Configuration Optimization with Limited Functional Impact

Edouard Guégain, Amir Taherkordi, Clément Quinton

Configuration Optimization with Limited Functional Impact

Edouard Guégain[1], Amir Taherkordi[2] and Clément Quinton[1]

[1] Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
{edouard.guegain,clement.quinton}@univ-lille.fr
[2] University of Oslo, Oslo, Norway
amirhost@ifi.uio.no

Abstract. Dealing with a large configuration space is a complex task for developers, especially when configurations must comply with both functional constraints and non-functional goals. In this paper, we introduce an approach to optimize any set of performance indicators for an existing configuration, while meeting functional requirements. The efficiency of this approach is assessed by exhaustively optimizing a configurable system, and by analyzing how the algorithm navigates through the configuration space. This approach proves especially efficient at optimizing configurations through a minimal number of changes, thus limiting the impact on their functional behavior.

Keywords: Software · Variability · Optimization · Performance.

1 Introduction

Modern software-intensive systems are highly configurable. Software engineers thus have to develop, test and maintain a significant number of options, or *features*, that are then combined together to produce a specific software configuration. As the number of features grows, the number of configurations (*i.e.*, the *configuration space*) consequently grows exponentially and modern systems face the ever-increasing complexity of their configurations [20]. Dealing with large configuration spaces is challenging, especially when configurations must comply with both functional constraints and non-functional performance goals. To avoid facing this complexity, developers may stick to default configurations or sub-optimal ones [12].

Based on this observation and inspired by our previous work on measuring the energy consumption of configurable systems [2], we propose in this paper 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 [4,7,9,10,22], 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 addresses 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 on a real-world system.

In the remainder of this paper, Section 2 explains fundamentals and a running example. Section 3 explains our optimization approach. Section 4 and Section 5 present the design and results of our experiments, respectively. Section 6 provides a critical discussion. Section 7 discusses related work and Section 8 concludes the paper.

2 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 [8], 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.

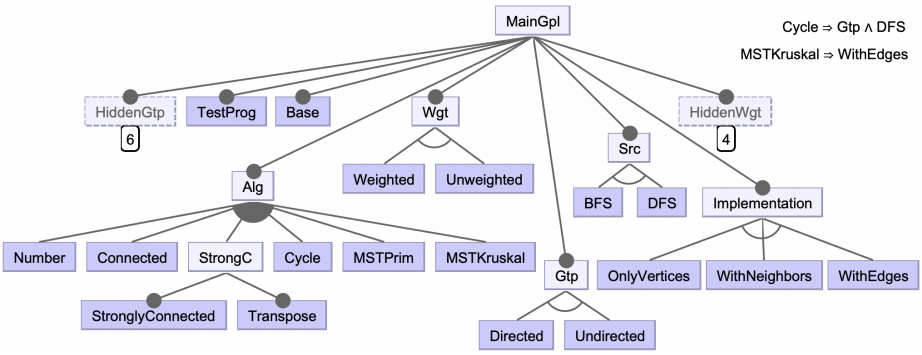

Fig. 1: Excerpt of the feature model of GPL-FH-Java.

Figure 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 [20]. 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 [12,10].

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.

3 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 [2], but the approach in this paper differs from the one in [2] in several aspects. In particular, the approach proposed in this paper addresses the limitations listed in [2]. 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 [2] only focused on switching selected features based on the feature tree structure.

3.1 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 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.

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

As interactions between features impact performance [15], 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 1, 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

Algorithm 1: ICO optimization algorithm

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

1 $candidates \leftarrow \emptyset$
2 $suggestions \leftarrow \emptyset$
3 $addable \leftarrow (features \backslash conf_{init}) \backslash constraints_{exclude}$
4 $removable \leftarrow conf_{init} \backslash constraints_{include}$
5 **for** $rem \in removable$ **do**
6 $\quad candidates \leftarrow candidates \cup$ `newConfig`$(conf_{init} \backslash rem)$
7 **end**
8 **for** $add \in addable$ **do**
9 $\quad candidates \leftarrow candidates \cup$ `newConfig`$(add \cup conf_{init})$
10 **end**
11 **for** $add \in addable, rem \in removable$ **do**
12 $\quad candidates \leftarrow candidates \cup$ `newConfig`$(addable \cup conf_{init} \backslash removable)$
13 **end**
14 $candidates \leftarrow$ `sortByPerfGain`$(candidates)$
15 **for** $c \in candidates$ **do**
16 $\quad$ **if** $\mathit{isValid}(c, \mathit{constraints}) \wedge \mathit{perf}(c) > \mathit{perf}(conf_{init})$ **then**
17 $\quad\quad suggestions \leftarrow suggestions \cup$ `diff`$(c, conf_{init})$
18 $\quad$ **end**
19 **end**
20 **return** suggestions

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, according to the performance model[3].

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 by 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.

3.2 Implementation

ICO has been implemented as a series of tools, namely the ICO tool suite. This tool suite has been built with the objectives of *(i)* providing developers with feedback about the performances of a given configuration and *(ii)* providing suggestions to optimize its performance by adding or removing a feature. ICO is composed of three software components: (1) ICOLIB, a Java implementation of the proposed approach; (2) ICOCLI, a command-line interface; and (3) ICOPLUGIN, an Eclipse plugin. The ICO tool suite takes as input a configuration, a feature model and performance files, and then returns optimized configurations based on the suggestions provided by Algorithm 1. Figure 2 provides an overview of the architecture of ICO: through either ICOCLI or ICOPLUGIN, user instructions are sent to ICOLIB which then performs various operations based on input files.

[3] The computation of the performance model is out of the scope of this paper. Yet, we discuss this particular point in Section 6.

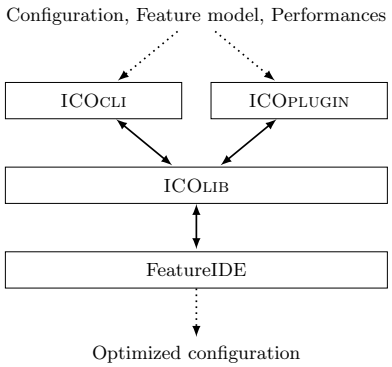

Fig. 2: The architecture of the ICO tool suite.

The tool suite is centered on ICOlib, a Java library that exposes the API managing all operations that can be performed with ICO: loading a project, displaying current performances, managing constraints (*i.e.*, the lists of features required or excluded by the developer), listing or applying improvement suggestions and saving the new configurations. In particular, ICOlib delegates to the FeatureIDE [19] library the responsibility to load, update, validate and save the configurations. Taken as a standalone component, ICOlib can be integrated as a Java dependency into any tool requiring an implementation of Algorithm 1. ICOplugin is an Eclipse Plugin developed to interact with ICOlib and implemented as an Eclipse view. It thus provides a GUI that assists developers when seeking optimized configurations, in particular by proposing visual feedback on suggested optimizations. ICOcli is a command-line interface to interact with ICOlib, enabling an in-depth exploration of the variability of the software and its performances. It can be used directly by the developer or integrated into automated processes such as CI/CD. The source code of ICO is publicly available[4], and [3] covers the specifics of its implementation.

4 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.

[4] `https://gitlab.inria.fr/ico`

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 2. 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 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 [2], *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 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.

5 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 3 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 1 to a given configuration results in either

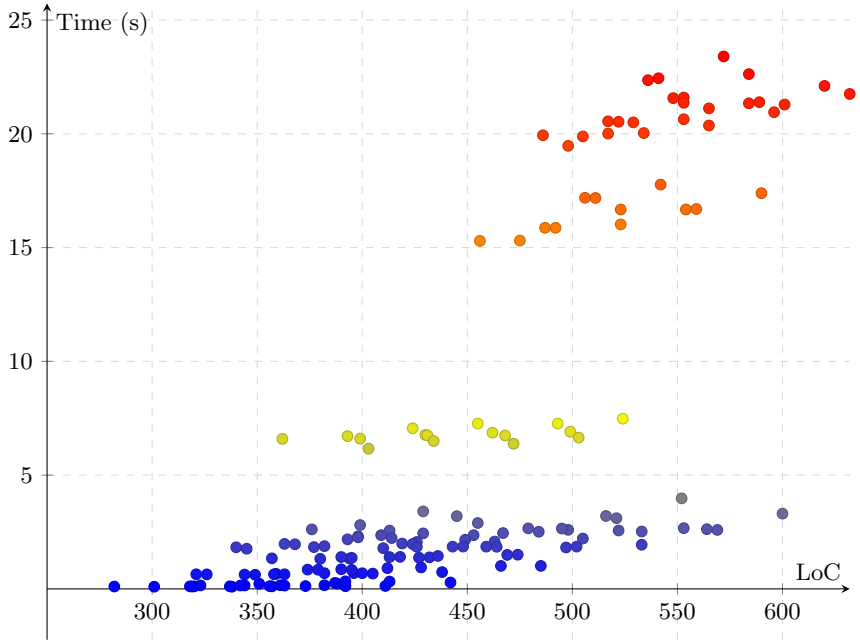

Fig. 3: Performance of each GPL-FH configuration *w.r.t* `LoC` and `time` (lower left corner is better).

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[5].

Table 1 summarizes the performance gains resulting from applying Algorithm 1 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.

> **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.

[5] Due to inaccuracies in the performance model. See Section 6 for further analysis.

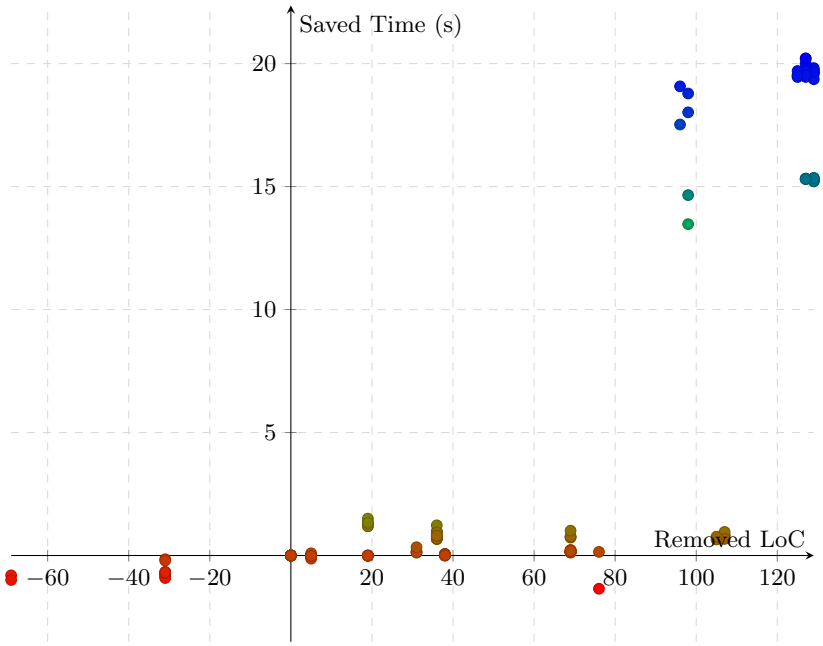

Fig. 4: Performance gains for each GPL-FH configuration *w.r.t* `LoC` and `time` (top right corner is better).

Investigating RQ2: How effective is the ICO *optimization approach?* Figure 4 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 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.

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

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 1 (see line 16), running the algorithm on each configuration of the configuration space thus results in a set 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 5 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 1: 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 1, situation S_3). These 18 configurations are thus local optima, and one of them is the global optimum.

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 2: Applying ICO on the GPL-FH configuration space.

Table 2 shows the number of iterations of Algorithm 1 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

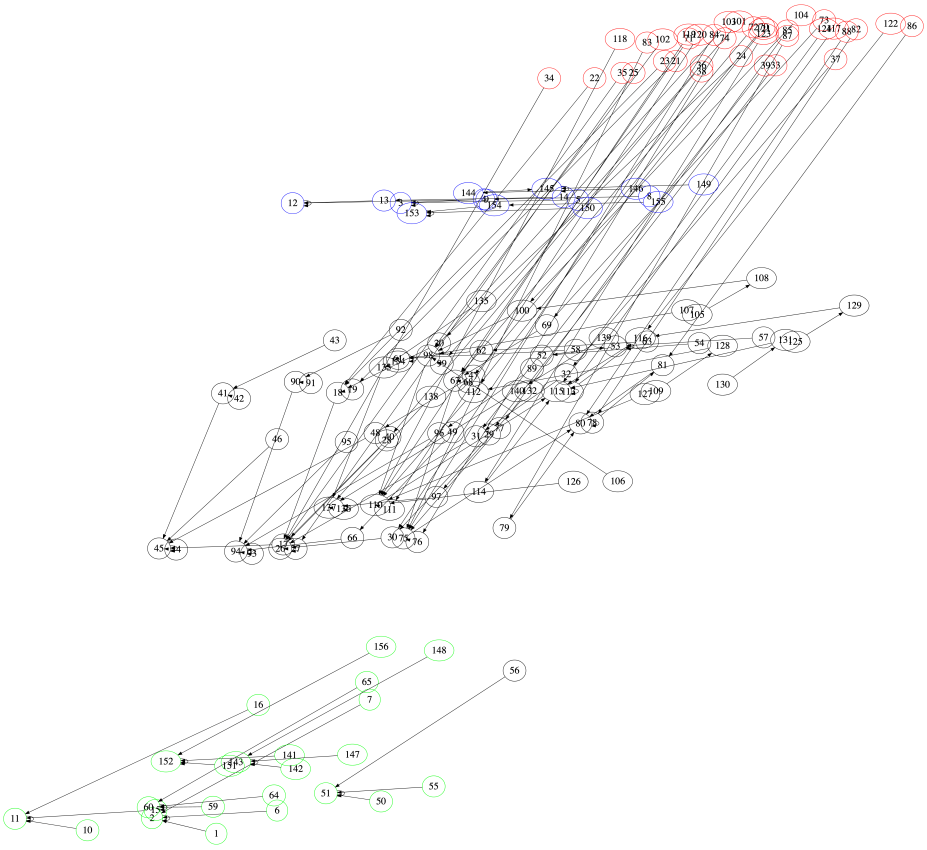

Fig. 5: ICO transition graph between configurations of GPL-FH. Configurations on a relative logarithmic scale for readability, `time` on the vertical axis, `LoC` on the horizontal axis, lower left is better.

convergence point. That is, after one iteration, 79 configurations (more than half the configuration space) have already converged. After a second iteration, 81% 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.

6 Discussion

A thorough analysis of the GPL-FH optimization graph depicted in Figure 5 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 $\texttt{Directed} \wedge \texttt{WithEdge} \leftrightarrow \texttt{DirectedWithEdge}$ cross-tree constraint. In addition, C_3 contains configurations whose couple of features `StronglyConnected` and `Transpose` are selected, complying with the $\texttt{StronglyConnected} \leftrightarrow \texttt{Transpose}$ 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 [2]. 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 as long as a feature model is provided. 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.

We ran the ICO optimization algorithm on the whole configuration space of GPL-FH.While relying on an exhaustive performance model was convenient, we acknowledge that this may not be practical for any case study, in particular regarding performance models of software systems exhibiting 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 [2,10,1].

7 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 [7,4,14]. 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 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.* [15,17] 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 [2], 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 [22,11,10] to tackle this challenge. Other authors like Hierons *et al.* rely on genetic algorithms to minimize the

number of measurements needed to optimize configurations [5], or leverage the performance predictions methods discussed above [16]. Such approaches do not take an initial configuration into consideration. For instance, Nair *et al.* [10] start their optimization process from a random configuration. In contrast to such approaches, the approach proposed in this paper 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 [18].

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 [13] based on proximity and similarity between features. Similarly, Zhang *et al.* [21] 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 [9] or [6], 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.

8 Conclusion

This paper 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.

As future work, we plan to improve the accuracy of performance suggestions provided by ICO by relying on more accurate performance models and performance prediction techniques. We also plan to extend the capabilities of ICO supporting multiple feature changes at once, *e.g.*, in highly heterogeneous systems such as Edge and Fog computing systems. In such systems, the wide range of possibilities (*e.g.*, component-level resource allocation and computation models) results in a huge configuration space that is not manually manageable.

Acknowledgements

The research leading to these results received funding from French Research Agency through the ANR-19-CE25-0003-01 KOALA project and from the Norwegian Research Council through the DILUTE project (Grant No. 262854/F20).

References

1. Acher, M., Martin, H., Lesoil, L., Blouin, A., Jézéquel, J.M., Khelladi, D.E., Barais, O., Pereira, J.A.: 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. p. 85–96. SPLC '22, Association for Computing Machinery, New York, NY, USA (2022)
2. Guégain, E., Quinton, C., Rouvoy, R.: On reducing the energy consumption of software product lines. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A. p. 89–99. SPLC '21 (2021)
3. Guégain, E., Taherkordi, A., Quinton, C.: The ICO Tool Suite: Optimizing Highly Configurable Systems (Nov 2022), `https://hal.archives-ouvertes.fr/hal-03874051`, preprint.
4. Guo, J., Czarnecki, K., Apel, S., Siegmund, N., Wasowski, A.: Variability-aware performance prediction: A statistical learning approach. pp. 301–311 (2013)
5. Hierons, R.M., Li, M., Liu, X., Segura, S., Zheng, W.: Sip: Optimal product selection from feature models using many-objective evolutionary optimization. ACM Trans. Softw. Eng. Methodol. **25**(2) (Apr 2016)
6. Horcas, J.M., Pinto, M., Fuentes, L.: Context-aware energy-efficient applications for cyber-physical systems. Ad Hoc Networks **82**, 15–30 (2019)
7. Kaltenecker, C., Grebhahn, A., Siegmund, N., Apel, S.: The interplay of sampling and machine learning for software performance prediction. IEEE Software **37**, 58–66 (2020)
8. Metzger, A., Pohl, K.: Software product line engineering and variability management: Achievements and challenges. In: Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014. pp. 70–84 (2014)
9. Metzger, A., Quinton, C., Mann, Z., Baresi, L., Pohl, K.: Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration. Computing (Mar 2022)
10. Nair, V., Yu, Z., Menzies, T., Siegmund, N., Apel, S.: Finding faster configurations using flash. IEEE Transactions on Software Engineering **46**(7), 794–811 (2020)
11. Olaechea, R., Stewart, S., Czarnecki, K., Rayside, D.: 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)
12. Pereira, J.A., Acher, M., Martin, H., Jézéquel, J.M., Botterweck, G., Ventresque, A.: Learning software configuration spaces: A systematic literature review. Journal of Systems and Software **182**, 111044 (2021)
13. Pereira, J.A., Matuszyk, P., Krieter, S., Spiliopoulou, M., Saake, G.: 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. pp. 120–131 (2016)
14. Siegmund, N., Grebhahn, A., Apel, S., Kästner, C.: Performance-influence models for highly configurable systems. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. p. 284–294. ESEC/FSE 2015 (2015)
15. Siegmund, N., Kolesnikov, S.S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., Saake, G.: Predicting performance via automated feature-interaction detection. pp. 167–177 (2012)
16. Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., Apel, S., Saake, G.: Spl conqueror: Toward optimization of non-functional properties in software product lines. Software Quality Journal **20**, 487–517 (2012)

17. Siegmund, N., Rosenmüller, M., Kästner, C., Giarrusso, P.G., Apel, S., Kolesnikov, S.S.: Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. Information and Software Technology **55**, 491–507 (2013)
18. Soltani, S., Asadi, M., Gašević, D., Hatala, M., Bagheri, E.: 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. p. 56–65. SPLC '12 (2012)
19. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: Featureide: An extensible framework for feature-oriented software development. Science of Computer Programming **79**, 70–85 (1 2014)
20. Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., Talwadker, R.: 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. p. 307–319. ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA (2015)
21. Zhang, S., Ernst, M.D.: Which configuration option should i change? In: Proceedings of the 36th International Conference on Software Engineering. p. 152–163. ICSE 2014, Association for Computing Machinery, New York, NY, USA (2014)
22. Švogor, I., Crnković, I., Vrček, N.: An extensible framework for software configuration optimization on heterogeneous computing systems: Time and energy case study. Information and Software Technology **105**, 30–42 (2019)