHub reported the creation of 60 million new projects during the year 2020 [Git20], and 85 million new projects in 2022 [Git22]. On the Python package repository Pypi, 22 000 projects were created in 2016, 30 000 in 2017, and 40 000 in 2018 [BB19]. The Node package manager NPM hosted 800 000 projects in February 2019, 1 million in September 2019, and 1.3 million in April 2020. The website mvnrepository.com, centralizing Java packages for Maven from different repositories, reported an index of 2 million projects in 2014, 4 million in 2016, and more than 10 million in 2018. In 2023, it reaches 33.6 million projects [Mvn23]. While some of these tools are designed for end users, most are technical components designed to assist developers in their work. Thus, when designing new systems, software architects are exposed to a constantly growing collection of languages, libraries, frameworks, tools, and architectures. Furthermore, each of these solutions comes with its own, internal, settings.

Thus, the most pragmatic way to design a system is to use consensual technologies, which ensures that developers have some experience with them, and to stick to the default configurations of such technologies [Xu+15]. Indeed, the developer's main goals are to minimize the development time and maximize the user experience. The performance of the overall system is not a main priority, as computing units are ever more powerful, and the infrastructure can be scaled up in a cloud environment if the system appears too slow for end users. Some software optimization may be considered if a specific bottleneck or regression appears during development, but in most situations, neither the technologies nor their configurations are selected for their performance.

However, this paradigm attracts some critics. To simplify development, new projects are created using heavyweight frameworks and extensive libraries. Thus, such projects are oversized, or bloated from very early in their development. Such bulky projects tend to raise issues for the developers themselves. For instance, they can be substantially slower to compile or boot. Each dependency follows its own life cycle, with inconsistent updates. Furthermore, each dependency

is a liability, as they may create vulnerability issues. Another example of oversize projects can be found in web-based desktop applications. Instead of developing a discreet rich application that users can install, some projects create web-based frameworks that embed both a web server and a web client, for instance, Node.js and Chromium, both installed on the user's device. Such an approach makes development easier, as creating a user interface is easier in web-based technologies. Furthermore, such frameworks can be cross-platform and support several operating systems. However, they are resource-intensive and tend to hoard memory and increase energy consumption. This problem was discussed as early as 1995, with Wirth's law, stating that software is getting slower more rapidly than hardware is getting faster.

More recently, environmental concerns also arose. Releasing software with poor efficiency, while providing performance through cloud scaling or high-performance devices, means that computing units will perform unnecessary work, and thus energy will be wasted. Scaling an infrastructure in the cloud appears as a slider in a web interface for the operator, but through a chain of events, it can lead the cloud provider to increase the capabilities of its data center, which in turn would trigger the manufacturing of new devices, and manufacturing such devices has a strong environmental impact. Software design can also have unintended consequences on end-user devices: for instance, the decision to end support for older operating systems, to ease development, can push end-user to upgrade an otherwise perfectly able device. A poorly designed software may also drain a device's –non-replaceable– battery, which hurts its lifespan and in turn the lifespan of the device itself.

Both software and hardware have environmental impacts. Information and Communication Technologies (ICT) are expected to take an increasing part in global greenhouse gas emissions. The think tank The Shift Project estimated in 2018 that ICT is responsible for 3.5% of global greenhouse gas emissions [Pro21]. The collective GreenIT.fr estimated this number to be 3.8% the same year [Gre19]. Such numbers are higher than the share of civil aviation in GHG emissions, which was 2.5% in 2019 [dat20]. Furthermore, these values are expected to increase substantially, reaching between 5% and 7% of global GHG emissions by 2025, according to the Shift Project. The power usage of devices is not the only one responsible for ICT GHG emissions, as the impact of usage on emissions is highly dependent on the energy mix of the country of usage. Rather, a substantial amount of emissions are caused by the manufacturing of devices: for televisions, computers, and smartphones, up to 75% of emissions are caused by manufacturing. The manufacturing process has negative impacts beyond GHG emissions: high-tech devices rely on a collection of materials that are increasingly higher to extract, which can cause geopolitical tensions if, or when, shortages arise. The extraction of such materials is a highly polluting process, with impacts on water, air, soil, and the health of the workers. Finally, at the end of their life, or rather once obsolete, only 17% of devices will be recycled [Uni20]. The rest will end up as e-waste in landfills, which they will continue polluting. Therefore, green IT can not be limited to reducing the energy consumption of software: it is also relevant to maximize the lifespan of hardware, in order to limit device manufacturing and decommission.

The software has a role to play in that regard, as bloatware can drive users toward more recent and more powerful hardware.

This introduction highlights a duality in Information and Communication Technologies: on one hand, there is an ever-increasing amount of software components, and a trend to use ever more computing power to perform similar tasks, using ever more resources. On the other hand, we are facing new constraints in the energy, water, and materials that we use. Thus, an increasing amount of initiatives have emerged, both in academia and in the industry, to work toward a more "green", or rather frugal, technological context. Numerous lines of work are coexisting and aim at this same goal, either by challenging the need for technological solutions, or by optimizing such solutions. This dissertation falls into the line of optimization approaches and aims at maximizing the efficiency of software systems through their configuration.

1.1 Problem statement

The short opening above introduced the importance of creating software with limited resource usage, and that maximizes the hardware's lifespan. In this context, all software must be considered to run in a constrained environment, and thus all their aspects must be optimized. While numerous approaches support the creation of green-by-design software [Our+21; LH22], they are only relevant when the project is under development, or when practitioners can refactor the source code. Thus, support is still missing to optimize legacy or closed-sourced software. This dissertation focuses on leveraging configuration to maximize the performance of such systems. In particular, it addresses the following concerns:

Lack of understanding of large configuration spaces Most software systems exhibit some form of configuration. A configuration is a set of options, and when such options have only two states, selected or unselected, n available options can be composed into up to 2^n configurations. Since adding one option may double the number of configurations, configurations are thus subject to combinatorial explosion, which prevents users from exploring options. Thus, users tend to stick to default configurations or suboptimal ones.

Lack of feedback about configurations Beyond the use of default or suboptimal configurations, users may not be aware of the impact of their configurations on the performance of their system. In particular, there is no easy way to know how a given configuration performs compared to the set of all valid configurations. This problem is replicated at the level of options: there is no easy way to know how a given option impact performance, and thus to know how to improve a configuration.

Lack of support to optimize configurations While the search for optimal or near-optimal configurations is an active research topic, there is little work on optimizing existing configurations. Replacing an existing configuration with an optimal configuration may not be feasible

when a system is in production. In the industry, any configuration change may break the system, causing financial or contractual issues. Therefore, when leveraging configurations to improve performance, there is a need for support to optimize a system through the least changes to its configuration.

In this context, this dissertation aims to provide support to optimize existing configurations for software, *w.r.t* one or many performance indicators, while minimizing the number of changes to this configuration. The research goal of this dissertation can be stated as follows:

> Given a set of options for a configurable system and a set of rules defining the compatibility between such options, quantify how each option impacts the system's performance. Then, given a configuration, optimize it toward one or many optimization goals. The configuration creation process must adapt to constraints, such as imposed options, independently of their performances. The optimized configuration must remain as similar as possible to the initial configuration.

In addition to attempting to answer this research goal, all the tools and data produced during the realization of this dissertation are publicly available as open-source or open-data, to ensure reproducibility of our results and to facilitate future work on the field.

1.2 Organization

This dissertation is divided into three parts. First, the introduction notes specify the goals of this dissertation, and then a state-of-the-art of software variability and software performance are presented. Then, the interplay between variability and performance is analyzed in an empirical case study. And finally, this dissertation presents and evaluates algorithms and tools to optimize performance in variable systems. The detailed outline is presented below.

- Chapter 2 introduces the concept of variability and the advantages it offers in industrial design. Then, it discusses how variability can be implemented in software systems, and finally how software variability became a field of software engineering

- Chapter 3 introduces the notion of software performance, with a specific focus on the challenges related to energy consumption. Then, it dives into the specifics of performance in software product lines.

- Chapter 4 is a case study of the impact of configuration on the performance of the configurable software JHipster, a web application generator. This chapter also introduces early optimization techniques and their limitations.

- Chapter 5 introduces more evolved techniques to optimize the performance of a system through configuration. In particular, such techniques rely on interactions phenomenon to yield better results.
- Chapter 6 is a generalization of the approach presented above to support multi-objective optimization while respecting given requirements. This chapter also analyses how optimization algorithms navigate the configuration space, to provide insight into their capabilities and limitations.

Chapters 4, 5, and 6 leverage material accepted or in pending review in peer-reviewed journals or conferences. Specifically, the content of Chapter 5 was published at SPLC'21 and the content of Chapter 6 at CAISE'23.

Chapter 2

Software variability

This chapter introduces the concept of variability and the advantages it offers in industrial design. Then, it discusses how variability can be implemented in software systems, and finally how software variability became a field of software engineering. The purpose of this section is not to be an exhaustive state of the art, but rather to introduce fundamental notions that will be reused throughout this document.

2.1 Definitions

The variability of a system qualifies the ability for some of its properties to change. Some properties of a variable system can thus be defined as ranges of possible values, instead of fixed values. Each property defined as a range of values is a variability point of the system or an **option** of the system. Specifying a value for each option creates a **configuration** of the system, and the **configuration space** of a system is the set of all the possible configurations of this system. A variable system can also have rules regarding relationships between options, such as mutual exclusion or requirement between features. Thus, a configuration can be **valid** if it respects the system's constraints, or **invalid** otherwise. A variable system is thus composed of a common, non-variable part that is shared between all configurations, a set of options that appear in some configurations, and a set of constraints defining relationships between options.

As an abstract concept, variability can be found in a wide range of domains, but variability is especially valuable in industrial design. In order to satisfy a range of market segments, an ad-hoc solution is to create a new product for each segment, with its own production line. This solution leads to production and maintenance costs proportional to the number of existing products. However, by creating a single variability-rich platform, composed of a common part and a set of options, the range of market segments can still be satisfied but the costs of production and maintenance are proportional to the number of options, not the number of products: the common part and each option have their own production line, but the number of products (*i.e.*, combinations of options) is superior to the number of options: for instance, with 4 options either present or absent, 16 different configurations can be generated. Such variable systems are

referred to as **product lines**. The variability can be explicitly exposed to end-users, allowing customers to select options during the buying process (*e.g.*, cars, or laptops in some online stores). The manufacturer can also select a set of configurations and sell them as different products (*e.g.*, smartphones).

A common example of variability exposed to end users can be found in the automotive industry, as end-user can be offered to configure the vehicle they buy. A brand can offer several car bodies (*e.g.*, hatchback, sedan, station wagon, SUV), several engines (*e.g.*, petrol or electric), several designs for the exterior (*e.g.*, black or white) and the interior (*e.g.*, regular or luxury), and wheel options (*e.g.*, 16" or 17"). In addition, cars with petrol engines can be manual or automatic. Likewise, an extended battery pack can be available for electric vehicles. Finally, the car can receive comfort options such as air conditioning or a GPS. This set of options, and the constraints organizing them, represent a **model** of the variability of this system. While the system offers a limited number of 16 options, the configuration space is larger: these options can be combined into 512 configurations, *i.e.*, 512 different vehicles. This example will be reused throughout this chapter and referred to as the Automotive Industry Example.

2.2 Implementations of software variability

Software systems development faces the same challenges as any system: they must adapt to different user needs and different hardware requirements, hence they must be variable. This variability must be provided while minimizing the cost of production for developers. In software systems, the options are referred to as **features**. Apel *et al.* [Ape+16] define a feature as a characteristic or end-user-visible behavior of a software system. As discussed below, software variability can be implemented in widely different ways, with different maturity levels.

Clone and own Software and source code have the specificity that some operations, such as copying a project, have limited cost. Thus, it is possible to create a new configuration of software by copying the project and implementing the new behavior in the new project [Dub+13]. This type of situation happens when the variability was not anticipated. For instance, some software may have been developed for a single client. When a new client asks for the same software, but with limited changes, developers might want to create a copy of the existing software and then implement the changes on this copy. While the initial cost is very limited, the differences between the two versions of the software will increase over time, and improvements to the common part must be synchronized between the two versions. Thus, the cost of handling such projects will increase over time, and over the number of clients asking for their specific version. Numerous examples of clones and own can be found in the open-source community in the form of forks[EEM10]. For instance, LibreOffice was forked from OpenOffice in 2010, and both are still in active development as of 2023. Likewise, the Firefox browser has been forked numerous times, including Waterfox, Pale Moon, Basilisk, LibreWolf, and the Tor browser.

Parameters and configuration In this kind of variability, a single program contains all the features, but they are enabled or disabled by software-defined switches [WG04]. Such switches can be controlled by command-line arguments or configuration files. This implementation is widespread and expected by users. Command-line arguments are standardized[1] and the variability can be explained by a manual page[2] or a specific argument, *e.g.*, `--help`. Some programs use a file-defined configuration, stored on the user's system. While there is no specific convention to store such files, Unix-based systems tend to rely on "dotfiles", such as `.bashrc` or `.vimrc`. In graphic software, the configuration files may be manipulated by a "settings" or "preferences" window. Such systems are provided as a one-size-fits-all, as the software contains all the existing features. Thus, it is possible that most users do not use – or do not know – most options offered by the system.

Conditional compilation Some programming languages can ignore specific blocks from the source code during compilation, depending on parameters given to the compiler. Such blocks are defined by specific instructions in the source code, such as #ifdef structures in C and C++. Specific flags given to the compiler define which code blocks must be compiled or not. Thus, changing the compilation flags allows the creation of different programs from a single code base. A famous example of variability implemented by conditional compilation is the Linux Kernel which is composed of 19000 features [Pet+19]. While conditional compilation is sufficient for simple systems, it scales poorly to higher numbers of features, or when features are interdependent. This complexity is called "Ifdef hell", and requires specific techniques to be tackled [Pre19].

Plugins and packages Some software systems are designed to be extended by their community, to broaden their capabilities. However, such systems must be designed to handle extensions, which can introduce a development cost front. To leverage this cost, the system can be developed as a plugin loader, and all the default features are developed as official plugins and shipped with the software[Ach+14; NKN14]. Plugins can be developed both by the developers of the system and by external developers. Such an open ecosystem allows for the quick delivery of new software as an extension of an initial, pre-existing system. For instance, the database administration tool DBeaver is developed as a set of packages for the Eclipse platform. However, such an ecosystem of plugins can lead to dependency or incompatibility between plugins, or even between specific versions of plugins.

Service-oriented architecture Software systems can share some behaviors, even in unrelated businesses. For instance, most cloud-based applications will require some form of authentication or user management system. In order to open such behaviors to reuse, developers can

[1] `https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html`
[2] `https://www.kernel.org/doc/man-pages`

adopt the service-oriented architecture (SOA), where systems are designed as a collection of services, instead of monoliths [End+04]. The different services interact over the network through standard protocols, thus ensuring interoperability. An SOA can contain services created by the system developers, designed to be reused across different applications, but also off-the-shelf services, such as monitoring tools or cache services. A further implementation of SOA can be found in micro-services, where the business part of the application is divided into a collection of services running as independent servers [GL18]. While SOA allows the reuse of components, it is a coarse-grain variability and is not designed to handle per-customer requirements [Thü+14b].

Feature-oriented programming When a software system is designed to be variable, it is possible to develop it as a set of features and to define how such features interact with each other [AK09]. Then, a product generator can compose the features selected in a configuration into a product [BBR06; AKL09]. This approach relies on software engineering techniques, such as method overriding or the design patterns Strategy or Decorator, in order to change the behavior of the products based on the features they contain.

	Clone and own	Parameters and configuration	Conditional compilation	Plugins and packages	Service-oriented architecture	Feature-oriented programming
Scalability issues	*		*			
Single code base		*	*			*
Tailored products	*		*	*	*	*
Fine feature-granularity	*	*	*	*		*
Explicit separation of features				*	*	*
Early design choice				*	*	*
High upfront cost				*	*	*

Table 2.1: Main characteristics of the differences between variability implementations.

A software system can also support many of these implementations of variability at once. For instance, the Eclipse IDE handles users' settings by command line arguments and configuration files and can be extended by packages, both official and unofficial. Finally, each of these implementations of variability offers advantages and drawbacks, which are summarized in Table 2.1. This table highlights that providing tailored products without facing scalability issues requires an explicit separation of features, which must be an early design choice and thus comes with a high upfront cost.

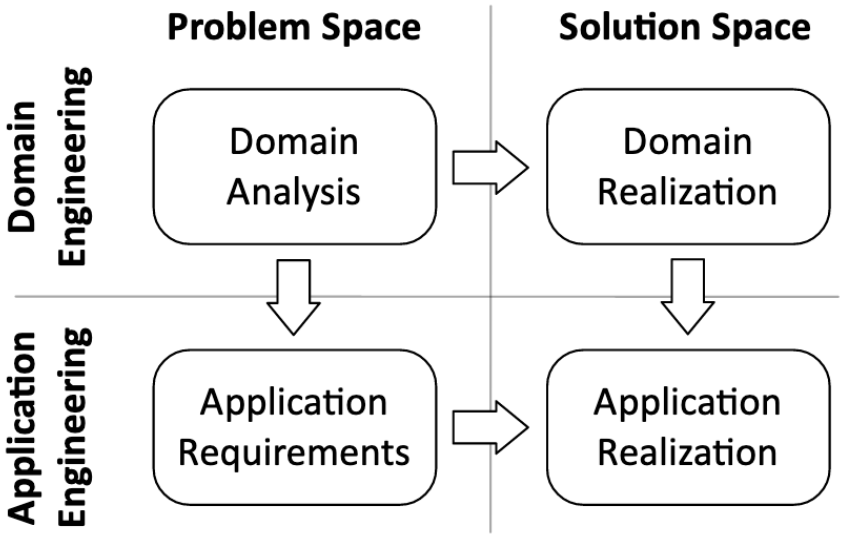


Figure 2.1: Overview of Software Product Line Engineering, from [Mei+17]

2.3 Software Product Lines and their Engineering

When the variability of a software system is planned during its design phase, specific processes can be implemented to tackle the complexity induced by variability. Designing such a system induces an upfront setup investment. However, this investment is expected to be leveraged in the long-term, through extensive reuse of components. A software system designed following such approaches is referred to as a **software product line** (SPL), and the practice of designing such systems is referred to as **software product line engineering** (SPLE). SPLE is composed of two major components: domain engineering, related to the creation of the variability of the system, and application engineering, related to the creation of `products` (*i.e.*, implementations of a configuration) corresponding to the needs of the customers. Figure 2.1 presents the interplay between these two aspects. Domain Engineering creates a model of the variability of the system during the domain analysis and implements the identified features during the domain realization. Application Engineering uses the variability identified during domain analysis to create the configurations of individual products and generate such products by assembling the features provided by the domain realization. The SPLE community offers tools and frameworks to manage software variability.

Feature models As an engineering practice, SPLE requires a standardized method to represent the architecture of the system under development. The variability of an SPL can be represented as a feature model, which describes the relationships between features, and thus how they can be composed into products. The relationships between features take the form of logical clauses, such as $A \vee B$, $A \vee B \wedge \neg(A \wedge B)$, or $A \Rightarrow B$. Such models can be visually

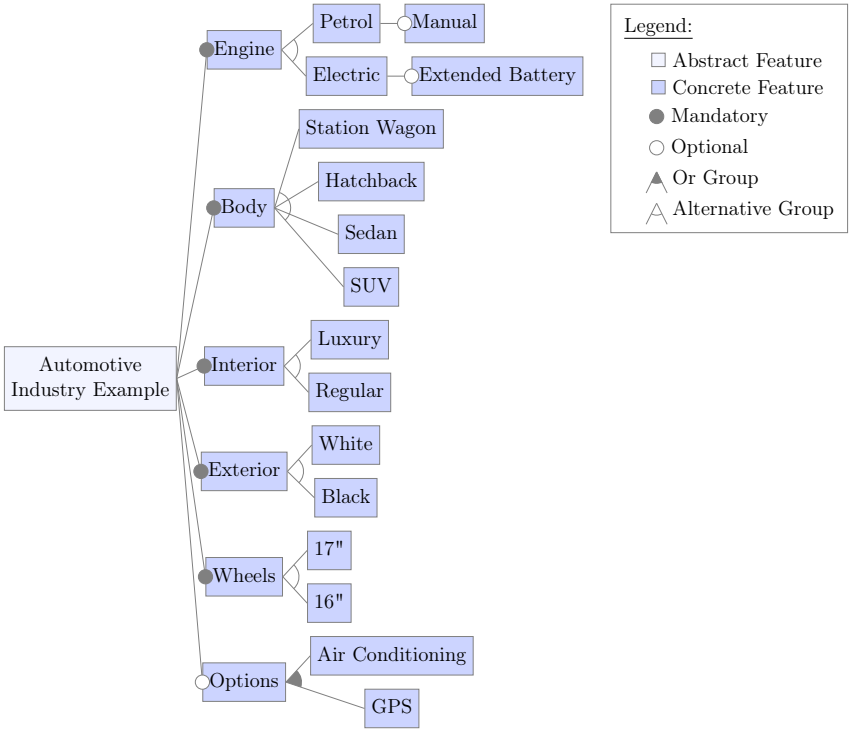


Figure 2.2: A representation of the feature model of the Automotive Industry Example.

represented as a tree of features and a set of remaining constraints not covered by the tree, *i.e.*, cross-tree constraints.

Figure 2.2 is a possible representation of the feature model of the Automotive Industry Example presented above. The mandatory features `engine`, `body`, `interior`, `exterior`, and `wheels` are marked by a black dot. The possible values for each of these features (*i.e.*, `petrol` or `electric` for `engine`) are marked as Alternative Groups to denote that a single implementation can be selected at once. A white dot on the features `Options`, `Manual`, and `Extended Battery` indicates that they are optional and can remain unselected. When `Options` is selected, both the `GPS` and `Air conditioning` features can be selected at once. They are thus in an Or Group.

Different extensions of feature models have emerged to cover additional use cases. Cardinality-based feature models allow the addition of cardinalities to features, thus turning the selection of a feature into the selection of several occurrences of this feature [QRD13]. Extended feature models allow the addition of attributes on features [BTR13]. Such attributes have a specific domain, *i.e.*, a range of possible values, and extra-functional features representing constraints between the attributes.

Sampling The number of valid products from an SPL can grow quickly due to combinatorial explosion. Such a high number of products can affect the execution time of automated validation, such as automatic testing, type consistency, or the evaluation of non-functional properties.

To alleviate this issue, different approaches rely on the evaluation of a subset of products sampled from the SPL configuration space [Thü+14a]. Sampling products induces new challenges of its own. The results of tests applied to the sampled products can only be generalized if the sampled products are representative of the configuration space. Thus, numerous approaches have emerged to generate representative sampling strategies. [Var+18] reported on 38 different sampling approaches in 2018. Such approaches have widely different implementations. A first approach is to manually select a set of configurations based on domain experts' knowledge [Al-+17; Tar+11]. Such approaches do not ensure uniformity of coverage of the source code or features. Moreover, as with any manual task, they scale poorly to larger configuration spaces. Thus, other approaches intend to automatically sample the configuration space based on specific objectives. Such objectives can be for instance the code coverage [Kim+10; Tar+11], ensuring that all the code is tested, or feature coverage [Al-+16b], ensuring that all features are tested at least once. Feature coverage can be extended to all pairs of features in pairwise analysis [Al-+16a; Mar+13; OMR10] or even tuples of larger size in t-wise analysis, where t is the size of the tuples[JHF11; Per+10]. Such coverage intends to cover feature interactions, *i.e.*, situations where the behavior or performance of specific features change depending on the presence of other features in the product. However, increasing the parameter t will increase the number of configurations generated, to a point where a new combinatorial explosion arises. The sampling process can also be parameterized with constraints such as the time allocated to generate products [Al-+16a], or a specific level of coverage [Tar+14]. The research on sampling methods is an open challenge, as larger feature models can contain more than 10000 features [Pet+19]. Without specific variability constraints, a pairwise algorithm applied to such a system can generate more than 50000000 products to test. Thus, more recent approaches are exploring different directions, such as machine learning [Per+21], to identify relevant configurations to test.

Analysis The complexity of an SPL grows proportionally to its feature and constraints count. Such complexity can hinder the assessment and validation of the system, such as the enumeration of valid configurations, or the detection of inconsistencies in the feature model or the source code. Hence, numerous approaches have emerged to automatically analyze SPLs. Such approaches are applied to the problem space (*i.e.*, the feature model and configurations) or the solution space (*i.e.*, the source code of features and the products) of an SPL. Analysis approaches on the problem space cover a large array of operations: detecting whether a feature model is void or not, *i.e.*, if any configuration can be selected at all, detecting if a partial or complete configuration is valid *w.r.t* a feature model, enumerating or counting all valid products, or detecting anomalies such as false-optional features, which are always selected, or dead features which can not be selected [BSR10; Mei+17]. Such approaches rely on propositional logic [Bou+23] or constraint programming [BTC05], or a combination of both, applied to the logical clauses of the feature model. Other approaches introduce specific algorithms to per-

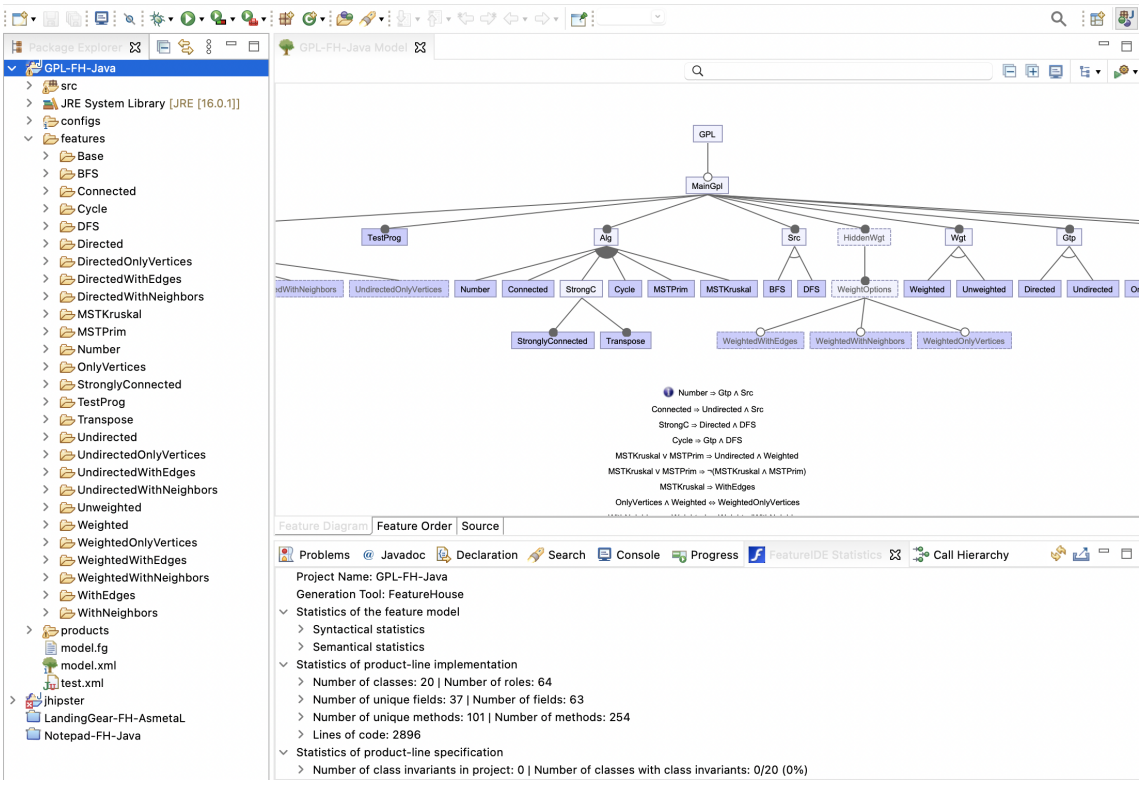


Figure 2.3: Screenshot of FeatureIDE.

form specific analysis operations[Hem08; VG09]. Other analysis approaches assess the solution space of an SPL relying on type checking, static analysis, and model checking. Such techniques are designed for analysis of a single program, *i.e.*, a single product, thus they can be applied individually to products generated by the product line, in addition to conventional software engineering tests such as unit tests or integration tests. As the number of valid products can grow exponentially with the number of features, the choice of products to analyze can be determined by the sampling methods presented above. However, such techniques have been extended to be variability-aware, and thus to be applicable across the features of an SPL. For instance, type checking can be applied on a specific product, but also on a specific feature or a complete SPL[Kol+13]. Similar generalizations can be found for static analysis[EKS20; Bod+13] and model checking[Cla+10]. Analyzing the SPL as a whole instead of individual products offers significant time savings, as the number of products is generally substantially higher than the number of features.

Integrated Development Environment As SPLE relies on a collection of tools and techniques, Integrated Development Environments (IDE) have emerged to centralize the development of such systems. For instance, Pure::Variant is a proprietary IDE developed by Pure-System. FeatureIDE is an open-source project started in 2004 that has since evolved with the SPLE community [Thü+14b]. Such IDE centralizes the different components of SPL. Assistance

is provided in the problem space with support for creating feature models and configurations. The solution space can be implemented with the help of different variability-oriented frameworks. For instance, conditional compilation can be implemented with the Munge preprocessor in C or the Antenna preprocessor in Java. Similarly, Feature-Oriented Programming can be implemented by frameworks such as AHEAD in Java, FeatureC++ in C++, or FeatureHouse in an array of languages including Java, C#, or Haskell. Such IDEs also embed analysis tools to provide insight and to ensure consistency between the different components of an SPL. These IDEs are central tools in the development of SPLs, thus numerous tools have been built on top of them[Fea17]. Figure 2.3 is a screenshot of FeatureIDE opened on the project GPL-FH-Java, an SPL designed to analyze graphs written in Java and using the FeatureHouse framework. On the left, the package explorer view from Eclipse lists the structure of the project. Inside the GPL-FH-Java project, a folder `configs` contains the existing configurations, created during the application engineering. The `features` folder contains the source code of each feature, implemented during the domain realization, inside their respective folder. The `src` folder, which usually contains the source code of Java applications, contains here the code generated by FeatureHouse during the application realization. As generated source code, it is not expected to be modified by developers. The file `test.xml` contains the unit tests and `model.xml` the feature model, which is open in the central view. This central view shows the feature model and allows its edition. Below, the FeatureIDE Statistics tab presents the results of different analysis methods on the feature model.

2.4 Summary

This section introduced the concept of variability and the different advantages it offers to both the creators and users of a system. Different possible implementations of variability in software were presented and compared. Finally, this section introduced how this software variability has evolved into an engineering practice, which provides numerous tools and methods to solve complex issues faced when managing variability.

Chapter 3

Software performance

This chapter introduces the notion of software performance, with a specific focus on the challenges related to energy consumption. Then, it dives into the specifics of performance in software product lines.

3.1 Definition

The Cambridge dictionary defines performance as *how well a person, machine, etc. does a piece of work or an activity.* In the case of software, this assessment can be made by measuring a set of **performance indicators**, each of them returning a specific value, or **metric**, quantifying how well the software executes its task.

Such performance indicators can be divided into two categories. The first category contains metrics related to the effectiveness of the software, *i.e.*, how successful it is at its purpose. They are referred to as **functional properties**, and the success rate is an example of such a metric. The second category contains metrics related to the quality of the system. This category contains objective metrics related to efficiency, *i.e.*, the number of resources used to execute its tasks. While time and memory usage are the most common efficiency metrics for software, other examples are energy consumption, CPU usage, or the size of software on storage drives. However, this category also contains more subjective criteria, such as reliability, security, or interoperability. Such metrics are referred to as **non-functional properties**.

Non-functional properties can be further divided into two categories, **static** and **dynamic** performance indicators. Performance indicators such as the size of the software on storage or the number of components in a software stack can be measured without executing the software. We will refer to such indicators as static performance indicators. By contrast, other indicators such as response time, energy consumption, or memory usage require the execution of the software, either in a benchmark or in production. Thus, we will refer to such indicators as dynamic performance indicators.

While measuring performance indicators at a specific time provides a set of metrics with no reference point, monitoring the evolution of performance indicators over time highlights the

evolution of performances. A software's performance can thus be monitored throughout its development to assess the evolution of its performance indicators. For instance, every new version of a server can be load-tested with tools such as Gatling to ensure that response times have not regressed. The performance of software can also be monitored over a single execution, for instance, to assign a performance issue to a specific code section, or to monitor the resource usage of a server running for extended periods.

The **optimization** of software refers to the practice of improving its non-functional properties. Optimization can be performed through numerous axes, such as reducing the algorithmic complexity, the number of files read or written, or the number of network requests. Software optimization serves different purposes and goals. Minimizing the latency of a server can improve the responsiveness of a system for its end-users. Optimizing the energetic performance of software running on battery-powered devices will extend the battery life of the device. Optimizing the energetic performance of a server can reduce the energy cost of running this server. Minimizing the CPU and memory usage of servers running on virtual machines can allow for virtual machine consolidation[DCM21], which in turn allows the unused servers to be turned off, reducing the cost of the infrastructure. Thus, optimization is relevant for all stakeholders of a system. However, beyond the low-hanging fruits, it is a difficult task and requires a specific time investment, without any guarantee of results.

3.2 Challenges related to power and energy optimization

The energetic performance of software systems is of high interest, due to its societal, financial, and environmental impact. The energetic performance of software systems can be measured as *power* or *energy* consumption. While power (P) measures the instantaneous consumption in Watts, energy (E, in Joules) reports an accumulated consumption over a given period [Pan+16]. The relationship between time, energy, and power is described by Formula 3.1.

$$E_{(J)} = P_{(W)} \cdot T_{(s)} \iff P_{(W)} = \frac{E_{(J)}}{T_{(s)}} \tag{3.1}$$

Optimizing for energy and optimizing for power have different purposes. When the analyzed software executes a specific task, requiring a specific time, it is relevant to optimize for energy, by reducing the amount of work performed to finish the task. By contrast, optimizing for power can lead to a false optimization. Indeed, power is the rate of energy consumption, thus, consuming energy at a slower rate (*e.g.*, by executing the same task over a longer period) will reduce the power usage of the software, but not its energy consumption. In this situation, time can not be used either to quantify the amount of work. In particular, time can be optimized by implementing parallelization. However, parallelization has an energetic overhead, and thus performing the same task in parallel can increase energy usage while reducing execution time. Thus, neither time nor power optimization alone ensures energy optimization in time-limited software. On the contrary, when the analyzed software is a server running for undefined periods

of time, neither energy nor time is measurable. It is thus relevant to optimize for power to consistently improve the energetic performance of the system.

While some performance indicators of software can easily be monitored, such as execution time or CPU usage, the energetic performance of software is still a challenging metric to monitor. The energy consumption of the hardware executing the software under analysis is distributed between different components, and in different proportions. Furthermore, a single device is often executing several programs at once, making the analysis of a specific program more challenging. Thus, the power and energy monitoring tools fall into one of two categories. In the first category are hardware tools, which provide coarse-grained readings of the energy consumption of a hardware system. The second category contains software tools, which provide a finer-grained reading of the energy consumption of a software system, at the cost of lower accuracy.

Hardware power meters Hardware power meters can be integrated between the power source and the components. For instance, Watts Up Pro is integrated between the outlet and the power supply of the device to monitor and logs the readings for posterior analysis. Power meters such as PowerMon 2 and PowerInsight can be integrated between the power supply of a device and its components to provide finer granularity. Such physical devices come at a cost, limiting systematic deployment in a large infrastructure. Such systems can also be applied to battery-powered devices, by plugging the power meter between the battery and the device. Such an approach is for instance used by Green Miner, paired with a USB monitoring tool, to gain insight into the energy consumption of Android applications.

Hardware power meters can also be implemented in the components themselves by their manufacturers, in an approach referred to as On-Chip power sensors. The RAPL device monitors the energy consumption of the CPU and DRAM of a device, for Intel CPUs since Sandybridge and for AMD CPUs since Zen. This device is able to provide high granularity about the energy consumption of subparts of the components, for instance by providing a separate reading for each core of multi-core CPUs. On Unix systems, the results of RAPL measures are accessible through the file system. Software interfaces provide easy access to such results. Specifically, JRAPL and Pyjoules are software packages, respectively in Java and Python, to embed RAPL readings into programs, thus allowing developers to monitor the energy consumption of a piece of code, up to the granularity of single lines of code. A similar On-Chip power sensor is available on some Nvidia GPUs, which can be queried through a C-based library.

Hardware power meters measure the energy consumption of a computer or a component rather than the one of a specific software [NRS13]. Thus, identifying the share of the software under study among the total energy consumption (*i.e.*, the *raw* energy consumption) is not straightforward. This issue can be addressed by sampling the CPU consumption for a specific duration (*e.g.*, one second) before launching the program under analysis. This measurement is defined as the *idle* power consumption P_{idle}, which refers to the average power consumption at

rest. The general idea is to remove the idle energy consumption from the raw measures to get rid of the external consumption. The resulting energy consumption, E_{net}, can then be associated with the software under study, as depicted in Figure 3.1 and presented in Equation (3.2).

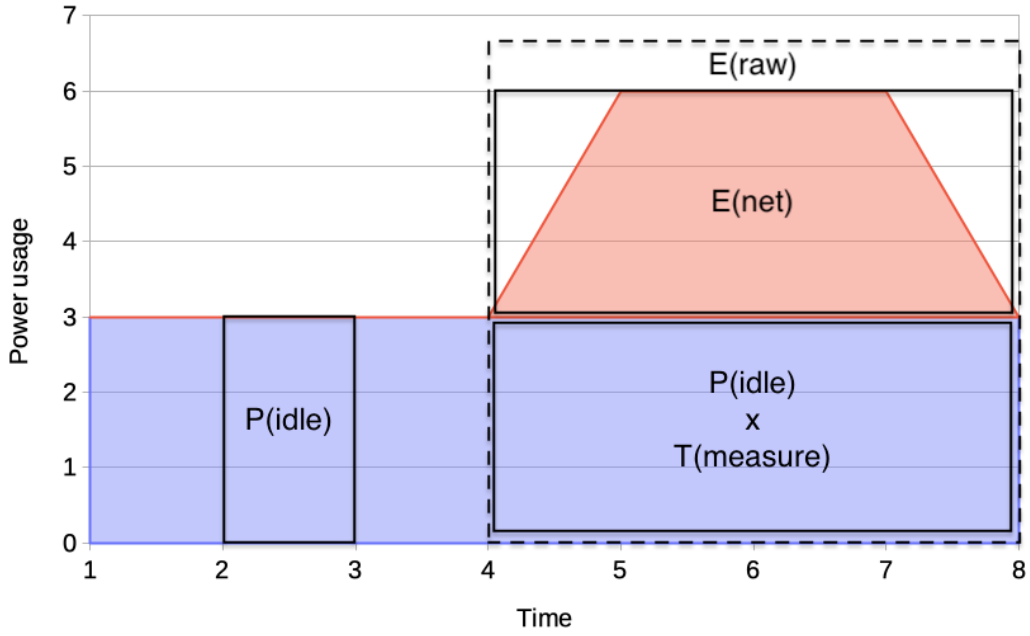


Figure 3.1: Raw *vs.* net energy consumption.

$$E_{net} = E_{raw} - (P_{idle} \times T_{measure}) \tag{3.2}$$

Software power meters While hardware tools are sufficient to estimate the energy consumption of a single software, they prove limited in more complex situations, such as when several software systems must be concurrently monitored on a single device. To tackle such issues, software power meters rely on models, such as empirical observations and statistical learning, to allocate the energy consumption measured by hardware power meters between the different software systems.

Tools such as PowerAPI [Bou+13; Col+18], SmartWatt [FRS20], and WattWatcher [LeB+15] use the results of hardware power meters in combination with a collection of indicators, such as the CPU frequency, cache misses, and events to build the power model. Such tools are able to provide insight into a specific process, or group of processes. They are composed of a sensor running on the measured system, and a server running on a separate hardware system. This design prevents the monitoring tool from altering the observed performance.

Other tools, such as Jolinar [NIB16] and Joulemeter [Jag+17], provide an estimation of the energy consumption of individual processes based on their resource usage, such as the CPU, RAM, or network usage. Such tools rely on a performance model of the hardware, such as the TDP of the CPU or the efficiency of storage drives, to provide their estimations.

Software power meters are necessary to understand the internal behavior of a system when several software components are running on the same device. However, as they rely on power models to provide their readings, their higher granularity comes at the cost of lower accuracy. Finally, they allow for the evaluation of software artifacts, such as functions and classes, but also functionalities by slicing the source code to isolate such functionalities [INB16]. Other studies focus on the identification of inefficient code ("energy hotspots" or anti-patterns) [NRS15; Per+20]. For instance, Pereira *et al.* [Per+20] reports on an approach to measure the energy consumption of a program facing different input workloads. They propose to model the consumption of methods by analyzing the consumption of the software and the number of calls for each method of this software. Once such hotspots are identified, energy consumption can be mitigated through refactoring operations, such as changes in the implemented data structures or algorithms [CAR17; CA18; Our+21].

3.3 Specificity of configurable software

For non-variable software, the assessment of performance is as straightforward as measuring performance indicators, and concluding that the resulting metrics represent the performance of the measured software. This process can also be applied to a specific product from an SPL, but the resulting metrics can not be generalized to other products from this SPL. Indeed, each feature of the SPL can affect the performance of the products, and thus different products, containing different features, may have different performances. For instance, variability implemented as parameters and configuration will go through different code blocks depending on user settings. In other approaches (clone and own, conditional compilation, plugin and packages, feature-oriented programming) the source code itself may differ depending on the configuration. Such differences in the executed code can impact the performance of variable software. Thus, the performance of configurable software can not be summarized as a simple measure of an indicator through a set configuration.

The performance of configurable software raises some specific challenges. In particular, the number of valid configurations can be very high, due to a combinatorial explosion between the features, thus excluding a systematic measurement of all valid configurations. Furthermore, the individual assessment of features is often impossible, due to the constraints of the feature model. For instance, considering an interface I accepting one of three implementations I_a, I_b or I_c, removing I_a requires the addition of I_b or I_c, thus preventing an isolated measure of I_a. Finally, an additional level of complexity is introduced by the phenomenon of **feature interactions**. Different features of a system can interact with each other, even if they have no functional or technical relationship. For instance, in the Automotive Industry Example presented above, the system offers air conditioning as an option. In electric vehicles, air conditioning is powered by the battery pack, not the engine. Thus, air conditioning in electric vehicles can impact the autonomy of the vehicle but not its power. However, in petrol-powered vehicles, air conditioning

relies on energy generated by the engine, thus impacting the vehicle's power. In that regard, the air conditioning option interacts with the petrol engine option in a way that impacts its performance, but not with the electric engine.

Thus, understanding the performance of a configurable system is a non-trivial task. However, it opens new opportunities, such as the ability to predict the performance of products without benchmarking them, and to create high-performing products. The performance of highly configurable systems has thus been widely studied in the last decade: Pereria *et al.* reports on 69 publications in the field between 2005 and 2019 [Per+21]. Research is mainly focused on the following three areas: *(i)* performance prediction, intending to estimate the performance of a configuration without actually measuring it, *(ii)* performance optimization, to generate optimal configurations of a system, and *(iii)* recommender systems, to assist developers during the feature selection process.

Performance prediction. Performance prediction approaches have been proposed by many researchers [Kal+20; Guo+13; Sie+15]. The aim of such approaches is to build an estimated performance model of the system's features. These approaches rely on machine learning techniques to infer performance data from a sample of configurations. One of their main objectives is to detect feature interactions – as they can have a significant impact on performances, and provide more accurate prediction than approaches that do not consider such interactions. Relying on such performance models, SPL Conqueror [Sie+12a; Sie+15; Sie+13] predict the performance of configurations as the sum of the impact of each feature on performances.

Other approaches rely on Classification and Regression Trees (CART), a statistical learning method used to predict the performance of products based on the feature they contain. CART divides the sampled configurations into clusters of similar performances [Per+21]. Then, given a new configuration, CART assigns it to the most similar cluster *w.r.t* to features and estimates the performances of this configuration as the performance of this cluster. CART prediction models can be improved over time by iteratively adding new configurations, and can thus provide early results with a sample of a limited size, and more accurate results as the sample size increased [Nai+20; Sar+15]. This approach is widely used, and numerous implementations or extensions have been proposed [Guo+18; Tem+17; MBM14].

Finally, some approaches attempt to assess the impact of the inputs on the performance of a program. Thus, the variability of the input data or the workload can be analyzed in addition to the variability of the software itself [Les+23]. Such approaches aim at selecting the best features based on specific properties of the input or the workload. The variability of the inputs can be discretized and added to the variability of the SPL itself [LME13]. Finally, input sensitivity can be applied to the adaptation of software to changes in their environments, as studied for instance in Dynamic Software Product Lines (DSPL) [Met+20; Göt+20].

Performance optimization. Many approaches have been proposed to address performance optimization for configurable systems. Such approaches strive to locate optimal or near-optimal

configurations *w.r.t* some performance indicators. Several studies provide deterministic approaches [ŠCV19; Ola+12; Nai+20] to tackle this challenge. Other authors like Hierons *et al.* rely on genetic algorithms to minimize the number of measurements needed to optimize configurations [Hie+16] or leverage the performance predictions methods discussed above [Sie+12b]. However, such approaches intend to find near-optimal configurations, rather than optimizing existing configurations, and thus they do not take an initial configuration into consideration. For instance, Nair *et al.* [Nai+20] start their optimization process from a random configuration. To the best of our knowledge, only Soltani *et al.* takes user's preferences into consideration when optimizing a configuration but yet does not support the optimization of pre-existing configurations [Sol+12].

Recommender systems. In recent years, we observed an increased interest in studies applying tools and approaches to assist users during the configuration of their systems. Such systems provide recommendations to the users based on their functional needs. For instance, Pereira *et al.* propose a visual recommender system [Per+16] based on proximity and similarity between features. Similarly, Zhang *et al.* [ZE14] use dynamic profiling and analyze the stack trace of the system to locate features that can be changed without altering the functional behavior of the system. Other approaches, such as [Met+22] or [HPF19], aim at updating the configuration of the system while it is running, in order to adapt it to the evolution of its environment. To the best of our knowledge, no recommender system provides suggestions based on both functional and performance considerations.

All these techniques allow for building performance models, and for creating near-optimal configurations. However, a current limitation is not addressed: in industrial settings, changing a configuration is a critical operation. It can lead to errors and substantial downtimes which in turn can cause financial losses. Thus, when optimizing the performance of a configuration, it is fundamental to minimize the difference between the initial and the improved configuration. Thus, specific work is needed to improve the performance of configurations with limited alterations.

3.4 Summary

This chapter introduced the concepts of software performance and optimization. A specific focus was made on the energetic performance of software, in particular the difference between energy and power, and the challenges of measuring such indicators. The different existing methods to measure, predict, or optimize configurable software were presented.

Finally, several approaches have been proposed to deal with the energetic performance of configurable software systems. In particular, this concern has been addressed by relying on *dynamic* SPL to reconfigure the system depending on context changes and ensure it continues meeting energetic requirements [Mun+19a; HPF19; DKB14]. These approaches take feature

interactions into account, but they rely on an exhaustive detection of such interactions which remains error-prone. Couto *et al.* also proposed different techniques to evaluate energy consumption in SPL using static analysis [CFS21; Cou+17]. These techniques estimate the energy consumption of features in the worst-case scenario by analyzing the source code of features to deduce the energy consumption of products and do not take feature interactions into account. Such approaches demonstrate an interest in the field, but the existing work is still limited.

Part II

Case study on Variable Software Performance

Chapter 4

On the Interplay Between Performance Indicators in Configurable Software: The JHipster Case Study

Modern applications have strong performance constraints. Stakeholders expect on the one hand to maximize user satisfaction, for instance with low latency, and on the other hand to minimize hosting costs, by reducing the CPU and RAM footprint or the energy consumption. Thus, developers are strongly incentivized to rely on software technologies that meet these performance goals [Our+20]. But which are these technologies? Assessing the performance of software components is a tedious and error-prone process: performances can be impacted by a multitude of factors, such as the environments in which the software is running and the workload that it executes. Furthermore, when the software under development offers is configurable (*i.e.*, variable), the number of valid configurations tends to increase exponentially. In this context, developers tend to leverage only a subset of options or stick to the default configuration [GTQ23a].

Therefore, there is a need to better understand the overall performance of configurable software: Which performance indicators are relevant? How to benchmark them? Can the benchmarks be simplified? How to create high-performing configurations? In this chapter, we exhaustively analyze the performance of a configurable software system, to better understand how to create high-performance configurations of this system. In addition, we provide the complete dataset of performance indicators, to facilitate future work related to performance in configurable software.

Such work requires the selection of a relevant configurable system to analyze. In particular, this system must be preexisting, open-source, documented, automatable, and offer different metrics to measure. The web stack generator JHipster has been selected as it fits all of these requirements, in addition to being used in the industry. JHipster has already been the subject of research papers [Hal+19; Hal+17; MPF22; Hor+21]. In particular, [Hal+19] is a comparison

between sampling methods. [Hal+17] is an attempt to build a software product line to deal with the variability of JHipster. Finally, [Hor+21; MPF22] rely on JHipster to validate their approaches, respectively to analyze and to reason on feature models. Such work was not focused on the performance of configuration or on their optimization. To the best of our knowledge, they did not execute any configuration of JHipster.

4.1 Experiment Setup

Our goal is to investigate the performance of a real-world highly-configurable software system, regarding various performance indicators. In particular, we aim to answer the following research questions:

RQ 1: How does the system's performance vary from one configuration to the other? Variability is known to impact configuration performances [Per+21]. We want to assess how the configuration of a system impacts its performance, *i.e.*, to what extent the performance of this system varies between configurations.

RQ 2: How are the different performance indicators related? Collecting a large set of performances is a tedious and error-prone process. Thus, if some performance indicators are strongly correlated, and if this correlation is known and understood, the collection of some performance indicators may prove unnecessary.

RQ 3: Do the different options of the system impact the performance of configurations in different ways? JHipster proposes different technologies to implement technical needs. Such options may exhibit different performances. Thus, analyzing the performance of configurations *w.r.t* options they contain may highlight how such options can impact performances.

RQ 4: Is it possible to drive the selection of high-performance configurations based on the performance of individual features? If different options exhibit different performances, then developers can leverage such knowledge to create highly efficient systems.

To conduct our empirical study and answer our research questions, we studied the JHipster system, an open-source development platform that facilitates the creation, development, and deployment of modern web applications and microservices. JHipster is built on top of popular technologies such as Angular, React, Spring Boot, and Docker and offers developers a set of tools, generators, and templates that automate the development process, from authentication to database configuration and deployment. JHipster is designed to quickly generate a complete, fully-functional application – either through an interactive command-line tool or via configuration files – which can then be easily customized to fit the developer's specific needs. We chose JHipster as it exhibits high variability and is known for its large industrial adoption and its automation capabilities. Moreover, JHipster has already been the subject of several research papers [Hal+19; Hal+17], making its variability model partly available.

JHipster offers an extensive range of options that can be combined into more than 100,000

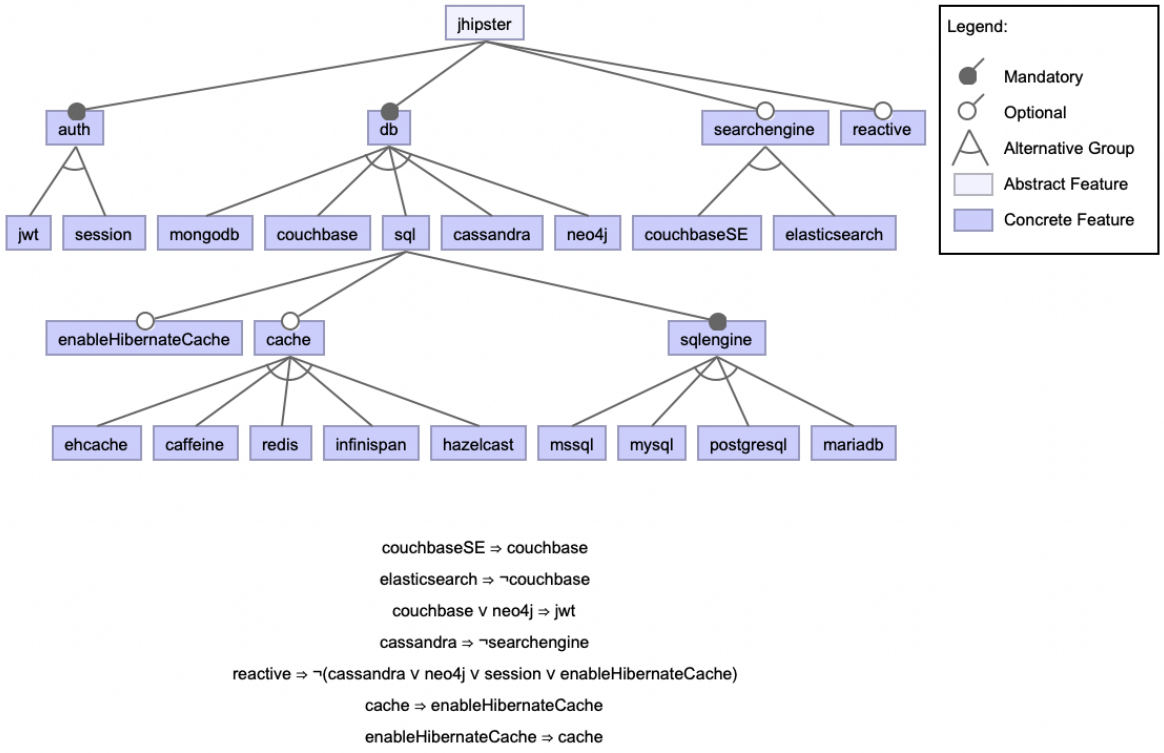


Figure 4.1: The variability of JHipster explored in this chapter.

configurations [MPF22]. In this chapter, we do not test exhaustively all JHipster configurations as our analysis consists in assessing the performance of the software stack in production. That is, our evaluation excludes options related to *(i)* building tools such as Maven or Gradle, *(ii)* development databases (H2disk, H2mem, etc.) and *(iii)* virtualization and orchestration tools, as we assume such options do not relate to nor impact software performance in production. As depicted by Figure 4.1, the JHipster variability under study covers options related to the database system, the cache system, the search engine, the authentication method, and the usage of reactive processing. This feature model was built based on the official documentation of JHipster, by extending and updating previously published JHipster feature models and through source-code mining on the JHipster GitHub repository[3]. This model defines 118 valid configurations, which were all ran and tested to ensure that they actually work and that the selected options are properly included.

[3] `https://github.com/jhipster`

4.2 Methodology

To answer our research questions, we empirically assessed the performances of all 118 JHipster configurations. This section introduces the performance indicators that were monitored and the experimental protocol we followed to yield our results. All the performance data that were monitored during our experiments are publicly available[4].

4.2.1 Measured Performance Indicators

The configurations were analyzed *w.r.t* a set of performance indicators. In particular, we measured and analyzed indicators related to response times, energy consumption, and static data.

Response times Response times were measured through Gatling, a benchmarking tool designed to simulate heavy loads on web applications and measure their performance and stability. Especially, we monitored requests such as the authentication request (`authentication`), the first authenticated request (`authenticated`), the retrieval of all objects belonging to a certain entity (`getall`), the creation of an object of a given entity (`create`), the retrieval of this object (`get`), and the deletion of this object (`delete`). For each of these indicators, we measured the mean response time in milliseconds. The number of simulated users and requests per user, as well as the profile of user arrival, were set to the default values generated by JHipster and will be detailed below.

Energy consumption The energetic performance of a configuration is defined as the energy consumption of the system implementing such a configuration, expressed in Joules. The energy consumption is measured during a fixed idle period `idle-total` once the stack is booted, and then during the Gatling benchmark, `wl-total`. As the idle periods and the Gatling benchmarks have fixed duration across configurations, the energy consumption indicators are converted into power usage indicators, expressed in Watts. To increase the granularity of energetic performance indicators, the total power usage is further distributed into two indicators, CPU power usage, and RAM power usage. Thus, six indicators are monitored: `idle-total`, `idle-cpu`, `idle-ram`, `wl-total`, `wl-cpu`, and `wl-cpu`.

Static indicators Some JHipster configurations include external services, such as Redis or Elasticsearch, which increase the number of concurrently running containers. Thus, the number of docker images `service` and their size `size` evolve with configurations, which may impact other performance indicators. The number of selected features `features`, which is different from the number of images, is also monitored. Finally, the stack's boot time `boot-time` is also accounted for in this category, as it is independent of the workload.

[4] `https://zenodo.org/deposit/8140600`

4.2.2 Experimental Settings

This section describes the technical environment and the protocol of the conducted experiment.

Technical Environment In order to ensure reproducibility, all experiments were conducted in the same technical setup. The experiments were performed on the Grid'5000 grid, on a mono-CPU device equipped with a 2.20GHz CPU and 96 GiB of RAM, running the operating system Ubuntu 20-04 minimal. The specific version of each software component is listed in the open data.

Data collection The assessment of a configuration is performed by the following process: First, the idle power usage of the device is monitored over a 30 seconds period. Then, the generated stack is started, and the startup duration is logged for future analysis. Once the stack is ready to accept requests, the power usage is monitored over a 30 seconds period to compute the `idle-total` indicator. Then, the stack is stressed with the load testing tool Gatling. The power usage is monitored during the Gatling benchmark to compute the `wl-total` indicator. The Gatling benchmark is the default scenario generated by JHipster to prevent user bias. Specifically, Gatling generated 100 users evenly distributed over a one-minute time window, with each user performing a scenario composed of 19 operations. The response-time indicators are extracted from the reports generated by Gatling. Energy consumption metrics are obtained using the Running Average Power Limit (RAPL) [Kha+18], a power monitoring tool embedded in some modern processors. This tool provides the total power usage of the device, and the specific power usage of the DRAM and CPU [DPW16]. To increase the accuracy of energetic indicators, all the power usages discussed in this chapter are cleaned of the idle power usage of the device, following the method presented in Chapter 3. The idle power usage of the device is measured before each benchmark to ensure the finest granularity. Finally, all experiments were performed after a warmup of the device, to avoid heat-related variations in the power usage [Wan+18].

4.3 Performance of Configurations

In this section, we analyze the performance of the 118 configurations of JHipster and look into the correlation between performance indicators.

4.3.1 Variability in Performances

Table 4.1 presents the variation in performance across configurations, for all the measured indicators. This table provides the unit, mean, standard deviation, minimum, maximum, and quartiles of each indicator. The ratio between the maximum value and minimum value is also displayed. While some performance indicators such as `authentication` response times are consistent across configurations, most exhibit substantial variations. In particular, the workload

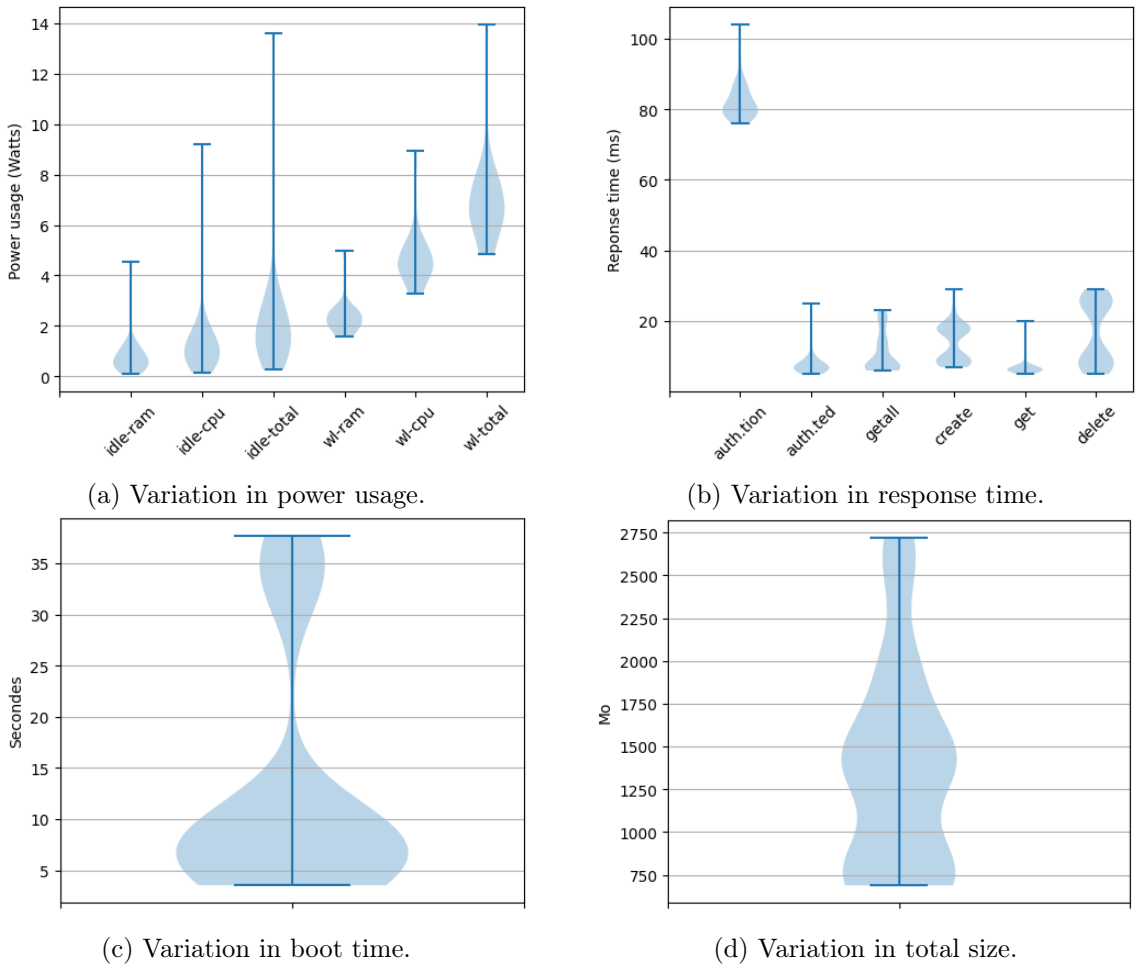
	unit	mean	std	min	25%	50%	75%	max	factor
size	Mo	1364.71	605.44	689.00	766.25	1373.50	1687.00	2719.00	3.95
services	–	2.63	0.61	2.00	2.00	3.00	3.00	4.00	2.00
boot-time	s	13.05	11.88	3.47	5.79	7.10	9.97	38.33	11.04
features	–	3.95	0.99	2.00	3.00	4.00	5.00	5.00	2.50
auth.tion	ms	82.86	5.30	76.00	79.00	81.50	86.00	103.00	1.36
auth.ted	ms	8.12	3.60	5.00	6.00	7.00	8.00	25.00	5.00
getall	ms	11.42	5.87	6.00	7.00	8.00	15.75	24.00	4.00
create	ms	13.91	5.30	7.00	9.00	14.50	18.00	29.00	4.14
get	ms	6.96	2.40	5.00	6.00	6.00	7.00	20.00	4.00
delete	ms	16.93	8.96	5.00	8.00	13.00	26.00	30.00	6.00
idle-cpu	W	2.04	1.85	0.20	0.92	1.70	2.53	10.83	53.066
idle-ram	W	0.95	0.81	0.15	0.50	0.73	1.19	4.74	32.53
idle-total	W	2.99	2.62	0.40	1.45	2.47	3.66	15.53	38.43
wl-cpu	W	4.30	1.23	2.26	3.58	4.06	4.81	9.00	3.98
wl-ram	W	2.29	0.65	1.36	1.95	2.22	2.56	5.08	3.72
wl-total	W	6.59	1.86	3.64	5.59	6.26	7.33	14.07	3.87

Table 4.1: Summary of the measured performance indicators.

power usage (`wl-total`) varies by a factor of 3.8, the size by a factor of 3.9, some boot times are 11 times higher than others, and the idle power usage varies by factors of 53. Therefore, the choice of a configuration appears to significantly impact the performance of JHipster.

Beyond the difference in performances across configurations, some indicators exhibit clustering patterns, as shown by Figure 4.2. Specifically, in Figure 4.2c the boot time is either below 10 seconds, or more than 30, with no in-between. Similar behavior is visible in Figure 4.2d, with `size` no configuration around 1250Mo or between 2250 and 2500Mo. In Figure 4.2b, `create` and `delete` response times appear to fall into two groups, centered around either 10 or 18ms for the `create` requests or around 10 or 25ms for the `delete` requests. Finally, in Figure 4.2a, most configurations are grouped around the lower values, while some outliers appear to have substantially higher power usage. This distribution raises some questions. Are the fastest configuration of `create` and `delete` the same configuration? Are they also the fastest to boot, the smallest, the most energy-efficient? Such questions can be generalized to performance indicators with no obvious clusters, such as `get`. Answering such questions requires an analysis of the correlation between all the measured performance indicators: if some or all of these performance indicators are strongly correlated, then the assessment and optimization of performance would be substantially simplified. Finally, it would be beneficial to identify if such behaviors are caused by specific options.

RQ 1: Some performance indicators exhibit very different performances across configurations. Thus, the performance of a system using JHipster appears to be highly dependent on its configuration.



(a) Variation in power usage.

(b) Variation in response time.

(c) Variation in boot time.

(d) Variation in total size.

Figure 4.2: Variations in the measured performance indicators across configurations.

4.3.2 Correlation in Indicator Groups

We analyzed correlations inside each of the indicator groups, *i.e.*, between static indicators, between energetic indicators, and between response times. The purpose of this analysis is to look for indicators that could be estimated based on other indicators. Such indicators could act as proxies for other indicators, and thus be used to simplify either the experimental setup or the analysis of the corresponding results. Unless specified otherwise, all reported correlations are Pearson Correlation Coefficients. The detail of all correlations is listed in Table 4.2.

Correlation between static indicators Static indicators include `size`, `services`, `boot-time`, and `features`. While all the correlations are positive, some are negligible: in particular, the correlations `boot-time` and `services` or `features` are respectively 0.05 and 0.17. By contrast, `boot-time` is strongly correlated to `size`, by 0.83. The indicators `services` and `features` are correlated by 0.52. This can be explained by the design of the variability: the services are represented as features in the feature model, but not all features are services. Thus, adding a

feature cannot reduce the number of services, and it has a certain probability of increasing the number of services. For instance, enabling the cache in a configuration can either increase the number of services (*e.g.*, with Redis) or not alter it (*e.g.*, with Ehcache). Finally, `services` and `features` appear slightly correlated to `size`, by respectively 0.45 and 0.33. Thus, `services` or `features` tends to increase `size`, which in turn affects `boot-time`. Therefore, `size` may be a relevant performance indicator to optimize, in order to also optimize `services`, `boot-time`, and `features`.

Correlation between energetic indicators All the energetic indicators are strongly correlated, with correlations ranging from 0.72 to 0.99. The CPU and RAM energy consumption are strongly correlated, both in idle (*i.e.*, 0.94) and under load (*i.e.*, 0.95). Across idle and workload, the performance of the CPU and DRAM are also correlated: their respective idle and workload power usages are correlated at respectively 0.72 and 0.91. Thus, the total power usages in idle and under load are also correlated at 0.80. The total energy consumption is computed as the sum of the CPU and DRAM energy consumption. Thus, the total power usage is strongly correlated to both metrics (*i.e.*, between 0.97 and 0.99, both in idle and under load). However, this correlation is stronger with the CPU consumption (*i.e.*, 0.99 in idle and under load) than with the DRAM consumption (*i.e.*, 0.97 in idle, 0.98 under load), meaning that the energy consumption of the CPU has more weight in the total power usage than the RAM. This is validated by the mean power usage of configurations presented in Table 4.1. The mean consumption of the CPU is higher than the consumption of the DRAM by a factor of 2.1 during idle and by a factor of 1.9 under load. Finally, the total power usage in idle is strongly correlated to the power usage under load (*i.e.*, 0.80). Yet, in Table 4.1 and Figure 4.2b, the highest idle power usage is higher than the highest power usage under load. This aberration can be explained by four outlying configurations exhibiting higher power usage in idle than under load. Such unexpected behavior, which is explained in Section 4.4, artificially reduces the correlation between `idle-total` and `wl-total`. Thus, idle power usage is a strong indicator of power usage under load. In future work, the power usage assessment under load may thus be substituted by the use of idle power usage as a proxy.

Correlation between response times The correlations between response times indicators are all positive, suggesting that faster configurations tend to be faster in all indicators. However, the correlations range from 0.06 to 0.93. The `authentication` and `authenticated` requests are strongly correlated (*i.e.*, 0.80). Both of them are also strongly correlated to the `get` request (*i.e.*, respectively 0.74 and 0.93). They are only weakly correlated to other requests such as `create` (0.33 and 0.38) or `delete` (0.11 and 0.09). The `get` indicator is also correlated to `create` (*i.e.*, 0.45), but its correlations to the remaining indicators are less significant, below 0.30. The `getall` request is weakly correlated to `authentication` (0.49), `authenticated` (0.30), and `get` (0.3). Finally, the `create` and `delete` requests are strongly correlated (*i.e.*, 0.92). Such a correlation tends to confirm the hypotheses formulated about the clustering of values

in Figure 4.2b: the slowest configurations on `create` are also the slowest in `delete`. The `delete` request is only weakly correlated to other response times, with a correlation lower than 0.15. The `create` request has stronger correlations with other indicators, in particular with `get` (0.45), `authentication` (0.33), and `authenticated` (0.38). Thus, the writing operations `create` and `delete` can be used as a proxy of each other, and `get` can be used as a proxy feature for the other reading operations *i.e.*, `authentication` and `authenticated`. While `getall` is also a reading operation, its correlations are too weak to highlight a specific proxy.

4.3.3 Correlations Across Indicators Groups

While some proxy indicators appeared inside each indicator group, assessing the performance of a configuration still requires a dedicated analysis for each indicator group. Therefore, to further simplify performance assessment and optimization, it appears relevant to explore proxy indicators that enable the estimation of performance across different groups of indicators.

Static indicators and time The response time to create or delete requests is strongly correlated to the static indicators, in particular the number of services (*i.e.*, 0.72 for `create` and 0.83 for `delete`). The correlations between `create` and `delete` and other static indicators may be the consequence of the internal correlations between static indicators. Such correlations can be explained by the behavior of each option, and are thus further discussed in Section 4.4. Aside from the `create` and `delete` indicators, the response times are only weakly correlated with static indicators. The correlations range from -0.34 between `authenticated` and `features`, to 0.35 between `getall` and `services`. `services` can thus be used as a proxy indicator for `create` and `delete`, but the static indicators do not provide a means to assess the response time in reading operations.

Static indicators and energy The correlation between energetic indicators and `size` ranges from 0.36 to 0.56. In particular, the power usage under load is more strongly correlated to size (*i.e.*, between 0.55 and 0.56) than the power usage in idle (*i.e.*, between 0.36 and 0.52). `size` is also more correlated to the total power usage than the specific power usage of the CPU or RAM. However, such correlations are too weak to highlight a potential proxy.

Despite internal correlations between energetic indicators and between static indicators, other static indicators are only weakly, or not at all, correlated to energetic indicators, as most correlations are lower than 0.26. In particular, the correlations between `features` and energetic indicators are between 0.08 and -0.07, and the correlations between the number of services and idle indicators are between 0 and -0.5. Overall, static indicators are more correlated to the power usage under load than to the power usage in idle. Correlations between static and energetic indicators are thus too weak to highlight any proxy indicator.

Time and energy Similarly to static indicators, response times are more strongly correlated to the power usage under load than to the one in idle. The idle power usages are correlated to only `authentication` by 0.23, and `getall` by 0.36. For these response times, the correlations increase to between 0.44 and 0.60 under load. With other response times, the correlations are between -0.12 and -0.02 in idle, and between 0.17 and 0.4 under load. Overall, the power usage of the RAM is less correlated to response time than the power usage of the CPU. Similarly, correlations between static and time indicators are thus also too weak to highlight any proxy indicator.

Summary This section reports on the correlations between performance indicators in JHipster. Some indicators exhibit strong correlations. In particular, all the energetic indicators are strongly correlated with each other. Static indicators are also correlated with each other, and in particular with `size`, which can thus act as a proxy indicator. The correlations across response time indicators are less explicit. The writing operations `create` and `delete` are strongly correlated. The indicators `authentication`, `authenticated` and `get`, all related to reading operations, are also strongly correlated. Across indicator types, only a limited set of indicators are correlated: the writing operations and `size`, `size` and power usage under load, and finally the power usage under load and some response times. However, such correlations are too weak to highlight any proxy across indicator groups.

> **RQ 2**: Some performance indicators exhibit strong correlations, in particular when they quantify measurements of similar natures. Such correlations allow the use of proxy indicators to simplify performance assessment and optimization. Correlations also exist across indicators of different nature but they tend to be weaker and less frequent. Thus, a specific performance assessment of each of the indicator groups is still required.

	size	services	boot-time	features	auth.tion	auth.ted	getall	create	get	delete	idle-cpu	idle-ram	idle-total	wl-cpu	wl-ram	wl-total
size	1.00	0.45	0.83	0.33	0.23	0.05	0.14	0.53	0.10	0.54	0.36	0.52	0.41	0.56	0.55	0.56
services	0.45	1.00	0.05	0.52	0.08	-0.02	0.35	0.72	0.00	0.83	-0.05	-0.00	-0.03	0.25	0.17	0.23
boot-time	0.83	0.05	1.00	0.17	0.15	-0.00	0.05	0.16	0.09	0.14	0.16	0.32	0.21	0.26	0.26	0.26
features	0.33	0.52	0.17	1.00	-0.13	-0.34	0.17	0.48	-0.14	0.57	-0.07	0.00	-0.05	0.08	0.05	0.07
auth.tion	0.23	0.08	0.15	-0.13	1.00	0.81	0.49	0.33	0.74	0.11	0.22	0.25	0.23	0.60	0.46	0.56
auth.ted	0.05	-0.02	-0.00	-0.34	0.81	1.00	0.30	0.38	0.93	0.09	-0.04	-0.04	-0.04	0.40	0.21	0.34
getall	0.14	0.35	0.05	0.17	0.49	0.30	1.00	0.11	0.30	0.06	0.34	0.37	0.36	0.53	0.46	0.51
create	0.53	0.72	0.16	0.48	0.33	0.38	0.11	1.00	0.45	0.92	-0.10	-0.03	-0.08	0.38	0.23	0.33
get	0.10	0.00	0.09	-0.14	0.74	0.93	0.30	0.45	1.00	0.15	-0.12	-0.09	-0.11	0.37	0.17	0.30
delete	0.54	0.83	0.14	0.57	0.11	0.09	0.06	0.92	0.15	1.00	-0.07	-0.02	-0.06	0.28	0.17	0.24
idle-cpu	0.36	-0.05	0.16	-0.07	0.22	-0.04	0.34	-0.10	-0.12	-0.07	1.00	0.94	0.99	0.72	0.80	0.76
idle-ram	0.52	-0.00	0.32	0.00	0.25	-0.04	0.37	-0.03	-0.09	-0.02	0.94	1.00	0.97	0.82	0.91	0.86
idle-total	0.41	-0.03	0.21	-0.05	0.23	-0.04	0.36	-0.08	-0.11	-0.06	0.99	0.97	1.00	0.76	0.84	0.80
wl-cpu	0.56	0.25	0.26	0.08	0.60	0.40	0.53	0.38	0.37	0.28	0.72	0.82	0.76	1.00	0.95	0.99
wl-ram	0.55	0.17	0.26	0.05	0.46	0.21	0.46	0.23	0.17	0.17	0.80	0.91	0.84	0.95	1.00	0.98
wl-total	0.56	0.23	0.26	0.07	0.56	0.34	0.51	0.33	0.30	0.24	0.76	0.86	0.80	0.99	0.98	1.00

Table 4.2: Correlations between the performance indicators

4.4 Performance of Options

This section aims at understanding the impact of options of a configurable system on its performance. The purpose of this investigation is to discover means to create efficient configurations through relevant option selection. The comparisons between services that are presented in this section are only applicable to the specific synthetic workload that was used as a benchmark.

4.4.1 Performance of Individual Options

While the previous section looked into the correlation between different performance indicators, this section compares the performance of various options available in JHipster: the choice of authentication methods, databases, and search engines, and the activation of hibernate cache and reactive processing. The purpose of this comparison is to analyse how each options impacts the performance of JHipster. The previous section highlighted that the CPU, the RAM, and the total power usage are strongly correlated (more than 0.94). For the sake of clarity, the power usage of CPU and RAM were thus ignored in this analysis, although present in the dataset.

Authentication In our experiment, we analyzed two authentication methods: session and JWT. Since some options (*i.e.*, Couchbase, Neo4J, and Reactive) are not compatible with the session authentication method, therefore, configurations containing such options were excluded from this analysis, to avoid biases. There is no significant difference *w.r.t* power usage and static indicators between the two authentication methods. However, some variations can be observed regarding response time. Specifically, the authentication and first authentication requests are slower with `session` than with `JWT`, respectively by 2% and 12%. Then, `session` is slightly faster on the remaining operations `getall` (6%), `create` (2%), `getcreated` (2%), and `delcreated` (2%).

Databases JHipster supports 8 databases: Mysql, Microsoft SQL Server (MSSQL), Postgres, MariaDB, Mongo, Cassandra, Neo4j, and Couchbase. However, the choice of database is loosely coupled to the activation of a search engine, the hibernate cache, and the JWT authentication method since selecting. For instance, selecting the database Cassandra prevents the use of a search engine, as defined by constraints in the feature model. Thus, configurations containing either of these were excluded from this analysis, to avoid biases.

The total size of the stack differs depending on the database. While the footprint of most configurations is between 700Mb and 900Mb, the average size of configurations containing Couchbase is 1730Mb, and 1925Mb for the ones containing MSSQL. Thus, the choice of database appears to have a strong impact on the size of the system, and can partially explain the spread in size in Figure 4.2d. Finally, while most configurations have boot times between 3.8 and 6.7 seconds, this value increases to 34 seconds for the one containing MSSQL. It thus appears that MSSQL is responsible for the clustering visible in Figure 4.2c.

		size	services	boot-time	features	auth.tion	auth.ted	getall	create	get	delete	idle-total	wl-total
Database	**Cassandra**	907.00	3.00	4.25	2.00	81.00	8.00	8.00	7.00	6.00	6.00	1.81	7.03
	Couchbase	1730.00	2.00	6.68	2.00	86.00	7.00	18.00	7.00	5.00	5.00	15.26	12.30
	MariaDB	695.00	2.00	5.19	2.00	80.00	9.00	7.00	8.00	7.00	7.00	1.42	4.48
	MongoDB	740.00	2.00	3.77	2.00	77.00	7.00	6.00	7.00	6.00	6.00	0.59	4.25
	MSSQL	1925.00	2.00	34.02	2.00	81.00	9.00	8.00	9.00	7.00	9.00	3.39	6.28
	MySQL	765.00	2.00	5.47	2.00	83.00	10.00	7.00	9.00	7.00	9.00	2.82	6.43
	Neo4j	869.00	2.00	6.67	2.00	86.00	13.00	11.00	13.00	9.00	10.00	3.62	6.17
	PostgreSQL	689.00	2.00	5.23	2.00	80.00	9.00	7.00	8.00	7.00	8.00	1.06	4.03
Authentication	**JWT**	1358.88	2.67	13.68	4.06	79.10	6.71	10.98	13.65	6.47	17.47	2.62	6.19
	Session	1358.71	2.67	13.77	4.06	84.14	7.51	10.33	13.39	6.31	17.18	2.66	6.21
Cache	**No cache**	1353.62	2.50	13.05	2.50	82.88	9.31	6.75	13.06	6.81	17.25	2.73	5.97
	Any cache	1385.29	2.70	14.56	4.50	81.56	6.62	11.74	13.82	6.34	17.75	2.70	6.27
	caffeine	1355.19	2.50	13.24	4.50	79.75	6.44	7.25	13.44	6.31	16.81	2.59	5.89
	ehcache	1355.38	2.50	13.22	4.50	79.88	6.12	7.25	13.44	6.12	16.81	1.95	5.70
	hazelcast	1371.31	2.50	16.32	4.50	82.75	6.81	14.69	13.81	6.31	17.75	3.86	7.02
	infinispan	1367.88	2.50	16.42	4.50	80.81	6.25	7.31	14.12	6.31	17.50	2.14	5.91
	redis	1476.69	3.50	13.63	4.50	84.62	7.50	22.19	14.31	6.62	19.88	2.96	6.85
CouchbaseSE	**No**	1733.50	2.00	6.18	2.50	86.50	7.50	20.00	7.00	5.00	5.00	15.40	12.92
	Yes	1733.50	2.00	6.39	3.50	87.00	7.00	19.00	7.00	5.00	5.00	14.75	13.73
Elasticsearch	**No**	1024.48	2.14	12.88	3.52	82.52	8.11	11.29	9.66	7.00	9.16	2.23	5.65
	Yes	1694.95	3.14	14.02	4.52	82.98	8.20	11.11	18.89	7.09	25.96	2.92	7.03
Reactive	**No**	1370.08	2.42	10.54	2.50	81.50	8.42	8.67	12.00	6.50	15.00	4.67	7.02
	Yes	1380.83	2.42	9.34	3.50	92.25	16.17	17.00	17.50	11.75	15.33	4.12	8.78

Table 4.3: Average performance of configurations containing each option

Regarding energetic indicators, Couchbase exhibits an outlying behavior: its power usage is higher in idle (15.3W total) than under load (12.3W total). This behavior is due to the design of Couchbase, which performs indexation in idle to optimize response times under load. All the other databases have idle power usage between 1.4W and 3.6W, except MongoDB with 0.6W, and load power usage between 4W and 6.4W, except for Cassandra with 7W.

As for response times, Neo4j is either the slowest or one of the slowest across all response times, except for `getall`. Contrarily, Couchbase is the fastest or one of the fastest across all response times, but the slowest for `getall` (*i.e.*, 18ms against 11ms for the second worst). MongoDB is the fastest to perform `getall`.

Hibernate cache JHipster supports the activation of Hibernate cache for SQL databases. Thus, configurations can be grouped into three categories: the ones where the Hibernate cache is activated, configurations where Hibernate cache is available but not activated, and configurations where Hibernate cache is not available. This analysis only accounts for configurations where Hibernate cache is available and activated. The configurations where the Hibernate cache is available and disabled serve as a baseline.

While activating the Hibernate cache has a limited impact on the idle power usage (-1%), it tends to increase the power usage under load (+5%). Configurations containing Hibernate cache have an average boot time 12% higher than configurations not containing it.

The Hibernate cache increases the response time for writing requests, by 6% for `create` and 3% for `delete`. This increase can be explained by the modification of the cached data, in addition to the modification of the database. Such overhead is leveraged in reading operations: the response times for `get` and `authenticated` are reduced by respectively 7% and 29%. However, the response time for `getall` is increased by 74%.

Cache provider When the Hibernate cache is enabled, a cache provider must be selected among the five ones offered by JHipster, *i.e.*, Caffeine, Ehcache, Hazelcast, Infinispan, or Redis. As the Hibernate cache is only available for SQL databases, non-SQL databases are excluded from this analysis. In Table 4.3, the “No cache" line refers to configurations where a cache system was available but was not selected and thus acts as a baseline. This paragraph is a deeper analysis of the previous paragraph, where the configurations with Hibernate cache are compared *w.r.t* the specific cache provider they contain.

Different performances can be observed in static indicators between these cache systems. Specifically, Redis increases the service count, as it is implemented as a separate docker image. Thus, it also increases the `size` of configurations. As for Infinispan and Hazelcast, they also slightly increase the total stack size but do not increase the service count, as they are implemented as libraries. They also increase the `boot-time` by 30%, whereas other providers increase this indicator by between 5% and 9%. Only Caffeine and Ehcache do not impact the total size of the stack.

The choice of the cache system also impacts the energetic performance of the stack. This impact differs when the system is idle or under load. The idle power usage of Hazelcast and Redis are higher than the baseline, by respectively 51% and 16%. However, under load, such energetic overheads are canceled and the difference to the baseline falls to respectively +3% and less than +1%. The remaining cache systems reduce the power usage by between 13% and 16% under load, compared to their baseline. Overall, configurations containing Ehcache exhibit the lowest power usage, both idle and under load.

Finally, the different cache provider also impact the response time in different ways. The time savings are consistent for requests such as `create`, with time savings between 9% and 15%, or `get`, with time savings between 30% and 35%. For `delete`, Hazelcast and Inifispan reduce the response times by less than 2%, Caffeine and Ehcache by 5%, while Redis increase response times by 12%. The impacts of cache systems are also inconsistent in `getall`. Caffeine, Ehcache, and Infinispan reduce response times by 29%, while Hazelcast increases them by 43% and Redis by 116%. Ehcache appears to have the best performance across response times, while Caffeine is a close second. Contrarily, Redis appears to provide the worst performance gains and can be counter-productive for specific requests.

Search engine JHipster supports Elasticsearch or Couchbase as search engines, providing support to research content in the database. However, the research feature is implemented as a specific route, which is not present in configurations with no search engine. The performance of search requests is thus a specific workload and is not assessed by this experiment. This analysis only compares the impact of adding a search engine on the performance of the other routes. Furthermore, Couchbase as a search engine (CouchbaseSE thereafter) is only available when the database is Couchbase, and Elasticsearch is only available when the database is not Couchbase or Cassandra. Thus, each search engine is compared to its respective baseline of configurations that could contain them but do not: configurations with CouchbaseSE are thus compared to configurations with Couchbase and no search engine, while configurations with Elasticsearch are compared to configurations without Elasticsearch, Couchbase, or Cassandra.

Enabling the Couchbase search engine has only a limited impact on power usage: -4% in idle and +6% under load. Contrarily, enabling Elasticsearch substantially increases power usage, by 31% in idle and 25% under load. Similarly, CouchbaseSE has no impact on `size` and `boot time`, whereas Elasticsearch increases them by respectively 65% and 9%. Thus, Elasticsearch can also be responsible for the spread in `size` visible in Figure 4.2d, in addition to the choice of database. Such results can be explained by the implementation of Elasticsearch, which runs in a separate container, inducing a substantial overhead in size, boot time, and power usage. While both search engines have very limited impact on response times, Elasticsearch increases the response time of `create` by 96% and `delete` by 183%. Such results may be caused, for instance, by an index requiring an update when the content of the database is modified, similar to cache providers.

Reactive Reactive is unavailable for configurations containing Cassandra, Neo4j, session, or a cache mechanism. Thus, configurations containing such features were excluded from this analysis. Reactive reduces the power usage in idle by 10% but increases the power usage under load by 25%. It has a limited impact on static indicators, by reducing the boot time by 11%. However, it substantially increases the response times. In particular, `authenticated`, `getall` and `get` are increasing by respectively 92%, 96%, and 81%. This difference is more limited for `create`, 46%. Only `delete` is unaltered.

> **RQ 3**: The different options have different impacts on the performance of the system. In addition, this impact is not consistent across performance indicators. Thus, an informed feature selection may allow the creation of efficient configurations.

4.4.2 Finding Optimal Configurations

The previous section provided insight into the performance of each option of JHipster. Such data can be used to compare options with each other, and thus to make an informed choice during the configuration process, to select options that optimize one or many performance indicators. For instance, PostgreSQL appears to be the database with the lowest power usage under load, thus it would make sense to select it in a configuration designed for low power consumption.

However, it is unclear if such configurations are actually the best-performing *w.r.t* their goal indicator, and how they will perform in other performance indicators. This behavior can be assessed by creating a set of configurations, each optimizing a specific indicator, and then by comparing such configurations with the set of all the les 118 valid configurations.

To void redundancy in the validation process, we selected performance indicators based on the correlations found in Section 4.3, *i.e.*, indicators that act as proxy for other indicators. The different selected optimization goals relate to the indicators the more strongly correlated to other indicators as described in Section 4.3, *i.e.*, the power usage during workload, the total size, and the response times for `getall` and `create` requests.

The creation of a configuration that optimizes a given indicator is made by selecting options that maximize this indicator. However, this selection process is constrained by the feature model, as the selection of an option can prevent the addition of other options later in the selection process. To limit such constraints, the options are selected by decreasing amount of implementations (sub-features in the feature model): first, the database type, then the cache system if applicable, and finally the authentication method, search engine, and reactive processing. In addition, we decided that if an optional feature (*i.e.*, a search engine or reactive) has no impact on the targeted performance indicator, it is included in the configuration to maximize capabilities.

Table 4.4 details the configuration for each goal. Each of the resulting configurations where benchmarked using the same protocol as the sample of configurations. However, as we aim at

Goal	Auth	DB	Cache	S. engine	React.
C_power	JWT	PostgreSQL	Ehcache	No	No
C_size	Session	PostgreSQL	No	No	No
C_getall	Session	MongoDB	NC	Yes	No
C_create	JWT	Couchbase	NC	Yes	No

Table 4.4: Best configurations for each optimization goal.

finding the specific performance of each individual configuration, each measure was repeated 20 times.

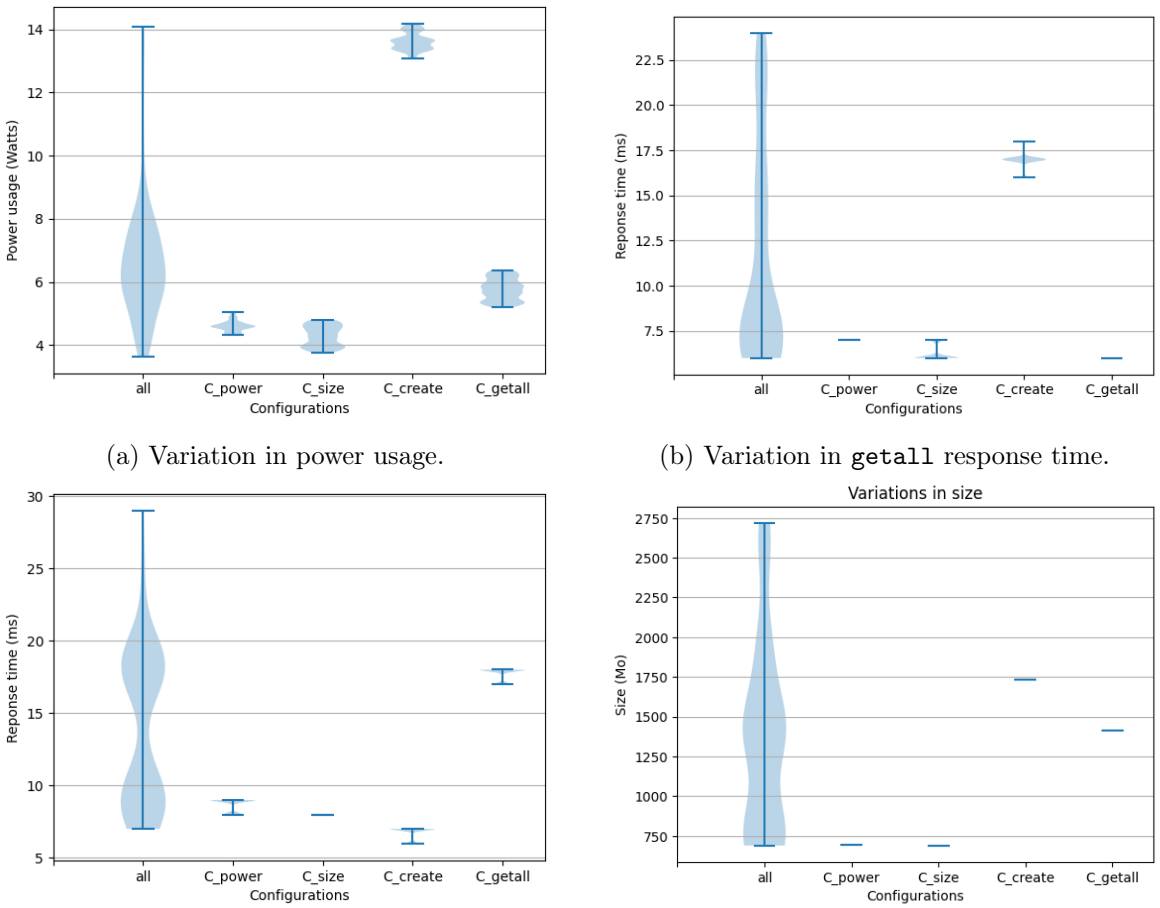
Figure 4.3 presents the performance of the different candidates, in each of the goal indicators. The performance of all the valid configurations in each goal indicator is also displayed, as a reference. With the exception of power, each candidate exhibits very strong performance on their respective goal indicator: C_getall offers the best response times in `getall`, C_create offers the best response times in `create`, and C_size has the smallest `size`. C_power, while lower than most of the initial configurations, is over-performed by C_size. Some of the initial configurations also appear to outperform C_power. This behavior can be explained by the complexity of obtaining accurate energetic measures [Hei+17]. In particular, the power usage of the initial configuration is estimated based on the execution time and the energy usage of the benchmark but also based on an estimation of the idle power usage of the system, which may fluctuate during the benchmark. Then, the performance of options is estimated based on the average performance of configurations containing them. The resulting performance model has some limitations, as the performance of some options is very closer. Finally, the power usage of the candidate optimal configurations is again estimated using the method presented above.

Thus, selecting relevant options during the configuration process helps improve the performance of the selected system. However, this selection process needs additional formalizing and more accurate performance models to ensure optimal configurations.

While the selected goal indicators are not strongly correlated to each other, as observed in Section 4.3, some of the different candidates have similar performances. In particular, C_power, and C_size have both low power usage and a small size. C_power, C_size, and C_getall have low response times for `getall`, and finally, C_power, C_size, and C_create have low response times for `create`. Such similarities in performance can be explained by the similarity in configuration, as some options are shared between the configurations.

Thus, it appears that creating configurations composed of the best-performing options in an indicator is a relevant method to create configurations that optimize for this performance indicator.

RQ 4: Given insight into the impact of each option on the performance of a system, it is possible to create configurations that are high-performance in a given indicator by selecting options that are high-performing in this indicator.



(a) Variation in power usage.

(b) Variation in `getall` response time.

(c) Variation in `create` response time.

(d) Variation in total size.

Figure 4.3: Performance of the optimal configurations.

4.5 Discussion

As an empirical study, the results presented in this chapter must be interpreted *w.r.t* to the methodology.

Generalization Under no circumstances can the results of our study be generalized as a truth about the performance of each of the services included in JHipster. The configurations were benchmarked on a specific, synthetic workload. Furthermore, the performance model for options, build on the results of this benchmark, is only an estimation of performance. Such a model was only built to validate that it is possible to build high-performance configurations for a given workload and a given performance indicator. Finally, all the services were hosted on a single device, which may not be representative of a real production deployment, in particular, *w.r.t* to response times and power usage.

Workload The workload used to benchmark the configurations was automatically generated by JHipster during the creation of the project. Using this generated benchmark prevents bias in the selection of the workload. However, this workload may not be representative of an actual production environment. This affects the extrapolation of our results to other contexts and workloads, in particular, *w.r.t* the performance of individual options discussed in Section 4.4. In addition, the performance was assessed over a single data model, affecting the generalization of our results to other data models.

Variability pruning Some options offered by JHipster were excluded from this analysis. In particular, the Oracle database was excluded for licensing reasons, and micro-services or deployment options due to limitations in the benchmarking tool. Thus, no data can be provided for such options, while they may impact performances.

4.6 Summary

This chapter is an empirical analysis of the variability and performance of the configurable stack generator JHipster. In particular, we observed that the different configurations of JHipster exhibit very different performances. However, some performance indicators are correlated, allowing for a simplification of future performance assessments.

By analyzing the performance of configuration *w.r.t* the options they contain, it was possible to build a performance model to better understand the impact of each option on performance. Using this model, it was possible to create new configurations, designed to maximize the performance of a given indicator. However, this approach is a manual process that may not scale to large or more complicated feature models.

Part III

Optimizing Performance in Variable Software

Chapter 5

Reducing the Energy Consumption of Software Product Lines

In the previous chapter, we showed that different configurations exhibit different performances and that such variations are caused by the options the configurations contain. We also demonstrated that it is possible to create high-performing configurations by selecting high-performing options. However, creating such configurations is a tedious and error-prone process since some of these configurations are invalid, as defined by the feature model. For instance, two high-performing options can be mutually exclusive, creating an ambiguity about which one to use. To tackle this problem, the high-performing configurations of JHipster were created following an ad-hoc algorithm. Such an approach may not scale to other configurable systems, in particular the largest one. Furthermore, this approach does not provide guidance to optimize pre-existing configurations, or to adapt to functional requirements.

In this chapter, we thus present a method to measure and optimize the performance of multiple products at once by sampling and analyzing a minimal set of products. In particular, our method distinguishes two approaches: one that considers the performance of individual features and one that takes pairs of features into account. The latter thus takes feature interactions into account when measuring the performance of a product, to highlight pairs of features that may cooperate or obstruct each other at a behavioral level, while altering the performance of the system. This approach also suggests candidate features whose interaction with user-required features exhibits a better performance than the one produced by the initial interaction.

Our method thus provides means to estimate the performance of individual options, highlight how feature interactions impact the performance of products, and propose products with better performance while still including user-required features. We implemented our method and demonstrated it on the energy consumption of a Java-based SPL, to assess and compare the approach considering the impact of feature interactions on performance with the approach only focusing on the performance of individual features.

In the remainder of the chapter, Section 5.1 describes our approaches to measure and reduce

the energy consumption of a product from an SPL. Section 5.2 presents the design and results of our experiments. Section 5.3 discusses the outcomes of our contributions, while Section 5.5 concludes the chapter.

5.1 Estimating and Reducing Energy Consumption for SPL

Unlike measuring the energy consumption of software, measuring the energy consumption of an SPL is a non-trivial task, as multiple related software—*i.e.*, the products of the SPL—must be measured. These products exhibit different properties, including energy consumption, while sharing several features that perform differently in different contexts. The context of a feature can either be external to the product containing this feature—*i.e.*, the environment hosting the product—or internal to the product. That is, a feature can exhibit different performances when combined with different sets of features. Inferring the energy consumption of a single feature by measuring it while running in *one* given product is therefore irrelevant and does not reflect the energy consumption of that feature in the SPL. On the other hand, measuring the energy consumption of a feature in each product individually is not feasible as some products may be complex to measure while measuring the consumption of each product from a large SPL is not an option. To tackle these issues, we thus propose two approaches that estimate the energy consumption of features by measuring products *sampled* from the configuration space of the SPL and then exploiting such sampled measures to reduce the energy consumption of any product from the SPL.

Both approaches improve the energy consumption of products by removing features or substituting them with other ones. However, some features are included in a product to ensure its validity with regard to the feature model, *e.g.*, in or and xor relationships. We thus define $F_f \subset F$ as the set of features that are valid substitute features for a given feature f. These substitute features are either sibling features of f in or and xor relationships or features involved in or and xor used in cross-tree constraints. On the other hand, some products may contain features due to functional constraints (*e.g.*, stakeholder requirements). Such features cannot be removed or substituted and are hereafter referred to as *required features*. Thus, from the stakeholder standpoint, all products are functionally equivalent if they contain the features required by this stakeholder.

5.1.1 Feature-wise Energy Analysis

Energy impact of individual features To estimate the energy consumption of each feature from the SPL, we first measure the energy consumption of every product from the sample, using the method presented in Section 3.2. The energy consumption of each product is then reported in a matrix $n \times m$ with n the features and m the products, by copying the energy consumption of the product in the columns of each included feature. For instance, Matrix (5.1) defines f_1 to f_n as the available features, p_1 to p_m the sampled products, and E_{xy} represents the energy

consumption of p_x, such as defined by 3.2, if it includes f_y, or is left empty otherwise.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ \vdots \\ p_m \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} f_1 & f_2 & \cdots & f_n \end{array} \\ \begin{bmatrix} E_{11} & E_{12} & \dots & E_{1n} \\ E_{21} & E_{22} & \dots & E_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ E_{m1} & E_{m2} & \dots & E_{mn} \end{bmatrix} \\ \begin{array}{cccc} \tilde{E}(f_1) & \tilde{E}(f_2) & \cdots & \tilde{E}(f_n) \end{array} \end{array} \tag{5.1}$$

By computing the median value of each column of the matrix, the relative energy consumption $\tilde{E}(f)$ of the feature f represented by this column can be estimated. The expected behavior is that extreme energy consumption will cancel out and all features will have similar median energy consumption. However, if the median consumption of a feature is higher or lower than the other medians, then the presence of this feature tends to impact the performance of the products that contain it. Although such a measure does not provide a very accurate reading, it can nevertheless be used to compare the energy consumption of different features and perform preliminary optimizations, *e.g.*, by selecting the less consuming feature among the substitutes of each feature F_f. In the remainder of the chapter, we will refer to this method as the *feature-wise analysis*.

Feature-wise mitigation Getting the most energy-efficient product including the required features follows a two-step process. First, all optional features of the products are removed, thus only including the required features and the features requiring a substitution. Then, by leveraging the feature-wise analysis, the energy consumption of the remaining non-required features can be compared with their respective substitutes, to identify the one with the lowest consumption among them. Each of the non-required features is replaced by the most energy-efficient substitute. This approach always converges toward an efficient product composed of no optional feature, including only the features with the lowest energy consumption within each feature substitution set.

The product resulting from this mitigation strategy is the one with the lowest energy consumption that can be obtained given an initial configuration.

5.1.2 Pairwise Energy Analysis

Energy impact of pairwise interactions Although measuring the energy consumption of each feature in isolation gives a general trend, it cannot be used to compute the energy consumption of a combination of features (*e.g.*, as the mean or the median of several individual consumptions) due to the feature interactions phenomenon [Sie+12a]. Indeed, numerous works have shown that features interact with each other, hence impacting performances of

products [Ape+10; Ape+11; Ape+13; Sie+15; Sie+12a; BHK11]. Therefore, restricting the energy consumption analysis to individual features does not provide a comprehensive landscape of each feature's consumption, and additional analysis that considers feature interactions must be performed to obtain additional details about the energy consumption. By analyzing how the consumption of a feature evolves when this feature is combined with different features, it is thus possible to highlight feature interactions leading to a positive or negative impact on the consumption of the product.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ \vdots \\ p_m \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} c_1 & c_2 & \cdots & c_n \end{array} \\ \begin{bmatrix} E_{11} & E_{12} & \dots & E_{1n} \\ E_{21} & E_{22} & \dots & E_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ E_{m1} & E_{m2} & \dots & E_{mn} \end{bmatrix} \\ \begin{array}{cccc} \tilde{E}(c_1) & \tilde{E}(c_2) & \cdots & \tilde{E}(c_n) \end{array} \end{array} \tag{5.2}$$

A possible means to take pairwise feature interactions into account is by creating a new matrix with as many columns as there are valid pairs of features in the SPL. The consumption of each pair of features can be quantified by reporting on the energy consumption of each product in the columns of the pairs of features that this product contains, as illustrated in Matrix (5.2). c_1 to c_n are all the valid pairs of features, p_1 to p_m the sampled products, and E_{xy} is the energy consumption of p_x if it contains c_y, or is left empty otherwise. To ensure proper interaction coverage—*i.e.*, that all valid pairs of features are measured—the sampling of products must be performed by an algorithm ensuring such coverage.

Following the same methodology as in the *feature-wise analysis*, the consumption of pairwise feature interactions can be inferred by computing the median energy consumption $\tilde{E}(c)$ of each pair of features c. In the remainder of the chapter, we will refer to this method as the *pairwise analysis*. It is worth noting that this method is not only valid for pairwise interactions but can also be used to deal with larger T-wise interactions of features.

Pairwise mitigation Instead of replacing each feature with the substitute feature with the lowest energy consumption, this second approach iteratively picks the alternative features whose interactions with other features of the product result in a more energy-efficient product. At each iteration, the approach identifies a feature to remove from the product and, if required, replaces this feature.

To identify the feature f to be removed from a given product P, our approach relies on a *scoring* system: the interaction score $\mathcal{I}$. The interaction score of a feature f is computed by considering all pairs of features containing f in the product P, and by summing the observed

median energy consumption $\tilde{E}(c)$ of these pairs, as described in Formula (5.3).[5]

$$\mathcal{I}(f, P) = \sum_{g \in P |\ g \neq f} \tilde{E}(gf) \tag{5.3}$$

The iterations of this approach are realized as described by Algorithm 1. This algorithm iterates over the set of features until no more improvement can be performed. That is, each iteration removes or changes one feature in the product. At each iteration, the feature with the highest interaction score in the product must be removed in priority. The algorithm starts by sorting features by decreasing interaction score (line 8) and considers the first feature of the list (line 9) as a removal candidate. If this removal feature is user-required and cannot be removed, it is skipped (line 13). If the removal candidate is not a required feature and must be replaced (line 17), the replacement feature is identified among all possible substitutes—*i.e.*, F_f—by computing the interaction score of alternative features with regards to all remaining features of the product—*i.e.*, all but the removal candidate (lines 18 to 23). The selected replacement feature is the one with the lowest interaction score among all alternative features. As a result of the iteration, a new product is created by including the replacement feature (line 26).

However, if the removal candidate has the lowest interaction score, the replacement is discarded and the algorithm skips the feature, which is kept in the product. If a feature is skipped—*i.e.*, it was either a requirement or already the best option, a new removal candidate is defined as the next feature in the ordered list (line 34 and 11). Other features of the product will be changed over the next iterations to accommodate this skipped feature. Once a modification has been applied, the algorithm proceeds to the next iteration, unless a stop criterion is met: if the same product appears twice over different iterations, or if all features were tested during an iteration and no optimization was found (line 36).

Once a stop criterion is met, the energy consumption of the product resulting from each iteration is measured to monitor the energy gain. As the different mutations of the product are based on empirical data, which may be subject to imprecision and noise, a specific iteration may worsen the performance of the product. For this reason, the last step of this algorithm measures the energy consumption of the products resulting from each iteration. The product finally returned by this algorithm is the one with the lowest energy consumption, which may be the initial product in the worst-case scenario (lines 38).

5.2 Empirical Validation

In the previous section, we introduced two approaches to reduce the energy consumption of a given product. In this section, we experimentally assess each of these approaches. In particular, we aim to answer the following research questions:

[5]The interaction score can also be used during the configuration process, *e.g.*, to assist the user when selecting the most energy-efficient features when dealing with a partial configuration.

Algorithm 1: Interaction mitigation

```
1  Input initialProduct : a product to improve
2  Output bestProduct : the best product from all iterations
3  iterations.addProd(initialProduct)
4  improvable ← true
5  while improvable do
6      currentProd ← iterations.lastItem()
7      currentProd.removeNonRequiredOptionalFeatures()
8      sortedFeat ← sortByInteractionScore(currentProd)
9      remCandIndex ← 0
10     noChangeFound ← true
11     while (remCandIndex < currentProd.size) ∧ noChangeFound do
12         remCandidate ← sortedFeat.get(remCandIndex)
13         if ¬isRequirement(remCandidate) then
14             prodCandidate ← copy(currentProd)
15             prodCandidate.remove(remCandidate)
16             subOptions ← allF_sub(remCandidate)
17             if subOptions then
18                 currentBest ← remCandidate
19                 for subCandidate ∈ subOptions do
20                     if I(subCandidate, prodCandidate) < I(currentBest, prodCandidate)
                        then
21                         currentBest ← subCandidate
22                     end
23                 end
24                 if currentBest ≠ remCandidate then
25                     prodCandidate.addFeat(currentBest)
26                     iterations.addProd(prodCandidate)
27                     noChangeFound ← false
28                 end
29             else
30                 iterations.addProd(prodCandidate)
31                 noChangeFound ← false
32             end
33         end
34         remCandIndex++
35     end
36     improvable ← (remCandIndex < currentProd.size) ∧ allDifferent(iterations)
37 end
38 return lowestEc(iterations)
```

RQ 1: *Do our different analyses detect feature interactions impacting energy consumption?* By applying the two approaches to the same set of products, it should be possible to determine whether feature interactions have been detected as the two analysis methods should provide different results.

RQ 2: *How effective are our approaches to reducing the energy consumption of a product?* The two proposed approaches rely on different analysis methods to mitigate the energy consumption of products. We propose two experiments to ensure both of them improve the consumption of the products given a set of required features and evaluate how they differ.

5.2.1 Methodology

To assess the effectiveness of our solution when measuring the energy consumption of a software product line, we performed our experiments on RobocodeSPL, a software product line designed to yield robots for Robocode [MTZ18]. Robocode is an environment in which community-developed robots fight against each other in battles. A battle is composed of several rounds, and rounds have a time granularity of turns. During a turn, each robot taking part in the battle computes its next action and sends it to the Robocode engine which executes them all and proceeds to the next turn. A round ends when only one robot survives, and the winner of a match is the robot that caused the most damage to its opponents through the different rounds.

The RobocodeSPL proposes several implementations for the 5 mandatory features a robot requires to run properly—*i.e.*, *radar*, *targeting*, *movement*, *enemy selection* and *gun*. For instance, a *movement* can follow linear or circular patterns, follow the walls, or ram the opponent, among others. There are also 3 optional features related to resource management (*e.g.*, not spending more in-game energy than the robots have), for a total of 92 features and 72 leaf features. The number of valid products is 1.3×10^6. Figure 5.1 depicts an excerpt of the feature model of RobocodeSPL.

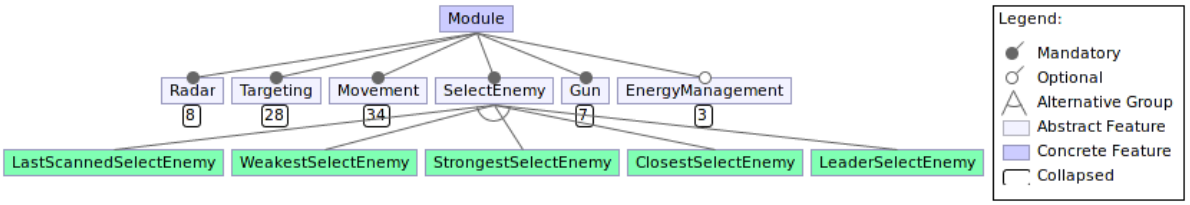


Figure 5.1: Excerpt of the Feature Model of RobocodeSPL.

To evaluate our approach, we launched multiple robot matches and ran our mitigation techniques to minimize the energy consumption of such matches. In particular, we launched matches opposing a sampled robot and a reference robot, the `sample.Wall` robot, considered as the strongest robot provided by Robocode[6]. As the goal was to minimize energy consumption, we were not interested in which robot wins or loses the match, but in the overall energy consumption of such a match. In order to fill the pairwise analysis matrix, the sample must contain several occurrences of each valid pair of features from the feature model. To ensure such coverage, we relied on the T-wise algorithm ICPL [JHF12] with $T = 2$ to sample the con-

[6]According to the Robocode Wiki: `https://robowiki.net/`

figuration space of RobocodeSPL. Another sampling technique may provide better uniform random samples [Kal+20; Mun+19b], but such techniques do not meet our coverage requirements. This algorithm generated 602 robots, hereafter referred to as the *training sample*. For each couple *(sampled robot, reference robot)*, we ran 10 matches to consolidate the performance data, resulting in a total of 6,020 matches of 1 round.

We used JJoules[7], a Java tool using the RAPL device of Intel CPU, to measure the energy consumption of the matches. JJoules is also able to monitor the energy consumption of the DRAM, while other tools can monitor other components, such as Hard Disk Drives. As Robocode is mainly CPU-intensive, we decided to focus on the energy consumption of the CPU. The energy consumption was monitored from the start to the end of each match, thus including the energy consumed by both robots, but excluding the energy consumption of the startup and shutdown of Robocode. All measurements were obtained from a machine running the Manjaro Linux distribution with an Intel i5 CPU at 2.9GHz and 8GB of RAM. Results, input data, and instructions to reproduce these experiments are available online.[8]

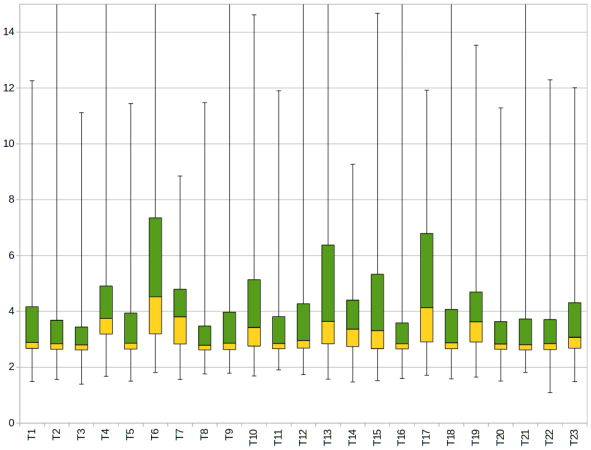
5.2.2 Results

Detecting interacting features The pairwise analysis relies on feature interactions to mitigate the energy consumption of products. The goal of this first experiment is therefore to ensure that the pairwise analysis is able to detect at least one occurrence of feature interaction. The first experiment thus compares the energy consumption of the Movement and Targeting features in different contexts—*i.e.*, in the presence of different sets of other features.

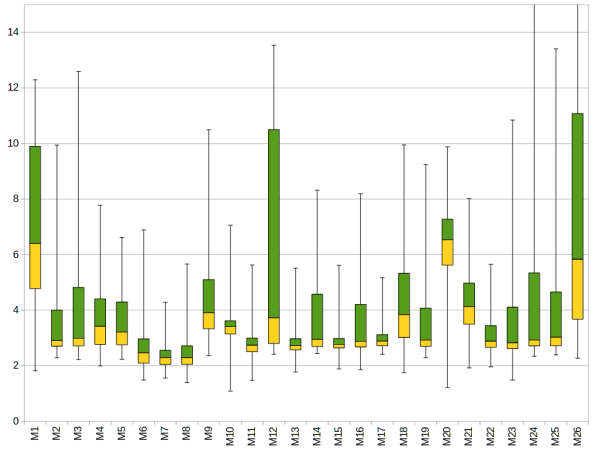
Figure 5.2 depicts how the energy consumption of different features evolves depending on the analysis method.[9] Figure 5.2a and Figure 5.2b report on the energy consumption (measured with the feature-wise analysis) of the products from our *training sample* containing respectively each targeting and movement feature. Among the targeting features, T4, T6, and T17 induce a higher energy consumption than the others, but most features show similar energy consumption, around 3 Joules. Among the movement features, M1, M20, and M26 impose the highest energy consumption, around 6 Joules, while M6, M7, and M8 report the lowest one, slightly above 2 Joules.

These differences in energy consumption can partially be explained by the functional behavior of these features. For instance, T6 (NoTargeting) performs no particular operation and always makes the robot shoot forward—*i.e.*, in the direction it is aiming at. This is not a smart behavior and the energy consumed by matches involving this feature depends on how fast the opponent is able to destroy this robot. By contrast, T13 (TargetAdvancingVelocitySegmentation) tries to anticipate the position of the opponent based on its speed and direction to ensure that the bullet and the opponent collide. Thus, a robot configured with T13 is able to win quickly,

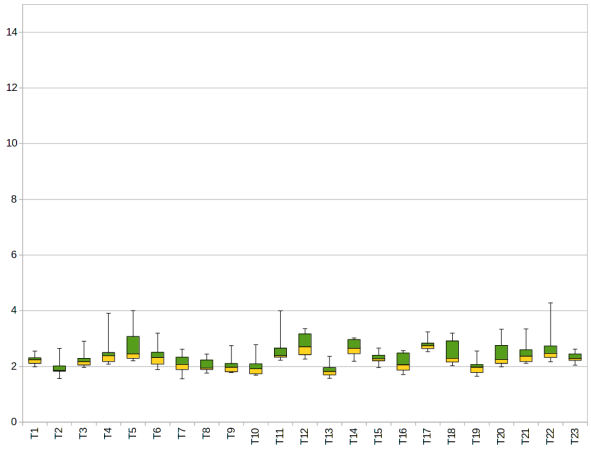
[7] https://github.com/powerapi-ng/j-joules
[8] https://doi.org/10.5281/zenodo.5048316
[9] Mapping to real feature names available in the open data



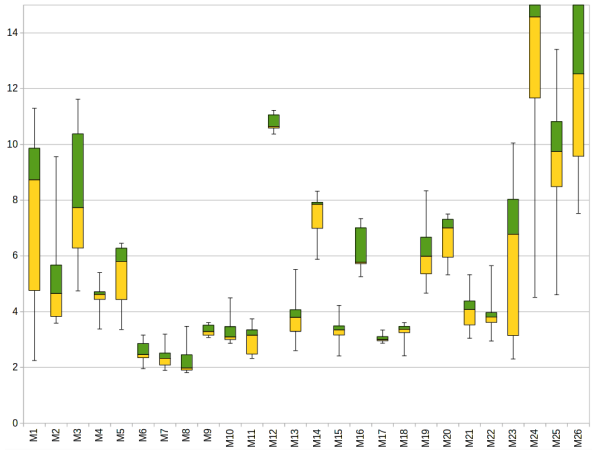
(a) Energy consumption per targeting feature.



(b) Energy consumption per movement feature.



(c) Energy consumption of targeting features when LinearRammingMovement (M7) is selected.



(d) Energy consumption of movement features when NoTargeting (T6) is selected.

Figure 5.2: Energy variations between the movement and targeting features and their potential interactions.

reducing energy consumption despite the additional computations required to anticipate the position of the opponent.

Figure 5.2c presents the energy consumption of each targeting feature when the feature LinearRammingMovement is selected—*i.e.*, M7, the most energy efficient movement feature. This figure is obtained by selecting all measurements of M7 in Figure 5.2b and breaking them down per targeting feature. Knowing that M7 is the best movement feature, the consumption of products containing each targeting feature is either improved or unchanged when M7 is selected. However, when targeting features are sorted by median energy consumption, their rank change depending on the context. For instance, T21 is ranked 3rd by the feature-wise analysis but becomes 16th when M7 is selected. T13 is ranked 19th out of 23 by the feature-wise analysis, but 1st in the pairwise analysis when M7 is selected. Furthermore, the median energy consumption of the couple of M7 and T13 is 1.8J, which is lower than the medians of both of these features alone, respectively 2.2 Joules and 3.6 Joules. Therefore, despite M7 being the best movement

feature, its performance can still be improved by selecting a relevant targeting feature.

As shown by the feature-wise analysis in Figure 5.2b, products including M9 and M12 have similar median energy consumption—*i.e.*, respectively 3.9 Joules and 3.7 Joules. However, when paired with NoTargeting (T6), one of the worst targeting features, their consumption evolves differently, as depicted in Figure 5.2d. The energy consumption of products including M9 is reduced from 3.9 Joules to 3.3 Joules, while the one for products including M12 dramatically increases from 3.7 Joules to 10.6 Joules. Thus, despite being considered a sub-optimal choice by the feature-wise analysis, T6 becomes an efficient choice when paired with M9. The pair composed of T6 and M8 is another occurrence of pairwise interaction outperforming both of its members (2 Joules instead of 4.5 Joules and 2.2 Joules, respectively). This result can be explained by the behavior of the features: M8 is a *ramming* movement feature, meaning that it is always moving toward the opponent. In this context, the behavior of NoTargeting—*i.e.*, always shooting forward—is very efficient, as it always hits the opponent.

Such changes in the resulting energy consumption with pairs of features outperforming both of their members alone shows that the energy consumption of targeting and movement features changes depending on how they are paired. Therefore, it highlights feature interactions between the targeting and movement features in RoboCodeSPL. This experiment thus unveiled occurrences of feature interactions allowing us to answer **RQ 1** positively.

> **RQ 1**: The pairwise analysis is able to detect interactions that impact significantly the energy consumption of products, and such interactions could not be detected by the feature-wise analysis.

Behavior without required feature The purpose of the second experiment is *(i)* to ensure the two mitigation approaches lead to a product different from the initial one, and *(ii)* to evaluate the energy consumption improvement resulting from these approaches. The first experiment showed that the feature-wise and pairwise analyses provide different results, due to their different granularity levels. It is yet to determine if the products resulting from their respective mitigation exhibit different energy consumption.

As explained in Section 5.1.2, the feature-wise analysis converges toward a specific product composed of no optional feature, and the features with the lowest energy consumption in each substitution set. In RoboCodeSPL, considering our optimization goal, *i.e.*, reducing the energy consumption against `sample.Wall`, and without any required feature, this product is composed of the features TurnMultiplierLock, DistanceSegmentation, LinearRammingMovement, StrongestSelectEnemy, and NoFireGun. Whatever initial product is considered for improvement, the feature-wise analysis will always return this product, hereafter referred to as the *Best Theoretic* product, BT_0.

By applying the pairwise analysis to this product, we can determine how the pairwise analysis compares to the feature-wise analysis in the absence of required features. The result of

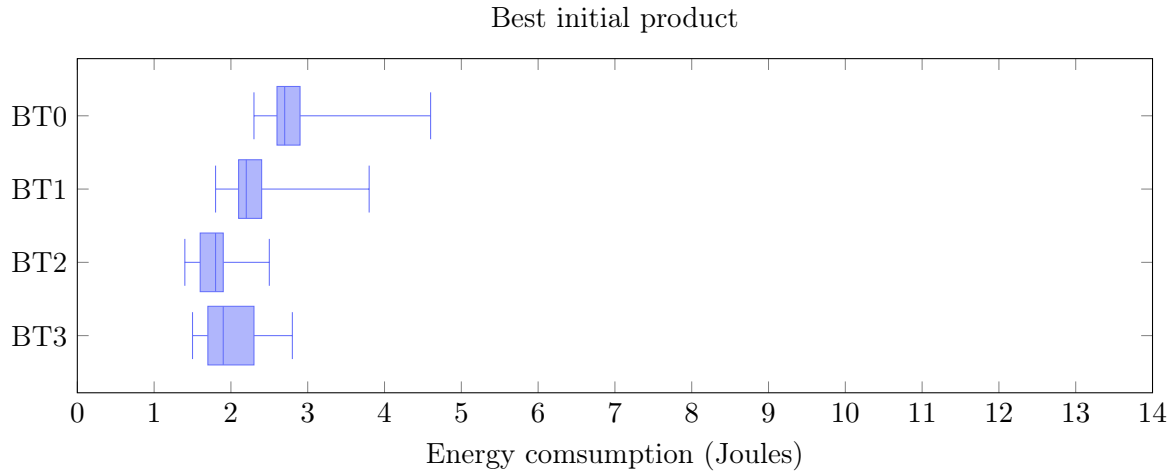


Figure 5.3: Improving the product resulting from the feature-wise analysis with the pairwise one.

this experiment is depicted in Figure 5.3, where BT_0 consumes 2.8 Joules. The pairwise analysis performed three iterations before reaching a stop criterion and returning the best product of the different iterations (BT_2), whose energy consumption is 1.9 Joules, *i.e.*, 29% lower than BT_0.

This experiment provides a partial answer to our second research question **RQ 2**: the pairwise analysis outperforms the best product of the feature-wise analysis by 30% when there is no required feature.

Behavior with required features To complete this partial answer, the third experiment is a variant of the previous experiment that takes the required features into account. The purpose of this experiment is *(i)* to ensure the changes our approaches perform on a product containing required features effectively reduce the energy consumption of such a product, and *(ii)* to evaluate these reductions. The products resulting from both approaches depend on the initial product, and on which features are required in this product. Therefore, in contrast to the previous experiment, it is not possible to assess our approaches with only one initial product. Thus, we used the FeatureIDE Product Generator to produce a sample of 520 random products—*i.e.*, 1 product tested for 2,500 products of the SPL—hereafter referred to as the *validation sample*. To mimic a real use case, we defined a random feature (based on the `java.util.Random` class) as a requirement in each of these products.

Relying on the consumption data measured on the *training sample* presented in Section 5.2.1, we applied our two energy mitigation approaches on each product from the *validation sample*. We evaluated how the products resulting from both approaches perform compared to their respective initial product. The feature-wise analysis generated 520 products, and the pairwise analysis generated 2,687 products—*i.e.*, a mean of 5 iterations per initial product. By design,

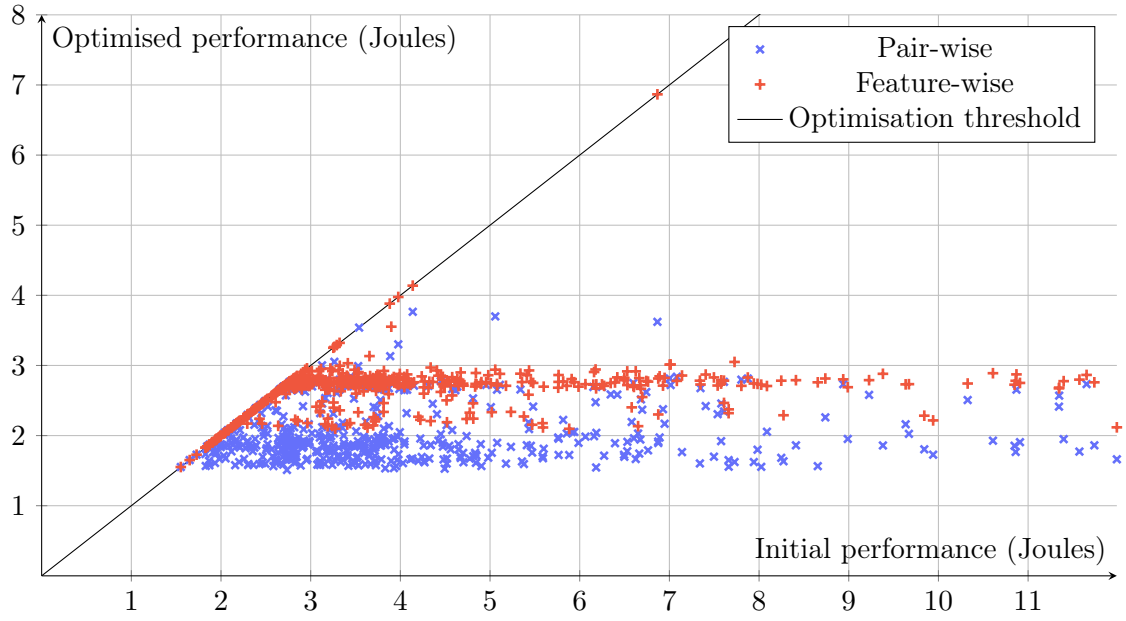


Figure 5.4: Energy consumption of the products resulting from both analyses.

the result of the first iteration of the pairwise analysis is the initial product, thus all initial products are included in these 3,207 products. We computed the performance of a product as its median energy consumption over 10 matches, for a total of 32,070 matches. Figure 5.4 presents the energy consumption of the products resulting from each analysis (on the vertical axis) depending on the energy consumption of the initial product (on the horizontal axis). Products that are on the improvement threshold line ($x = y$, identity line) performed the same as the initial product, meaning that the corresponding approach failed to reduce its consumption and returned the initial product. Products that are strictly below the improvement threshold line performed better than their respective initial products.

The feature-wise analysis improved the performance of 375 products from the *validation sample* (72%), while the pairwise analysis improved the performance of 501 products (96%). For 127 products (24%), the pairwise analysis found improvement when the feature-wise analysis failed. For 1 product (0.2%), the feature-wise analysis found improvement while the pairwise did not. Additional analysis on this specific product tends to exclude noise or measurement error as a cause for this exception.

To get a better view of the efficiency differences between the two approaches, Figure 5.5 depicts their respective relative gains—*i.e.*, by how much they reduced the energy consumption of the initial products. In the feature-wise analysis, the end of the first quartile is still at 0%, as it improved 72% of the products, whereas, with the pairwise analysis, the end of the first quartile is already near a 24% gain. The median gain of the feature-wise analysis is 20%. Regarding the pairwise analysis, such a gain is reached before the second quartile. Therefore, only half of

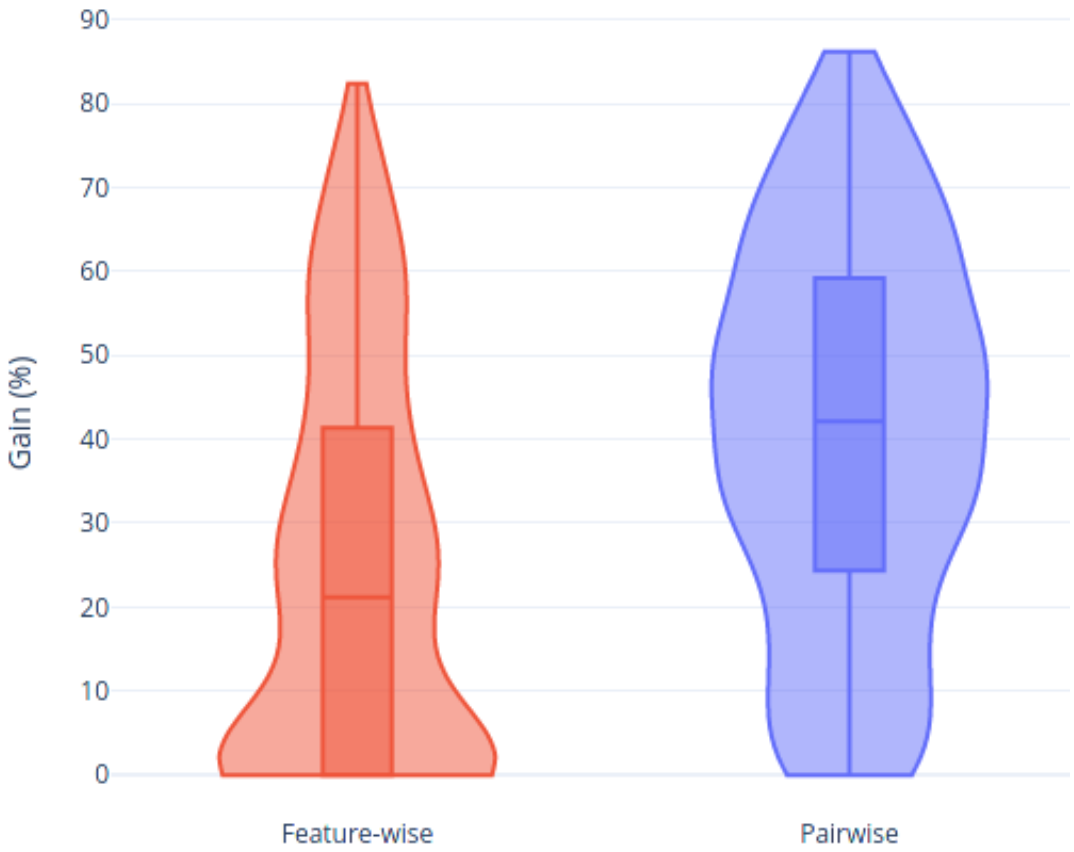
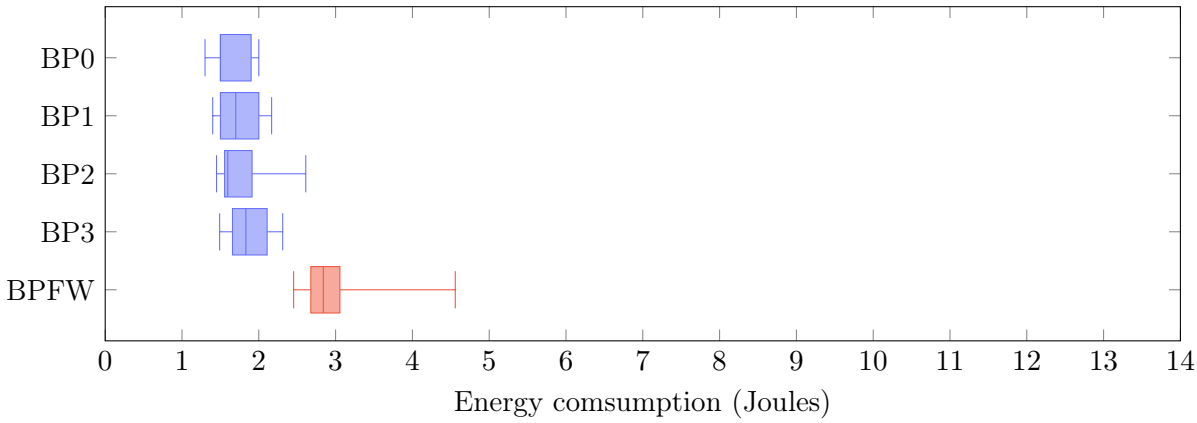


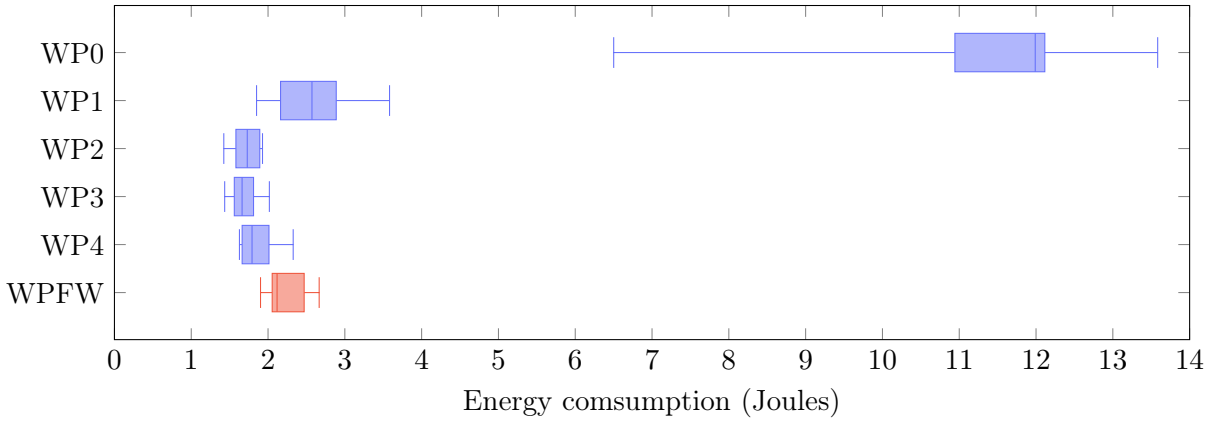
Figure 5.5: Relative gains of the pairwise and feature-wise analysis.

the products resulting from the feature-wise analysis obtained gains higher than 20%, while the pairwise analysis improved more than three-quarters of products by such a gain. Similarly, half of the products improved by the pairwise analysis were improved by 40% or more, while the feature-wise analysis had such a gain for only a quarter of the products. The maximum gain is similar for both approaches: 82% and 86%, respectively.

Figure 5.6 presents how both approaches performed on two products: those with the worst and best initial energy consumption in the *validation sample*. The initial product is designated with the subscript 0 (*e.g.*, WP_0 in Figure 5.6a). The different iterations of the pairwise analysis on the two products are designated with their respective index (WP_1 to WP_4 and BP_1 to BP_3), while the result of the feature-wise analysis used as a comparison is designated with the subscript FW ($WPFW$ and $BPFW$, respectively). Figure 5.6a depicts the product WP_0 with the worst initial energy consumption, 12 Joules. The pairwise analysis performed 4 iterations before meeting one of the stop criteria. Most of the gains are obtained after the first iteration, with WP_1 reducing the energy consumption by 78%. WP_2 and WP_3 brought additional gains of 32% and 3% on their preceding iteration, respectively. However, WP_4 increased the energy consumption by 8%, resulting in WP_3 being returned by the pairwise analysis, with an energy consumption 86% lower than WP_0. The feature-wise analysis returned a product $WPFW$ with



(a) Worst initial product.



(b) Best initial product.

Figure 5.6: Focus on the best and worst initial products from the *validation sample*.

an energy consumption 82% lower than for WP_0. The energy consumption of the product WP_3 resulting from the pairwise analysis is 21% lower than the product $WPFW$ resulting from the feature-wise analysis. Figure 5.6b depicts the product BP_0 with the best initial energy consumption, 1.6 Joules. This product is more challenging for both of our approaches, as none of them found any optimization. The pairwise analysis performed 3 iterations with energy consumption 14%, 2%, and 18% higher than BP_0, respectively. The energy consumption of the product $BPFW$ resulting from the feature-wise analysis is 83% higher than BP_0. As both approaches fail to find optimization, they return the initial product BP_0. Such results complete the partial answer to our second research question **RQ 2**.

> **RQ 2**: Both approaches can improve products, with and without required features. However, the pairwise approach outperforms the features-wise approach, by improving more products and by providing higher performance gains.

Overall, both of our approaches successfully improved products from RobocodeSPL, with or without constraints. The feature-wise and pairwise analysis thus provided useful input data about the energy consumption of features and pairs of features, which could then be used to improve products through the feature-wise and pairwise mitigation processes. The pairwise analysis improved more products than the feature-wise analysis and led to higher gains. However, although less efficient than the pairwise analysis, the feature-wise analysis is more straightforward to set up and can be used as a first intent to reduce energy consumption. It is especially relevant in the absence of feature interactions, or in systems where pairs of features are too numerous to be exhaustively measured.

5.3 Discussion

Threats to Validity To assess our approach, we ran our experiments on a specific SPL (RobocodeSPL) to measure and reduce the energy consumption of real-world products derived from this SPL. Results, such as the success rate or the relative gains, are thus only related to this single system and cannot be generalized. Nonetheless, our contribution can be easily applied to any SPL. The improvements resulting from applying our approaches to other SPL will depend on the initial energy consumption of products and the impact of feature interactions on these products. We leave the evaluation of our approach across a larger set of domains to a future study. A second threat to validity lies in the *training sample* considered. To avoid measuring all products of the SPL, we sampled the configuration space and measured 602 products, which is only 0.05% of all valid products. Such a small sample may prevent the detection of some feature interactions and therefore, energy optimization hotspots. Still, it is worth noting that despite the low number of analyzed products, significant gains were obtained on the vast majority of products using our approaches.

Limitations During the pairwise mitigation process, products are changed over several iterations. However, it might be possible that the optimal change in a given iteration prevents further improvements in the next iterations, *e.g.*, a sub-optimal change in that iteration might lead to further and greater improvements in the long term. Furthermore, this algorithm removes all non-required optional features, without taking into account their hypothetical positive interactions in the product. It does not either consider the possibility to add an optional feature to improve the energy consumption of the product.

In the SPL community, extensions to feature models have been developed to convey information about features [BTR13]. Extended feature models can be used to assign consumption data on features, to automatically apply optimizations. However, the adoption of such extensions may raise some challenges when dealing with consumption metrics associated with pairs of features.

In the green computing domain, a commonly-used means to reduce the energy consumption

of software is by refactoring inefficient code—*i.e.*, making it more efficient without changing its functional behavior. Although our analysis methods highlight features or pairs of features with high energy consumption, they do not provide fine-grained feedback nor means to identify what causes such non-energy-efficient products at low-level, *e.g.*, inefficient code or unexpected behavior.

The energy consumed to obtain the measurements for our experiments (*i.e.*, the *training sample* of 602 products) amounts to 24,328 Joules. In comparison, the highest energy saving among the 520 products of the validation sample is 10 Joules per match. Hence, we can consider that our approach is profitable in the best-case scenario after 2,433 matches with high energy savings. This might seem a significant number at first, but this result must be considered keeping in mind the 1.3×10^6 products of the SPL that can benefit from these measurements. In addition, it cannot be generalized to other SPL since this profitability threshold tightly depends on the number of features and products of the analyzed SPL.

Finally, the pairwise analysis method relies on a sample of products containing all pairs of features. As the number of features in the SPL grows, the number of pairs had a quadratic growth. For larger feature models, the use of heuristics to identify interactions between pairs of features may prove necessary.

5.4 Related Work

The approaches presented in this chapter lie at the intersection of software product lines and so-called green computing. This section discusses related work belonging to both of these research fields.

Green software Green computing approaches applied to software are mainly focused on the evaluation of energy consumption [NRS13]. Rather than evaluating software artifacts, such as functions or classes, Islam *et al.* [INB16] evaluates the energy consumption of functionalities by slicing the source code to isolate such functionalities. Then, they measure the energy consumption of these functionalities by executing and measuring the consumption of the related sliced code. While this approach works well to measure the energy consumption of features in isolation, it does not take feature interaction into consideration, as proposed by our contribution.

Other studies focus on the identification of inefficient code ("energy hotspots" or anti-patterns) [NRS15; Per+20]. For instance, Pereira *et al.* [Per+20] reports on an approach to measure the energy consumption of a software facing different input workloads. They propose to model the consumption of methods by analyzing the consumption of the software and the number of calls for each method of this software. Once such hotspots are identified, energy consumption can be mitigated through refactoring operations, such as changes in the implemented data structures or algorithms [CAR17; CA18; Our+21]. These approaches have not been designed to take variability (especially code shared among features) into account, but they can

be considered complementary to our approaches. They provide a fine-grained analysis of the energy consumption of the software at the code level, while our measures rely on the behavior (energy-wise) of the features.

Green SPL Recently, several approaches have been proposed to deal with the energy consumption of highly configurable software systems. In particular, this concern has been addressed by relying on *dynamic* SPL to reconfigure the system depending on context changes and ensure it continues meeting its green requirements [Mun+19a; HPF19; DKB14]. These approaches also take feature interactions into account, but they rely on an exhaustive detection of such interactions. This detection can be done by tools implementing dedicated heuristics, or by domain experts. Thus, this detection may be error-prone. In this chapter, the pairwise mitigation process assumes that each feature interacts with all other features, and is thus unaffected by detection errors.

Couto *et al.* also proposed different techniques to evaluate energy consumption in software product lines using static analysis [CFS21; Cou+17]. These techniques estimate the energy consumption of features in the worst-case scenario by analyzing the source code of features to deduce the energy consumption of products, whereas our approaches measure the median energy consumption of running products. Contrarily to our approaches, they did not aim at suggesting improvements to products based on their estimations, nor took feature interactions into account.

Performances Energy consumption can be generalized as a performance indicator. Considering the general problem of optimizing product performances, numerous work has been done to model and predict such performances. Siegmund *et al.* provided various contributions related to performance models and performance predictions in software product lines [Sie+15; Sie+13]. These approaches take feature interactions into account via a systematic identification. Statistical analysis of a sample of products has also already been used by Guo *et al.* to predict the performance of a product based on its configuration [Guo+13]. In this prediction approach, feature interactions are detected using the systematic approach presented in [Sie+12b]. Such works are designed to predict performances but do not suggest optimizations for poorly-performing products.

Different authors provide multi-objective optimization frameworks for configurations [Sie+12a; Ola+12; Hie+16]. Such frameworks are designed to optimize multiple performance indicators, and energy consumption can be one of them. Soltani *et al.* [Sol+12] proposes an approach relying on artificial intelligence to configure products meeting stakeholders' functional requirements, preferences, and performance goals. However, they also rely on a systematic identification of feature interactions. This approach is complementary to ours, as we do not take stakeholders' preferences into account, and this approach does not take energy consumption or feature interactions into account. By contrast to these works, our approaches do not require a system-

atic identification of feature interactions. We assume that each feature interacts with all other features.

5.5 Summary

In this chapter, we proposed a method to measure and reduce energy consumption in software product lines. This contribution is twofold. First, we showed that it is possible to estimate the energy consumption of a single feature by measuring the consumption of a small set of products containing this feature. Second, we provided a means to identify the energy consumption of pairs of features to take feature interactions into account without detecting them in a systematic way. We applied our approach to RobocodeSPL and improved the energy consumption of 96% of randomly sampled products. In particular, half of these products have seen their energy consumption reduced by at least 40%.

However, this approach is limited by the fact that the optimized configurations can be widely different from the initial configurations. Such behavior affects its applicability in real-world use cases, where any change to a configuration is a liability. Therefore, this approach can be further improved, by maximizing performance while also minimizing the number of changes to the initial configurations.

Chapter 6

Configuration Optimization with Limited Functional Impact

In the previous chapter, we introduced an approach to optimize the energy consumption of configurable software. While this approach proved very efficient, it exhibits some limitations. In particular, the approach is particularly efficient to optimize some type of variability constraints (*e.g.*, xor relationship) but is biased against other (*e.g.*, fully optional features). Furthermore, the algorithms tend to improve performance by replacing all options of a configuration, which limits its applicability in real-world use cases, where any configuration change is a liability. Finally, while energy consumption is an environmental concern, it is not the only one. Other indicators, such as the size or the resource usage, are also relevant to optimize, for instance, to allow for VM consolidation in data centers [DCM21].

Based on this observation, we propose in this chapter an approach that optimizes a configuration regarding multiple performance objectives. Contrarily to prior work that samples or predicts performance models seeking for the best configuration of the whole configuration space [Guo+13; Kal+20; Met+22; Nai+20; ŠCV19], our approach optimizes existing configurations by maximizing performance gains while minimizing changes to such configurations. The objective is to provide the developer with the best-performing configuration by altering as little as possible the initial one, in order to remain as close as possible to the developer's functional requirements. Our contribution is threefold. First, we propose ICO, a novel optimization approach for configurable systems that address the aforementioned objective. Second, we release an up-and-running Java-based implementation of the approach. Third, we provide an in-depth analysis of the behavior of our approach and assess its efficiency in a real-world system.

In the remainder of this chapter, Section 6.1 explains fundamentals and a running example. Section 6.2 explains our optimization approach. Section 6.3 and Section 6.4 present the design and results of our experiments, respectively. Section 6.5 provides a critical discussion and Section 6.7 concludes the chapter.

6.1 Motivation and Running Example

Feature models are commonly used to define the configuration space of highly-variable software systems. A feature model is a tree or a directed acyclic graph of features [MP14], organized hierarchically in parent / sub-feature(s) relationships. Features can be mandatory, optional, or alternative and the selection of a feature may require or exclude the selection of other features. While most of these relationships can be encoded in a feature tree, *require* and *exclude* relationships are usually defined using cross-tree constraints. Therefore, the feature model describes the configuration space of a software system encoded both as a feature tree and a set of cross-tree constraints. It thus defines, in an implicit yet compact way, the set of possible configurations for that software.

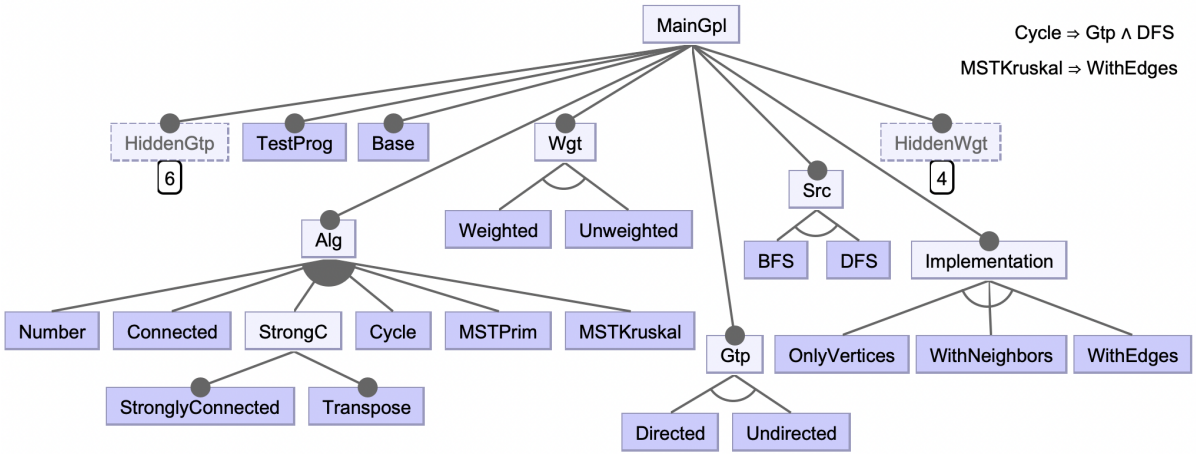


Figure 6.1: Excerpt of the feature model of GPL-FH-Java.

Figure 6.1 presents an excerpt of the GPL-FH feature model (some features, like `HiddenWgt`, are collapsed and only two constraints are shown). GPL-FH is a testbed, used in particular to evaluate different implementations and algorithms that can be executed on a graph. The graph under test is generated at runtime through the `TestProg` feature. GPL-FH exhibits 156 configurations for 37 features and 14 constraints. These features represent different characteristics of the generated graph, such as `Weighted` or `Unweighted`, and cross-tree constraints define what algorithm can be run depending on the implementation of the graph, *e.g.*, `MSTKruskal` can only be run with a `WithEdges` implementation.

When running a configuration, a few questions arise regarding its performance, such as: *Are there better (*e.g.*, faster regarding* GPL-FH*) configurations? If yes, is there one that is close enough so it still complies with the user's requirements? What would be the gain of running this configuration? How to make sure changing feature(s) will not result in a worst configuration?* These questions arise for several reasons. In particular, the large number of configurations makes picking the *best* configuration on the first try almost impossible, unless having the proper background knowledge of the configuration space. Developers usually do not have this background knowledge and only consider less than 20% of the available configurations [Xu+15]. Another

reason is the use of the default configuration or a legacy one, *e.g.*, to make sure functional requirements are met. Running such a configuration does not guarantee running the optimal one; On the contrary, it may result in running worst or incorrect configurations [Per+21; Nai+20].

In both cases, it is necessary to explore the configuration space to seek configurations providing better performance. Yet, the size of the configuration space increases exponentially with the number of functionalities, making this exploration impractical manually. There is thus a need for an approach that optimizes the performance of an existing configuration while minimizing the impact on functional requirements for such a configuration.

6.2 ICO: Iterative Configuration Optimization Approach

To address these challenges, we propose the Iterative Configuration Optimization (ICO) approach. The core idea is as follows: From an initial configuration, ICO explores the remaining configuration space in search of configurations that *(i)* are neighbors of the initial configuration, *(ii)* comply with the user's functional requirements (*i.e.*, features that have to be selected or excluded) and *(iii)* optimize given performance indicators. It then provides optimization suggestions to the developer. ICO is inspired by the energy consumption optimization approach presented in Chapter 5, but the approach in this chapter differs from the one in Chapter 5 in several aspects. In particular, the approach proposed in this chapter addresses the limitations listed in Chapter 5. That is, we propose an approach that is feature model agnostic and supports multi-objective optimization, in contrast to an optimization method that was tightly coupled to the feature model under test and dedicated to only one performance indicator, the energy consumption. In addition, ICO supports cross-tree constraints in its optimization process, while the approach presented in Chapter 5 only focused on switching selected features based on the feature tree structure.

6.2.1 Optimizing Configurations

To perform the optimization process, ICO relies on the performance of each feature regarding all the considered metrics. That is, as shown by Equation 6.1, the overall performance P of a feature f with respect to n metrics is the sum, for each metric, of p_{if} the normalized performance of the feature regarding this metric, multiplied by w_i the weight associated to this metric and by d_i the objective optimization for this metric, *i.e.*, 1 or -1, respectively to maximize or minimize.

$$P_f = \sum_{i=1}^{n} d_i w_i p_{if} \tag{6.1}$$

As interactions between features impact performance [Sie+12a], ICO is able to optimize configurations *w.r.t* tuples of features of any size, in which case f defines a tuple of features instead of a single one. The performance of a configuration is then computed as the average

performance of features - or tuples of features in interaction-wise optimization - contained in this configuration.

The ICO approach is realized by Algorithm 2, which takes the set of features, the list of constraints, and the initial configuration as input to compute a set of improvement suggestions. The algorithm starts by creating a set of candidate configurations for the configuration to optimize (lines 5– 13). Candidate configurations are the set of configurations that are one change away from the initial configuration, *i.e.*, *neighbor* configurations since they differ by the selection/deselection of one feature. For instance, a GPL-FH configuration for a `Weighted` graph is a neighbor of the same configuration where `Unweighted` graph is selected since both features are mutually exclusive. In a general way, each unselected feature leads to a candidate configuration where this feature is selected (lines 5– 7), each selected feature leads to a candidate configuration where this feature is unselected (lines 8– 10), and each exclusive relationship of both a selected and unselected features leads to a candidate configuration where the selected feature is deselected and the unselected one is selected (lines 11– 13). Candidate configurations are then ordered by performance gain (line 14), and finally filtered regarding their validity and performance (line 16), to ensure that the returned suggestions *(i)* cannot turn a valid configuration into an invalid one and *(ii)* can only improve the performance of the configuration,

Algorithm 2: ICO optimization algorithm

Input: features, constraints, $conf_{init}$;
Output: suggestions

```
1  candidates ← ∅
2  suggestions ← ∅
3  addable ← (features\conf_init)\constraints_exclude
4  removable ← conf_init\constraints_include
5  for rem ∈ removable do
6  |  candidates ← candidates ∪ newConfig(conf_init\rem)
7  end
8  for add ∈ addable do
9  |  candidates ← candidates ∪ newConfig(add ∪ conf_init)
10 end
11 for add ∈ addable, rem ∈ removable do
12 |  candidates ← candidates ∪ newConfig(addable ∪ conf_init\removable)
13 end
14 candidates ← sortByPerfGain(candidates)
15 for c ∈ candidates do
16 |  if isValid(c, constraints) ∧ perf(c) > perf(conf_init) then
17 |  |  suggestions ← suggestions ∪ diff(c, conf_init)
18 |  end
19 end
20 return suggestions
```

according to the performance model[10].

For each candidate configuration, the algorithm then computes the difference between this candidate configuration and the initial one (line 17). This difference takes the form of a feature to add or a feature to remove – or both, and its estimated performance gain. As a result, the algorithm provides a set of improvement suggestions, ordered by potential performance gains. For instance, a possible suggestion for a GPL-FH configuration is to replace the `Undirected` feature with the `Directed` feature which offers better performances, while other features remain unchanged.

The approach can thus be entirely automated by applying, while new suggestions are provided, the one providing the highest performance gain. ICO also offers an interactive mode, where developers select the suggestion to apply according to their functional requirements and domain knowledge.

6.2.2 Implementation

The ICO algorithm has been implemented as a series of tools, namely the ICO tool suite. The ICO tool suite[11] is a set of software components interacting together to help developers optimize the configuration – *w.r.t* performance indicators – of the software being developed. The ICO tool suite is composed of three tools:

- ICOLIB, a library that performs the optimizations;
- ICOCLI, a command-line tool to interact with ICOLIB;
- ICOPLUGIN, an Eclipse plugin to interact with ICOLIB.

Figure 6.2 presents the architecture of ICO. ICO executes the user's instructions regarding *(i)* the configuration to optimize, *(ii)* the feature model encoding the configuration space of the software, and *(iii)* its related performance data.

The architecture of the tool suite is flexible enough to be extended by any front-end components interacting with ICOLIB. These components take as input performance data as CSV files. Performance files can describe the performance of individual features as well as the performance of pairs of features, in order to take feature interactions into account [Sie+12a; CM06]. Such data can be a direct assessment of the feature's performances, *e.g.*, the number of lines of code, or an evaluation of their impact on configurations' performances, *e.g.*, time or energy. Through either ICOCLI or ICOPLUGIN, the user's instructions are sent to ICOLIB which in turn relies on the FeatureIDE library [Thü+14b] to perform an automated analysis of the configurations. In particular, the library checks the validity of the resulting optimized configurations returned by ICOLIB. Relying on this library also makes ICO more versatile, as it provides support for

[10]The computation of the performance model is out of the scope of this chapter. Yet, we discuss this particular point in Section 6.5.

[11]The source code is available at `https://gitlab.inria.fr/ico`

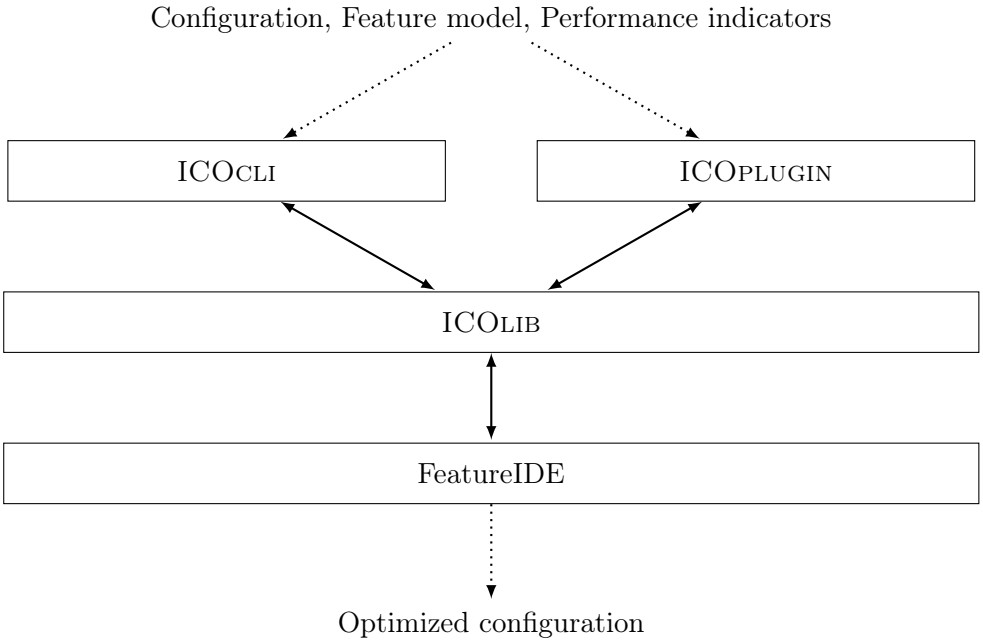


Figure 6.2: The architecture of the ICO tool suite.

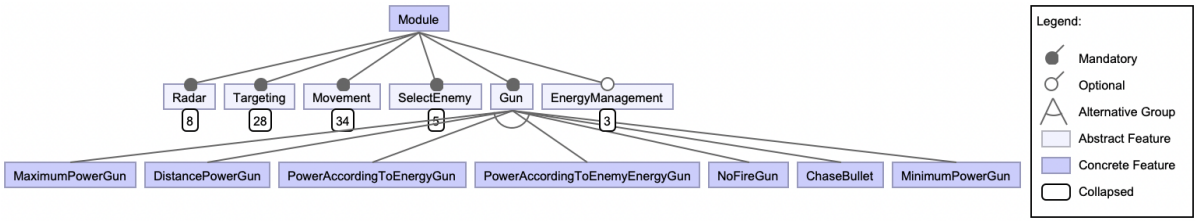
a wide range of feature model and configuration file formats, such as CNF and DIMACS for feature models or XML and Equation for configurations.

Based on the user's inputs (*e.g.*, features that have to be included or excluded for functional reasons), ICOLIB provides suggestions to improve the current configuration by maximizing its performance indicators. Precisely, ICO suggests either an addition or removal of a feature, or a substitution of a feature with another one. Such suggestions are presented to the user either after completing the configuration or in real-time during the configuration process, *e.g.*, by indicating which feature should be added next to make the configuration both valid and more efficient. As such, ICO can thus be considered both an optimizer and a recommender system [Per+16; Per+18].

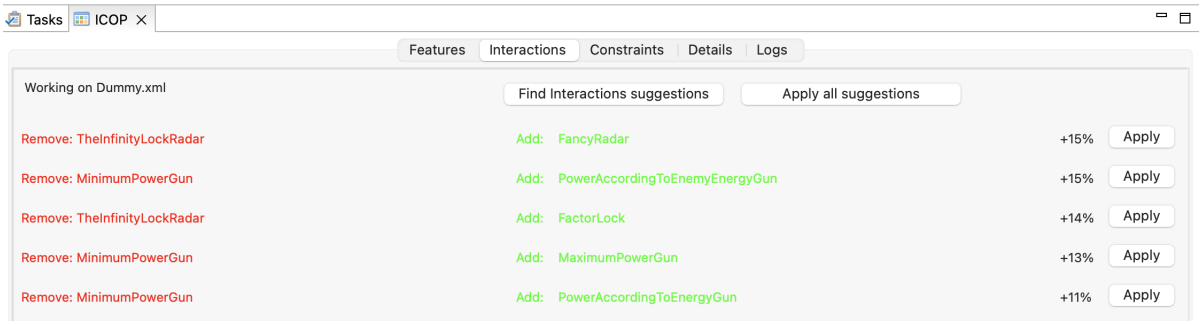
ICOlib ICOLIB is the central component of the tool suite. It provides a facade exposing the API handling all operations that can be performed with ICO: `load` a project, `display` current performances, `manage` constraints (*i.e.*, the lists of features that have to be included or excluded from the configuration), `list` or `apply` improvement suggestions and `save` the new configuration. While managing constraints and listing/applying suggestions are core functionalities of ICO, the processes of loading, validating, and saving configuration are delegated to the FeatureIDE library. Relying on the strategy pattern, ICOLIB is provided with two optimization approaches, implementing respectively the feature-wise and pairwise approaches from Chapter 5. Thanks to this architectural design, new optimization algorithms can seamlessly be integrated to ICO. The time required to generate suggestions is highly dependent on the structure of the feature model, especially on the number of features and cross-tree constraints (and

their complexity). However, as suggestions are independent of each other, their generation can be parallelized. ICOLIB current Java implementation relies on `Stream.parallelStream()` for such a task.

The ICOtool suite comes with two ICOLIB clients: the first one as an Eclipse[12] Plugin, ICOPLUGIN; the second one as a command line tool, ICOCLI.



(a) An excerpt of a feature model.



(b) The interaction suggestion tab of ICOPLUGINwhile optimizing a configuration from the feature model.

Figure 6.3: Usage of ICOPLUGIN

ICOPlugin ICOPLUGIN is an Eclipse plugin developed to interact with ICOLIB and implemented as an Eclipse view. Whenever a configuration file is selected by the user, the ICOPLUGIN view displays the five following tabs:

- Features, to list and apply feature-based suggestions;
- Interactions, to list and apply interaction-based suggestions;
- Constraints, to manage the exclusion and inclusion of features;
- Details, to monitor the current configuration performances;
- Logs, to monitor ICOLIB execution logs.

Figure 6.3b shows the Interactions suggestion tab of ICOPLUGIN. Both Features and Interactions tabs give access to the Apply all suggestions and Find suggestions functionalities of their respective mode. When listed, each suggestion exhibits an Apply button to let the developer

[12] `https://www.eclipse.org/`

Listing 6.1: Single-line mode

```
./ICOcli -P ./project -C config.xml -e feat1,feat2 -i feat3 -F -A -S -N
```

Listing 6.2: Interactive mode

```
./ICOcli
> load -P ./project -C config.xml
> exclude --add feat1,feat2
> include --add feat3
> apply -F -A
> save
> exit
```

Figure 6.4: The two modes of ICOcli.

perform the proposed suggestion. The optimized configuration belongs to the configuration space encoded by the feature model presented in Figure 6.3a. The suggestions are ordered according to their improvement rate, *i.e.*, applying the first suggestion will have the most effective impact regarding the considered performance indicators. While each suggestion can be applied individually by clicking on its related Apply button, the Apply all suggestions functionality automates the process of applying all the best suggestions at once. Precisely, ICO computes new suggestions every time the configuration is improved and applies the best suggestion recursively until no further improvements can be made. Suggestions must be recomputed to avoid overwriting improvements *e.g.*, with a first suggestion "replace A by B (+15%)" and a second suggestion "replace A by C (+10%)", applying all suggestions without recomputing them would replace A by B and then by C, which is suboptimal.

ICOcli Another means to interact with ICOlib is by using ICOcli, a lightweight command line interface program. ICOcli proposes two modes: the single-line mode, where instructions are given as parameters, and the interactive mode where instructions are given sequentially by the developer in a shell. It is also possible to mix the two modes by giving some instructions as parameters (*e.g.*, the project path) and then running the remaining ones in the shell. The single-line mode is relevant for automation (*e.g.*, for CI/CD or in an automated process) or to perform a single task, whereas the interactive mode provides a more human-friendly interaction with the tool suite, enabling an in-depth exploration of the variability of the software and its performances, *e.g.*, comparing the performance of features and interactions or the impact of constraints on suggestions.

Figure 6.4 presents the same set of tasks computed in the two different modes. The developer loads the software project `./project` and the configuration file `./project/config.xml` respectively with parameters `-P` and `-C`. Then, features `feat1` and `feat2` are added to the exclusion list and feature `feat3` is added to the inclusion list using parameters `-e` and `-i` in single-line mode and the `exclude`/`include` commands in interactive mode. All the best feature-related suggestions are then applied with parameters `-F -A` in single-line mode or by using the `apply`

`-F -A` command in interactive mode. Finally the configuration `./default.xml` is overwritten by `-S` in single-line mode or `save` in interactive mode. The developer quits the program with `exit` in interactive and mixed modes, while in single-line mode the `-N` parameter prevents the shell from opening.

Finally, ICOcli offers the ability to perform arbitrary edits to the configuration (*e.g.*, removing a feature), a functionality also offered by ICOplugin through the FeatureIDE API.

6.3 Experimental Methodology

Our goal is to assess the validity and effectiveness of our approach. In particular, we aim to answer the following research questions:

RQ 1: Can any configuration be optimized? Considering a configuration space, we investigate whether or not any configuration from that space can be optimized using our approach.

RQ 2: How effective is the ICO optimization approach? When the ICO approach provides a better configuration, we measure the performance discrepancy between that configuration and the initial one.

RQ 3: How many iterations does it take to optimize a configuration? We evaluate the number of iterations of ICO required to converge from an initial configuration to its respective optimal one.

We evaluate our approach on the real-world configurable system GPL-FH presented in Section 6.1. This system was selected for several reasons. First, both its source code and feature model are publicly available, and they seamlessly integrate as GPL-FH can be run from the command line. Second, its feature model (presented in Figure 6.1) exhibits 156 configurations, thus providing a large-enough configuration space for the optimization process to be significant.

The experiments consist in optimizing all 156 configurations regarding a pair of performance indicators, namely the execution time (`time`) and the number of lines of code (`LoC`). This exhaustive optimization highlights how the approach navigates through the configuration space. To not interfere with the `time` measurements, the logging functionality that comes as a default option of the GPL-FH system was disabled, as it might misrepresent the actual execution time. The GPL-FH default number of vertices was changed from 10 to 3500 to yield a larger graph and be able to properly measure the `time`, thus getting meaningful readings. The building time of the graph itself is excluded from the `time` measurement, since constant across configurations. In order to consolidate the measure of the `time` of each configuration, the experiment was repeated 20 times. Beyond that point, the average execution time converges.

The performance of each feature *w.r.t* `LoC` and `time` is computed according to the method proposed in Chapter 5, *i.e.*, the performance of a feature *w.r.t* a metric is the average performance in this metric of configurations containing this feature. The global performance of each

feature (*i.e.*, the performance taking all metrics into consideration) is then calculated using Formula 6.1. Both metrics were given the same weight, while the optimization goal was set to a minimization of both performance indicators. The optimization algorithm has then been applied on each of the 156 configurations of GPL-FH: for each initial configuration, it seeks for a better neighbor configuration that minimizes `LoC` and `time`. All measurements were performed on a machine with an Intel Core i5 CPU at 2.9GHz and 8GB of RAM.

6.4 Results

The configuration space of GPL-FH has been exhaustively measured, providing insight into the performance of each of the 156 configurations *w.r.t* `LoC` and `time`. Figure 6.5 presents such performances. The best and worst `time` are respectively 0.09 and 23.4 seconds, while `LoC` ranges from 282 to 632. The optimization of a configuration should thus provide higher variations in `time` than in `LoC`, as the ratio between the worst and best readings for `time` (260) is orders of magnitude higher than the one for `LoC` (2.2).

Investigating RQ1: Can any configuration be optimized? Applying the best suggestion (if any) provided by Algorithm 2 to a given configuration results in either one of the following situations: (S_1) the configuration improved regarding both performance indicators; (S_2) the configuration improved regarding one performance indicator and worsened regarding the other; (S_3) the configuration did not improve nor worsen, *i.e.*, ICO returned no suggestion; (S_4) the configuration worsened on both indicators[13].

Table 6.1 summarizes the performance gains resulting from applying Algorithm 2 on the GPL-FH configuration space regarding the four situations discussed above. Out of the 156 configurations, 138 were modified while 18 remained unchanged. Among the 138 modified configurations, 110 were improved regarding both performance indicators and 16 regarding only one. As a matter of fact, all these 16 single-indicator optimizations relate to an improvement of `LoC` at the expense of `time`. The remaining 12 configurations worsened on both performance indicators.

[13]Due to inaccuracies in the performance model. See Section 6.5 for further analysis.

Performance change	Configurations		Removed LoC			Saved Time (s)		
w.r.t indicators (*Situation*)	Count	%	worst	med.	best	worst	med.	best
Optimized - both indicators (S_1)	110	70	5	69	129	∼0	0,78	20,21
Optimized - one indicator (S_2)	16	10	5	5	76	-1,35	-0,01	∼0
Unchanged (S_3)	18	12	-	-	-	-	-	-
Worsened - both indicators (S_4)	12	8	-69	-31	-31	-0,99	-0,74	-0,15

Table 6.1: The effect of ICO on the GPL-FH configuration space.

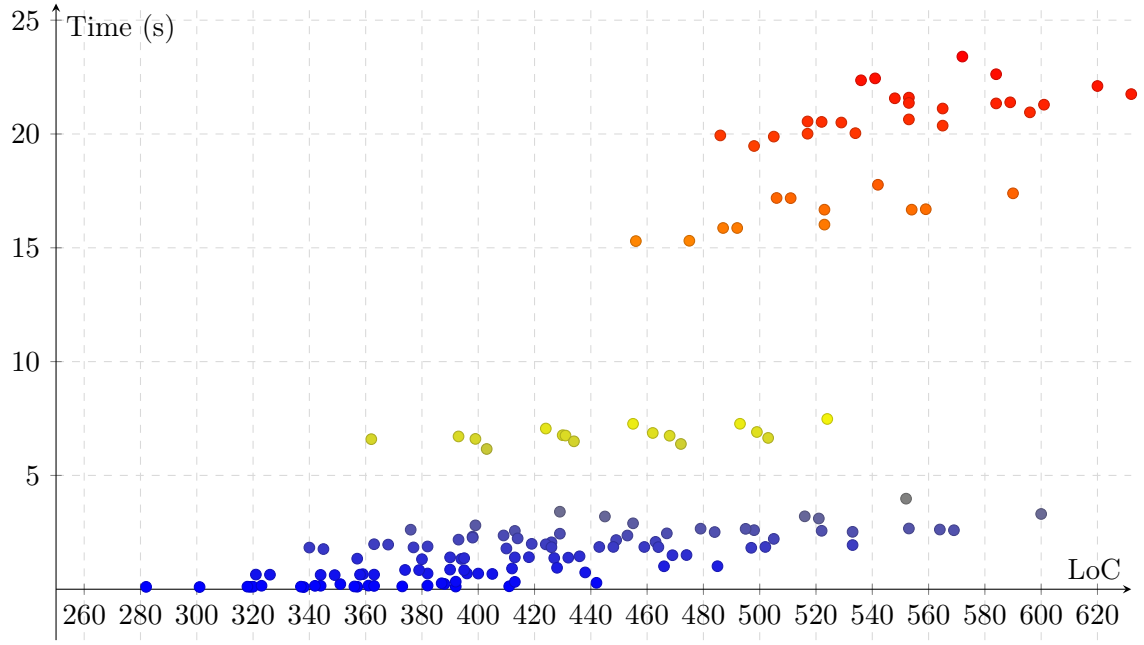


Figure 6.5: Performance of each GPL-FH configuration *w.r.t* `LoC` and `time` (lower left corner is better).

> **RQ 1**: These results show the efficiency of ICO: only 8% of the configuration space could not be improved by our approach. 12% remained unchanged as there was no way to further optimize them, and 80% were successfully optimized.

Investigating RQ2: How effective is the ICO optimization approach? Figure 6.6 shows the performance gains when running ICO on the GPL-FH configuration space. As anticipated above, variations in `time` were more significant than the `Loc`-related ones, *i.e.*, ranging from +96,6% to -133,6% regarding `time` and from +26,5% to -17,7% regarding `Loc`. The 12 configurations discussed in Table 6.1 worsen both performance indicators (situation S_4) thus represent a negative gain and as depicted below the horizontal axis and the left side of the vertical axis. The figure highlights that the performance loss on such features is very limited when compared to the performance gains in other situations.

> **RQ 2**: ICO provides efficient optimizations, especially for poorly performing configurations, but can sometimes worsen configurations' performance. Nevertheless, although worsened, these configurations remain in the top-tier performance ranking.

Investigating RQ3: How many iterations does it take to optimize a configuration? Since an initial configuration cannot be turned into an invalid one by Algorithm 2 (see line 16), running the algorithm on each configuration of the configuration space thus results in a set

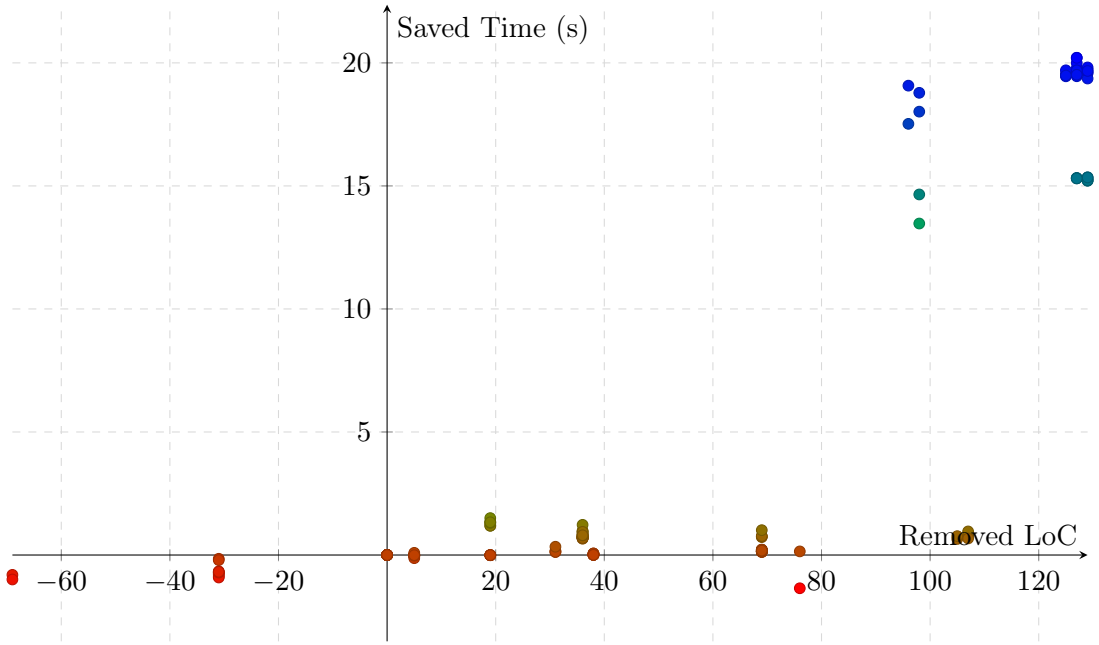


Figure 6.6: Performance gains for each GPL-FH configuration *w.r.t* `LoC` and `time` (top right corner is better).

of optimized configurations which are a subset of the initial configurations. These optimized configurations cannot be further optimized, as they have no neighbor configuration with better performances. Based on this inclusion, it is then possible to build a directed graph representing all successive iterations of the algorithm.

Figure 6.7 depicts such a graph for the GPL-FH case study, where each node represents a configuration. For the sake of readability, nodes are placed on a relative logarithmic scale representing their related configuration's `time` and `LoC`, respectively on the vertical axis and on the horizontal axis. Each edge represents the application of the first suggestion returned by Algorithm 2: the initial configuration is the source node for that edge, while the optimized configuration resulting from applying this first suggestion is the target node. Thus, an edge represents the removal of a feature, the addition of a feature, or the substitution of a feature by another one. This graph is composed of 18 disconnected sub-graphs. Each sub-graph converges towards one of the 18 configurations that could not be optimized and remained unchanged (see Table 6.1, situation S_3). These 18 configurations are thus local optima, and one of them is the global optimum.

Table 6.2 shows the number of iterations of Algorithm 2 required by all configurations to converge towards their related optimized configuration. As explained before, 18 configurations remain unchanged and therefore do not need any iteration of the ICO algorithm to reach their convergence point. Regarding the 138 other configurations, a single iteration drives 61 of them (44.2%) toward their convergence point. That is, after one iteration, 79 configurations (more than half the configuration space) have already converged. After a second iteration, 81%

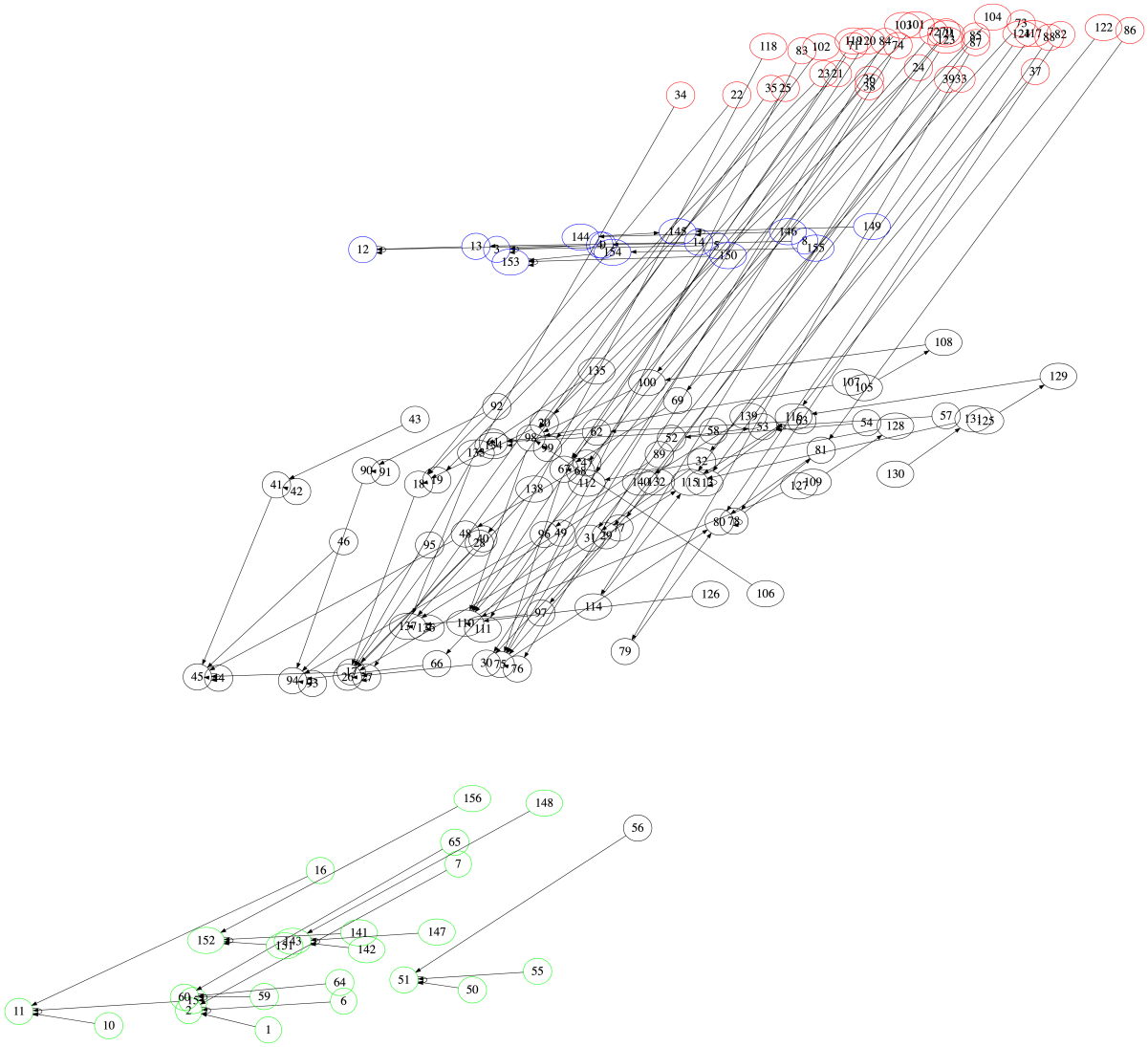


Figure 6.7: ICO transition graph between configurations of GPL-FH. Configurations on a relative logarithmic scale for readability, `time` on the vertical axis, `LoC` on the horizontal axis, lower left is better.

of configurations have reached their convergence point. Up to five iterations are required to optimize the whole set of configurations, but the last two iterations only apply to 3.8% of the configurations.

RQ 3: The number of iterations required by ICO to optimize a configuration is very limited, as *(i)* half of configurations are optimized after a single iteration and *(ii)* the number of configurations yet to be optimized decreases dramatically after each iteration. In this experiment, only 1 configuration required the maximum number of five iterations to be optimized.

Nb Iterations	0	1	2	3	4	5
Nb Configurations	18	61	48	22	6	1
Nb Configurations, cumulative	18	79	127	149	155	156
% remaining configurations	11.5	44.2	62.3	75.8	85.7	100
% total configurations	11.5	39.1	30.7	14.1	3.8	0.6
% total configurations, cumulative	11.5	51	81	95	99	100

Table 6.2: Applying ICO on the GPL-FH configuration space.

6.5 Discussion

A thorough analysis of the GPL-FH optimization graph depicted in Figure 6.7 provides additional findings regarding the ICO approach. The graph is composed of four clusters of nodes: (C_1) the green nodes at the bottom left; (C_2) the black nodes in the center; (C_3) the "horizontal" cluster of blue nodes in the upper part above cluster (C_2); and (C_4) the red nodes in the upper right corner. In particular, we observe that all the configurations located in C_4 move to C_2 once optimized. Such configurations thus share the same optimization. In particular, they are optimized by removing the `MSTPrim` feature, which is characterized by both the worst `LoC` and `time` performance.

Feature Model Design. One can also notice that no optimization edge enters or leaves clusters C_1 and C_3. These two clusters are characterized by the presence of features that are mutually dependent such as `Directed`, `WithEdges` and `DirectedWithEdges` (a sub-feature of the collapsed `HiddenGtp` feature), which are tightly-coupled by the `Directed` $\wedge$ `WithEdge` $\leftrightarrow$ `DirectedWithEdge` cross-tree constraint. In addition, C_3 contains configurations whose couple of features `StronglyConnected` and `Transpose` are selected, complying with the `StronglyConnected` $\leftrightarrow$ `Transpose` cross-tree constraint, thus preventing the removal or addition of these features. The best-performing configuration from C_3 is actually similar to a configuration from C_1, with the addition of `Transpose` and `StronglyConnected` features that, as explained before, must be removed together. It is thus impossible to enter or leave these clusters without changing two or three features at once, or without turning the configuration into an invalid transitional state, which is not supported by the ICO approach. Configurations from these clusters can only be optimized by changing other features, and configurations from C_2 and C_4 cannot be optimized towards C_1 or C_3. Feature relationships and cross-tree constraints also explain why the optimization process converges towards 18 different configurations: these configurations only contain mutually-dependent features and cannot be further improved while remaining valid. The shape of the feature model and related cross-tree constraints can thus hinder the capacity of ICO to optimize the entire configuration space.

Performance Model. To perform its optimization process, ICO relies on a performance model. This model provides an estimated performance for each feature, measured based on the method proposed in Chapter 5. As the performance model is estimated, it may contain measurement inaccuracies which in turn may impact the efficiency of the approach. For instance, we observed that, while optimizing GPL-FH, the performance of twelve configurations worsened after an iteration. When analyzing the initial and "optimized" configurations, we found out that the twelve performance regressions were caused by the addition of either the feature `Number` or `Cycle`. Both of these features happen to be present in all the configurations from C_1, the cluster of best-performing configurations. However, such good performances are actually not related to `Number` or `Cycle` alone, but to the presence of other features in combination. The performance model seems thus biased toward `Number` and `Cycle`, which causes inaccuracies during the execution of ICO.

Validity Threats. To assess our approach, we ran our experiments on a specific configurable system (GPL-FH) and optimized it based on specific metrics, *i.e.*, minimizing the execution time and the number of lines of code of configurations from this system. Results such as the performance gains or the number of iterations are thus only related to this single system, and cannot be generalized. Nonetheless, our contribution can be easily applied to any configurable system. The optimization gains resulting from applying our approach to other configurable systems will depend on the initial performance of each configuration for such systems. We leave the evaluation of our approach across a larger set of domains to a future study.

In this chapter, the performance model is built by an exhaustive assessment of GPL-FHconfigurations. While such an approach is convenient, we acknowledge that it may not be practical for other use cases, in particular, for systems with larger configuration spaces. Yet, it is still possible to use our approach by sampling or predicting performance models for such larger spaces, using approaches such as [Nai+20; Ach+22].

6.6 Related Work

The management of highly configurable systems has been widely studied in the last decade. Research has mainly addressed one of the following three areas: *(i)* performance prediction, intending to estimate the performance of a configuration without actually measuring it, *(ii)* performance optimization, to generate optimal configurations of a system, and *(iii)* recommender systems, to assist developers during the feature selection process.

Performance prediction. Performance prediction approaches have been proposed by many researchers [Kal+20; Guo+13; Sie+15]. The aim of such approaches is to build an estimated performance model of the system's features. These approaches rely on machine learning techniques to infer performance data from a sample of configurations. One of their main objectives is to detect feature interactions – as they can have a significant impact on performances, and

provide more accurate prediction than approaches that do not consider such interactions. Relying on such performance models, Siegmund *et al.* [Sie+12a; Sie+13] predict the performance of configurations as the sum of the impact of each feature on performances. Such approaches are complementary to the current performance model of ICO, which originates from Chapter 5, and can extend its current implementation.

Performance optimization. Many approaches have been proposed to address performance optimization for configurable systems. Such approaches strive to locate optimal or near-optimal configurations *w.r.t* some performance indicators. Several studies provide deterministic approaches [ŠCV19; Ola+12; Nai+20] to tackle this challenge. Other authors like Hierons *et al.* rely on genetic algorithms to minimize the number of measurements needed to optimize configurations [Hie+16] or leverage the performance predictions methods discussed above [Sie+12b]. Such approaches do not take an initial configuration into consideration. For instance, Nair *et al.* [Nai+20] start their optimization process from a random configuration. In contrast to such approaches, the approach proposed in this chapter aims at optimizing a set of performance indicators while remaining as close as possible to the initial user-defined configuration. This optimization goal is shared with the approach of Soltani *et al.*, which takes user's preferences into consideration to optimize a configuration, but yet does not support the optimization of pre-existing configurations [Sol+12].

Recommender systems. Some approaches are designed to assist users during the configuration of their systems, by providing recommendations based on the user's functional needs. For instance, Pereira *et al.* propose a visual recommender system [Per+16] based on proximity and similarity between features. Similarly, Zhang *et al.* [ZE14] use dynamic profiling and analyze the stack trace of the system to locate features that can be changed without altering the functional behavior of the system. Other approaches, such as [Met+22] or [HPF19], aim at updating the configuration of the system at runtime, to adapt it to the evolution of its environment. To the best of our knowledge, no recommender system provides suggestions based on both functional and performance considerations.

6.7 Summary

This chapter introduced ICO, an iterative approach to optimize configurations regarding defined performance indicators. Considering an initial configuration to optimize, ICO estimates the performance of neighbor configurations, *i.e.*, configurations that distinguish from the initial one by a single feature (de)selection change. ICO provides performance improvement suggestions to drive the optimization process, which can be run either in fully automated mode or based on the developer's inputs. We evaluated our approach on a real-world example by running ICO on its entire configuration space, and our experiments showed that ICO significantly improved 80% of the configurations.

Chapter 7

Conclusion

This chapter concludes this dissertation and summarizes the different contributions that were produced. Then, the short-term and long-term perspectives are discussed.

7.1 Summary of this dissertation

This section recalls the research goals and summarizes the related results that were presented in previous chapters. The overall goal of this work is to leverage the variability of software systems, *i.e.*, their configuration, to optimize their performance. Specifically, such optimizations aim at reducing their environmental impact, for instance by reducing their power usage or their resource usage.

To tackle this research goal, this dissertation first attempted to better understand the performance of configurable systems. Then, an optimization algorithm was formalized in order to optimize such systems. Finally, this algorithm was generalized to adapt to additional constraints, and in particular to reduce the number of configuration changes. The content of this dissertation is summarized as follows:

JHipster Chapter 4 provides an empirical study of the performance of a configurable software system. In particular, the configurations of JHipster, a configurable web stack generator, are exhaustively assessed to answer the following research questions:

RQ 1: How does the system's performance vary from one configuration to the other?

RQ 2: How are the different performance indicators related?

RQ 3: Do the different options of the system impact the performance of configurations in different ways?

RQ 4: Is it possible to drive the selection of high-performance configurations based on the performance of individual features?

This chapter demonstrates the significant influence of configuration choices on performance outcomes and highlights the presence of strong correlations between certain performance indicators, indicating redundancy. Furthermore, it investigates the effects of each option on

performance and illustrates that selecting high-performing options enables the creation of near-optimal configurations.

On Reducing the Energy Consumption of Software Product Lines Chapter 5 introduces an approach to optimize the energy consumption of configurable software. It provides a method to assess the performance of each option of the system, and an algorithm to improve a configuration using such performance data. Configurable software brings specific challenges, in particular, due to interaction between options. Thus, a specific focus is made on this issue with the following research questions:

RQ 1: Do our different analyses detect feature interactions impacting energy consumption?

RQ 2: How effective are our approaches to reducing the energy consumption of a product?

The assessment of our analysis methods confirms that feature interactions have a noticeable impact on energy consumption. Nevertheless, it is important to note that these interactions are not necessarily problematic, they can be leveraged to further enhance performances. In particular, configurations that were optimized *w.r.t* interactions exhibited a 30% decrease in energy consumption compared to configurations optimized solely based on individual features.

Configuration Optimization with Limited Functional Impact Chapter 6 is a deepening of the previous chapters and addresses several of its limitations. In particular, the reality of configurable software is that configurations are created for a specific reason, and cannot be changed freely to maximize performance. Otherwise, the configuration would not meet the functional constraints of the system. Thus, this chapter introduces an algorithm to iteratively create optimization suggestions, which can then be curated by the user. The effectiveness and behavior of this algorithm is assessed with the following research questions:

RQ 1: Can any configuration be optimized?

RQ 2: How effective is the ICOoptimization approach?

RQ 3: How many iterations does it take to optimize a configuration?

The main highlight of this chapter is that initial attempts to optimize a configuration produce the most significant outcomes. Even an inefficient configuration can become near-optimal with very few iterations of the algorithm, thus reducing the risk of breaking the configuration.

The objective of this dissertation is to improve the understanding of performance in variable software. Across the chapters, this document presents a progression from initial efforts to comprehend the impact of options on performance to the development of an automated, open-source optimization tool.

7.2 Contributions

The writing of this dissertation was punctuated with several intermediary contributions. In particular, the following research papers were published during this period.

- **[SPLC'21] On reducing the energy consumption of software product lines.** Édouard Guégain, Clément Quinton, and Romain Rouvoy. Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A. 2021. [GQR21] Rewarded with the *Artifacts available* and *Artifacts evaluated and functional* ACM badges.

- **[CAISE'23] Configuration optimization with limited functional impact.** Édouard Guégain, Amir Taherkordi, and Clément Quinton. International Conference on Advanced Information Systems Engineering. Cham: Springer Nature Switzerland, 2023. [GTQ23a]

- **[ASEW'23] ICO: A Platform for Optimizing Highly Configurable Systems.** Édouard Guégain and Clément Quinton. 2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). IEEE, 2023. [GTQ23b]

- **On the interplay between performance indicators in configurable software: the JHipster case study** Édouard Guégain, Alexandre Bonvoisin and Clément Quinton. Submitted, 2023.

In addition, several tools and datasets were shared with the community.

- **Performance and replication data of RobocodeSPL**, containing the results and the tools to replicate the experiments of [SPLC'21]. `https://doi.org/10.5281/zenodo.5048316`

- **Performance and replication data of GPL-FH-java**, containing the results and tools to replicate the experiments of [CAISE'23]. `https://doi.org/10.5281/zenodo.8144454`

- **Performance data of JHipster**, containing the results of "On the interplay between performance indicators in configurable software: the JHipster case study." `https://doi.org/10.5281/zenodo.8140600`

- **The ICO tool suite**, a set of open-source tools to automate the optimization of configurable software. `https://gitlab.inria.fr/ico/`

7.3 Short term perspectives

This dissertation explores the optimization of configurable software and presents a validated optimization algorithm from various perspectives. However, the focus of the analysis was narrow, resulting in a limited exploration of use cases and applications. This section thus presents work that was either planned or considered but did not directly align with the primary objective of this dissertation. Yet, these works remain relevant and can be pursued in the future.

Exploring new use cases In this dissertation, only a limited set of configurable systems is considered. The project JHipster falls into the category of Service-oriented systems, while RobocodeSPL and GPL-FH-java were developed following the feature-oriented programming philosophy. Thus, the other implementations of variability disussed in Chapter 2 such as Plugin, Packages, Parameters and Configuration, were not assessed and require additional validation. In particular, the variability of the Linux Kernel, implemented as Conditional Compilation, is notoriously challenging due to its size. Therefore, additional work is needed to assess the performance of the algorithm on such large systems. In particular, beyond the quality of the suggestions, the execution time of the algorithm has not yet been empirically assessed on larger variability models. The configuration of a software component is not the only factor in its performance. In the literature, the impacts of the tasks given to a variable software on its performance have already been studied [Les+23; LME13]. However, such measures were not performed *w.r.t* energetic performance. Thus, such experiments could be replicated to assess the impact of a certain task on the energetic performance of configurable systems. Such works can be extended by discretizing the different tasks into categories, in order to detect which options are the most relevant for each category. Such work could provide additional performance gain, by creating software configurations that are optimized specifically for their current task.

Beyond software variability, the optimization algorithm may be relevant to other types of variability. In particular, cyber-physical systems are variable with regard to both their software and their hardware. Thus, the performance of such systems is conditioned by both software performance and hardware suitability. Furthermore, the adequacy between the software configuration and the hardware configuration may impact the performance of the systems. In particular, the field of the Internet of Things, Fog computing, and Edge computing has a growing share of ICT greenhouse gas emissions and is thus a particularly relevant domain. However, modeling hardware variability may require an extension of variability models. For instance, physical devices can have quantitative constraints, such as their number, their range, or their spacial coordinate [QRD14]. Thus, the algorithm will need additional improvement to support such extended variability models.

Improving the algorithm The main limitation of the optimization algorithm presented in this dissertation is related to the number of changes performed on a configuration in a single iteration, limited to at most one addition and one deletion. As discussed in Chapter 6, this behavior prevents the algorithm from exploring more ways to improve performances. In particular, the design of the variability model, *i.e.*, the representation of the variability, not the variability itself, can lead the configuration toward a local optimum, rather than a global optimum.

Therefore, the algorithm could be significantly improved by allowing more changes in a single iteration. The challenge here is not technical but instead functional. Indeed, the more changes are performed to the configuration at once, the further the improved configuration is

from the initial configuration. With no limit to this distance, the algorithm will, in a single iteration, remove all the options from the configuration, and select the best options. The new behavior does not reflect the purpose of the algorithm, which is to stay as close as possible to the initial configuration.

A solution to this dilemma would be to let the user define a maximal number of changes in an iteration. In such a scenario, no objective criteria could help the user decide on such a threshold. Furthermore, the user may wonder if increasing this threshold could yield better performance gains. Such a manual process also goes against the purpose of the algorithm. Thus, additional research is necessary to identify such an objective criterion and thus to define the threshold.

Another solution would be to perform the algorithm without a change limit and to provide the suggestions to the user, sorted by their number of changes as a first criterion, and then by their expected performance gain as a second criterion. The downside of this solution is that the number of suggestions may become very high and thus deter the user from exploring them. Thus, a new criterion is necessary to sort the suggestions, for instance by assessing their relevance. There is no current solution to perform this task, and thus additional research is also necessary to explore this solution.

7.4 Long-term perspectives

The short-term perspectives, presented above, can be addressed with the current state of the art. However, during the redaction of this dissertation, new perspectives arose that will require extensive intermediary work, both in research and in engineering. Such projects are thus classified as long-term perspectives, and they are introduced in this section.

Unifying variability In this dissertation, different granularities of variability were considered. JHipster offers a coarse-grained variability, where the configuration allows the selection of whole services, such as a database management service or a cache service. In that regard, the variable system is a software stack, and each option is a component. On the other hand, RobocodeSPL and GPL-FH-java offer fine-grained variability, where the configuration alters the source code of the system. The variability of such systems is thus internal to a single component.

When configuring a software stack, the internal configuration of each component is invisible. Thus, there is no way to ensure that the internal configuration of each component is relevant and ensures high performance. Therefore, there is a research gap in the unification of different levels of granularity. A comprehensive variability of software would encompass the hardware and network context, the variability of the operating system, the virtualization layers, the software stack, and the internal configuration of each component of the stack. Such a system would allow for high-performance configuration, with regard to both individual options and interactions.

However, building such a system is challenging. In particular, building such an exhaustive variability model may prove unfeasible. The number of options of such a model would be the sum of all the options of the subsystems (*i.e.*, the operating system, the virtualization layer, the software stack, and each component of the stack). Furthermore, some constraints may appear across different systems, such as an incompatibility between a given option from a specific component of the stack, and a certain option of the operating system. Such cross-layer constraints may prove impossible to detect a priori without additional research.

Furthermore, such a system would contain different types of variabilities: for instance, the software stack may be implemented as a service-oriented architecture, while each component relies on parameters and configuration, and finally the operating system on a conditional compilation. Even within a single type of variability, each individual system may have specific requirements regarding how configurations are managed. Thus, converting a specific configuration (composed of options related to the operation system, the virtualization layers, the software stack, and each component of the stack) into an executable product is a strong technical challenge, that may require specific research.

Automating the experimental setup Optimizing a configurable system can be decomposed into a set of tasks. First, the practitioner needs a model of the variability of the system. If the system was designed to be variable, such a model may exist. It is rarely the case, and thus the model must be built. While some automation tools exist, they only work for the internal variability of a software component, with constraints such as the supported languages. Thus, the process remains largely manual. Then, the performance of each option must be assessed. In this dissertation, the performance of each option is estimated by a regression of the performance of the configuration containing them. Thus, options are assessed by measuring a set of configurations. However, the total number of configurations can be too high to be systematically measured, and some sampling techniques may be required. The criterion to choose between a systematic measure and sampling is unclear, but this decision may impact the precision of the measures. Then, each configuration must be evaluated. The time required for such a task can remain limited for static indicators. However, regarding dynamic indicators, the measurement process may last minutes or even hours for each configuration. Such a time frame requires specific experiment devices, and a level of automation to launch the configurations and gather the results. Finally, the analysis of the results is work-intensive. For instance, several validations must be performed to ensure that all options were properly measured and that all the performance indicators were properly collected. The interpretation of the results is also challenging. In particular, there is no clear consensus about the criteria to detect feature interactions.

The complexity of this process makes it hard to automatically assess a large set of configurable systems. Therefore, there is a need for an abstraction of the experimental setup, to simplify the replication of a specific experiment and the transfer of skills to new variable systems. Such an abstraction would encompass building the variability model of a system, de-

ciding whether to use sampling or not, deploying the benchmark, evaluating each configuration, computing the performance of each option, validating this performance, and finally providing insight into the results, such as the best or worst options, or the presence of interactions.

Bibliography

[Ach+14] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. "Extraction and evolution of architectural variability models in plugin-based systems". In: *Software & Systems Modeling* 13 (2014), pp. 1367–1394.

[Ach+22] Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Olivier Barais, and Juliana Alves Pereira. "Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size". In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*. SPLC '22. Graz, Austria: Association for Computing Machinery, 2022, pp. 85–96. ISBN: 9781450394437.

[AK09] Sven Apel and Christian Kästner. "An overview of feature-oriented software development." In: *J. Object Technol.* 8.5 (2009), pp. 49–84.

[AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. "Featurehouse: Language-independent, automated software composition". In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 221–231.

[Al-+16a] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. "IncLing: efficient product-line testing using incremental pairwise sampling". In: *ACM SIGPLAN Notices* 52.3 (2016), pp. 144–155.

[Al-+16b] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. "Tool demo: testing configurable systems with featureIDE". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2016, pp. 173–177.

[Al-+17] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. "Delta-oriented product prioritization for similarity-based product-line testing". In: *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*. IEEE. 2017, pp. 34–40.

[Ape+10] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kastner. "Detecting Dependences and Interactions in Feature-Oriented Design". In: *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*. ISSRE '10. 2010, pp. 161–170. ISBN: 9780769542553.

[Ape+11] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. "Detection of Feature Interactions Using Feature-Aware Verification". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. 2011, pp. 372–375. ISBN: 9781457716386.

[Ape+13] Sven Apel, Alexander Von Rhein, Thomas Thüm, and Christian Kästner. "Feature-Interaction Detection Based on Feature-Based Specifications". In: *Comput. Netw.* 57.12 (Aug. 2013), pp. 2399–2409. ISSN: 1389-1286.

[Ape+16] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines*. Springer, 2016.

[BB19] Ethan Bommarito and Michael Bommarito. "An empirical analysis of the python package index (pypi)". In: *arXiv preprint arXiv:1907.11073* (2019).

[BBR06] Don Batory, David Benavides, and Antonio Ruiz-Cortes. "Automated analysis of feature models: challenges ahead". In: *Communications of the ACM* 49.12 (2006), pp. 45–47.

[BHK11] Don Batory, Peter Höfner, and Jongwook Kim. "Feature Interactions, Products, and Composition". In: *SIGPLAN Not.* 47.3 (Oct. 2011), pp. 13–22. ISSN: 0362-1340.

[Bod+13] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. "Spllift: Statically analyzing software product lines in minutes instead of years". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 355–364.

[Bou+13] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "Powerapi: A software library to monitor the energy consumed at the process-level". In: *ERCIM News* 2013.92 (2013).

[Bou+23] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. "Reasoning on Feature Models: Compilation-Based vs. Direct Approaches". In: *arXiv preprint arXiv:2302.06867* (2023).

[BSR10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. "Automated analysis of feature models 20 years later: A literature review". In: *Information systems* 35.6 (2010), pp. 615–636.

[BTC05] David Benavides, Pablo Trinidad, and Antonio Ruiz Cortés. "Using Constraint Programming to Reason on Feature Models." In: *SEKE*. Vol. 5. 2005, pp. 677–682.

[BTR13] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. "Automated reasoning on feature models". In: *Seminal Contributions to Information Systems Engineering: 25 Years of CAiSE* (2013), pp. 361–373.

[CA18] Luis Cruz and Rui Abreu. *Using Automatic Refactoring to Improve Energy Efficiency of Android Apps*. 2018. arXiv: `1803.05889 [cs.SE]`.

[CAR17] Luis Cruz, Rui Abreu, and Jean-Noël Rouvignac. "Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring". In: *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. MOBILESoft '17. 2017, pp. 205–206. ISBN: 9781538626696.

[CFS21] Marco Couto, João Paulo Fernandes, and João Saraiva. "Statically Analyzing the Energy Efficiency of Software Product Lines". In: *Journal of Low Power Electronics and Applications* 11.1 (2021), p. 13.

[Cla+10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. "Model checking lots of systems: efficient verification of temporal properties in software product lines". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 335–344.

[CM06] Muffy Calder and Alice Miller. "Feature interaction detection by pairwise analysis of LTL properties—A case study". In: *Formal Methods in System Design* 28 (3 May 2006), pp. 213–261. ISSN: 0925-9856.

[Col+18] Maxime Colmant, Romain Rouvoy, Mascha Kurpicz, Anita Sobe, Pascal Felber, and Lionel Seinturier. "The next 700 CPU power models". In: *Journal of Systems and Software* 144 (2018), pp. 382–396.

[Cou+17] Marco Couto, Paulo Borba, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva. "Products Go Green: Worst-Case Energy Consumption in Software Product Lines". In: *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*. SPLC '17. 2017, pp. 84–93. ISBN: 9781450352215.

[dat20] Our World in data. *Co2 emissions from aviation*. `https://ourworldindata.org/co2-emissions-from-aviation/`. Accessed: 2023-05-18. 2020.

[DCM21] Alexandre HT Dias, Luiz HA Correia, and Neumar Malheiros. "A systematic literature review on virtual machine consolidation". In: *ACM Computing Surveys (CSUR)* 54.8 (2021), pp. 1–38.

[DKB14] Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. "Probabilistic Model Checking for Energy Analysis in Software Product Lines". In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. 2014, pp. 169–180. ISBN: 9781450327725.

[DPW16] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. "A Validation of DRAM RAPL Power Measurements". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. Alexandria, VA, USA: Association for Computing Machinery, 2016, pp. 455–470. ISBN: 9781450343053.

[Dub+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. "An exploratory study of cloning in industrial software product lines". In: *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE. 2013, pp. 25–34.

[EEM10] Neil A Ernst, Steve Easterbrook, and John Mylopoulos. "Code forking in open-source software: a requirements perspective". In: *arXiv preprint arXiv:1004.2889* (2010).

[EKS20] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. "Fast static analyses of software product lines: An example with more than 42,000 metrics". In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. 2020, pp. 1–9.

[End+04] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, and Tony Newling. *Patterns: service-oriented architecture and web services*. IBM Corporation, International Technical Support Organization New York, NY ..., 2004.

[Fea17] FeatureIDE. *MS Windows NT Kernel Description*. 2017. URL: `https://featureide.github.io/` (visited on 03/08/2023).

[FRS20] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. "Smartwatts: Self-calibrating software-defined power meter for containers". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, pp. 479–488.

[Git20] Github. *Github Octoverse 2020*. `https://octoverse.github.com/2020/`. Accessed: 2023-05-18. 2020.

[Git22] Github. *Github Octoverse 2022*. `https://octoverse.github.com/2022/developer-community/`. Accessed: 2023-05-18. 2022.

[GL18] Javad Ghofrani and Daniel Lübke. "Challenges of Microservices Architecture: A Survey on the State of the Practice." In: *ZEUS* 2018 (2018), pp. 1–8.

[Göt+20] Hendrik Göttmann, Lars Luthmann, Malte Lochau, and Andy Schürr. "Real-time-aware reconfiguration decisions for dynamic software product lines". In: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*. 2020, pp. 1–11.

[GQR21] Édouard Guégain, Clément Quinton, and Romain Rouvoy. "On reducing the energy consumption of software product lines". In: ACM, Sept. 2021, pp. 89–99. ISBN: 9781450384698.

[Gre19] GreenIT.fr. *Empreinte environmental du numérique mondial.* `https://www.greenit.fr/empreinte-environnementale-du-numerique-mondial/`. Accessed: 2023-05-18. 2019.

[GTQ23a] Edouard Guégain, Amir Taherkordi, and Clément Quinton. "Configuration optimization with limited functional impact". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2023, pp. 53–68.

[GTQ23b] Edouard Guégain, Amir Taherkordi, and Clément Quinton. "ICO: A Platform for Optimizing Highly Configurable Systems". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE. 2023, pp. 62–67.

[Guo+13] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. "Variability-Aware Performance Prediction: A Statistical Learning Approach". In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE'13. 2013, pp. 301–311. ISBN: 9781479902156.

[Guo+18] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. "Data-efficient performance learning for configurable systems". In: *Empirical Software Engineering* 23 (2018), pp. 1826–1867.

[Hal+17] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Patrick Heymans. "Yo variability! JHipster: a playground for web-apps analyses". In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*. 2017, pp. 44–51.

[Hal+19] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. "Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack". In: *Empirical Software Engineering* 24 (2019), pp. 674–717.

[Hei+17] Franz C Heinrich, Alexandra Carpen-Amarie, Augustin Degomme, Sascha Hunold, Arnaud Legrand, Anne-Cécile Orgerie, and Martin Quinson. "Predicting the performance and the power consumption of mpi applications with simgrid". In: (2017).

[Hem08] Adithya Hemakumar. "Finding Contradictions in Feature Models." In: *SPLC (2)*. 2008, pp. 183–190.

[Hie+16] Robert M. Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zheng. "SIP: Optimal Product Selection from Feature Models Using Many-Objective Evolutionary Optimization". In: *ACM Trans. Softw. Eng. Methodol.* 25.2 (Apr. 2016). ISSN: 1049-331X.

[Hor+21] Jose-Miguel Horcas, José A Galindo, Ruben Heradio, David Fernandez-Amoros, and David Benavides. "Monte Carlo tree search for feature model analyses: a general framework for decision-making". In: *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A*. 2021, pp. 190–201.

[HPF19] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. "Context-aware energy-efficient applications for cyber-physical systems". In: *Ad Hoc Networks* 82 (2019), pp. 15–30.

[INB16] Syed Islam, Adel Noureddine, and Rabih Bashroush. "Measuring energy footprint of software features". In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 2016, pp. 1–4.

[Jag+17] Erik Jagroep, Giuseppe Procaccianti, Jan Martijn van der Werf, Sjaak Brinkkemper, Leen Blom, Rob van Vliet, et al. "Energy efficiency on the product roadmap: an empirical study across releases of a software product". In: *Journal of Software: Evolution and process* 29.2 (2017), e1852.

[JHF11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. "Properties of realistic feature models make combinatorial testing of product lines feasible". In: *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings 14*. Springer. 2011, pp. 638–652.

[JHF12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. "An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models". In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. SPLC '12. 2012, pp. 46–55. ISBN: 9781450310949.

[Kal+20] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "The interplay of sampling and machine learning for software performance prediction". In: *IEEE Software* 37.4 (2020), pp. 58–66.

[Kha+18] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. "RAPL in Action: Experiences in Using RAPL for Power measurements". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3.2 (2018), pp. 1–26.

[Kim+10] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. "Reducing configurations to monitor in a software product line". In: *Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings 1*. Springer. 2010, pp. 285–299.

[Kol+13] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. "A comparison of product-based, feature-based, and family-based type checking". In: *ACM SIGPLAN Notices* 49.3 (2013), pp. 115–124.

[LeB+15] Michael LeBeane, Jee Ho Ryoo, Reena Panda, and Lizy Kurian John. "Watt watcher: fine-grained power estimation for emerging workloads". In: *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2015, pp. 106–113.

[Les+23] Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. "Input sensitivity on the performance of configurable systems an empirical study". In: *Journal of Systems and Software* 201 (2023), p. 111671. ISSN: 0164-1212.

[LH22] Olivier Le Goaer and Julien Hertout. "ecoCode: a SonarQube Plugin to Remove Energy Smells from Android Projects". In: *37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–4.

[LME13] Max Lillack, Johannes Müller, and Ulrich W Eisenecker. "Improved prediction of non-functional properties in software product lines with domain context". In: *Software Engineering 2013* (2013).

[Mar+13] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. "Practical pairwise testing for software product lines". In: *Proceedings of the 17th international software product line conference*. 2013, pp. 227–235.

[MBM14] I Made Murwantara, Behzad Bordbar, and Leandro L Minku. "Measuring energy consumption for web service product configuration". In: *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services*. 2014, pp. 224–228.

[Mei+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering software variability with FeatureIDE*. Springer, 2017.

[Met+20] Andreas Metzger, Clément Quinton, Zoltán Ádám Mann, Luciano Baresi, and Klaus Pohl. "Feature model-guided online reinforcement learning for self-adaptive services". In: *Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings 18*. Springer. 2020, pp. 269–286.

[Met+22] Andreas Metzger, Clément Quinton, Zoltán Mann, Luciano Baresi, and Klaus Pohl. "Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration". In: *Computing* (Mar. 2022).

[MP14] Andreas Metzger and Klaus Pohl. "Software Product Line Engineering and Variability Management: Achievements and Challenges". In: *Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*. 2014, pp. 70–84.

[MPF22] Daniel-Jesus Munoz, Mónica Pinto, and Lidia Fuentes. "Quality-Aware Analysis and Optimisation of Virtual Network Functions". In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*. SPLC '22. Graz, Austria: Association for Computing Machinery, 2022, pp. 210–221. ISBN: 9781450394437.

[MTZ18] Jabier Martinez, Xhevahire Tërnava, and Tewfik Ziadi. "Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study". In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. SPLC '18. 2018, pp. 132–142. ISBN: 9781450364645.

[Mun+19a] Daniel-Jesus Munoz, José A. Montenegro, Mónica Pinto, and Lidia Fuentes. "Energy-aware environments for the development of green applications for cyber–physical systems". In: *Future Generation Computer Systems* 91 (Feb. 2019), pp. 536–554. ISSN: 0167739X.

[Mun+19b] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. "Uniform random sampling product configurations of feature models that have numerical features". In: *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. 2019, pp. 289–301.

[Mvn23] Mvnrepository. *mvnrepository.com*. `https://mvnrepository.com/repos/`. Accessed: 2023-05-18. 2023.

[Nai+20] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. "Finding Faster Configurations Using FLASH". In: *IEEE Transactions on Software Engineering* 46.7 (2020), pp. 794–811.

[NIB16] Adel Noureddine, Syed Islam, and Rabih Bashroush. "Jolinar: analysing the energy footprint of software applications". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016, pp. 445–448.

[NKN14] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. "Exploring variability-aware execution for testing plugin-based web applications". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 907–918.

[NRS13] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "A Review of Energy Measurement Approaches". In: *SIGOPS Oper. Syst. Rev.* 47.3 (Nov. 2013), pp. 42–49. ISSN: 0163-5980.

[NRS15] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "Monitoring energy hotspots in software". In: *Automated Software Engineering* 22 (3 Sept. 2015). ISSN: 0928-8910.

[Ola+12] Rafael Olaechea, Steven Stewart, Krzysztof Czarnecki, and Derek Rayside. "Modelling and Multi-Objective Optimization of Quality Attributes in Variability-Rich Software". In: *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*. NFPinDSML '12. 2012. ISBN: 9781450318075.

[OMR10] Sebastian Oster, Florian Markert, and Philipp Ritter. "Automated incremental pairwise testing of software product lines". In: *Software Product Lines: Going Beyond: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings 14*. Springer. 2010, pp. 196–210.

[Our+20] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. "On reducing the energy consumption of software: From hurdles to requirements". In: *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2020, pp. 1–12.

[Our+21] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. "Tales from the code# 1: The effective impact of code refactorings on software energy consumption". In: *ICSOFT 2021-16th International Conference on Software Technologies*. 2021.

[Pan+16] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. "What Do Programmers Know about Software Energy Consumption?" In: *IEEE Software* 33.3 (2016), pp. 83–89.

[Per+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. "Automated and scalable t-wise test case generation strategies for software product lines". In: *2010 Third international conference on software testing, verification and validation*. IEEE. 2010, pp. 459–468.

[Per+16] Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. "A feature-based personalized recommender system for product-line configuration". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2016, pp. 120–131.

[Per+18] Juliana Alves Pereira, Sandro Schulze, Sebastian Krieter, Márcio Ribeiro, and Gunter Saake. "A Context-Aware Recommender System for Extended Software Product Line Configurations". In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems* (2018).

[Per+20] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. "SPELLing out energy leaks: Aiding developers locate energy inefficient code". In: *Journal of Systems and Software* 161 (Mar. 2020), p. 110463. ISSN: 01641212.

[Per+21] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. "Learning software configuration spaces: A systematic literature review". In: *Journal of Systems and Software* 182 (2021), p. 111044. ISSN: 0164-1212.

[Pet+19] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. "Product sampling for product lines: the scalability challenge". In: *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. 2019, pp. 78–83.

[Pre19] Christopher Preschern. "Patterns to Escape the #ifdef Hell". In: *Proceedings of the 24th European Conference on Pattern Languages of Programs*. EuroPLop '19. Irsee, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362061.

[Pro21] The Shift Project. *Sythèse numérique et 5G*. `https://theshiftproject.org/wp-content/uploads/2021/03/Synthese_Numerique-et-5G_30-mars-2021.pdf`. Accessed: 2023-05-18. 2021.

[QRD13] Clément Quinton, Daniel Romero, and Laurence Duchien. "Cardinality-based feature models with constraints: a pragmatic approach". In: *Proceedings of the 17th International Software Product Line Conference*. 2013, pp. 162–166.

[QRD14] Clément Quinton, Daniel Romero, and Laurence Duchien. "Automated selection and configuration of cloud environments using software product lines principles". In: *2014 IEEE 7th International Conference on Cloud Computing*. IEEE. 2014, pp. 144–151.

[Sar+15] Atrisha Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. "Cost-efficient sampling for performance prediction of configurable systems (t)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 342–352.

[ŠCV19] Ivan Švogor, Ivica Crnković, and Neven Vrček. "An extensible framework for software configuration optimization on heterogeneous computing systems: Time and energy case study". In: *Information and Software Technology* 105 (2019), pp. 30–42. ISSN: 0950-5849.

[Sie+12a] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. "Predicting Performance via Automated Feature-Interaction Detection". In: *Proceedings of the 34th International*

Conference on Software Engineering. ICSE '12. Zurich, Switzerland, 2012, pp. 167–177. ISBN: 9781467310673.

[Sie+12b] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. "SPL Conqueror: Toward optimization of non-functional properties in software product lines". In: *Software Quality Journal* 20.3 (2012), pp. 487–517.

[Sie+13] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. "Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption". In: *Information and Software Technology* 55.3 (2013), pp. 491–507. ISSN: 0950-5849.

[Sie+15] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-Influence Models for Highly Configurable Systems". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. 2015, pp. 284–294. ISBN: 9781450336758.

[Sol+12] Samaneh Soltani, Mohsen Asadi, Dragan Gašević, Marek Hatala, and Ebrahim Bagheri. "Automated Planning for Feature Model Configuration Based on Functional and Non-Functional Requirements". In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. SPLC '12. 2012, pp. 56–65. ISBN: 9781450310949.

[Tar+11] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. "Configuration coverage in the analysis of large-scale system software". In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. 2011, pp. 1–5.

[Tar+14] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Static Analysis of Variability in System Software: The 90, 000# ifdefs Issue." In: *USENIX Annual Technical Conference*. 2014, pp. 421–432.

[Tem+17] Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, and Olivier Barais. "Learning contextual-variability models". In: *IEEE Software* 34.6 (2017), pp. 64–70.

[Thü+14a] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. "A classification and survey of analysis strategies for software product lines". In: *ACM Computing Surveys (CSUR)* 47.1 (2014), pp. 1–45.

[Thü+14b] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. "FeatureIDE: An extensible framework for feature-oriented software development". In: *Science of Computer Programming* 79 (2014), pp. 70–85.

[Uni20] International Telecommunication Union. *Global Ewaste Monitor 2020*. `https://www.itu.int/en/ITU-D/Environment/Pages/Spotlight/Global-Ewaste-Monitor-2020.aspx/`. Accessed: 2023-05-18. 2020.

[Var+18] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. "A classification of product sampling for software product lines". In: *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*. 2018, pp. 1–13.

[VG09] Pim Van Den Broek and Ismênia Galvão. "Analysis of Feature Models using Generalised Feature Trees." In: *VaMoS*. 2009, pp. 29–35.

[Wan+18] Yewan Wang, David Nörtershäuser, Stéphane Le Masson, and Jean-Marc Menaud. "Potential effects on server power metering and modeling". In: *Wireless Networks* (2018), pp. 1–8.

[WG04] Diana L Webber and Hassan Gomaa. "Modeling variability in software product lines with the variation point model". In: *Science of Computer Programming* 53.3 (2004), pp. 305–331.

[Xu+15] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma 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*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 307–319. ISBN: 9781450336758.

[ZE14] Sai Zhang and Michael D. Ernst. "Which Configuration Option Should I Change?" In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 152–163. ISBN: 9781450327565.