On Reducing the Energy Consumption of Software Product Lines

Édouard Guégain, Clément Quinton, Romain Rouvoy

▶ **To cite this version:**

Édouard Guégain, Clément Quinton, Romain Rouvoy. On Reducing the Energy Consumption of Software Product Lines. SPLC'21: 25th ACM International Systems and Software Product Line Conference, Sep 2021, Leicester, United Kingdom. pp.89-99, 10.1145/3461001.3471142 . hal-03269168

HAL Id: hal-03269168

https://inria.hal.science/hal-03269168v1

Submitted on 1 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On Reducing the Energy Consumption of Software Product Lines

Édouard Guégain
edouard.guegain@univ-lille.fr
Univ. Lille, UMR 9189 CRIStAL
CNRS, UMR 9189
Inria
Lille, France

Clément Quinton
clement.quinton@univ-lille.fr
Univ. Lille, UMR 9189 CRIStAL
CNRS, UMR 9189
Inria
Lille, France

Romain Rouvoy
romain.rouvoy@univ-lille.fr
Univ. Lille, UMR 9189 CRIStAL
CNRS, UMR 9189
Inria, IUF
Lille, France

ABSTRACT

Along the last decade, several studies considered green software design as a key development concern to improve the energy efficiency of software. Yet, few techniques address this concern for *Software Product Lines* (SPL). In this paper, we therefore introduce two approaches to measure and reduce the energy consumption of a SPL by analyzing a limited set of products sampled from this SPL. While the first approach relies on the analysis of individual feature consumptions, the second one takes feature interactions into account to better mitigate energy consumption of resulting products.

Our experimental results on a real-world SPL indicate that both approaches succeed to produce significant energy improvements on a large number of products, while consumption data was modeled from a small set of sampled products. Furthermore, we show that taking feature interactions into account leads to more products improved with higher energy savings per product.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Software Product Lines, Energy, Consumption, Measurement, Mitigation

ACM Reference Format:
Édouard Guégain, Clément Quinton, and Romain Rouvoy. 2021. On Reducing the Energy Consumption of Software Product Lines. In *25th ACM International Systems and Software Product Line Conference - Volume A (SPLC '21), September 6–11, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3461001.3471142

1 INTRODUCTION

Energy consumption of software systems is becoming a major concern for the environment and the society, especially with the emergence of highly-configurable and large-scale distributed systems, such as Fog, IoT, or cyber-physical systems. Along the last decade, several studies showed that software has a significant impact on the energy consumed and considered green software design as a key development concern to improve the energy efficiency of software systems at large [14, 15, 26]. While tools and approaches have thus been published to understand, measure, and anticipate the energy consumption of software [22], few techniques address this concern for *Software Product Lines* (SPL) [6]. Measuring the energy consumption of a SPL is a challenging task due to *(i)* the large number of features and products that must be considered, and *(ii)* the time and cost required to efficiently handle such a variability. To the best of our knowledge, only Couto *et al.* proposed an approach to measure energy consumption in SPL [6]. Similarly to their work, we acknowledge that measuring the energy consumption of each product of the SPL is not conceivable, in particular when considering large configuration spaces of highly-configurable software systems. But, rather than simulating and predicting the energy consumption of a single product as in [6], we propose to measure the energy consumption of a set of products.

In this paper, we thus present a method to measure and reduce the energy consumption 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 energy consumption of individual features and one that takes pairs of features into account. The latter thus takes feature interactions into account when measuring the energy consumed by a product to highlight pairs of features that may cooperate or obstruct each other at a behavioral level, while altering the energy spent to complete a task. This approach also suggests candidate features whose interaction with user-required features exhibits lower energy footprint than the one produced by the initial interaction.

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

In the remainder of the paper, Section 2 explains fundamentals on software energy consumption and measurement. Section 3 describes our approaches to measure and reduce the energy consumption of a product from the SPL. Section 4 presents the design and results of our experiments. Section 6 analyzes related work. Section 5 discusses the outcomes of our contributions, while Section 7 concludes the paper.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SPLC '21, September 6–11, 2021, Leicester, United Kingdom
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8469-8/21/09…$15.00
https://doi.org/10.1145/3461001.3471142

2 MEASURING ENERGY CONSUMPTION OF SOFTWARE

While the energy consumption of hardware components has been widely studied, software consumption only recently gained interest. By driving and managing such hardware, software is now considered as a central concern when aiming at reducing energy consumption [22]. When dealing with green concerns of software systems, the impact of such systems is often measured as *power* or *energy* consumption. While power (P) measures the instantaneous consumption in Watts, energy (E) reports on a accumulated consumption over a given period in Joules [25]. Throughout this paper, we will propose an approach to estimate and reduce energy consumption of SPL, expressed in Joules. Thus, all our measurements represent the total energy consumed by products from this SPL, independently of their execution time, which can vary depending on products.

Energy measurement tools usually estimate the energy consumption of the CPU rather than the one of a specific software [21]. Thus, identifying the share of the software under study among the total energy consumption is not straightforward. To address this issue, we first sample the CPU consumption for one second before the program starts. We define this measurement as the *idle* energy consumption P_{idle}, which refers to the average power consumption at rest. The general idea is to measure the energy consumed by the running software, E_{raw}, and then substract $E_{idle} = P_{idle} \times T_{measure}$ from E_{raw} to get rid of the environment consumption.

$$E_{net} = E_{raw} - (P_{idle} \times T_{measure}) \tag{1}$$

The resulting energy consumption, E_{net}, can then be associated to the software under study, as depicted in Figure 1 and presented in Equation (1).

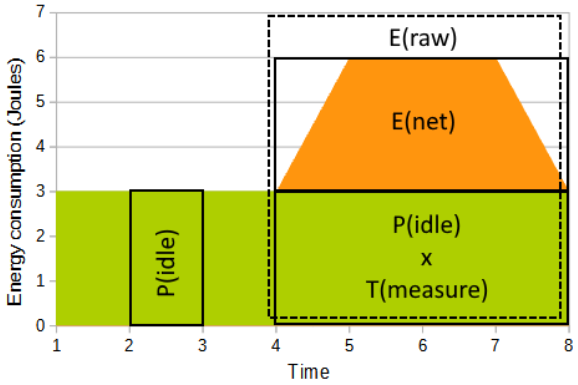


Figure 1: Raw *vs.* net energy consumption.

3 ESTIMATING AND REDUCING ENERGY CONSUMPTION FOR SPL

Unlike measuring the energy consumption of a software, measuring the energy consumption of a 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 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's 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 contains the features required by this stakeholder.

3.1 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 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 (2) 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 if it includes f_y, or is left empty otherwise.

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

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 paper, 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-steps 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 are 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.

3.2 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 consumption) due to the feature interactions phenomenon [28]. Indeed, numerous work have shown that features interact with each other, hence impacting performances of products [1–4, 27, 28]. Therefore, restricting the energy consumption analysis to individual features does not provide a comprehensive landscape of a feature consumption, and additional analysis that consider 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 positive or negative impact on the consumption of the product.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ \vdots \\ p_m \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} c_1 & c_2 & \cdots & c_n \end{array} \\ \begin{bmatrix} E_{11} & E_{12} & \dots & E_{1n} \\ E_{21} & E_{22} & \dots & E_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ E_{m1} & E_{m2} & \dots & E_{mn} \end{bmatrix} \\ \begin{array}{cccc} \tilde{E}(c_1) & \tilde{E}(c_2) & \cdots & \tilde{E}(c_n) \end{array} \end{array} \tag{3}$$

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 (3). 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 a 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 paper, 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 by the substitute feature with the lowest energy consumption, this second approach iteratively picks the alternative features whose interactions with other features of the product results 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 feature containing f in the product P, and by summing the observed median energy consumption $\tilde{E}(c)$ of these pairs, as described in Formula (4).[1]

$$\mathcal{I}(f, P) = \sum_{g \in P \mid g \neq f} \tilde{E}(gf) \tag{4}$$

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 a user-required feature 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

[1] 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)
```

iterations to accommodate this skipped feature. Once a modification has been applied, the algorithm proceeds to the next iteration, unless a stop criteria is met: if a 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 criteria is met, the energy consumption of the product resulting from each iteration is measured in order 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, it is possible that a specific iteration worsens 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).

4 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:

RQ 1: *Do our different analysis detect feature interactions impacting energy consumption?* By applying the two approaches on 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 reduce the energy consumption of a product?* The two proposed approaches rely on different analysis methods to mitigate 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.

4.1 Methodology

To assess the effectiveness of our solution when measuring energy consumption of a software product line, we performed our experiments on RobocodeSPL, a software product line designed to yield robots for Robocode [18]. 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 which caused the most damages 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 2 depicts an 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[2]. As the goal was to minimize the 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

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

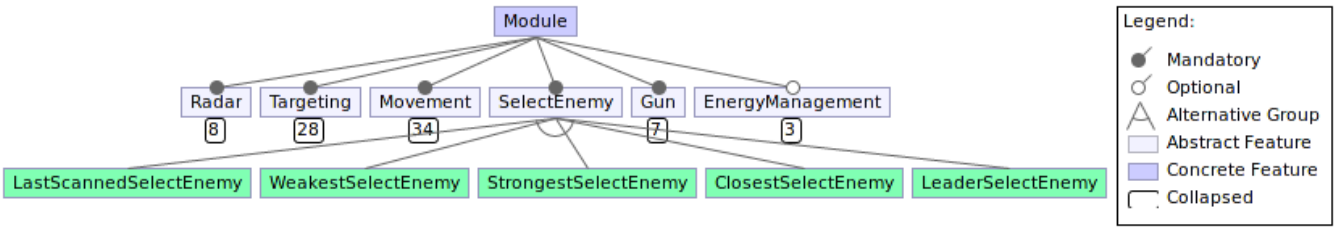


Figure 2: Excerpt of the Feature Model of RobocodeSPL.

pair of features from the feature model. To ensure such a coverage, we relied on the T-wise algorithm ICPL [16] with $T = 2$ to sample the configuration space of RobocodeSPL. Another sampling technique may provide better uniform random samples [17, 20], 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,[3] 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.[4]

4.2 Results

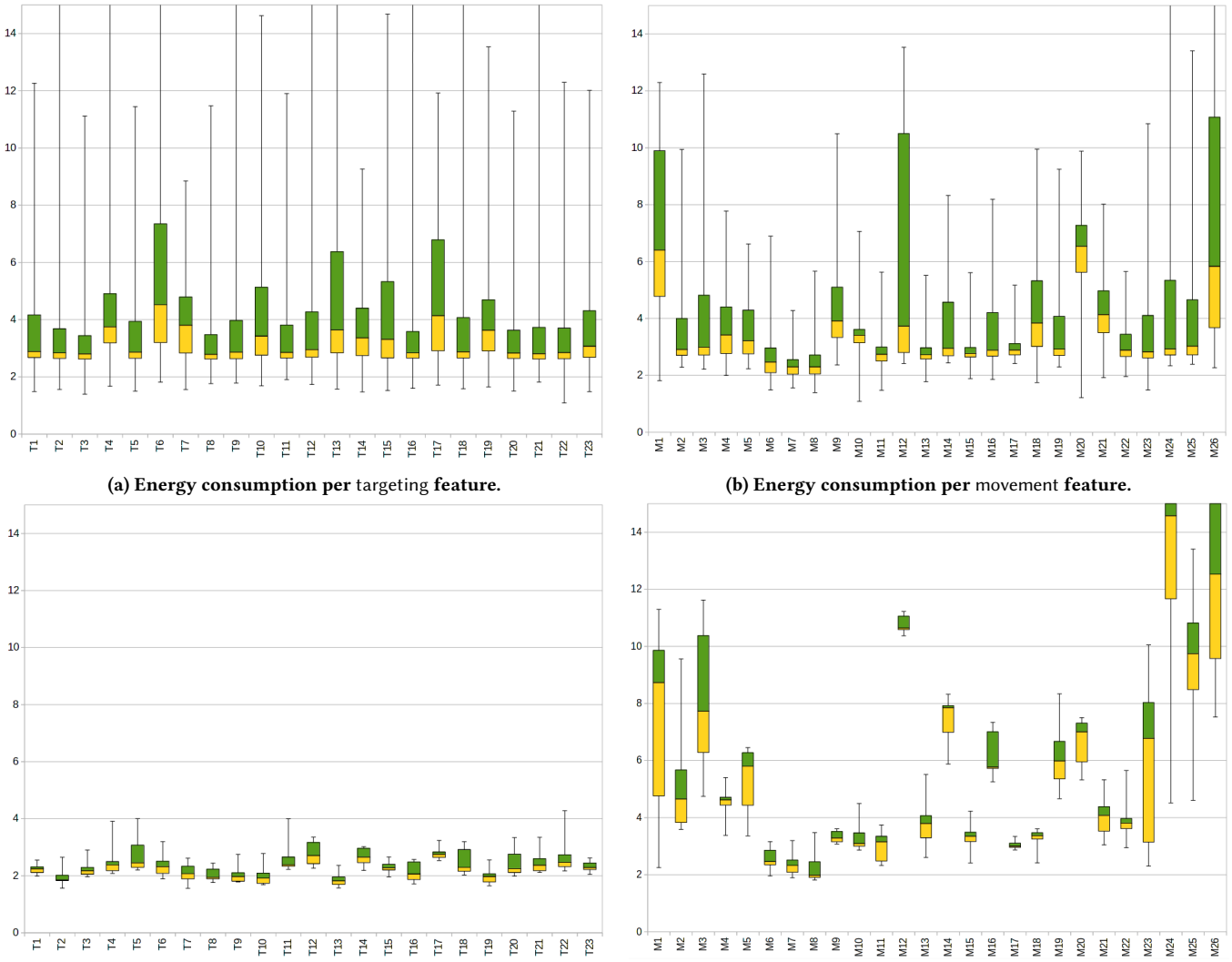
Detecting interacting features. The pairwise analysis relies on feature interactions to mitigate energy consumption of products. The goal of this first experiment is therefore to ensure that the pairwise analysis is able to detect at least one occurrence of feature interaction. The first experiment thus 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 3 depicts how the energy consumption of different features evolves depending on the analysis method.[5] Figures 3a and 3b 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 an 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 on 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, reducing the energy consumption despite the additional computations required to anticipate the position of the opponent.

Figure 3c 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 3b, and breaking them down per targeting feature. M7 being 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 3b, 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 evolve differently, as depicted in Figure 3d. 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

[3] https://github.com/powerapi-ng/j-joules
[4] https://doi.org/10.5281/zenodo.5048316
[5] 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 per targeting **features when feature** LinearRammingMovement **(M7) is selected.**

(d) Energy consumption per movement **features when feature** NoTargeting **(T6) is selected.**

Figure 3: Energy variations between the movement **and** targeting **features and their potential interactions.**

of NoTargeting—*i.e.*, always shooting forward—is very efficient, as it always hits the opponent.

Such changes in the resulting energy consumption with couple of features outperforming both of their members alone show 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: the pairwise analysis is able to detect interactions significantly impacting the energy consumption of products, and such interactions were not 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 analysis 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 3.2, 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 the initial product considered for improvement, the feature-wise analysis will

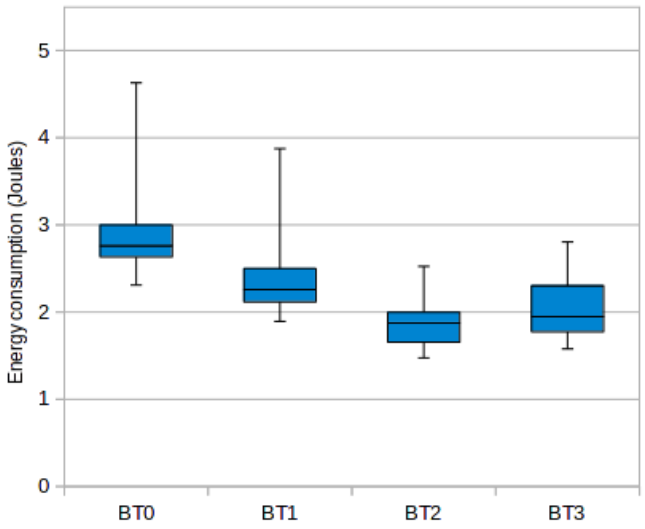


Figure 4: Improving the product resulting from the feature-wise analysis with the pairwise one.

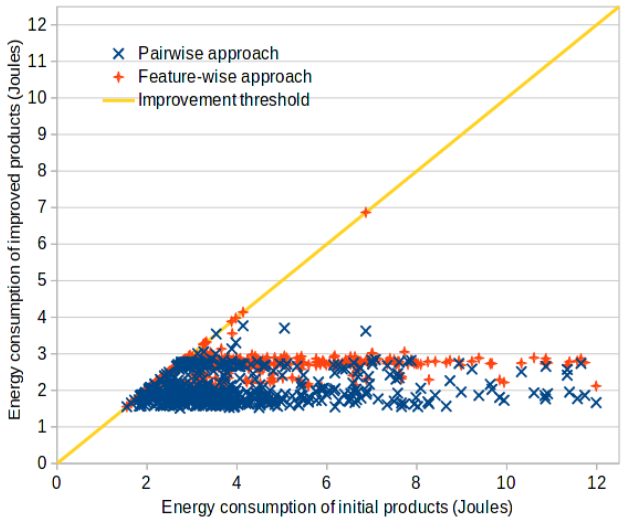


Figure 5: Energy consumption of the products resulting from both analysis.

always return this product, hereafter referred to as the *Best Theoretic* product, BT_0.

By applying the pairwise analysis on this product, we can determine how the pairwise analysis compares to the feature-wise analysis in the absence of required features. The result of this experiment is depicted in Figure 4, where BT_0 consumes 2.8 Joules. The pairwise analysis performed three iterations before reaching a stop criteria 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 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, by 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 4.1, 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, 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 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 tend to exclude noise or measurement error as a cause for this exception.

To get a better view on the efficiency differences between the two approaches, Figure 6 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

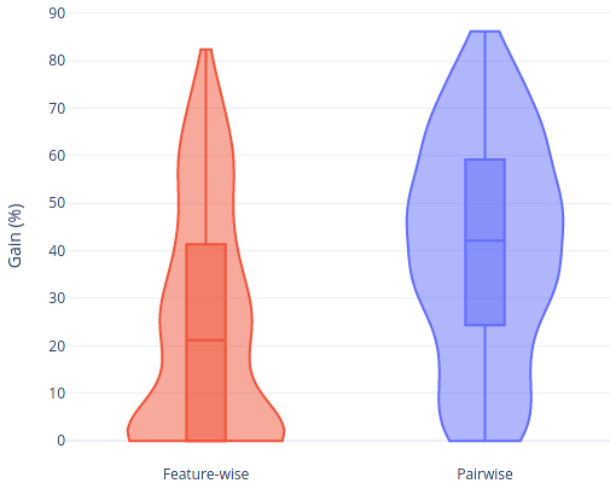


Figure 6: Relative gains of the pairwise and feature-wise analysis.

the second quartile. Therefore, only half of 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 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 products. The maximum gain is similar for both approaches: 82% and 86%, respectively.

Figure 7 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 7a). 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 comparison is designated with the subscript FW ($WPFW$ and $BPFW$, respectively). Figure 7a depicts the product WP_0 with the worst initial energy consumption, 12 Joules. The pairwise analysis performed 4 iterations before meeting a 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 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 7b 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 approach fail to find optimization, they return the initial product BP_0.

These results complete the partial answer to our second research question **RQ 2**: Both approaches are able to improve products, with and without required features, and the pairwise analysis outperforms the features-wise approach.

Overall, both of our approaches succeed in improving products from RobocodeSPL, with or without constraints. The feature-wise and pairwise analysis thus provided useful input data about energy consumption of features and couple of features, that 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 setup, 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.

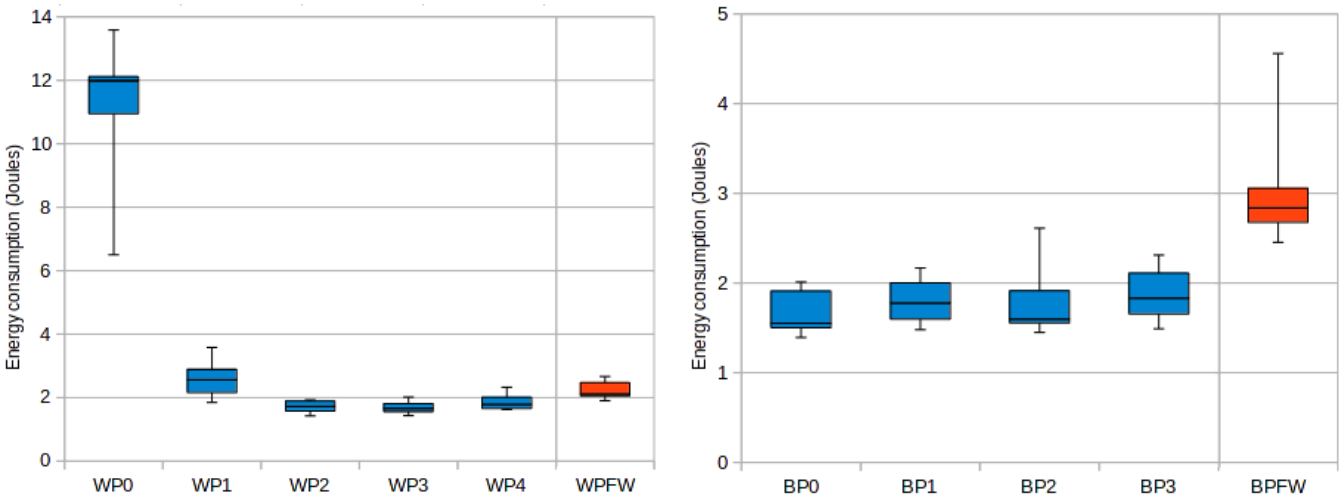
5 DISCUSSION

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 [5]. Extended feature models can be used to assign consumption data on features, in order to automatically apply optimizations. However, the adoption of such extensions may raise some challenges when dealing with consumption metrics associated to pairs of features.

In the green computing domain, a commonly-used means to reduce energy consumption of software is by refactoring inefficient



(a) Worst initial product. **(b) Best initial product.**

Figure 7: Focus on the best and worst initial products from the *validation sample*.

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-inefficient 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 after 2,433 matches in the best case scenario where energy savings are high. 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 others 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 feature may proves necessary.

6 RELATED WORK

The approaches presented in this paper 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 [21]. Rather than evaluating software artefacts, such as functions or classes, Islam *et al.* [14] 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 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) [22, 26]. For instance, Pereira *et al.* [26] report 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 [8, 9, 24]. These approaches have not been designed to take variability (especially code shared among features) into account, but they can be considered as complementary to our approaches. They provide a fine-grained analysis of the energy consumption of the software at 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 relying on *dynamic* SPL to reconfigure the system depending on context changes and ensure it continues meeting its green requirements [10, 13, 19]. 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 paper, 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

[6, 7]. These techniques estimate the energy consumption of features in the worst case scenario by analyzing the source code of features to deduce 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 [27, 30]. 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 [11]. In this prediction approach, feature interactions are detected using the systematic approach presented in [29]. 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 [12, 23, 28]. Such frameworks are designed to optimize multiple performance indicators, and energy consumption can be one of them. However, they also rely on a systematic identification of feature interactions. Soltani *et al.* [31] propose an approach relying on artificial intelligence to configure products meeting stakeholders' functional requirements, preferences and performance goals. 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 systematic identification of feature interactions. We assume that each feature interacts with all other features.

7 CONCLUSION

In this paper, we proposed a method to measure and reduce energy consumption in software product lines. Our 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 energy consumption of couples of features to take feature interactions into account without detecting them in a systematic way. We applied our approach on 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%.

As future work, we plan to improve the pairwise mitigation to tackle its current limitations. We also plan to use our approaches on widely used variable systems, such as the Linux kernel or Apache Web Server.

ACKNOWLEDGMENTS

This work was partially funded by the ANR-19-CE25-0003-01 KOALA project. We thank Baptiste Lewandoski for his help during the development of the tools supporting our experiments.

REFERENCES

[1] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kastner. 2010. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10)*. 161–170. https://doi.org/10.1109/ISSRE.2010.11

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

[3] Sven Apel, Alexander Von Rhein, Thomas Thüm, and Christian Kästner. 2013. Feature-Interaction Detection Based on Feature-Based Specifications. *Comput. Netw.* 57, 12 (Aug. 2013), 2399–2409. https://doi.org/10.1016/j.comnet.2013.02.025

[4] Don Batory, Peter Höfner, and Jongwook Kim. 2011. Feature Interactions, Products, and Composition. *SIGPLAN Not.* 47, 3 (Oct. 2011), 13–22. https://doi.org/10.1145/2189751.2047867

[5] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*. Springer, 491–503.

[6] Marco Couto, Paulo Borba, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva. 2017. 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)*. 84–93. https://doi.org/10.1145/3106195.3106214

[7] Marco Couto, João Paulo Fernandes, and João Saraiva. 2021. Statically Analyzing the Energy Efficiency of Software Product Lines. *Journal of Low Power Electronics and Applications* 11, 1 (2021), 13.

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

[9] Luis Cruz, Rui Abreu, and Jean-Noël Rouvignac. 2017. 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)*. 205–206. https://doi.org/10.1109/MOBILESoft.2017.21

[10] Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. 2014. Probabilistic Model Checking for Energy Analysis in Software Product Lines. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. 169–180. https://doi.org/10.1145/2577080.2577095

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

[12] Robert M. Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zheng. 2016. SIP: Optimal Product Selection from Feature Models Using Many-Objective Evolutionary Optimization. *ACM Trans. Softw. Eng. Methodol.* 25, 2, Article 17 (April 2016), 39 pages. https://doi.org/10.1145/2897760

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

[14] Syed Islam, Adel Noureddine, and Rabih Bashroush. 2016. Measuring energy footprint of software features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–4. https://doi.org/10.1109/ICPC.2016.7503726

[15] Erik A. Jagroep, Jan Martijn van der Werf, Sjaak Brinkkemper, Giuseppe Procaccianti, Patricia Lago, Leen Blom, and Rob van Vliet. 2016. Software Energy Profiling: Comparing Releases of a Software Product. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. 523–532. https://doi.org/10.1145/2889160.2889216

[16] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. 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)*. 46–55. https://doi.org/10.1145/2362536.2362547

[17] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2020. The interplay of sampling and machine learning for software performance prediction. *IEEE Software* 37, 4 (2020), 58–66.

[18] Jabier Martinez, Xhevahire Tërnava, and Tewfik Ziadi. 2018. 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)*. 132–142. https://doi.org/10.1145/3233027.3233038

[19] Daniel-Jesus Munoz, José A. Montenegro, Mónica Pinto, and Lidia Fuentes. 2019. Energy-aware environments for the development of green applications for cyber–physical systems. *Future Generation Computer Systems* 91 (2 2019), 536–554. https://doi.org/10.1016/j.future.2018.09.006

[20] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. 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*. 289–301.

[21] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2013. A Review of Energy Measurement Approaches. *SIGOPS Oper. Syst. Rev.* 47, 3 (Nov. 2013), 42–49. https://doi.org/10.1145/2553070.2553077

[22] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2015. Monitoring energy hotspots in software. *Automated Software Engineering* 22 (9 2015). Issue 3. https://doi.org/10.1007/s10515-014-0171-1

[23] Rafael Olaechea, Steven Stewart, Krzysztof Czarnecki, and Derek Rayside. 2012. 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)*. Article 2, 6 pages. https://doi.org/10.1145/2420942.2420944

[24] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. 2021. Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption. In *ICSOFT*. https://hal.archives-ouvertes.fr/hal-03202437

[25] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. 2016. What Do Programmers Know about Software Energy Consumption? *IEEE Software* 33, 3 (2016), 83–89. https://doi.org/10.1109/MS.2015.83

[26] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2020. SPELLing out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software* 161 (3 2020), 110463. https://doi.org/10.1016/j.jss.2019.110463

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

[28] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting Performance via Automated Feature-Interaction Detection. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. Zurich, Switzerland, 167–177.

[29] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal* 20, 3 (2012), 487–517.

[30] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. 2013. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology* 55, 3 (2013), 491–507. https://www.sciencedirect.com/science/article/pii/S0950584912001541

[31] Samaneh Soltani, Mohsen Asadi, Dragan Gašević, Marek Hatala, and Ebrahim Bagheri. 2012. 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)*. 56–65.