Automatic Non-functional Testing and Tuning of Configurable Generators

Mohamed Boussaa

ANNÉE 2017

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Mohamed BOUSSAA

préparée à l'unité de recherche INRIA
INRIA Rennes Bretagne Atlantique
ISTIC

Automatic Non-functional Testing and Tuning of Configurable Generators

Thèse soutenue à Rennes
le 6 Septembre 2017

devant le jury composé de :

Hélène WAESELYNCK
Directrice de recherche, LAAS-CNRS Toulouse / *Rapporteuse*

Philippe MERLE
Chargé de recherche, INRIA Lille / *Rapporteur*

Erven ROHOU
Directeur de recherche, INRIA Rennes / *Examinateur*

Franck FLEUREY
Directeur de recherche, SINTEF Oslo / *Examinateur*

Jean-Marie MOTTU
Maître de conférences, Université de Nantes / *Examinateur*

Gerson SUNYÉ
Maître de conférences, Université de Nantes / *Examinateur*

Benoit BAUDRY
Directeur de recherche, INRIA Rennes / *Directeur de thèse*

Olivier BARAIS
Professeur, Université de Rennes 1 / *Co-directeur de thèse*

Acknowledgements

This thesis would not have been completed without the help of others. I would like to take this opportunity to express my gratitude towards them and acknowledge them.

First of all, I would like to offer my deepest gratitude to my supervisors *Olivier Barais* and *Benoit Baudry* for their useful advices, and for guiding my steps over more than three years. Special thanks are also due to my "third" supervisor *Gesron Sunyé* that always knew how to motivate me and help me to overcome the difficult times in my thesis.

I owe special thanks to my Master's supervisor *Marouane Kessentini*, with whom I discovered the software engineering research world, for encouraging me to continue in the research field as a PhD candidate.

I would also like to thank the members of my jury. My examiners, *Hélène Waeselynck* and *Philippe Merle* for having accepted to serve on my examination board and for the time they have invested in reading and evaluating this thesis. Thanks are also due to *Erven Rohou* for agreeing to preside over my jury. I would also like to thank *Jean-Marie Mottou* and *Franck Fleury* for attending as members of the jury.

I greatly appreciate and wish to thank all the (former and current) members of the DiverSE research team for providing a great and friendly work atmosphere, and for all the different discussions, seminars, and DiverSE coffees.

Finally, I would like to thank my family. Even so far away from home, they are always present in my life during these three years. I first express my gratitude to my parents *Najoua* and *Noureddine* for their support, comprehension, confidence and, especially for their patience. You have always been behind me and believed in me. Then, thanks to my sisters *Imene* and *Jihene* for encouraging me to succeed in everything I do. By now, we can say that we have a Doctor in the *Boussaa* family !

Contents

Résumé en Français

Contexte

Les techniques émergentes de l'ingénierie dirigée par les modèles et de la programmation générative ont permis la création de plusieurs générateurs (générateurs de code et compilateurs). Ceux-ci sont souvent utilisés afin de faciliter le développement logiciel et automatiser le processus de génération de code à partir des spécifications abstraites. En effet, les utilisateurs sont devenus capables de synthétiser rapidement des artefacts logiciels pour une large gamme de plateformes logicielles et matérielles. En outre, les générateurs modernes comme les compilateurs C, sont devenus hautement configurables, offrant de nombreuses options de configuration à l'utilisateur de manière à personnaliser facilement le code généré pour la plateforme matérielle cible. Par conséquent, la qualité logicielle est devenue fortement corrélée aux paramètres de configuration ainsi qu'au générateur lui-même. Dans ce contexte, il est devenu indispensable de vérifier le bon comportement des générateurs. D'une part, il est important de tester les programmes générés afin de détecter des anomalies de générateurs et les corriger. D'autre part, il est crucial d'aider les utilisateurs à bien paramétrer les générateurs afin de satisfaire les exigences des plateformes logicielle et matérielle cibles.

Motivations

Aujourd'hui, les générateurs populaires tels que GCC, LLVM, etc., sont fréquemment utilisés dans l'industrie afin de traduire le code source écrit en langage humain vers un code machine de bas niveau (p. ex. binaires, exécutables). Ces générateurs, connus sous le nom compilateurs, offrent une large gamme d'options de configuration aux utilisateurs pour contrôler le comportement du générateur. Différentes catégories d'options peuvent être activées pour aider les développeurs à déboguer, optimiser l'application générée ou encore

sélectionner l'architecture matérielle cible. Par exemple, GCC version 4.8.4 offre une large sélection d'options en ligne de commande, y compris plus de 150 options d'optimisation. Cela constitue un espace de configuration très large avec 2^{150} combinaisons d'optimisation possibles pouvant être appliquées par l'utilisateur pour optimiser la qualité du code généré (comme le temps d'exécution, la taille du binaire, le temps de compilations, etc.). En outre, la construction d'une seule séquence d'optimisation pour tous les programmes en entrée est impossible car les interactions entre les optimisations sont trop complexes et difficiles à définir. De plus, l'impact des optimisations est fortement corrélé au matériel cible et au code source d'entrée. Cet exemple montre combien il est difficile de paramétrer les générateurs (tel que les compilateurs) dans le but de produire un code machine qui satisfait différentes propriétés non-fonctionnelles.

Il est aussi essentiel de tester les générateurs afin d'assurer que la génération automatique de code n'introduit aucune anomalie dans le produit final. Toute anomalie dans le code généré entrainerait une perte de confiance de la part des utilisateurs qui n'utiliseront plus ces générateurs lors du développement logiciel. En conséquence, les développeurs de générateurs doivent bien vérifier et tester le bon comportement du code automatiquement généré. Contrairement aux compilateurs qui disposent de plusieurs solutions de test [YCER11, LAS14], il en existe peu pour évaluer automatiquement les comportement des générateurs de code, dont aucune ne se base sur le test des propriétés non-fonctionnelles. En effet, les générateurs de code sont moins utilisés et expérimentés dans l'industrie par rapport aux compilateurs. Ils sont également difficiles à tester puisqu'ils impliquent un ensemble de technologies très complexes et hétérogènes [GS15, GS14]. Le test des générateurs de code implique principalement les créateurs/experts de l'outil. Néanmoins, les utilisateurs sont également responsables de cette validation puisqu'ils reportent continuellement les anomalies rencontrées lors de la génération de code. De nombreuses approches ont été proposées [SCDP07, YCER11] afin de tester le comportement fonctionnel du code généré. Cependant, peu de solutions évaluent les propriétés non-fonctionnelles, à savoir les propriétés liées à la performance et l'utilisation des ressources du code généré.

En résumé, les générateurs (compilateurs et générateurs de code) sont des composants essentiel pour la production automatique du code. La qualité du logiciel généré est directement corrélée à la qualité du générateur lui-même. Tant que la qualité des générateurs est maintenue et améliorée, la qualité des artefacts logiciels générés s'améliore également puisque toute anomalie avec ces générateurs affecte directement la qualité du produit logiciel final. En particulier, lorsque la génération automatique de code est utilisée, nous identifions deux problèmes majeurs qui menacent la qualité du logiciel généré: D'une part, l'espace de configuration très large pose un important défi aux utilisateurs qui doivent sélectionner les meilleures options d'optimisation, répondant à certaines exigences non-

fonctionnelles. D'autre part, la complexité des générateurs de code ainsi que le manque de solutions pour évaluer les propriétés non-fonctionnelles du code généré représente un obstacle pour les créateurs de générateurs qui veulent assurer la qualité du logiciel automatiquement généré.

Problématiques

Nous avons identifié trois problématiques pour le test et la configuration automatique des générateurs.

Le test des générateurs de code (le problème de l'oracle):

Le test des générateurs de code relève le problème de l'oracle. Un oracle de test est le mécanisme par lequel un testeur peut déterminer si un test passe ou pas. En ce qui concerne les tests non-fonctionnels des générateurs de code, ce problème devient très récurrent parce que il est assez difficile de déterminer le comportement non-fonctionnel attendu du code à tester (p. ex. la consommation de mémoire). Déterminer si les mesures non-fonctionnelles correspondent à une anomalie de générateur ou non, n'est pas aisé. Pour contourner ce problème, des techniques telles que les tests métamorphiques [CCY98] sont appliquées pour tester des programmes sans définir explicitement un oracle. Cette technique emploie des relations métamorphiques de haut-niveau pour vérifier automatiquement les résultats des tests. Alors, quels types d'oracles pouvons-nous définir? Comment pouvons-nous détecter automatiquement des anomalies de générateurs? Toutes ces questions soulèvent des défis importants dans le test des générateurs de code.

La configuration automatique des compilateurs (exploration de l'espace de recherche des optimisations):

La plupart des compilateurs fournissent un très grand nombre d'optimisations qui permettent d'améliorer la qualité du code généré. Cependant, pour explorer un tel espace d'optimisation, les utilisateurs doivent évaluer leurs effets selon une propriété non-fonctionnelle spécifique tel que le temps d'exécution, la taille du binaire, etc. Du fait de la difficulté de la construction d'une séquence d'optimisation donnée, de la complexité des interactions et de l'effet imprévisible des optimisations, les utilisateurs trouvent des difficultés à choisir la meilleure configuration de compilateur qui satisfait une exigence non-fonctionnelle spécifique.

La surveillance de la consommation des ressources du code généré (gérer la diversité des plateformes logicielles et matérielles):

Pour évaluer les propriétés liées à l'utilisation des ressources du code généré (par des compilateurs ou des générateurs de code), les développeurs doivent compiler, déployer et exécuter les artefacts logiciels générés sur différentes plateformes d'exécution. Ensuite, ils utilisent souvent plusieurs profileurs, débogueurs et outils de surveillance spécifiques à la plateformes [GS14, DGR04] afin de trouver des anomalies lors de l'exécution du code généré. En raison de l'hétérogénéité des plateformes d'exécution logicielle et matérielle, la collecte d'informations sur l'utilisation des ressources du code généré devient une tâche très couteuse, puisque les développeurs doivent analyser et vérifier le code généré pour chacune des plateformes cibles à l'aide d'outils spécifiques.

Les défis de cette recherche peuvent être résumés dans les questions de recherche suivantes:

RQ1. Comment pouvons-nous aider les développeurs à tester automatiquement le code généré et détecter des anomalies non-fonctionnelles de générateurs de code ?

RQ2. Comment pouvons-nous aider les utilisateurs des compilateurs à choisir automatiquement la configuration d'optimisation adéquate qui satisfait une exigence non-fonctionnelle spécifique?

RQ3. Comment pouvons-nous faciliter la mise en place d'outils de surveillance de la consommation des ressources dans un environnement hétérogène?

Contributions

Cette thèse établit trois contributions principales. Elles sont brièvement décrites dans le reste de cette section.

Contribution I: détection automatique des inconsistances dans les familles de générateurs de code.

Dans cette contribution, nous abordons le problème de l'oracle dans le domaine du test non-fonctionnel des générateurs de code. La disponibilité de multiples générateurs de code avec des fonctionnalités comparables (c.-à-d. familles de générateurs de code) nous permet d'appliquer l'idée du test métamorphique [ZHT+04] en définissant des oracles de test de haut-niveau (c.-à-d. relation métamorphique) pour détecter des inconsistances. Nous définissons la relation métamorphique comme la comparaison entre les variations de performance et de l'utilisation des ressources des différentes versions, générées à partir de

la même famille de générateurs de code. Une inconsistance est détectée lorsque le code généré présente un comportement inattendu par rapport à toutes les implémentations équivalentes de la même famille. Nous évaluons notre approche en analysant la performance de Haxe, un langage de programmation de haut niveau impliquant un ensemble de générateurs de code multi-plateformes. Nous évaluons les propriétés liées à la performance et à l'utilisation des ressources du code généré pour cinq plateformes logicielles cibles. Les résultats expérimentaux montrent que notre approche est capable de détecter plusieurs inconsistances qui révèlent des problèmes réels dans cette famille de générateurs de code.

Contribution II: une approche pour l'auto-configuration des compilateurs.

Comme nous l'avons indiqué précédemment, le grand nombre d'options de compilation nécessite une méthode efficace pour explorer l'espace d'optimisation. Ainsi, nous appliquons, dans cette contribution, une méta-heuristique appelée *Novelty Search* [LS08] pour l'exploration de cet espace de recherche. Cette approche aide les utilisateurs à paramétrer automatiquement les compilateurs pour une architecture matérielle cible et pour une métrique non-fonctionnelle spécifique tel que la performance et l'utilisation des ressources. Nous évaluons l'efficacité de notre approche en vérifiant les optimisations fournies par le compilateur GCC. Nos résultats expérimentaux montrent que notre approche permet d'auto-configurer les compilateurs en fonction des besoins de l'utilisateur et de construire des optimisations qui surpassent les niveaux d'optimisation standard. Nous démontrons également que notre approche peut être utilisée pour construire automatiquement des niveaux d'optimisation qui représentent des compromis optimaux entre plusieurs propriétés non-fonctionnelles telles que le temps d'exécution et la consommation des ressources.

Contribution III: Un environnement d'exécution léger pour le test et la surveillance de la consommation des ressources des logiciels.

Enfin, nous proposons une infrastructure basée sur les micro-services pour assurer le déploiement et la surveillance de la consommation des ressources des différentes variantes du code généré. Cette contribution traite le problème de l'hétérogénéité des plateformes logicielles et matérielles. Nous décrivons une approche qui automatise le processus de génération, compilation, et exécution du code dans le but de faciliter le test et l'auto-configuration des générateurs. Cet environnement isolé repose sur des conteneurs système, comme plateformes d'exécution, pour une surveillance et analyse fine des propriétés liées à l'utilisation des ressources (CPU et mémoire). Cette infrastructure constitue un terrain de jeu pour tester et régler les générateurs. Cette contribution répond principalement à *RQ3*, mais celle-ci est particulièrement utilisée pour valider les expériences portées dans *RQ1* et *RQ2*.

Chapter 1

Introduction

1.1 Context

Modern software systems rely nowadays on a highly heterogeneous and dynamic interconnection of platforms and devices that provide a wide diversity of capabilities and services. These heterogeneous services may run in different environments ranging from cloud servers with virtually unlimited resources down to resource-constrained devices with only a few KB of RAM. Effectively developing software artifacts for multiple target platforms and hardware technologies is then becoming increasingly important. As a consequence, we observe in the last years [CE00b], that generative software development received more and more attraction to tame with the runtime heterogeneity of platforms and technological stacks that exist in several domains such as mobile or Internet of Things [BCG11].

Generative programming [CE00a] offers a software abstraction layer that software developers can use to specify the desired system behavior (*e.g.*, using domain-specific languages DSLs, models, etc.), and automatically generate software artifacts. As a consequence, the new advances in hardware and platform specifications have paved the way for the creation of multiple *generators* that serve as a basis for automatically generating code to a broad range of software and hardware platforms.

Automatic code generation involves two kinds of generators: source code generators and compilers. On the one hand, code generators are needed to transform the high-level system specifications (*e.g.*, textual or graphical modeling language) into conventional source code programs (*e.g.*, general-purpose languages GPLs such as Java, C++, etc). On the other hand, compilers bridge the gap between the input programs (*i.e.*, written using GPLs)

and the target execution environment, producing low-level machine code (*i.e.*, binaries, executables) for a specific hardware architecture.

With full automatic code generation, it is now possible to develop the code easily and rapidly, improving the quality and consistency of a program as well the productivity of software development [KJB+09]. In addition, today's modern generators (*e.g.*, C compilers) become highly configurable, offering (numerous) configuration options (*e.g.*, optimization passes) to users in order to tune the produced code with respect to the target software and/or hardware platform.

In this context, it is crucial that the software being automatically generated undergoes an appropriate testing technique to verify the correct behavior of generators. Hence, users can trust the code generator and gain confidence in its correct operation. Verifying the correctness of generated code can be either functional (*e.g.*, verifying that the generated code exhibits the same functional behavior as described in the input program), or non-functional (*e.g.*, verifying the quality of generated code).

1.2 Motivation

As we stated in the context of this thesis, today's modern generators are highly configurable, letting the user to easily customize the automatically generated code. In the meantime, efficiently testing configurable generators poses important challenges since it is too costly to manually or automatically execute and test all configurations. For instance, popular generators such as GCC, LLVM, etc., are widely used in software development and they offer a large selection of configuration options to control the generator behavior. Different categories of options can be enabled (*i.e.*, option flags) to help developers to: debug, optimize, and tune application performance, select language levels and extensions for compatibility, select the target hardware architecture, and perform many other common tasks that configure the way the code is generated. The huge number of generator configurations, versions, optimizations, and debugging utilities make the task of choosing the best configuration set very difficult and time-consuming. As an example, GCC version 4.8.4 provides a wide range of command-line options that can be enabled or disabled by users, including more than 150 options for optimization. This results in a huge design space with 2^{150} possible optimization combinations that can be enabled by the user. In addition, constructing one single optimization sequence that improves the code quality for all programs is impossible since the interactions between optimizations are too complex and difficult to define. As well, the optimization's impact is highly dependent on the hardware and the input source code. This example shows how painful it is for the users to

tune generators such as compilers (through optimization flags) in order to satisfy different non-functional properties such as execution time, compilation time, code size, etc.

Before tuning generators, it is crucial to test if the code generation works properly. If so, users will trust the tool and will more likely to continue using it for production code generation. Contrarily, any issue with the generated code leads to a loss of confidence in generators and users will unlikely continue to use them during software development. As a consequence, checking the correctness of generated code has to be done with almost the same expensive effort as it is needed for manually written code. In this context, compared to compiler testing [YCER11,LAS14], code generators lack of testing solutions to automatically evaluate their correct behavior, especially for the non-functional properties. That is because code generators are less used and experienced in industry compared to compilers. In addition, they are difficult to test since they involve a set of complex and heterogeneous technologies that are internally managed in a very complex way [GS15, GS14]. Testing code generators is principally the tool experts responsibility. Nevertheless, users (*e.g.*, customers) are also responsible of this validation since they will continuously report the faults encountered during code generation. Faulty code generators can generate defective software artifacts which range from uncompilable or semantically dysfunctional code that causes serious damage to the target platform; to non-functional bugs which lead to poor-quality code that can affect system reliability and performance (*e.g.*, high resource usage, high execution time, etc.). Numerous approaches have been proposed [SCDP07, YCER11] to verify the functional outcome of generated code. However, there is a lack of solutions that pay attention to evaluate the properties related to the performance and resource usage of automatically generated code.

In summary, from the user's point of view, generators (compilers and code generators) are black box components that can be used to facilitate the software production process. The quality of the generated software is directly correlated to the quality of the generator itself. As long as the quality of generators is maintained and improved, the quality of generated software artifacts also improves. Any bug with these generators impacts on the software quality delivered to the market and results in a loss of confidence on the end users. In particular, when automatic code generation is used, we identify two major issues that threaten the quality of generated software: on the one hand, highly configurable generators control the quality of generated code through numerous optimization options than can be enabled/disabled by users. This huge configuration space poses an important challenge for users to select the best optimization options that meet some non-functional requirements. On the other hand, the complexity of code generators as well as the lack of solutions for evaluating the non-functional properties of generated code represent an obstacle for the users who need more evidence to continue using them during software development.

1.3 Scope of the thesis

In this thesis, we seek to evaluate the quality of the generated code in terms of performance and resource usage. On the one hand, we provide facilities to the code generator creators/maintainers to monitor the execution of generated code for different target software platforms and have a deep understanding of its non-functional behavior in terms of resource usage. Consequently, we automatically detect the non-functional issues caused by some faulty code generators. On the other hand, we provide a mechanism that helps users (*i.e.*, software developers) to select the best optimization sets that satisfy specific resource usage or performance requirements for a broad range of programs and hardware architectures.

This thesis addresses three problems:

(1) **The problem of non-functional testing of code generators:** We benefit from the existence of multiple code generators with comparable functionality (*i.e.*, code generator families) to automatically test the generated code. We leverage the metamorphic testing [CCY98] to detect non-functional inconsistencies in code generator families. We focus in this contribution on testing the performance and resource usage properties (*e.g.*, intensive resource usage). An inconsistency is detected when the generated code exhibits an unexpected behavior in terms of performance or resource usage compared to all equivalent implementations in the same code generator family.

(2) **The problem of generators auto-tuning:** We exploit recent advances in search-based software engineering in order to provide an effective approach to effectively tune generators (*e.g.*, GCC compilers) through optimizations. We also demonstrate that our approach can be used to automatically construct optimization levels that represent optimal trade-offs between multiple non-functional properties such as execution time and resource usage requirements.

(3) **The problem of software platforms diversity and hardware heterogeneity in software testing:** Running tests and evaluating the resource usage in heterogeneous environments is tedious. To handle this problem, we benefit from the recent advances in lightweight system virtualization, in particular container-based virtualization, in order to offer effective support for automatically deploying, executing, and monitoring code in heterogeneous environment, and collect non-functional metrics (*e.g.*, memory and CPU consumptions).

In the rest of the thesis, we use the term **"compilers"** to refer to the traditional compilers that take as input a source code and translate it into machine code like GCC, LLVM, etc. Similarly, **"code generators"** designate the software programs that transform an input program into source code like Java, C++, etc. Finally, we use the term **"generators"** to designate both, code generators and compilers.

1.4 Challenges

In existing solutions that aim to test and auto-tune generators, we find three important challenges. Addressing these challenges, which are described below, is the objective of the present work.

- **Code generators testing: the oracle problem:** One of the most common challenges in software testing is the oracle problem. A test oracle is the mechanism by which a tester can determine whether a program has failed or not. When it comes to the non-functional testing of code generators, this problem becomes very challenging because it is quite hard to determine the expected output of automatically generated code (*e.g.*, memory consumption). Determining whether these non-functional outputs correspond to a generator anomaly or not is not obvious. That is why testing the generated code becomes very complex when testers have no precise definition of the oracle they would define. To alleviate the test oracle problem, techniques such as metamorphic testing [CCY98] are widely used to test programs without defining an explicit oracle. Instead, it employs high-level metamorphic relations to automatically verify the outputs. So, which kind of test oracles can we define? How can we automatically detect inconsistencies? All these questions pose important challenges in testing code generators.

- **Auto-tuning compilers: exploring the large optimizations search space:** The current innovations in science and industry demand ever-increasing computing resources while placing strict requirements on many non-functional properties such as system performance, power consumption, size, reliability, etc. In order to deliver satisfactory levels of performance on different processor architectures, compiler creators often provide a broad collection of optimizations that can be applied by compiler users in order to improve the quality of generated code. However, to explore the large optimization space, users have to evaluate the effect of optimizations according to a specific performance objective/trade-off. Thus, constructing a good set of optimization levels for a specific system architecture/target application becomes very

challenging. Moreover, due to the complex interactions and the unknown effect of optimizations, users find difficulties to choose the adequate compiler configuration that satisfies a specific non-functional requirement.

- **Runtime monitoring of generated code: handling the diversity of software and hardware platforms:** To evaluate the properties related to the resource usage of generated code (by compilers or code generators), developers use generally to compile, deploy and execute the generated software artifacts on different execution platforms. Then, after gathering the non-functional metrics, they report issues related to the code generation process such as program crash, abnormal termination, memory leaks, etc. In fact, developers often use several platform-specific profilers, debuggers, and monitoring tools [GS14, DGR04] in order to find some inconsistencies or bugs during code execution. Due to the heterogeneity of execution platforms and hardware, collecting information about the resource usage of generated code becomes very hard and time-consuming tasks since developers have to analyze and verify the generated code for different target platforms using platform-specific tools.

The challenges this research tackle can be summarized in the following research questions. These questions arise from the analysis of the challenges presented in the previous paragraphs.

RQ1. How can we help code generator creators/maintainers to automatically detect non-functional inconsistencies in code generators?

RQ2. How can we help compiler users to automatically choose the adequate compiler configuration that satisfies a specific non-functional requirement?

RQ3. How can we provide efficient support for resource consumption monitoring in heterogeneous environment?

1.5 Contributions

This thesis establishes three core contributions. They are briefly described in the rest of this section.

Contribution I: Automatic detection of inconsistencies in code generator families. In this contribution, we tackle the oracle problem in the domain of code generators testing. The availability of multiple generators with comparable functionality (*i.e.*, code generator families) allows us to apply the idea of metamorphic testing [ZHT+04] by defining high-level test oracles (*i.e.*, metamorphic relation) to detect issues. We define the metamorphic relation as a comparison between the variations of performance and resource usage of code, generated from the same code generator family. We evaluate our approach by analyzing the performance of Haxe, a popular high-level programming language that involves a set of cross-platform code generators. We evaluate the properties related to the resource usage and performance for five target software platforms. Experimental results show that our approach is able to detect several performance inconsistencies that reveal real issues in this code generator family.

Contribution II: An approach for auto-tuning compilers. As we stated earlier, the huge number of compiler options requires the application of a search method to explore the large design space. Thus, we apply, in this contribution, a search-based meta-heuristic called *Novelty Search* [LS08] for compiler optimizations exploration. This approach helps compiler users to efficiently explore the large optimization search space and auto-tune compilers according to the performance and resource usage properties and that, for a specific hardware architecture. We evaluate the effectiveness of our approach by verifying the optimizations performed by the GCC compiler. Our experimental results show that our approach is able to auto-tune compilers according to the user requirements and construct optimizations that yield to better performance results than standard optimization levels. We also demonstrate that our approach can be used to automatically construct optimization levels that represent optimal trade-offs between multiple non-functional properties such as execution time and resource usage requirements.

Contribution III: A lightweight execution environment for software testing and monitoring. Finally, we propose a micro-service infrastructure to ensure the deployment and monitoring of the different variants of generated code. This contribution addresses the problem of software and hardware heterogeneity. Thus, we describe an approach that automates the process of code generation and compilation, execution, and monitoring in order to facilitate the testing and auto-tuning process. This isolated and sand-boxing environment is based on system containers, as execution platforms, to provide a fine-grained understanding and analysis of the properties related to the resource usage (CPU and memory). This approach constitutes the playground for testing and tuning generators. This contribution answers mainly *RQ3* but the same infrastructure is particularly used to validate the carried experiments in *RQ1* and *RQ2*.

1.6 Overview of this thesis

The remainder of this thesis is organized as follows:

Chapter 2 first contextualizes this research, situating it in the domain of generative programming. We discuss several concepts and stakeholders involved in the field of generative programming as well as an overview of the different aspects of automatic code generation. Finally, we discuss the different problems that make the task of generators auto-tuning and testing very difficult.

Chapter 3 presents the state of the art of the thesis. This chapter provides a survey of the most used techniques for tuning and testing generators. This chapter is divided into three sections. First, we study the different techniques used to test the functional and non-functional properties of code generators. Second, we study the previous approaches that have been applied for auto-tuning compilers. Finally, we present several research efforts that used the container-based architecture for software testing and monitoring. At the end, we provide a summary of the state of the art and we discuss several open challenges.

Chapter 4 presents our approach for non-functional testing of code generator families. It shows an adaptation of the idea of metamorphic testing to automatically detect code generator issues. We conduct a statistical analysis and we report the experimental results conducted using an example of code generator families. The inconsistencies we detect are related to the performance and memory usage of generated code. We also propose solutions to fix the detected issues.

Chapter 5 resumes our contribution for compiler auto-tuning. Thus, we present an iterative process based on a search technique called Novelty Search for compiler optimization exploration. We provide two adaptations of this algorithm: mono and multi-objective optimization. We also show how this technique can easily help compiler users to efficiently generate and evaluate compiler optimizations. The non-functional metrics we are evaluating are the performance, memory, and CPU usage. We evaluate this approach through an empirical study and we discuss the results.

Chapter 6 shows the testing infrastructure used across all experiments. It shows the usefulness of such architecture, based on system containers, to automatically execute and monitor the automatically generated code.

Chapter 7 draws conclusions and identifies future directions for testing and auto-tuning generators.

1.7 Publications

- Mohamed Boussaa, Olivier Barais, Gerson Sunyé, Benoît Baudry: **Automatic Non-functional Testing of Code Generators Families**. In *The 15th International Conference on Generative Programming: Concepts & Experiences (GPCE 2016)*, Amsterdam, Netherlands, October 2016.

- Mohamed Boussaa, Olivier Barais, Benoît Baudry, Gerson Sunyé: **NOTICE: A Framework for Non-functional Testing of Compilers**. In *2016 IEEE International Conference on Software Quality, Reliability & Security (QRS 2016)*, Vienna, Austria, August 2016.

- Mohamed Boussaa, Olivier Barais, Gerson Sunyé, Benoît Baudry: **A Novelty Search-based Test Data Generator for Object-oriented Programs**. In *Genetic and Evolutionary Computation Conference Companion (GECCO 2015)*, Madrid, Spain, July 2015.

- Mohamed Boussaa, Olivier Barais, Gerson Sunyé, Benoît Baudry: **A Novelty Search Approach for Automatic Test Data Generation**. In *8th International Workshop on Search-Based Software Testing (SBST@ICSE 2015)*, Florence, Italy, May 2015.

Part I

Background and State of the Art

Chapter 2

Background

In this chapter, the context of this thesis and the general problems it faces are introduced. The objective of this chapter is to give a brief introduction to different domains and concepts in which our work takes place and used throughout this document. This includes an overview of the generative software development process as we see in the context of this thesis, the main actors and their roles for tuning and testing configurable generators, and the main challenges we are facing.

The chapter is structured as follows:

In Section 2.1, we present the problem of software diversity and hardware heterogeneity caused by the continuous innovation in science and technology.

Section 2.2 aims at providing a better understanding of the generative programming concept. We present the different steps of automatic code generation involved during software development as well as the different stakeholders and their roles in testing and tuning generators. We highlight then, the main activities that the software developer goes through from the software design until the release of the final software product.

Section 2.3 gives an overview of the different types of code generators used in the literature and we show the complexity of testing code generators.

Similarly, in Section 2.4, we describe some compiler optimizations and we illustrate the compiler auto-tuning complexity by presenting the different challenges that this task is posing.

Finally, in Section 2.5, we conclude by providing a summary of the relevant challenges for testing and tuning configurable generators.

2.1 Diversity in software engineering

The history of software development shows a continuous increase of complexity in several aspects of the software development process. This complexity is highly correlated with the actual technological advancement in the software industry as more and more heterogeneous devices are introduced in the market [BCG11]. Generally, heterogeneity may occur in terms of different system complexities, diverse programming languages and platforms, types of systems, development processes and distribution among development sites [GPB15]. System heterogeneity is often led by software and hardware diversity. Diversity emerges as a critical concern that spans all activities in software engineering, from design to operation [ABB+14]. It appears in different areas such as mobile, web development [DA13], security [ABB+15], etc.

However, software and hardware diversity leads to a greater risk for system failures due to the continuous change in configurations and system specifications. As a matter of a fact, effectively developing software artifacts for multiple target platforms and hardware technologies is then becoming increasingly important. Furthermore, the increasing relevance of software and the higher demand in quality and performance contribute to the complexity of software development.

In this background introduction, we discuss two different dimensions of diversity: (1) hardware heterogeneity, and (2) software diversity.

2.1.1 Hardware heterogeneity

Modern software systems rely nowadays on a highly heterogeneous and dynamic interconnection of devices that provide a wide diversity of capabilities and services to the end users. These heterogeneous services run in different environments ranging from cloud servers to resource-constrained devices. Hardware heterogeneity comes from the continuous innovation of hardware technologies to support new system configurations and architectural design (*e.g.*, addition of new features, a change in the processor architecture, new hardware is made available, etc). For example, until February 2016[1], the increase in capacity of microprocessors has followed the famous Moore's law[2] for Intel processors. Indeed, the number of components (transistors) that can be fitted onto a chip doubles every two years, increasing the performance and energy efficiency. For instance, Intel Core 2 Duo processor was introduced in 2006 with 291 millions of transistors and 2.93 GHz clock speed. Two

[1] https://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time/
[2] https://en.wikipedia.org/wiki/Moore%27s_law

years later, Intel has introduced the Core 2 Quad processors which came up with 2.66 GHz clock speed and the double number of transistors introduced in 2006 with 582 millions of transistors.

In the last years, modern processors becomes more and more heterogeneous, using more than one kind of processor or cores, called "co-processors". The CPU can even use different instruction set architectures (ISA), where the main processor has one and the rest has another, usually a very different architecture. Operations performed by the co-processor may be floating point arithmetic, graphics, signal processing, string processing, encryption or I/O Interfacing with peripheral devices. As an example, the ARM big.Little processor architecture[3] released in 2011 (see Figure 2.1), is a power-optimization technology where high-performance ARM CPU cores are combined with the most efficient ARM CPU cores to deliver peak-performance capacity and increased parallel processing performance, at significantly lower average power. It can save 75% of CPU energy and can increase performance by 40% in highly threaded workloads. The intention of this architecture is to create a multi-core processor that can adjust better to dynamic computing needs. Threads with high priority or computational intensity can in this case be allocated to the "Big" cores while threads with less priority or less computational intensity, such as background tasks, can be performed by the "Little" cores. This model has been implemented in the Samsung Exynos 5 Octa in 2013[4].

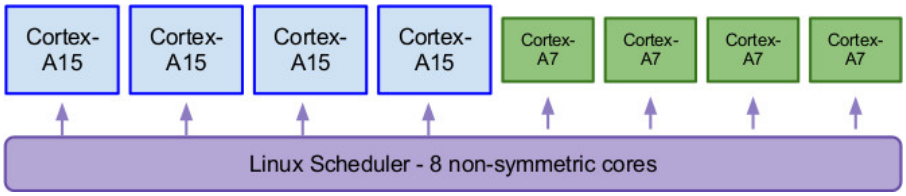


Figure 2.1: ARM Big.Little heterogeneous multi-processing

Given the complexity of new emerging processor architectures (x86, x64, multi-core, etc) and CPU manufacturers such as ARM, AMD, and Intel, some of the questions that developers have to answer when facing hardware heterogeneity: Is it easy to deliver satisfactory levels of performance on modern processors? How is it possible to produce machine code that can exploit efficiently the continuous hardware changes?

[3] https://en.wikipedia.org/wiki/ARM_big.LITTLE

[4] http://www.embedded.com/electronics-news/4419448/Benchmarking-ARM-s-big-little-architecture

2.1.2 Software diversity

In todays software systems, different software variants are typically developed simultaneously to address a wide range of application contexts and customer requirements [SRC+12]. Therefore, software is built using different approaches and languages, depending on the application domain.

In the literature, Baudry *et al.* [BM15b] and Schaefer *et al.* [SRC+12] have presented an exhaustive overview of the multiple facets of software diversity in software engineering. According to their study, software diversity can emerge in different types and dimensions such as diversity of operating systems, programming languages, data structures, components, execution environments, etc. Like all modern software systems, software need to be adapted to address changing requirements over time supporting system evolution, technology and market needs like considering new software platforms, new languages, new customer choices, etc.

In order to understand the skills and capabilities required to develop software on top of different classes of devices and application domains, we queried a popular open-source repository *GitHub* to evaluate the diversity of existing programming languages. The following sets of keywords were used: *1) Cloud:* server with virtually unlimited resources, *2) Microcontroller:* resource constrained node (few KB RAM, few MHz), *3) Mobile:* an intermediate node, typically a smartphone, *4) Internet of Things:* Internet-enabled devices, *5) Distributed systems*, and *6) Embedded systems*, as a large and important part of the service implementations will run as close as possible to physical world, embedded into sensors, devices and gateways.

Figure 2.2 presents the results of those queries. The queried keywords are presented on the *x-axis* together with the number of matches for that keyword. For each keyword, the *y-axis* represents the popularity (in per cent of the total number of matches) of each of the 10 most popular programming languages that we encountered.

This simple study indicates that no programming language is popular across the different areas. A general trend indicates that Java and JavaScript (and to some extent, Python and Ruby) are popular in cloud and mobile, whereas C (and to some extent, C++) is a clear choice for developers targeting embedded and microcontroller-based systems. Other languages do not score more 10% for any of the keywords. For all keywords except Cloud, the combined popularity of Java, JavaScript and C/C++ (*i.e.*, the sum of the percentages) is above 70%. For Cloud, we observe a large use of Python, Ruby also being very popular, so the combined popularity of Java, JavaScript and C/C++ is only 50%. It is also worth noticing that the most popular language for a given keyword scores very poorly (less than

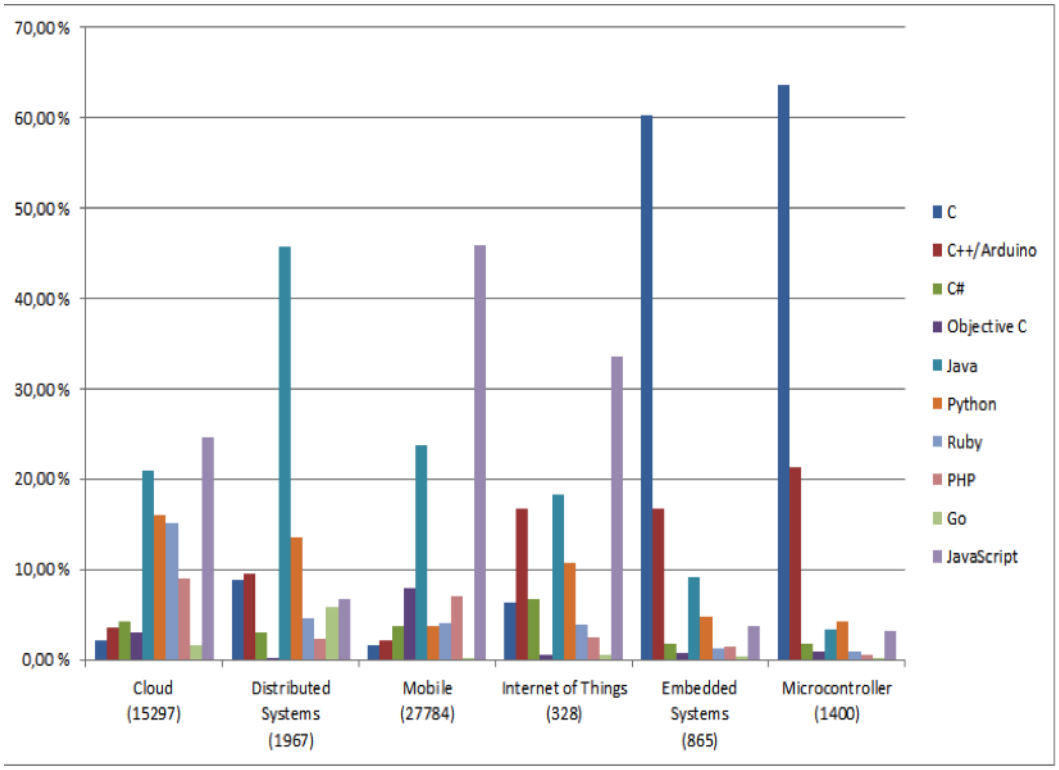


Figure 2.2: Popularity of 10 programming languages in different application domains

5%) for at least another keyword. While it might appear that a combination of C/C++, JavaScript and Java should be able to cover all the areas, in practice it does not exclude the need for other programming languages. For example, the Fibaro Home Center 2 (a gateway for home automation based on the Z-Wave protocol) uses Lua as scripting language to define automation rules. Another example is the BlueGiga BlueTooth Smart Module, which can be scripted using BGScript, a proprietary scripting language. This shows that each part of an infrastructure might require the use of a niche language, middleware or library to be exploited to its full potential.

In summary, the variation of programming languages for the different kinds of devices and application domains induces a high *software diversity*. Accordingly, we propose the following definition of software diversity in the context of this thesis: *Software diversity is the generation or implementation of the same program specification in different ways/manners in order to satisfy one or more diversity dimensions such as the diversity of programming languages, execution environments, functionalities, etc.*

2.1.3 Matching software diversity to heterogeneous hardware: the marriage

2.1.3.1 Challenges

The hardware and software communities are both facing significant changes and major challenges. Figure 2.3 shows an overview of the challenges that both communities are facing. In fact, hardware and software are pulling us in opposite directions.

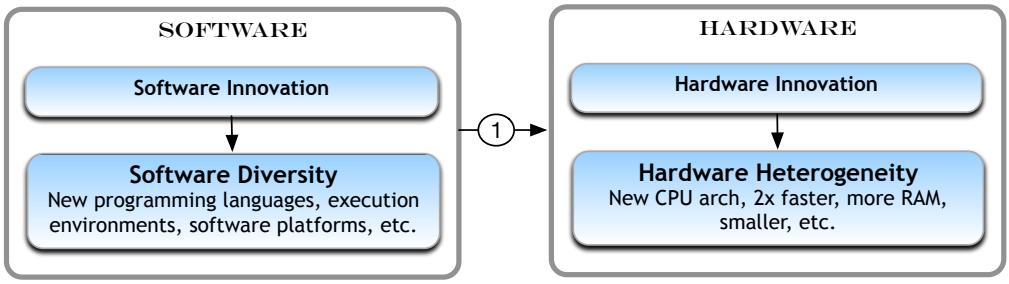


Figure 2.3: Matching software to hardware

On the one hand, software is facing challenges of a similar magnitude, with major changes in the way software is deployed, is sold, and interacts with hardware. Software diversity, as discussed in Section 2.1.2, is driven by software innovation, driving the software development toward highly configurable and complex systems. This complexity is carried by the huge diversity of software technologies, customer configurations, execution environments, programming languages, etc. This explosion of configurations that software is facing makes the activity of testing and validation very difficult and time consuming. As a consequence, software becomes higher and higher level, managing complexity and gluing lot of pieces together to give programmers the right abstraction for how things really work and how the data is really represented.

On the other hand, hardware is exposing us to more low-level details and heterogeneity due to the continuous hardware innovation. Hardware innovation offers us energy efficiency, performance improvement but exposes a lot of complexity for software engineers and developers. For example, in [He10], authors argue that system software is not ready for this heterogeneity and cannot fully benefit from new hardware advances such as multi-core and many-core processors. Although multi-core processors have been used in everyday life, we still do not know how to best organize and use them. Meanwhile, hardware specialization for every single application is not a sustainable way of building chips.

2.1.3.2 Mapping software to hardware

Matching software to hardware is ensured by the efficient translation of the high-level software programs into a machine code that better exploit the hardware changes (relation 1 in Figure 2.3). This is exactly what a compiler is intended to do.

Configuring existing compilers

Gone are the days where we used to write the assembly code by hand and from scratch. Now, it is up to the compilers to handle this heterogeneity and to efficiently generate and optimize the code for a particular microprocessor.

As shown in Figure 2.4, a compiler is typically divided into two parts, a front-end and a back-end. The compiler front-end verifies the syntax and semantics and analyzes the source code to build an internal representation of the program, called the intermediate representation or IR. For example, the GNU Compiler Collection (GCC) and LLVM support many front-ends with programming languages such as C, C++, Objective-C, Objective-C++, Fortran, Java, Ada, and Go, among others. The compiler back-end generates the target-dependent assembly code and performs optimizations for the target hardware architecture. Typically, the output of a back-end is a machine code specialized for a particular processor and operating system (*e.g.*, ARM, Sparc processors, etc). As a consequence, people who are writing compilers have to continuously enhance the way these executables are produced by releasing new compiler versions to support new hardware changes (*i.e.*, introducing new optimization flags, instruction sets, etc.).

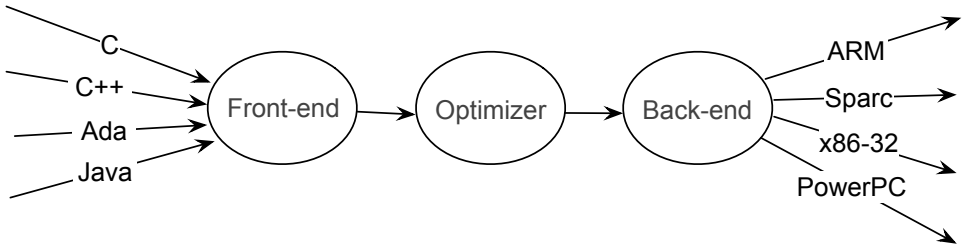


Figure 2.4: Compiler architecture

Let's take the GCC example. GCC is able to generate code automatically for approximately **more than 40 different processor architectures**. Hence, GCC becomes highly configurable, allowing the compiler user to enable multiple flags to customize the generated code. For instance, one important compiler flag is *-march*. It tells the compiler what

code it should produce for the system's processor architecture. It tells GCC that it should produce code for a certain kind of CPU. Using *-march=native* enables all the optimization flags that are applicable for the native system's CPU, with all its capabilities, features, instruction sets, and so on. There exits many other configuration options for the target CPU like *–with-arch=i7*, *–with-cpu=corei7*, etc. Generally, each time a new family of processors is released, compiler developers release a new compiler version with more sophisticated configuration options for the target platform. For example, old compilers produce only 32-bit programs. These programs still run on new 64-bit computers, but they may not exploit all processor capabilities (*e.g.*, they will not use the new instructions that are offered by x64 CPU architecture). For instance, the current x86-64 assembly language can still perform arithmetic operations on 32-bit registers using instructions like addl, subl, andl, orl, etc, with the l standing for "long", which is 4 bytes/32 bits. 64-bit arithmetic is done with addq, subq, andq, orq, etc, with q standing for "quadword", which is 8 bytes/64 bits.

Another example is that compilers need to support parallelism. In fact, we can see that modern computers today can do many things at once and modern CPUs become highly parallel processors with different levels of parallelism (*e.g.*, the ARM Big.Little in Figure 2.1). We find parallelism everywhere from the parallel execution units in a CPU core, up to the SIMD (Single Instruction, Multiple Data) instruction set and the parallel execution of multiple threads. One of the commonly applied optimizations by modern compilers in parallel computing is vectorization. It constitutes the process of converting an algorithm from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation, which processes one operation on multiple pairs of operands at once. Programmers can exploit vectorization using compilers to speedup certain parts of their code. One hot research topic in computer science is the search for methods of automatic vectorization [NRZ06]: seeking methods that would allow a compiler to convert scalar algorithms into vectorized algorithms without human intervention.

In short, to cope with heterogeneous hardware platforms, software developers use these highly configurable compilers (for compiled languages such as C or C++) in order to efficiently compile their high-level source code programs and execute them on top of a board range of platforms and processors.

Masking hardware heterogeneity

Sometimes, software developers try to avoid the hardware heterogeneity. Thus, they use for example managed languages such as Java, Scala, C#, etc to favor software portability. Instead of compiling to native machine instruction set, these languages are compiled into an intermediate language or IL, which is similar to a binary assembly language. These

instructions are executed by a JVM, or by .NET's CLR virtual machine, which effectively translates them to native binary instructions specific to the CPU architecture and/or OS of the machine. By using managed code, memory management such as type safety checking and destruction of unneeded objects (*i.e.*, garbage collector) are handled internally within this sandbox runtime environment. Thus, developers focus on the business logic of applications to provide more secure and stable software without taking too much care of the hardware heterogeneity. However, using managed languages has drawbacks. It includes slower startup speed (the managed code must be JIT compiled by the VM). It can also be slower than native code and generally more greedy in terms of system resources. For example, we can see in Figure 2.2, that the C language is the most widely used programming language in the context of embedded systems[5] where the system is really resource-constrained. Contrarily to managed languages, C utilizes the hardware to its maximum by multi-processing and multi-threading APIs provided by POSIX. It also controls the memory management and uses less memory (which allows more freedom on memory management compared to the use of garbage collector).

Building new DSLs and compilers

An alternative approach for matching software to hardware is to build new languages and compilers for a specific domain from scratch. For example, Hou *et al.* [HZG10] have presented a container-based programming language for heterogeneous many-core systems. This DSL allows programmers to write unified programs that are able to run efficiently on heterogeneous processors. To map this DSL to such hardware processors, they provide a set of compilers and runtime environments for the x86 CPUs and CUDA GPUs. Similarly, Chafi *et al.* [CDM+10, CSB+11] proposed leveraging DSLs to map high-level application code to heterogeneous devices. Results show that the presented DSL can achieve high performance on heterogeneous parallel hardware with no modification required to the source code. They compared this language performance to MATLAB code and they showed that it outperformed it in nearly all cases.

In short, hardware heterogeneity raises many challenges for the software community that need to create or deal with highly configurable generators (*i.e.* Compilers) to truly take advantage of the new chip with more advanced optimizations for the new hardware settings.

[5] http://www.eetimes.com/author.asp?doc_id=1323907

2.2 From classical software development to generative programming

In comparison to the classical approach where software development was carried out manually, todays modern software development requires more automatic and flexible approaches to face the continuous innovation in industry production, as described in the previous sections. Hence, more generic tools, methods and techniques are applied in order to keep the software development process as easy as possible for testing and maintenance and to handle the different requirements in a satisfyingly and efficient manner. As a consequence, generative programming (GP) techniques are increasingly applied to automatically generate and reuse software artifacts.

Definition (Generative programming). *Generative programming is a software engineering paradigm based on modeling software families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge [CE00b].*

Generative software development consists in using higher-level programming techniques such as meta-programming, modeling, DSL, etc. in order to provide a new integrated software engineering solution that enables the exploitation of the different dimensions of software diversity (*e.g.*, through automatic code generation).

In principle, the generative programming concept can be seen as a mapping between a problem space and a solution space [Cza05] (see Figure 2.5).

The problem space is a set of domain-specific abstractions that can be used by application engineers to express their needs and specify the desired system behavior. This space is generally defined as DSLs or high-level models.

The solution space consists of a set of implementation components, which can be composed to create system implementations (*e.g.*, the generation of platform-specific software components written using GPLs such as Java, C++, etc.).

The configuration knowledge constitutes the mapping between both spaces. It takes a specification as input and returns the corresponding implementation as output. It defines the construction rules (*i.e.*, the translation rules to apply in order to translate the input model/program into specific implementation components) and optimizations (*i.e.*, optimization can be applied during code generation to enhance some of the non-functional

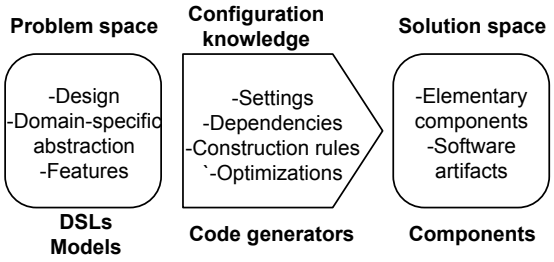


Figure 2.5: Generative programming concept

properties such as execution speed). It defines also the dependencies and settings among the domain specific concepts and features.

This schema integrates several powerful concepts from model driven engineering, such as domain-specific languages, feature modeling, and code generators.

Some commonly benefits of such software engineering process are:

- It reduces the amount of re-engineering/maintenance caused by specification requirements.
- It facilitates the reuse of components/parts of the system.
- It increases the decomposition and modularization of the system.
- It handles the heterogeneity of target software platforms by automatically generating code.

An example of generative programming application is the use of Software Product Lines (SPL) [SRC+12]. SPL-based software diversity is often coupled to generative programming techniques [CE00b] that enable the automatic production of source code from variability models. This technique implies the use of automatic code generators to produce code that satisfies user requirements (SPL models). This technique enables one to manage a set of related features in order to build diverse products in a specific domain. Thus, this solution is able to control software diversity by handling the diversity of requirements such as user requirements or environmental constraints or changes.

2.2.1 Overview of the generative software development process

The generative software development process involves many different technologies. In this section, we describe in more details the different activities and stakeholders involved to

automatically transform the high-level system specifications into executable programs and that from design time to runtime.

Figure 2.6 gives an overview of this process, as we see in the context of this thesis. We distinguish four main tasks necessary for ensuring the automatic code generation:

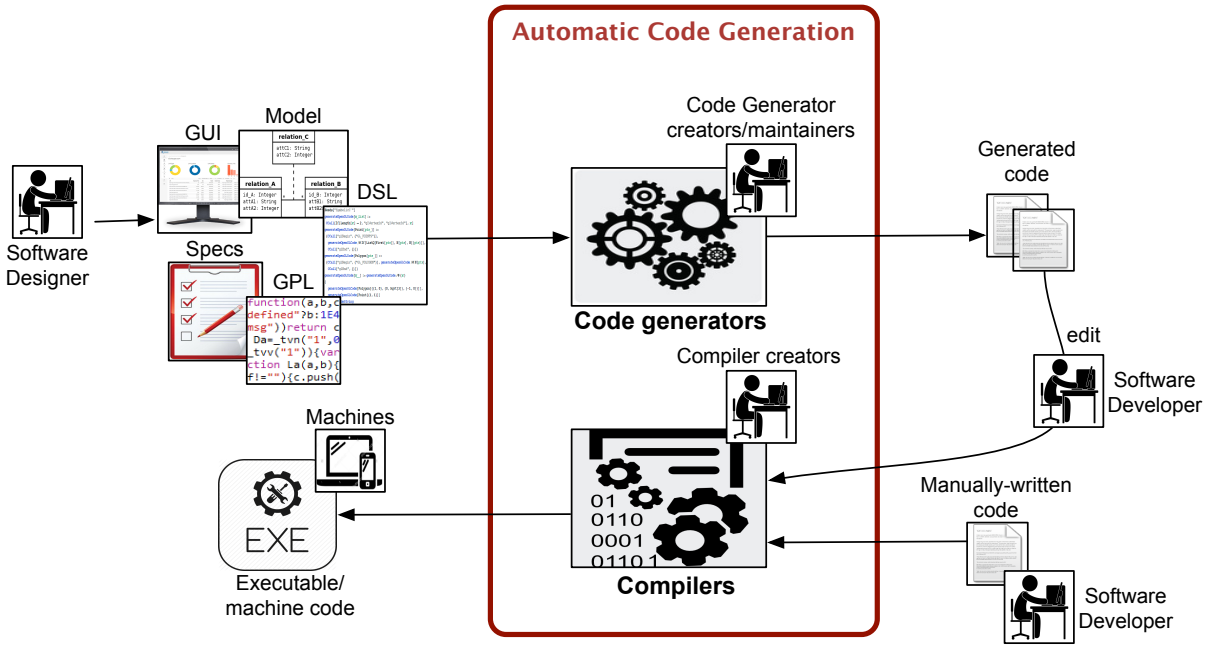


Figure 2.6: Overview of the generative software development process

1. ***Software design:*** As part of the generative programming process, the first step consists on representing the system behavior. On the input side, we can either use code as the input or an abstract form that represents the design. It depends on the type of the code generator and on the input source program it requires. These programs can range from a formal specification of the system behavior to abstract models that represent the business logic. For example, designers can define, at design time, the software behavior using for example Domain-Specific Models (DSMs). A DSM is a system of abstractions that describes selected aspects of a sphere of knowledge and real-world concepts pertinent to the domain that needs to be designed. These models are specified using high-level abstract languages (*i.e.*, DSLs).

2. ***Code generation:*** Code generation is the technique of building code using programs. The common feature of the generator is to produce code that the software developer would otherwise write by hand. Code generators are generally seen as a black box which requires as input a program and generate as output a source code for a specific target software platform/language. Code generation can build code for one or more target language, once or multiple times. There are different varieties of code generation aspects and it highly depends on the input category as described in the previous step. For example, code generator developers use model-driven engineering techniques in order to automatically generate code. Instead of focusing their efforts on constructing code, they build models and, in particular, create model transformations that transform these models into new models or code. Thus, the code generation process starts by taking the previously defined specification in order to translate a model to an implementation in a target language. We will see in Section 2.3.2 the different types of code generators.

3. ***Software development:*** Software development may be divided into two main parts. On the one hand, software developers may follow the two previous steps in order to automatically generate code for a specific target software platform. In this case, they might edit the system specification described in the first step (at a high level) and re-generate code each time needed by calling a specific generator. In some cases, generated code can even be edited by the end software developers. This task depends on the complexity of the generated code. Sometimes, it requires the help of domain experts who have enough expertise and knowledge to easily update and maintain the automatically generated code. On the other hand, they may manually implement the source code from scratch without going through any abstractions or code generation aspects. In this case, they may integrate the manually-written code with the automatically generated in order to deliver the final software product.

4. ***Compilation:*** Once code is generated or implemented, a classical compiler is used (if needed) to translate the generated code into an executable one. This translation depends on the target hardware platforms and it is up to the software developer to select the adequate compiler to use. Compilers are needed to target heterogeneous and diverse kinds of hardware architectures and devices. As an example, cross compilers may be used to create executable code for a platform other than the one on which the compiler is running. In case the generated code needs to run on different machines/devices, the software developer needs to use different compilers for each target software platform and deploy the generated executables within different machines which is a tedious and time-consuming task.

2.2.2 Automatic code generation in GP: a highly configurable process

Among the main advantages that GP offers, is the automatic code generation, highlighted with red box in Figure 2.6. Automatic code generation emerges in two principal aspects:

1. The use of code generators to cope with software diversity and automatically generate code to a broad range of software platforms, as we discussed in Section 2.1.2.

2. The use of compilers to cope with hardware heterogeneity and automatically generate code to a broad range of hardware platforms, as we discussed in Section 2.1.1.

Both, compilers and code generators, are responsible for the automatic code generation in GP. To satisfy the different software and hardware requirements, modern generators provide many configuration options to easily tune the generated code:

Compilers, on the one hand, become highly configurable and very user-friendly, letting the user to easily introduce optimizations and customize the machine code to fit with target hardware settings. As an example, Table 2.1 depicts the number of optimizations available in three popular compilers. The user can configure the compiler by selecting one of the 2^n possible optimization sequences, where n is the number of optimizations available in the compiler. We can see that the configuration space is very large.

Table 2.1: Number of optimizations in LLVM, GCC, and ICC

Compiler	#Optimizations	#Combinations
LLVM	100	2^{100}
GCC	250	2^{250}
ICC	75	2^{75}

On the other hand, code generators offer the possibility to customize the generated code for the target software platform. Code generators provide general configuration options necessary for building software artifacts (*e.g.*, select the target programming language, dependencies, platform settings, libraries, etc.). As an example, JHipster[6] is a concrete example of generative programming application in industry. JHipster is an application generator based on YO generator which provides tools to generate quickly modern web applications using Java stack on the server side (using Spring Boot) and a responsive

[6] https://jhipster.github.io/

Web front-end on the client side (with AngularJS and Bootstrap). The generated web application can be quite different from one user to another. It really depends on the options/choices selected by the user to build a configured application. The selected parameter values will configure the way the JHipster code generators will produce code. For example, Figure 2.7 shows a feature model of some configuration examples that the user can select. When building applications, the user may select the database type he would generate, the Java version, the network protocol, etc. Using this feature model, **more than 10k diverse architecture types** of project can be selected which means that 10k program variants may be generated depending on the different criteria. Whatever configuration selected by the user, the application behavior will not change and the generated application will share a similar architecture and fundamental code-base.

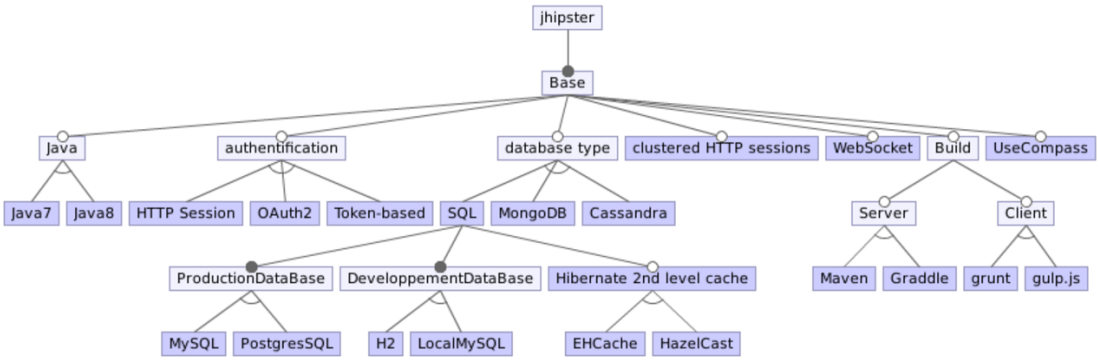


Figure 2.7: Example of JHipster feature model

Well, we can see that both generators are highly configurable. However, in practice, code generators are less used in industry compared to compilers. First, because compilers are required for machine code production and optimization. Then, users of popular compilers such as GCC or LLVM have enough experience and confidence on the correct translation of the code. Code generators on the other side are less used because users do not have enough experience with them and need to gain confidence on their correct operation by rigorously testing them.

In summary, automatic code generation in GP faces two major challenges. On the one hand, configurable generators need to be efficiently tuned in order to produce high-quality software products. On the other hand, they have to be rigorously tested in order to provide evidence to the users of the efficiency of generated code.

We describe in the next section the main stakeholders involved in the automatic code generation in GP and their roles for validating this process.

2.2.3 Stakeholders and their roles for testing and tuning generators

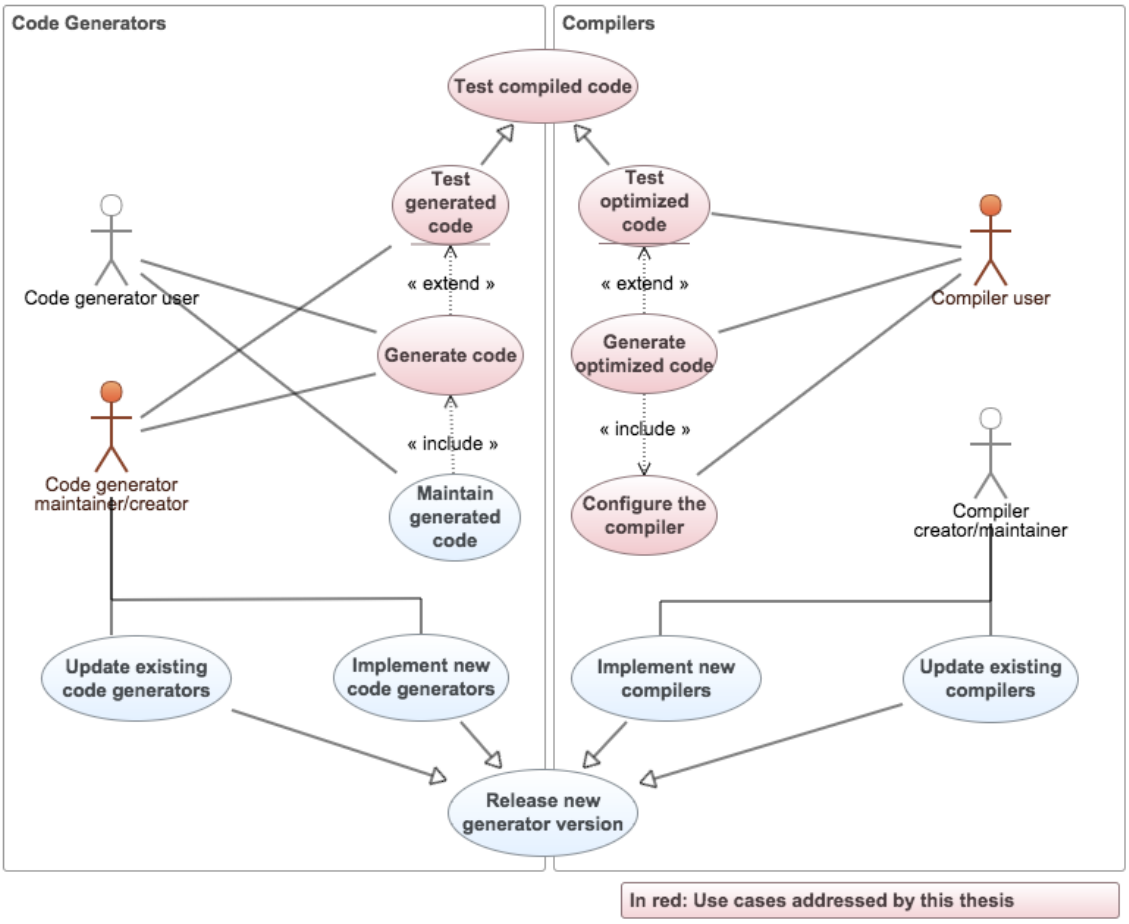


Figure 2.8: Use case diagram of the different actors/roles involved in testing and tuning generators

Software development involves several stakeholders that play different roles for validating and testing the software development chain described previously. Figure 2.8 depicts a use case diagram that describes these different concerns, actors, and roles for testing and tuning generators.

Basically, we distinguish two stakeholders: generator users and creators/maintainers. As shown in the bottom of Figure 2.8, creators/maintainers are responsible of the correct

functioning of generators. They use their expertise and knowledge associated to the software and hardware technologies, resulting in efficient code generation. They contribute to the software development community by creating and providing new generator updates (*e.g.*, introduce new optimizations, build new platform-specific generators or enhance existing ones).

On the other side, generator users represent the group of software developers that have no knowledge/expertise about the way code is generated. Thus, they are unable to edit or maintain the internal behavior of generators (*e.g.*, the case of commercial and off-the-shell generators). In this case, generators are used as black box components by engineers during software development to ease code production. Developers may configure generators in order to efficiently produce code for the target hardware platform (*e.g.*, by providing a set of optimization options) or maintain/edit the generated code in case of automatic source code generation.

The use cases highlighted in red in Figure 2.8 constitute the main tasks that we are addressing in this thesis. Our main concern is to evaluate the non-functional properties of automatically generated code. We involve two generator stakeholders, creators/maintainers and users (highlighted with red stereotypes). On the one hand, we would help code generator creator/maintainers to test the generated code and detect code generator issues by evaluating the resource usage and performance properties. This task may also involve code generator users but it is mainly the task of generator experts. On the other hand, we would help compiler users to auto-tune compilers through the use of optimizations provided by compiler experts. It consists in evaluating the impact of these configurations on the performance and resource usage properties in order to find the best set of optimizations for a specific program and target hardware architecture.

2.3 Testing code generators

In this thesis, we focus on testing the automatic code generation process (highlighted in red in the left side of Figure 2.8). To do so, we introduce in this section some basis about code generators. We give an overview of the different types of code generators and we discuss their complexity which constitutes a major obstacle for testing.

2.3.1 Testing workflow

The main goal of generators is to produce software systems from higher-level specifications. As stated before, the code generation workflow is divided into two levels. It starts by trans-

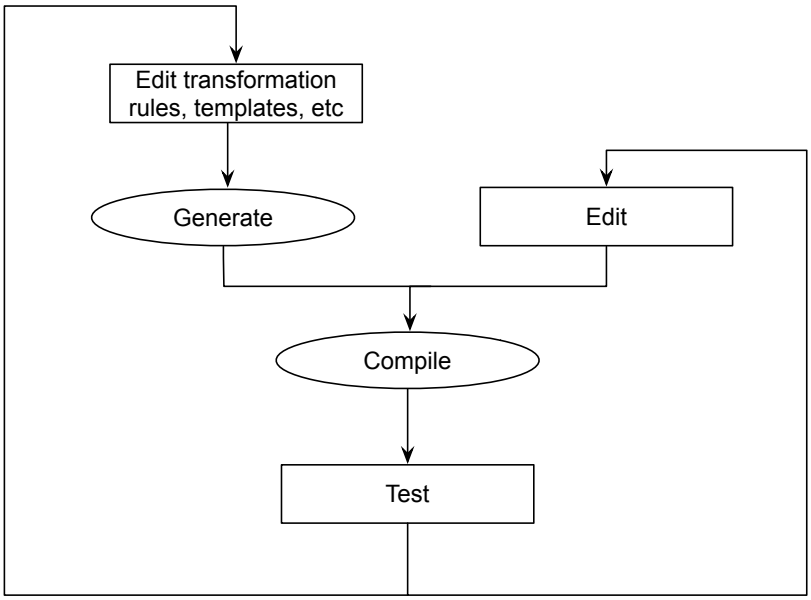


Figure 2.9: Code generation workflow

forming the system design into source code through the use of code generators. Afterwards, source code is transformed into executables using compilers. Thus, software developers use to generate code, edit it (if needed), compile it and then test it. If changes are applied to compilers or generators, the cycle is repeated. Figure 2.9 presents an overview of this testing cycle. The right-hand side of the figure shows the classic workflow for developing and debugging code which is *edit, compile, and test*. The user writes or edits an existing code, compiles it using specific compilers, and tests it. Code generation adds a few new workflow elements in the left-hand side of the figure where generator creators edit the templates and transformation rules (or the generator itself) and then run the code generator to create new output files. The output files are then compiled and the application is tested.

2.3.2 Types of code generators

There are many ways to categorize generators. We can differentiate them by their complexity, by usage, or by their input/output. According to [Her03], there are two main

categories of automatic code generation: passive or active. Passive code generators build the code only once, then it is up to the user to update and maintain the code. The most common use of passive code generators are wizards.

Active code generators, run on code multiple times during the lifecycle. With active code generators, there is code that can be edited by the users, and code that should only be modified by the code generator. Active code generators are widely referenced in the literature [PBG05, AE09]. We focus in this thesis, on testing this class of code generators. In the literature [Her03, HT00, FB08, BNS13], researchers define six categories of active code generators:

- **Code munger:** A code munger reads code as input and then builds new code as output. This new code can either be partial or complete depending on the design of the generator. A code munger is the most common form of code generators and are used widely. This kind of generators are often used for automatically generating documentations. A source-to-source compiler, transcompiler or transpiler[7] can also be defined as code mungers. A transcompiler takes a code written in some programming language and translates it to a code written in some other language. **Our contribution related to code generators testing will focus on this kind of generators to validate our approach for automatically detecting inconsistencies.**

  Examples: C2J, JavaDoc, Jazillian, Closure Compiler, Coccinelle, CoffeeScript, Dart, Haxe, TypeScript, and Emscripten

- **Inline code expander:** This model reads code as input and builds new file that uses the input code as a base but with sections of the code expanded, based on the design of the original one. It starts with designing a new language. Usually this new language is an existing language with some syntax extensions. The inline code expander is then used to turn this language into production code in a high-level language.

  Examples: Embedded SQL languages such as SQLJ (for Java) and Pro*C (for C). The SQL can be embedded in the C or Java code. The generator builds production C code by expanding the SQL into C code which implements the queries for example.

- **Mixed code generator:** This model has the same processing flow as the inline code expander, except that the input file is a real source file that can be compiled and ran. The generated output file keeps the original markup that will denote where the generated code is placed. It enables code generation for multiple small code

[7] https://en.wikipedia.org/wiki/Source-to-source_compiler

fragments within a single file or distributed throughout multiple files. Generally, transformation rules are defined using regular expressions.

Examples: Codify is a commercial mixed-code generator which can generate multiple code fragments in a single file from special commands. Another example is the replacement of comments in the input file by the corresponding code.

- **Partial class generator:** A partial class generator takes an abstract definition as input instead of code (*e.g.*, UML class diagram) and then builds the output code. User then can extend it by creating derived classes and adding methods to complete the design. Turning models into code is done through a sequence of transformations. For example, platform-independent model (PIM) is transformed into a platform specific model (PSM). Then code generation is performed from PSM by using some sort of template-based code transformations.

  Examples: ArgoUML and Codegen translate UML class diagrams to general-purpose languages such as C#, Java and C++. They do not generate complete implementations, but they try to convert the input UML class diagrams into skeleton code that the user can easily edit it. EMF GenCode is also a partial generator.

- **Tier generator:** In this model the generator builds a complete set of output code from an abstract definition. It has the same concept as partial class generator. The big difference between tier and partial class generation is that in the tier model the generator builds all the code for a tier. This code is meant to be used without extension. The partial class generator model however lets the engineer create the rest of the derived classes that will complete the functionality for the tier.

  Examples: Database access layer, web client layer, data export, import, or conversion layers.

- **Full-domain language:** Domain languages are basically new languages that have types, syntax and operations and they are used for a specific type of problem. Domain languages are the extreme end of automatic code generation because developers have to write a compiler for each problem domain and language.

  Examples: Matlab is a domain-specific math language used to represent mathematical operations, DSLs such as ThingML[8] and its code generators.

[8] http://thingml.org/

2.3.3 Why testing code generators is complex?

Testing code generators raise different challenges. In the following, we discuss some of them:

- **The oracle problem**:

  To automate the testing process, test oracles are required to assess whether a test has passed or not. A test oracle checks whether the result of executing a test is as expected. In case of functional testing of code generators, the test oracle can be easily defined. For example, it can be defined as the comparison result between the simulated or executed model and its corresponding implementation. However, in case of non-functional testing of code generators, the test oracle is complex to define. In fact, the generated code has to meet certain performance requirements (*e.g.*, execution speed, utilization of resources, etc.). Proving that the generated code respects one of these non-functional requirements is not obvious.

- **Infeasibility of unit testing**:

  It is infeasible to test a whole code generator exhaustively with traditional software test approaches due to the complexity of the tool. When it comes to the unit testing of code generators, each translation function would have to be detached from the software system and surrounded by a test harness. This means decoupling each translation rule and testing it separately. This is, however, infeasible because it is difficult to address this specific functionality separately when testing a system as a whole. Consider, for example, functional testing of the translation function for the sum operator (+). According to [BR04], there are more than 2000 ways of implementing the function a = b + c since the operation depends on data types and whether data limiting is enabled or not.

- **Complexity of code generators**:

  Code generators can be difficult to understand since they are typically composed of numerous elements, whose complex interdependencies pose important challenges for developers performing design, implementation, and maintenance tasks. Given the complexity and heterogeneity of the technologies involved in a code generator, developers who are trying to inspect and understand the code generation process have to deal with numerous different artifacts. As an example, in a code generator maintenance scenario, a developer might need to find all chained model-to-model and model-to-text transformation bindings, that originate a buggy line of code to

Table 2.2: Metrics of the TargetLink code generator

Metrics	**TargetLink 2.0**
No. of classes	3000
No. of files	6000
No. of functions	51 000
Lines (total)	1 800 000
Lines of code	990 000
Lines of comments	560 000

fix it. This task is error prone when done manually [GS15]. Table 2.2 shows, as an example, some metrics of the TargetLink code generator version 2.0. TargetLink[9] is a code generator that generates production code (C code) straight from the MATLAB/Simulink/Stateflow graphical models. This table shows how huge is the code generator base code. With more than 1 800 000 lines of code, it is very hard to test the whole system.

– **Non-executable source model**:

Code generators do not always support executable source models. Sometimes, code generators such as partial class generator, generate only structural code through a series of transformations from a non-executable model (*e.g.*, UML diagrams). It is up to the users next, to extend the generated code by implementing the derived classes. In case of non-executable models, it becomes challenging to automatically verify the correct behavior of produced code as it is described in the model specification because we can not compare its execution to a simulated model for example.

2.4 Compilers auto-tuning

Compilers have a major role in automatically producing fast and efficient target machine code. This is not a trivial task because potentially many variants of the machine code exist for the same program. Hence, the task of the compiler is to find and produce the best version of the machine code for any given program. For this reason, compilers generally attempt to automatically optimize the code to improve its performance. This process is called code optimization.

[9] https://www.dspace.com/en/inc/home/products/sw/pcgs/targetli.cfm

2.4.1 Code optimization

Code optimization within a compiler is the process of transforming a source code program into another functionally equivalent code for the purpose of improving one or more of its non-functional properties. The most common outcome of optimizations is the performance improvement. Other less common non-functional properties are code size, memory usage and power consumption. There exist many types of optimizations such as loop unrolling, automatic parallelization, code-block reordering, and functions inlining, among others. The hardware characteristics that influence on the optimization impact may include: the number of CPU registers (the more registers, the easier it is to optimize for performance), cache size, CPU architecture, etc.

Optimization can be categorized broadly into two types: machine independent and machine dependent:

- **Machine-independent optimization:**

  Intermediate code generation inside compilers may introduce many inefficiencies such as extra copies of variables and using variables instead of constants. This kind of optimization removes such inefficiencies and improves code. Thus, the compiler takes in the intermediate code and transforms a part of the code regardless of any CPU registers or memory locations. These optimizations generally change the structure of programs. Optimizations that are applied on abstract programming concepts (structures, loops, objects, functions) are independent of the machine targeted by the compiler.

  Examples: Eliminate redundancy, loop unrolling, eliminate useless and unreachable code, function inlining, dead-code elimination, etc.

- **Machine-dependent optimization:**

  Machine-dependent optimizations are applied after generating the target code and when the code is transformed according to the target machine architecture. They take advantage of special hardware features to produce code which is shorter or which executes more quickly on the machine such as instruction selection, register allocation, instruction scheduling, parallelism, etc. They mostly involve CPU registers and memory references. Machine-dependent optimizers put efforts to take maximum advantage of the memory hierarchy. They are more effective and have better impact on performance than independent optimizations because they best exploit special features of the target hardware.

Examples: Register allocation optimizations for efficient utilization of registers, branch prediction, loop optimization, etc.

2.4.2 Why compilers auto-tuning is complex?

Today, modern compilers implement a broad number of optimizations. Each optimization tries to improve the performance of the input application.

On the one hand, optimizing compilers becomes quite sophisticated nowadays. Creating compiler optimizations for a new microprocessor is a hard and time-consuming work because it requires a comprehensive understanding of the underlying hardware architecture as well as an efficient way to evaluate the optimization impact on the non-functional properties.

On the other hand and from the compiler user perspective, applying and evaluating optimizations is challenging because the determination of the optimal optimization set has been identified as a major research problem in the literature [KKO02].

We resume, in the following, several issues that make the activity of compiler auto-tuning very complex:

- **Conflicting objectives:** Compilers optimizations have to support a variety of conflicting objectives, such as execution time/compilation speed, execution time/resource usage, etc. It is difficult to define a set of optimizations that satisfies all properties.
- **Optimization interactions:** The interaction between optimization phases as well their application order make it difficult to find an optimal sequence.
- **Huge number of optimizations:** The huge number of optimizations is also an issue for the compiler user to choose the best optimization sequence since an exhaustive search is impossible (we count $2^{number\ of\ optimizations}$ possible combination to evaluate).
- **Non universal optimizations:** There is no universal optimization sequence that will enhance the performance of all programs. Optimization impact depends on the hardware and on the input program. Thus, constructing an optimization sequence for different programs and hardware architectures becomes very hard.

– **Compiler bugs:** Applying optimizations may introduce errors in the compiled code and results in compiler bugs [LAS14, YCER11]. Therefore, tuning compiler must not cause any change in the program behavior.

– **Optimization impact:** Optimized code should be fast and efficient. Optimizations have to improve some non-functional properties, not to induce performance regressions/overhead.

– **Tuning compilers need expertise:** In case the compiler user has no knowledge and expertise about the compiler technology and its optimizations, it will be quite hard to select the set of optimization sequences to apply.

2.5 Summary: challenges for testing and tuning configurable generators

We resume in this section the main testing and tuning challenges we have identified throughout this chapter:

- **Heterogeneous execution environments**: The diversity of existing software environments and platforms as well as the hardware heterogeneity make the testing activity of generators very difficult. Deploying and executing the automatically generated software artifacts on top of a bench of platforms is time consuming. Thus, an effective mean is needed to facilitate generators testing and tuning. **How can we leverage the new advances in software engineering technologies to face the continuous hardware and software innovation when testing/tuning generators?**

- **The oracle problem when testing code generators**: Automatic code generation offers many gains over traditional software development methods (*e.g.*, speed of development, productivity, etc.). However, code generators are complex pieces of software that may themselves contain bugs. Thus, they need to be rigorously tested. The test oracle problem as discussed in Section 2.3.3 is one of the main challenges related to the automatic non-functional testing of code generators. This problem occurs also in automatic functional testing, when dealing with non-executable models. **Proving that the generated code is functionally correct is not enough to claim the effectiveness of the code generator under test. How about the**

non-functional requirements such as resource consumption? How can we efficiently detect the non-functional issues?

- **Large optimization search space when auto-tuning compilers**: Compilers may have a huge number of potential optimization combinations, making it hard and time-consuming for software developers to find/construct the sequence of optimizations that satisfies user specific key objectives and criteria. It also requires a comprehensive understanding of the available optimizations of the compiler and their interactions. Moreover, it is difficult to find the optimization sequence that represents a trade-off between two conflicting objectives. **So, how can we help compiler users to automatically tune compilers and choose the optimization set that satisfies some specific non-functional requirements?**

- **Monitoring the resource usage of automatically generated code:** Analyzing the resource usage of optimized or generated code requires a dynamic and adaptive solution that efficiently extracts those properties. Due to the software diversity and hardware heterogeneity, monitoring the resource usage of each execution platform becomes challenging and time-consuming. **So, how can we ease this process and provide efficient support to help generator users/experts to evaluate the non-functional behavior of generated code in terms of resource usage?**

Chapter 3

State of the art

In this chapter, we present the state of the art of this thesis. We discuss previous efforts in three research areas: (1) code generator testing, (2) compiler auto-tuning, and (3) lightweight virtualization for software testing and monitoring.

We first discuss existing techniques related to code generators testing. We start by studying the state of the art approaches related to the automatic functional testing of code generators. In a second stage, since there are few research efforts that investigate the automatic non-functional testing of code generators, we rather focus on studying the oracle problem in software testing and the different methods applied to alleviate this problem. We end this section by providing a summary table of these approaches.

Afterwards, we move to present a brief introduction to the iterative compilation research filed and we identify several problems that have been investigated in this field. We discuss as well several techniques applied to the compiler auto-tuning process. To sum up, we provide a summary table, showing the most important research efforts in iterative compilation.

Finally, we discuss in the last section the system-level virtualization technology as means of automatic software deployment, monitoring, and testing. We provide then, examples of existing solutions in academia and industry that opted for this technology to automate software testing and monitoring.

This chapter is structured as follows:

Section 3.1 reviews existing techniques for code generators testing and the principal categories of oracles.

In Section 3.2, we provide a survey of the most used compiler auto-tuning techniques to construct the best set of optimization options.

Then, in Section 3.3, we discuss the use of containers as a lightweight execution environment. In particular, we present several research efforts that used this solution for software testing and monitoring.

Finally, Section 3.4 provides a summary of the state of the art and we discuss the challenges we are addressing in this thesis.

3.1 Testing code generators

Testing the manually written code has always been a crucial task to ensure that the code is correct. It aims to prove that the code is functionally correct using techniques such as unit testing, integration testing, acceptance testing, etc. These techniques help to find errors that engineers make when developing code. When a code generator is used, adequate testing approaches are needed to detect errors caused by the automatic code generation. Verifying that the code generator is correct will increase the confidence in the tool and users will continue to use it for production code generation.

The key objective of this section is to present the existing research efforts that have been presented to address:

- **The problem of automatic code generator testing:** We provide an overview of the approaches that aimed to automatically test code generators in terms of functional properties.
- **The problem of automatic non-functional testing of code generators** Automatic code generator testing poses different challenges, especially for the test oracle definition. Therefore, we provide an overview of the commonly known test oracle definition approaches in software testing.

3.1.1 Functional testing of code generators

Most of the previous work on code generator testing focuses on checking the correct functional behavior of generated code [SCDP07, ZSP+06, Con09, CMKSP10, JS14, BR04, SC03].

In the case of automatic code generation against **executable models**, various approaches have been proposed to automatically verify the model-to-code translation. Verification purpose is to check that the generated code correctly implements the designed model. Thus, the model is tested against its requirements and the code can be verified

against the executable model by means of dynamic testing. For this purpose, both the model and the generated code are executed to be later exploited. This approach is presented and discussed in several research efforts [SWC05, SCDP07, CMKSP10, JS14, BR04]. Authors of these papers argue that this approach is not only applicable for model-based code generators but also for all kinds of code generators, unless the input model/source code is executable.

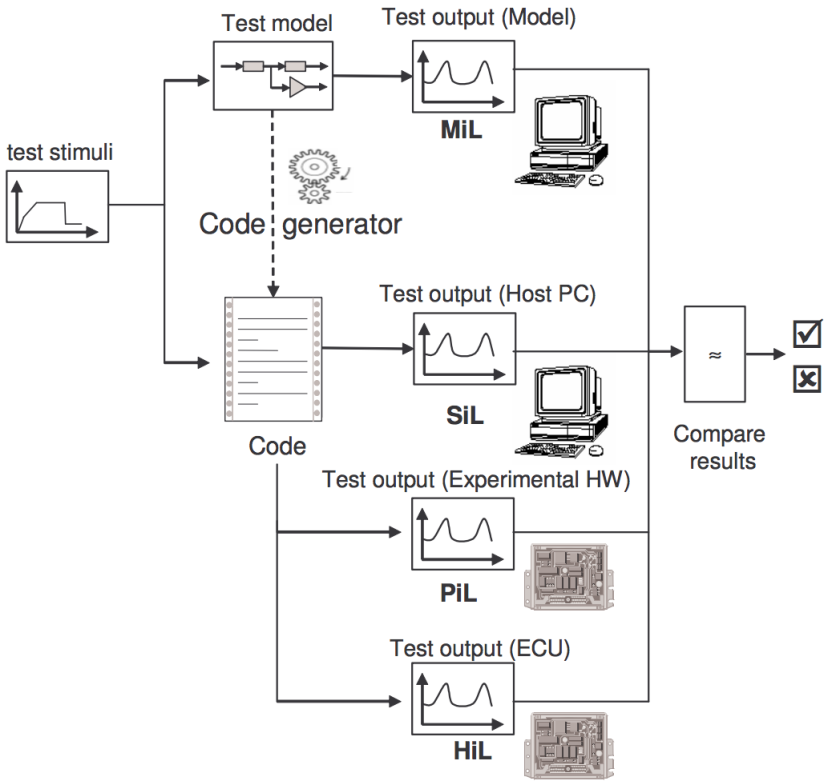


Figure 3.1: Automatic functional testing of code generators

As it is shown in Figure 3.1, both the generated executable and the simulated model are executed with the same input. The determination of the input test stimuli can use a structural testing criteria on model level (model coverage) and code level (code coverage) to generate high-quality test vectors. Afterwards, the two outputs are compared with respect to certain acceptance criteria. The comparison procedure is known in the software testing community as equivalence, comparative, or back-to-back testing approach [Vou90, McK98].

The great advantage of this approach is that the test oracle is simple to define. It rep-

resents the comparison between two or more output results. According to [SH09,SCDP07], there are four stages of comparison that can be performed. They are described as follows (see Figure 3.1):

- Model-in-the-Loop (MiL): The simulation of the model on the host machine is termed MiL. The MiL test purpose is to generate a reference test results (expected values). Moreover, MiL simulation captures the specified behavior of the model that is to be implemented in general-purpose language later on. It also checks the validity of the model with respect to the functional requirements. The only problem that could occur during a MiL execution is that the model would fail to execute.

- Software-in-the-Loop (SiL): The execution of the generated object code on the host machine with the same stimuli used for the MiL is termed as SiL. The execution results should be comparable to the results obtained during MiL. The aim of SiL is to detect translation errors such as arithmetical problems, and to measure code coverage. Once the system detects a defect, the testing environment should provide the tester with a suitable navigation tool to jump to the erroneous data variable in order to fix it.

- Processor-in-the-Loop (PiL): PiL tests the object code on the target processor. It generates the cross compiled source code and executes it on the target processor machine. Of course, compiler optimizations can be applied to enhance the code quality. Then, the test scenario is executed on the target processor (*e.g.*, target embedded systems as in [SH09]). The aim of PiL is to verify the code behavior on the target processor and to measure code efficiency (*e.g.*, profiling, memory usage, etc.).

- Hardware-in-the-Loop (HiL): Finally, during HiL, the software embedded into the target chip is executed. For that purpose, the target hardware is connected to a real-time simulation system simulating the plant. The model originally developed no longer simulates the physical environment signals, a dedicated hardware is specially designed for this purpose. The aim of HiL is to check the correct software behavior on real hardware.

In this context, Conrad *et al.* [CMKSP10] applied the approach described above by presenting an automated testing approach to assess the equivalence between Simulink models and the generated code. This approach is called Code Generation Verification (CGV). CGV assesses the numerical equivalence between the model used (*i.e.*, Simulink models)

and the generated code (*i.e.*, the executables derived from the generated C Code). In fact, each individual model-to-code translation is followed by a verification phase to assess that the input Simulink model used for code generation and the output (*i.e.*, the object code derived from the model via code generation and compilation) produce the same numerical results when stimulated with identical inputs. In their equivalence testing approach, they use to run the model used for code generation using simulation and the generated code with the same input stimuli (test vectors) followed by a numerical comparison of the outputs (result vectors). Then, they check whether or not the semantics of the model have been preserved during code generation, compilation, and linking, by comparing the result vectors, which are the outputs resulting from stimulation with identical test vectors of the model and the generated code. More precisely, the simulation results should be similar to the execution results. However, when defining the result vector comparison, they tolerate limited differences between both results. They argue that some factors between simulation and execution may cause a small difference between both executions such as limited precision of floating point numbers, target optimized code constructs, etc. Thus, they define an application-dependent threshold. So, two result vectors are considered sufficiently similar when their difference is less than a specific threshold value. They illustrate the CGV-based translation validation in the context of embedded automative software by using Simulink and Real-Time Workshop Embedded Coder for verification. They assess their approach by verifying the numerical equivalence between Simulink models and C generated code. They calculate the absolute difference between simulation results and execution results. Then, they compare this difference to the defined tolerance threshold. They show that for some input test suites there exist mismatched signals (with high variation value) which represent an inconsistency between designed models and executed signal.

In [Con09], the automatic code generation tool is certified to a particular safety standard (IEC 61508-3). Compliance of the model with a standard helps to demonstrate that the model is well-formed according to the certification and that it meets all requirements for later code generation. The process of code generator evaluation (described above) is used to show that the generated code is equivalent to the model (that respect the IEC 61508-3 safety standard)

Stuermer *et al.* [SCDP07] present a systematic test approach for model-based code generators. They investigate the impact of optimization rules for model-based code generation by comparing the output of code and model executions. If these outputs are equivalent, it is assumed that the code generator works as expected. They evaluate the effectiveness of this approach by means of testing optimizations performed by the TargetLink code generator. Test vectors are generated using a structural coverage of the model and generated code. They have used Simulink as a simulation environment of models.

In [JS14], authors present a testing approach of the Genesys code generator framework which tests the translation performed by a code generator from a semantic perspective rather than just checking for syntactic correctness of the generation result. Basically, Genesys realizes back-to-back testing by executing both the source model as well as the generated code on top of different target platforms. Both executions produce traces and execution footprints are then compared with each other.

Another alternative for validating a code generator would be to use formal proofs [BDF08, Ler09]. This involves mathematically proving that the code generation transformation process is correct and that each code generation rule preserves the model semantic. Denney *et al.* [DF05] extend an existing code generator such that it produces all logical annotations that are required for formal safety proofs. These proofs certify that the program does not violate certain conditions during its execution. This approach is integrated into the AUTOBAYES and AUTOFILTER code generators. They used it to prove that the generated code satisfies both language-specific properties such as array-bounds safety or proper variable initialization and domain-specific properties such as vector normalization, matrix symmetry, or correct sensor input usage.

3.1.1.1 Summary of functional testing approaches

Inspired by the work of Sturmer *et al.* [SWC05], we provide a summary of the existing techniques that are applied to test the automatic code generation, some of them are described above:

Table 3.1: Summary of several approaches applied for testing code generators [SWC05]

Level	Testing technique	Objectives
Model	Functional MiL simulation/testing	–Verify that the model reflects its functional requirement specification –Check validity of the model within the development environment without resource limitations of target environment
	Structural MiL testing (model coverage)	–Explore possible pathways within the model by determining test cases on the basis of the model structure
	Adoption of modeling guidelines	–Rely on experiences and expert knowledge –Reveal design errors at an early development stage
Code generator	Tool certification	–Independent approval which guarantees that techniques, applied for developing and verifying the tool, are in compliance with the requirements of a certification standard
	Testing	–Ensure that the code generator has been tested rigorously –Validate that specific translation functions (*e.g.*, optimisations) behave as expected
	Formal proof	–Show by means of mathematical proofs that each code generation (rule) preserves the model semantics
	Adoption of standards and guidelines	–Ensure that the code generator has been developed following a systematic development process / quality management system
Generated code	Functional SiL testing	–Detect translation errors
	Functional PiL testing	–Check validity of the code behavior taking into account the target CPU architecture
	Functional HiL testing	–Check behavior of the code when it is deployed in the target hardware device
	Structural MiL/HiL/PiL testing	–Determine test cases on the basis of the code structure and explore possible execution pathways
	Static analysis	–Check that code conforms to coding guidelines/standards –Detect optimization opportunities such as dead code, etc.

3.1.2 Non-functional testing of code generators

Previous work on non-functional testing of code generators focuses on comparing, as oracle, the non-functional properties of hand-written code to automatically generated code [SP15, RFBJ13]. As an example, Strekelj *et al.* [ŠLG15] implemented a simple 2D game in both the Haxe programming language (a high-level language) and the target programming language, and evaluated the difference in performance between the two versions of code. They showed that the generated code through Haxe has better performance than the hand-written one.

In [Ajw07], authors compare some non-functional properties of two code generators, the TargetLink code generator and the Real-Time Workshop Embedded Coder. They also compare these properties to manually written code. The code run on a C166 microprocessor as a target which is an embedded system. The metrics used for comparison are ROM and RAM memory usage, execution speed, readability, and traceability. Many test cases are executed to see if the controller behaves as expected. The comparison results show that the generated code by TargetLink is more efficient than the manually-written code and the other generated code in terms of memory and execution time. They also show that the generated code can be easily traced and edited.

Cross-platform mobile development has been also part of the non-functional testing goals since many code generators are increasingly used in industry for automatic cross-platform development. In [PV15, HSD11], authors compared the performance of a set of cross-platform code generators and presented the most efficient tools.

3.1.2.1 The oracle problem

One of the most important aspects we are interesting in while testing code generators is the *test oracle*. It is the mechanism that verifies whether the outputs of the program for the executed test cases are correct or not.

Compared to many aspects of test automation, the problem of automating the test oracle is still challenging and less well solved. Only few techniques are available to generate test oracles. In most of the cases, designing and implementing test oracles are still manual and expensive activities. That is because the test oracles are not always available and may be hard to define or too difficult to apply [BM+15a]. This is commonly known as the *"oracle problem"*. As pointed out in [MK01], the oracle problem has been *"one of the most difficult tasks in software testing"* but it is often ignored by researchers.

In this context, the automatic testing of code generators particularly implies the oracle problem. When testing compilers, for example, it is quite difficult to automatically verify

the equivalence between the source code and object code. This task becomes even more complicated when some optimizations are applied to the machine code. When testing code generators, the verification of the equivalence between the high-level system specification and the generated code is also challenging.

As we discussed in Section 3.1.1 about the automatic functional testing of code generators, the test oracle is often defined as a back-to-back comparison between the output of the system specification and the generated code. When it comes to test the non-functional properties such as the resource usage or execution speed, this problem becomes more critical. There is no clear definition about how the oracle should be defined except the few research efforts, discussed in Section 3.1.2.

The research community has proposed several approaches [HMSY13, BM+15a] to alleviate the oracle problem. In a recent survey, Harman *et al.* [HMSY13] classify test oracles in three categories *specified oracles*, *implicit oracles*, and *derived oracles*. We give an overview of these three categories as they have described:

- Specified oracles:

  Specified oracles can be generated from several kinds of specifications, such as algebraic specifications or formal models of the system behavior. For example, Stocks *et al.* [SC96, RAO92] discuss an approach for deriving test cases and oracles from specification. The idea is that the formal specification of a software can be used as a guide for designing functional tests. Then, test oracles can be associated with individual test templates (test case specifications). Thus, they construct abstract models of expected outputs, called oracle templates. The approach is illustrated with test oracle templates for the Z specification.

  Specified oracles are effective in identifying errors. However, the task of defining and maintaining specifications is very expensive and time consuming. The applicability of specified oracles is therefore limited and they are also less adopted in industry.

- Implicit oracles:

  Implicit oracle refers to the detection of obvious faults such as a program crash, abnormal termination, or execution failure. Thus, oracle definition does not require any domain knowledge or formal specification to implement, and as a consequence, it does not need any prerequisites about the behavior or semantics of the program under test.

  Implicit oracles [HMSY13, BM+15a] are easy to obtain at practically no cost. At the same time, implicit oracles are mostly incomplete, since they are not able to identify

internal problems and complex failures, but they help to detect, in a black-box way, general errors like system crashes or un-handled exceptions.

As an example, fuzz testing [MFS90] is one of the methods where implicit oracles are used to find anomalies, such as crashes. The idea of fuzzing is to generate random inputs and pass them to the system under test to find anomalies. Bugs detection is based on the efficiency of generated inputs/data. If an anomaly is detected, the tester reports it by identifying the input triggering it. Fuzzing is commonly used to detect security vulnerabilities, such as buffer overflows, memory leaks, exceptions, etc. [BBGM11].

Kropp *et al.* [KKS98] present an approach to test the robustness of the system under test using implicit oracles. This approach relies on the creation and execution of invalid input robustness tests. Specifically, these tests are designed to detect crashes and hangs caused by invalid inputs to function calls. The results show that between 42% and 63% of components on the POSIX systems measured had robustness problems.

Ricca and Tonella [RT06] focus on developing patterns to detect anomalies. They consider a subset of possible anomalies that can be found in web applications such as navigation problems, hyperlink inconsistencies, etc. Their empirical results show that 60% of the web applications considered in their study exhibit anomalies and execution failures.

- Derived oracles:

  Derived oracles are derived from various artifacts (*e.g.*, documentation, system executions) or properties other than specifications.

  For example, in regression testing, oracles can be derived from the executions of previous versions of the software under test. In this case, the derived oracles will verify if the new software version behaves as the original one [MPP07]. For example, EvoSuite and Randoop derive test oracles from previous versions of the system under test.

  Oracles can also be automatically derived from program invariants [ECGN00]. Invariants can help programmers characterizing aspects of program execution and identifying program properties that must be preserved when modifying code. They report properties that were true over the observed executions of all programs such as *"y = 2*x+3"*, *"array a is sorted"*, etc. The Daikon invariant detector[1] is an open source tool that applies machine learning techniques to infer these invariant properties.

[1] https://plse.cs.washington.edu/daikon/

Another type of derived oracles are pseudo-oracles (also known as differential testing, dual coding, and N-version programming [PCC+16]). The concept of a pseudo-oracle is introduced by Davis and Weyuker [DW81]. A pseudo-oracle is program that is capable of providing the expected outputs and check the correctness of the system by comparing the outputs of multiple independent implementations. It checks the consistency of the results of the different software versions of the systems, when the same functionality is executed. An inconsistency can be detected when one or more versions of the system trigger failures. For example, in compiler testing, different versions of the same program are generated by applying some optimizations. The functionality of the program under test remains the same for all versions. The oracle is defined, in this case, as a comparison between the functional outputs of the different versions [YCER11].

Additionally, oracles can be derived from properties of the system under test. For instance, a metamorphic testing (MT) method has been proposed to alleviate the oracle problem [CHTZ04]. MT is an automated testing method that employs expected properties of the target functions to test programs without human implication. MT exploits the relation between the inputs and outputs of special test cases of the system under test to derive metamorphic relations (MRs) defined as test oracles for new test cases. MT recommends that, given one or more test cases (called source or original test cases) and their expected outcomes, one or more follow-up test cases can be constructed to verify the necessary properties (*i.e.*, metamorphic relations) of the system or function to be implemented. For a given problem, usually more than one MR can be identified. It is therefore important to select suitable MRs for effective bugs detection.

MT was recently applied for compiler testing. Le *et al.* [LAS14] present an approach called equivalence modulo inputs (EMI) testing. The idea of this approach is to pass different program versions (with same behavior) to the compiler in order to inspect the output similarity after code compilation and execution. So, given a deterministic program P and a set of input values I, the authors propose to create equivalent versions of the program by profiling its execution and pruning un-executed code (by identifying the statements not covered by I and mutating or deleting a subset of the dead statements of P). Once a program and its equivalent variant are constructed, both are used as input to the compiler under test and then, inconsistencies in their results are checked. Inconsistencies represent, in this case, deviations in the functional behavior. This method has detected 147 confirmed bugs in two open source C compilers, GCC and LLVM. EMI testing is an example of metamorphic testing. In fact, the program variants are in a metamorphic relationship with one another and with P,

with respect to I. Another application of the equivalence-based method is presented by Tao *et al.* [TWZS10] to test the semantic-soundness property of compilers. They use three different techniques in generating equivalent source code programs and then test the mutants with the original programs, such as replacing an expression with an equivalent one. Empirical results show that their approach is able to detect real issues in GCC and ICC compilers. A metamorphic approach has also been used to test GLSL compilers via opaque value injection [DL16].

Pseudo-oracles and metamorphic oracles have similar concept. Pseudo-oracles need different implementations of the same program specification while in metamorphic testing, follow-up test cases (program variants) must be derived from original program under test through program transformations.

3.1.2.2 Summary: oracle definition approaches

We provide in Table 3.2 a summary of the several oracle definition techniques described above:

Table 3.2: Summary of test oracle approaches

Oracle	Method	Objectives
Specified oracles	–Assertions and contracts –Specification-based languages –Algebraic specification languages	–Use of notions of specifications as a source of oracle information.
Implicit oracles	–Fuzz testing –Load testing –Robustness checking	–Identify obvious faults such as crashes, memory leaks, un-handled exceptions, abnormal program termination, etc.
Derived oracles	–Metamorphic testing –N-version programming –Regression testing –Back-to-back testing –Invariant detection	–Oracles are derived from various artifacts (*e.g.*, documentation, system executions) or properties (*e.g.*, metamorphic relations) of the system under test. – Check the consistency of the results of the different versions of the systems, when the same functionality is executed.

3.2 Compilers auto-tuning techniques

3.2.1 Iterative compilation

Iterative compilation, *also known as optimization phase selection, adaptive compilation, or feedback directed optimization* [TVVA03], consists in applying software engineering techniques to produce better and more optimized programs. The key objective of iterative compilation is to find the best optimization sequence that leads to the fastest and highest-quality machine code.

The basic idea of iterative compilation is to explore the compiler optimization space by measuring the impact of optimizations on software performance. Several research efforts have investigated this optimization problem, such as Search-Based Software Engineering (SBSE) techniques, to guide the search towards relevant optimizations regrading performance, energy consumption, code size, compilation time, etc. Experimental results have been usually compared to standard compiler optimization levels.

It has been proven that optimizations are highly dependent on the target platform and on the input program which makes the task of searching for the best optimization sequence very complex [TVVA03].

In the following sections, we describe the classical iterative compilation process and we discuss the relevant techniques and approaches that have been presented to tackle some of the challenges related to compiler optimization we have identified in the previous chapter, namely:

- **The problem of optimization-space exploration:** We present several approaches that have addressed the combinatorial explosion problem of possible compiler optimizations.
- **The problem of multi-objective optimization:** We present several techniques that aimed to find trade-offs between multiple non-functional properties.

3.2.2 Implementation of the iterative compilation process

The implementation of an iterative compilation system consists mainly on applying a sequence of steps to enhance the quality of the generated code. Figure 3.2 shows a general overview of the main steps needed to ensure the implementation of the iterative compilation process.

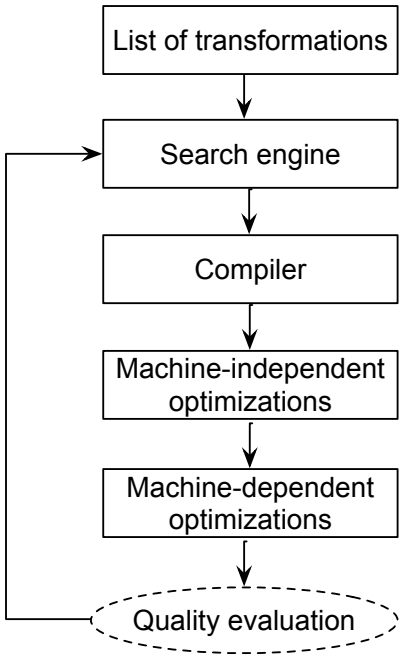


Figure 3.2: Overview of the iterative compilation process

– **List of transformations:**

The iterative process starts by defining the optimization space. It represents the list of optimizations that the compiler has to evolve during the search for the best optimization sequences.

– **Search engine:**

It applies a search algorithm or method to efficiently explore the large optimization search space. In fact, it takes as input the previously defined list of transformations and decides which optimizations will be retained at the end of the search.

– **Compiler:**

Once the optimization sequence is defined, the target compiler (*e.g.*, GCC, LLVM) is called to compile the input program and also perform initial machine independent optimizations.

– **Machine-independent optimizations:**

This results in an initial machine independent optimized program. These optimizations are performed during code generation and impact all target systems. It includes optimizations that are applied during the parse tree mapping to the intermediate code and the optimization applied to the intermediate code itself.

– **Machine-dependent optimizations:**

For further optimization, the compiler applies from the provided optimization sequence the machine dependent optimizations. This includes optimizations applied during the mapping of intermediate code to assembler and optimizations applied directly on the generated object code.

– **Quality evaluation:**

It consists on evaluating the quality of the optimized code. Many non-functional properties can be evaluated like code size, execution time, resource usage, power consumption, etc.

This model represents the classical and typical iterative compilation process. Of course, there exist many ways and adaptations to implement this process. The implementation of the iterative process depends on the algorithm used, the problem addressed, the technologies used, etc. Our contribution regarding compiler auto-tuning respects this model. The goal of the next subsections is to present the different state of the art approaches related to iterative compilation.

3.2.3 Iterative compilation search techniques

In Section 2.4.2 of Chapter 2, we presented several issues with optimizing compilers that make the activity of compiler auto-tuning very complex such as the huge number of optimizations, conflicting objectives, optimization impact, etc. In this section, we discuss the available tools and approaches dedicated to the automatic search for optimal compiler settings, and give an overview of known approaches that addressed these several compiler optimization challenges. In each subsection, we identify and discuss a particular problem and we present the best known approaches proposed to solve it.

3.2.3.1 Auto-tuning: a mono objective optimization

The compiler auto-tuning technique has been used in many optimization scenarios. What all of this prior work on iterative compilation have in common is that it focuses on a single

objective function to be optimized. For example, researchers typically focus on speeding up the performance of compiled code which constitutes the major optimization objective for most iterative compilation approaches [PE06, HKW05, ACG+04, FOK02, CHE+10, CSS99].

So, the problem has been often adapted as a mono-objective search problem where the speedup is the main concern for most of the previous work. Genetic algorithms (GA) [SOMA03, BY07] present an attractive solution to this problem. GA-based approaches compute an initial population using a set of optimizations, generally defined under the standard compiler levels -Ox. Then, at each iteration, the individuals (*i.e.*, option sets) that comprise the generation are evaluated by measuring the execution time of each solution. The results are sorted and pass through a breeding and mutation stage to form the next generation. This process continues until a termination condition is reached. At the end, the algorithm returns the best optimization set that led to the highest performance.

For example, ACOVEA[2] (Analysis of Compiler Options via Evolutionary Algorithm), is an open source tool that applies GAs to find the best options for compiling programs with the GCC compiler. In this context, best solutions define those options that produce the fastest executable program from a given source code. This tool was even included in the Gentoo Linux repository to help users to find the best set of optimizations.

The ESTO framework [BY07] studies the application of GA to the problem of selecting an optimal option set for a specific application and workload. ESTO regards the compiler as a black box, specified by its external-visible optimization options. ESTO supports a GA variant named budget-limited genetic algorithm which reduces the population size exponentially and then reduce the time needed to evaluate the different evaluations. The authors ran experiments on the SPEC2000 benchmark suite and tested 60 optimization options within three compilers: GCC, XLC and FDPR-Pro. Results of ESTO are compared to GCC -O1 and -O3, to XLC -O3, and to FDPR-Pro -O3. The results show that ESTO is capable to construct optimization levels that yield to better performance than standard options.

3.2.3.2 Escaping local optimum

A common problem in iterative compilation is the local optimum. In fact, the search space of optimizations for a specific program could be very huge and it generally contains many local minima in where the search algorithm could be trapped [BKK+98]. Therefore, researchers in this field try to build robust techniques and algorithms to avoid such problem. In [BKK+98], Bodin *et al.* tried to analyze this search space and they found

[2] https://github.com/Acovea/libacovea

that the optimization space is highly non-linear containing many local minima and some discontinuities. This paper focuses on parameterized transformations. The small area of the transformation space considered in this paper is composed of three parameterized optimizations: loop unrolling (with unroll factors from 1 to 20), loop tiling (with tile sizes from 1 to 100) and padding (from 1 to 100). They focus on reducing the compilation and execution time of optimized programs and use a simulator to target embedded processors. They use a compiler framework developed to optimize multimedia programs for embedded systems. They analyze these optimizations across four CPU architectures (UltraSparc, R10000, Pentium Pro, and TriMedia-1000) and the matrix multiplication is selected as the program to optimize. The proposed search algorithm visits a number of points at spaced intervals, applying the appropriate transformation, executing the transformed program and evaluating its worth by measuring the execution time. Those points lying between the current global minimum and the average are added to an ordered queue. Iteratively, such points are removed from the queue and points within the neighboring region are investigated, again at spaced intervals. This process is continued until a specific number of points are evaluated and the fastest transformed program is reported. They show that in the case of large transformation spaces, they can achieve within 0.3% of the best possible time by visiting, less then 0.25% of the space. They find the minimum after visiting up to less than 1% of the space.

Cooper *et al.* [CGH+06] describe their experience exploring the search space of compilation sequences. They report the results of exhaustively enumerating several search spaces of sequences of length 10 chosen from 5 transformations. They show that the search space has many local minima, and that random-restart hill climbing is an effective strategy to overcome this problem.

Another way to efficiently explore the large search space in compiler optimization is the Design Space Exploration (DSE) technique [MND+14, MNC+16]. The DSE is based on a clustering approach for grouping functions with similarities and exploration of a reduced search space resulting from the combination of optimizations previously suggested for the functions in each group. The identification of similarities between functions uses a data mining method that is applied to a symbolic code representation. They compare their approach to the GA and their experimental results reveal that the DSE-based approach achieves a significant reduction in the total exploration time of the search space (20x over a GA approach) and a performance speedup (41% over the baseline).

3.2.3.3 Phase ordering problem

Phase ordering is also an important problem in iterative compilation which explores the effect of different orderings of optimization phases on program performance. In fact, when using some compilers such as LLVM, it is important to define the right order of applying optimizations. Thus, researchers in this field try to apply search techniques in order to find the right optimization sequence. However, reordering optimization phases is extremely hard to support in most production systems, including GCC due to their use of multiple intermediate formats and complex inherent dependencies between optimizations. So generally, compilers manage internally the order of applying optimizations and do not give the hand to the user to choose this order, avoiding conflicts and compilation issues.

When the order is managed by the users, exhaustively evaluating all orderings of optimization phases is infeasible due to the huge number optimization phases. This problem becomes more complex by the fact that these phases interact with other optimizations in a complex way. For example, even if we keep the same set of optimizations for an input program, varying the order of applying these optimization phases can produce different code, with potentially significant performance variations amongst them.

In this field, Whitfield [WS90] developed a framework based on axiomatic specifications of optimizations, including both *pre* and *post* conditions before and after applying optimizations. For a selected set of optimizations, the framework is used to determine those interactions among the optimizations that can create conditions and those that can destroy conditions for applying other optimizations. Then, from these interactions, an application order is derived to obtain the potential benefits of the optimizations that can be applied to a program. This framework was employed to list the potential enabling and disabling interactions between optimizations, which were then used to derive an application order for the optimizations

Kulkarni *et al.* [KWTD09,KWTD06] proposed an exhaustive search strategy to find optimal compilation sequences for each function of a program. They exhaustively enumerated all distinct function instances for a set of programs that would be produced from different phase-orderings of 15 optimizations. This exhaustive enumeration was made possible by detecting which phases were active and whether or not the generated code was unique, making the actual optimization phase order space much smaller than the attempted space. This exhaustive enumeration allowed them to construct probabilities of enabling/disabling interactions between different optimization passes in general and not specific to any program. They use this idea to prevent the combinatorial explosion of the total number of sequences to be tested. They were able to find all possible function instances that can be produced by different phase orderings for 109 out of the 111 total functions they evaluated.

Cooper *et al.* [CSS99] adapts the GA to solve the optimization phase ordering problem. They target embedded systems and focus on reducing the code size. They choose 10 program transformations to evolve in Fortran compiler. The solutions generated by their algorithm are compared to solutions found using a fixed optimization sequence. Their technique was successful for reducing the code size by 40% compared to the standard sequence.

In another work [CGH+06], the same authors explored phase orders at program-level with randomized search algorithms based on GAs, hill climbers, and randomized sampling. They target a simulated abstract RISC-based processor with a research compiler. They report properties of several generated sub-spaces of phase ordering and the consequences of those properties for the search algorithms.

3.2.3.4 Evaluating iterative optimization across multiple data sets

Most iterative optimization studies find the best compiler optimizations through repeated runs on the same data set. The problem is that if we select the best optimization sequence for an input data set through the iterative process, we do not know if it will be the best for the same program but with another data sets. Thereby, researchers in this field try to investigate this problem by evaluating the effectiveness of iterative optimization across a large number of data sets. In particular, since there is no existing benchmark suite with a large number of data sets, Chen *et al.* [CHE+10] attempt to collect 1000 data sets called KDataSets for 32 programs, mostly derived from the MiBench benchmark. Then, they exercise iterative optimization on these collected data sets in order to find the best optimization combination across all data sets. They use random search to generate random optimization sequences for the ICC compiler (53 flags) and the GCC compiler (132 optimizations). They demonstrate that for all 32 programs (from MiBench), they were able to find at least one combination of compiler optimizations that achieves 86% or more of the best possible speedup across all data sets using ICC (83% for GNU's GCC). This optimal combination is program-specific and yields speedups up to 1.71 on ICC and 2.23 on GCC over the highest optimization level (-Ofast and -O3, respectively). This means that a program can be optimized on a collection of data sets and it can retain near optimal performance for most other data sets. However, they tested their approach on only one single benchmark and one target architecture.

3.2.3.5 Conflicting objectives: a multi-objective optimization

Several research efforts attempt to find trade-offs between two (or more) non-functional properties [ACG+04, HE08, PE06, PHB15, CFH+12, MND+14, LCL08, MÁCZCA+14].

In COLE [HE08], the authors considered that the problem of compiler optimizations can be seen as a multi-objective problem where two non-functional properties can be enhanced simultaneously. Thus, they investigated the standard levels of compiler optimization by searching for pareto optimal levels that maximize both performance and compile time. They show that by using the multi-objective genetic algorithm (in their experiment they used SPEA2), it is possible to find a set of compiler optimization sequences that are more pareto-effective in terms of performance and compile time than the standard optimization levels (-O1, -O2, -O3, and -Os). The motivation behind this approach is that these standard levels were set up manually by compiler creators based on fixed benchmarks and data sets. For the authors, these universal levels may not be always effective on unseen programs and there exist higher levels that provide better trade-offs in terms of code quality. They used the SPEC2000 CPU benchmark, which is a popular benchmark suite for evaluating the compiler performance. They evolved 60 optimization flags that are defined in the standard levels -O1, -O2, -O3, and -Os. They run iterative compilation on one single machine shipped with Intel CPU Pentium 4 and they compared the proposed algorithm (SPEA2) to random search as well as to standard optimization levels. The experimental results using GCC (v4.1.2) show that the automatic construction of optimization levels is feasible in practice, and in addition, yields better optimization levels than GCC's manually derived optimization levels. However, they do not provide a guarantee that the new explored optimization levels will be optimal for other applications.

Martinez *et al.* [MÁCZCA+14] propose an adaptive worst-case execution time aware compiler framework for automatic search of compiler optimization sequences. Compared to the previously described approaches, authors in this work focus on generating efficient code for embedded systems. Therefore, they focus on crucial properties for real-time systems such as average-case execution time (ACET), worst-case execution time (WCET), power consumption and code size. They explore the performance of compiler optimizations with conflicting goals. Thus, they try to find suitable trade-offs between these objectives in order to identify pareto optimal solutions using multi-objective algorithms. The objective functions try to minimize the WCET-ACET and WCET-Code size properties. They apply three evolutionary multi-objective algorithms namely IBEA, NSGA-II and SPEA2 and compare their results to standard levels (-O1, -O2 and -O3). They evolve 30 optimizations within the WCC compiler and perform experiments on top of one single machine shipped with Intel Quad-Core CPU processor. They pick up 35 programs from various bench-

marks such as DSPstone, MediaBench, MiBench, etc. They find that NSGA-II is the most promising algorithm for the given problem. The discovered optimization sequences significantly outperform standard optimization levels. In fact, the highest standard optimization level -O3 can be outperformed for the WCET and ACET on average by up to 31.33% and 27.43%, respectively. The same approach performs as well for the WCET-Code size optimization with a 30.6% WCET reduction over -O3. However, the code size increases by 133.4%. They argue that the WCET and the code size are typical conflicting goals. If a high improvement of one objective function is desired, a significant degradation of the other objective must be accepted.

In [PMV+13], the TACT framework is presented. Compared to previous approaches, TACT is designed primarily for automatic tuning on embedded systems running Linux. Thus, the target CPU architecture for this tool is the ARM architecture (ARM Cortex-A9) and 200 options are used in the GCC compiler for ARM. TACT supports multiple optimization objectives, so it can tune either for a single optimization parameter, or for two objective functions simultaneously, for example, for performance and code size (or compile time). So, it applies the SPEA2 algorithm and GA for mono-objective optimizations. The results show how the SPEA2 outperforms the standard GCC levels (-O2, -O3 and -Os) across several open-source popular applications such as C-Ray, Crafty Chess, SciMark, x264 and zlib.

3.2.3.6 Predicting optimizations: a machine learning optimization

Machine learning has been also proposed by several research efforts to tune compilers. Compared to evolutionary algorithms, using machine learning in compiler optimization has the potential of reusing knowledge across the different iterative compilation runs, gaining the benefits of iterative compilation to learn the best optimizations across multiple programs and architectures.

Generally, machine learning applications create in an offline phase a prediction model, which will be used to determine the compiler optimization set that should be applied to unseen programs in the online phase. The main advantage of this technique is to reduce the number of required program evaluations.

In the Milepost project [FKM+11] for example, authors start from the observation that similar programs may exhibit similar behavior and require similar optimizations, so it is possible to correlate program features and optimizations together to predict good transformations for unseen programs, based on previous optimization experience. Thereby, they provide a modular, extensible, self-tuning optimization infrastructure that can automati-

cally learn how to best optimize programs for configurable heterogeneous processors based on the correlation between program features, run-time behavior and optimizations. The proposed infrastructure is based on a machine learning compiler that presents an Interactive Compilation Interface (ICI) and plugins to extract program features (such as the number of instructions in a method, number of branches, etc) and select optimization passes.

The Milepost framework proceeds in two distinct phases: training and deployment. During the training phase, information about the structure of programs (input training programs) is gathered, showing how they behave under different optimization settings. Such information allows machine learning tools to correlate aspects of program structure, or features, with optimizations, building a strategy that predicts good combinations of optimizations. After running an iterative process that evaluates different combinations of optimizations on top of the training programs/features, predictive models are created to correlate a given set of program features with profitable program transformations. Then, in the deployment phase, the framework analyzes new unseen programs by determining the program features and passes them to the new created models to predict the most profitable optimizations to improve execution time or other metrics depending on the user's optimization requirements. GCC was selected as the compiler infrastructure. They evolved 100 optimization flags under -O1, -O2 and -O3 levels and compared their results to the -O3 level and to the random search. The experimental results show that it is possible to improve the performance of the MiBench benchmark suite automatically using iterative compilation and machine learning on several platforms, including x86: Intel and AMD, and the ARC configurable core family. Using the machine learning-based framework, they were also able to learn a model that automatically improves the execution time of some individual MiBench programs by a factor of more than 2 while improving the overall MiBench suite by 11% on reconfigurable ARC architecture, without sacrificing code size or compilation time. Furthermore, their approach supports general multi-objective optimization where a user can choose to minimize not only the execution time but also the code size and compilation time.

3.2.3.7 Summary: auto-tuning compiler techniques

We provide in Table 3.3 a summary of several iterative compilation approaches, most of them are described above. We classify these approaches based on the problem they are tackling. Sometimes, the research papers address more than one issue in iterative compilation. This is not an exhaustive study of all iterative approaches but, it gives an overview of several research efforts involved in different areas during the past 20 years.

Table 3.3: Summary of iterative compilation approaches

	Local optimum	Phase ordering	Multiple data sets	Conflicting objectives	Learning Methods	Multiple CPUs	Multiple-compilers
Whitfield *et al.*'90 [WS90]	-	+	-	-	-	-	-
Bodin *et al.*'98 [BKK+98]	+	-	-	-	-	+	-
Kulkarni *et al.*'06 [KWTD06]	-	+	-	-	-	-	-
Cooper *et al.*'06 [CGH+06]	+	+	-	-	-	+	-
Bashkansky *et al.*'07 [BY07]	-	-	-	-	-	+	+
Hoste *et al.*'08 [HE08]	-	-	-	+	-	-	-
Lokuciejewski *et al.*'10 [LPF+10]	-	-	-	+	-	-	-
Chen *et al.*'10 [CHE+10]	-	-	+	-	-	+	+
Fursin *et al.*'11 [FKM+11]	-	-	-	+	+	+	-
Pekhimenko *et al.*'11 [PB11]	-	-	-	+	+	-	-
Plotnikov *et al.*'13 [PMV+13]	-	-	-	+	-	-	-

3.3 Lightweight system virtualization for automatic software testing

The key objective of this section is to present examples of the existing research efforts that have been presented to address:

- **The problem of hardware heterogeneity and software diversity:** For instance, we show the benefit of using a container-based infrastructure to tackle this problem. Thus, we present several research efforts that opted for this technology to facilitate software testing.
- **The problem of resource usage monitoring:** We discuss existing solutions applied for automatic resource usage extraction in a micro-services infrastructure.

The use of virtualization such as Virtual Machines (VMs) is very useful to tackle the problem of hardware and software platforms heterogeneity. In industry, a number of com-

mercial usage scenarios benefit from virtualization techniques to provide services to the end users. For example, Amazon EC2 makes VMs available to the customers who can use them to run their own computer applications or services on the cloud. Thus, a user can create, launch and terminate new VMs as needed. However, VMs are known to be very expensive in terms of system resources and performance [SCTF16]. In fact, each new VM instance constitutes of a virtual copy of all the hardware of the host machine which increases resource usage and overhead [Mer14]. Container-based virtualization presents an interesting alternative technology to VMs. The container technology is an operating-system-level virtualization which imposes little to near zero overhead. Programs in virtual instances use the operating system's system call interface and do not need to be emulated or ran in an intermediate virtual machine, as it is the case for VMware, QEMU or Xen. For instance, Docker[3] is a popular engine that offers the ability to deploy applications and their dependencies into lightweight containers that are very cheap to create. Processes executed in a Docker container are isolated from other processes running in the host OS or in other Docker containers. The Docker solution aims to address the challenges of resource usage and performance overhead caused by the full-stack virtualization.

Several authors [SCTF16, SPF+07, Mer14, FFRR15] compared the performance of the traditional VM solution to the container-based operating system technology. They showed that containers result in better performance than VMs since they induce less overhead.

3.3.1 Application in software testing

In software development, the container technology becomes more and more used in order to create a portable, consistent operating environment for development, deployment, and testing in the cloud [LTC15]. In the following, we discuss some of the state of the art approaches that choose the container technology as a mechanism to solve some research testing problems.

Marinescu *et al.* [MHC14] have used Docker as technological basis in their repository analysis framework Covrig to conduct a large-scale and safe inspection of the revision history from six selected Git code repositories. For their analysis, they run each version of a system in isolation and collect static and dynamic software metrics, using a lightweight container environment that can be deployed on a cluster of local or cloud machines. Each container is used to configure, compile, and test one program revision, as well as collect the metrics of interest, such as code size and coverage. The motivation of using such infrastructure is to provide a clean and configurable execution environment to run experiments.

[3] https://www.docker.com

According to the authors, the use of Docker as a solution to automatically deploy and execute the different program revisions has clearly facilitated the testing process.

Another Docker-based approach is presented in the BenchFlow2 project which focuses on benchmarking BPMN 2.0 engines [FIP15]. This project is dedicated to the performance testing of workflow engines. In this work, Ferme *et al.* present a framework for automatic and reliable calculation of performance metrics for BPMN 2.0 Workflow Management Systems (WfMSs). According to the authors, benchmarking WfMSs raises many challenges: 1) the system deployment complexity due to the distributed nature of these models execution; 2) the high number of configuration options required to integrate the deployment of the system under test, *i.e.*, the WfMS; 3) the complexity of the execution behaviors that can be expressed by modern modeling and execution languages such as BPMN2. Therefore, to address these problems, BenchFlow exploits Docker as a containerization technology, to enable the automatic deployment and configuration of the WfMSs. Thus, the WfMSs are automatically deployed and undeployed using Docker. Each component involved in the benchmark are packaged as Docker images to be deployed and executed on different servers connected by a dedicated local network. For each Docker instance, a new instance of the business models set is executed by the Wf during the experiment. Thanks to Docker, BenchFlow automatically collects all the data needed to compute the performance metrics, and to check the correct execution of the tests (metrics related the RAM/CPU usage and execution time). Their experimental results show that a simple business model running on two popular open-source WfMSs reveal important performance scalability issues.

Hamdy *et al.* [HIH16] propose Pons, a web based tool for the distribution of pre-release mobile applications for the purpose of manual testing. Pons facilitates building, running, and manually testing of Android applications directly in the browser. Based on Docker technology, this tool gets the developers and end users engaged in testing the applications in one place, alleviating the tester's burden of installing and maintaining testing environments, and providing a platform for developers to rapidly iterate on the software and integrate changes over time. Thus, it speeds up the testing process and reduces its cost. Pons utilizes Docker by predefining Docker images that contain the required services and tools to build Android applications, starting from the operating system up to the software development kit. A container is then built using one of these images to store the source code of a mobile application at a specific moment of history in a sandbox environment. Pons creates then, an Android emulator inside the Docker container to run the tests. The results are streamed at runtime in the web browser.

3.3.2 Application in runtime monitoring

Runtime monitoring is important in the area of cloud computing [ABDDP13]. Like VMs before them, containers require a monitoring mechanism. It should provide both historical and timely information about the resource usage of containers.

In industry, many commercial solutions are proposed to efficiently monitor applications running inside containers. For example, the Datadog[4] and cAdvisor[5] agents use the native Docker accounting metrics to gather the CPU, memory, network, and I/O metrics of the running containers. CAdvisor allows to monitor containers running in the same host machine. As an alternative, Scout[6] is used to aggregate metrics from different hosts and containers in a distributed architecture. As an orchestration system for Docker containers, we cite the open project Kubernetes[7]. It allows to quickly and efficiently respond to customer demand by deploying applications using multiple hosts and containers on the cloud. This clustering framework is shipped with a monitoring tool called Heapster[8] that provides a base monitoring platform on Kubernetes. Heapster collects and interprets various signals like resource usage, lifecycle events, etc, and exports cluster metrics via REST endpoints. It supports a pluggable storage backend such as InfluxDB with Grafana and Google Cloud Monitoring. Most of these tools provide web-based dashboards to visualize resource consumption at runtime as well as alerting mechanism that can be triggered if metrics go above or below a configured threshold. Other examples of Docker monitoring tools exist such as: Sensu Monitoring Framework, Prometheus, Sysdig Cloud, etc.

Runtime monitoring of containers has been also applied to solve research problems. As an example, Kookarinrat *et al.* [KT15] have investigated the problem of auto-sharding in NoSQL databases using a container-based infrastructure for runtime monitoring. The auto-sharding technique is used to divide data in the database and distribute it over multiple machines in order to scale it horizontally. In fact, selecting the right key is challenging. It could lead to either an improvement of the performance and capability of a database or to performance issues (*i.e.*, by selecting a wrong key) which could lead to a system halt. Therefore, authors analyzed and evaluated such suggested properties by studying how the variation of a shard key's choices could impact the DB performance. They simulated an environment using Docker containers and measured the read/write performance of variety of keys. Inside each container, they executed write/read queries into the MongoDB

[4] www.datadoghq.com
[5] https://github.com/google/cadvisor
[6] https://scoutapp.com
[7] https://kubernetes.io
[8] https://github.com/kubernetes/heapster

database and used Docker stats to automatically retrieve information about the memory and CPU usage.

Sun *et al.* [SWES16] present a tool to test, optimize, and automate cloud resource allocation decisions to meet QoS goals for web applications. Their infrastructure relies on Docker to gather information about the resource usage of deployed web servers.

Containers monitoring has been applied in other research efforts related especially to cloud computing and virtualization [PHP16, MRA+16].

3.4 Summary & open challenges

The analysis of the state of the art reveals several challenges. We describe below some of the challenges we have identified in both areas of iterative compilation and code generator testing:

- **Limits of existing work when testing code generators (the oracle problem):** Most of the work related to the automatic testing of code generators define an equivalence functional oracle to compare the result of MiL, SiL and PiL. In case of non-executable models, this comparison becomes impossible. Particularly, the oracle problem for testing the non-functional properties of the generated code has been avoided and not addressed by existing research efforts. The only comparison that has been made consists in comparing the hand-written code to the automatically generated code. The key objective of this comparison is to show that the generated code has better or equivalent performance properties compared to the human code.
  ⇒ **We believe that more advanced testing techniques as described in Section 3.1.2.1 can be applied to automatically detect code generator issues.**

- **Limits of existing techniques when exploring the large optimization search space (a multimodal problem):** As showed earlier, different techniques are applied during the iterative compilation process to explore the large optimization search space. It has been showed that the optimization search space is a multimodal problem, containing many local optima. For effective search, some authors use to involve only few optimizations (2 to 10) or to prune some paths in order to reduce this large search space. Genetic Algorithms are largely applied in most of the previous works. However, this technique may fall in the local optimum problem.
  ⇒ **The optimization search space is multimodal and very large. An alternative solution to the classical approaches is needed to tackle this problem.**

- **Lack of solutions that deal with conflicting objectives when auto-tuning compilers (a multi-objective optimization problem):** When trying to optimize software performance, many non-functional properties and design constraints must be involved and satisfied simultaneously to better optimize code. However, improving program execution time can result in a high resource usage which may decrease system performance especially for resource-constrained devices. Therefore, it is important to construct optimization levels that represent multiple trade-offs between resource usage and performance, enabling the user to choose among different optimal solutions which best suit the system requirements. Actually, there are only few work that address the compiler optimization as a multi-objective optimization problem.
  ⇒ **To deal with conflicting objectives, it is important to find a consensus between several non-functional properties when optimizing code to handle both, the resource usage and performance requirements**

- **Limits of existing approaches to handle the software platform and hardware requirements (the problem of software diversity and hardware heterogeneity):** Testing generators requires the execution of the generated code on different hardware and software platforms. Most of the existing research techniques apply a naive approach by running the generated code on different machines or using simulators for some configurations. Configuring the target execution environment to run the generated code and test it is time-consuming. Particularly, for generator users, it becomes very difficult to deploy and test the generated software artifacts in front of the increasing diversity of hardware and software platform settings.
  ⇒ **A highly configurable execution environment is needed to handle the software and hardware requirements.**

- **Lack of solutions that evaluate the resource usage properties when evaluating the generated code:** There is almost no research effort to deal with the non-functional properties of automatically generated code such as the resource usage. Most of the related work (in code generator testing) focus on the functional correctness of the generated code without putting too much emphasizes on the quality of the generated code. We claim that evaluating properties such as resource usage is very important in order to ensure an efficient production code generation. In iterative compilation, most of the existing work tend to reduce the execution time, code size, compilation time, etc. There is almost no work that evaluates the memory and CPU usage of the optimized code.
  ⇒ **An effective monitoring solution is needed to evaluate the resource usage properties of automatically generated code**

Part II

Contributions

To the reader: summary of contributions

In the rest of this thesis, we present our approaches that contribute to achieve our goal: automatic non-functional testing and tuning of configurable generators. Figure 3.3 depicts an overview of how the different contributions we propose are connected to each other and how they contribute to address the challenges of the state of the art, described earlier.

This thesis makes three main contributions:

- **Contribution I: Automatic non-functional testing of code generator families (in blue):**

  In this contribution (Chapter 4), we propose an approach for automatic non-functional testing of code generators. As discussed earlier, existing solutions lack of automation and efficiency to find code generator issues. Particularly, the oracle problem as well as the test of non-functional properties are not addressed. In this contribution, we address the limitations of the state of the art, so we describe an approach based on metamorphic testing and statistical analysis to efficiently detect inconsistencies within code generator families. Starting from a set of high-level benchmarks and test suites, we automatically generate source code to five target software platforms (*i.e.*, using a code generator family). We execute the generated code and evaluate the resource usage metrics using the lightweight testing infrastructure presented in contribution III. Inconsistencies are then reported for further inspection.

- **Contribution II: NOTICE, An approach for auto-tuning compilers (in red):**

  We address in this contribution two main problems: exploring the large optimization search space and finding trade-offs between conflicting objectives. Thus, we

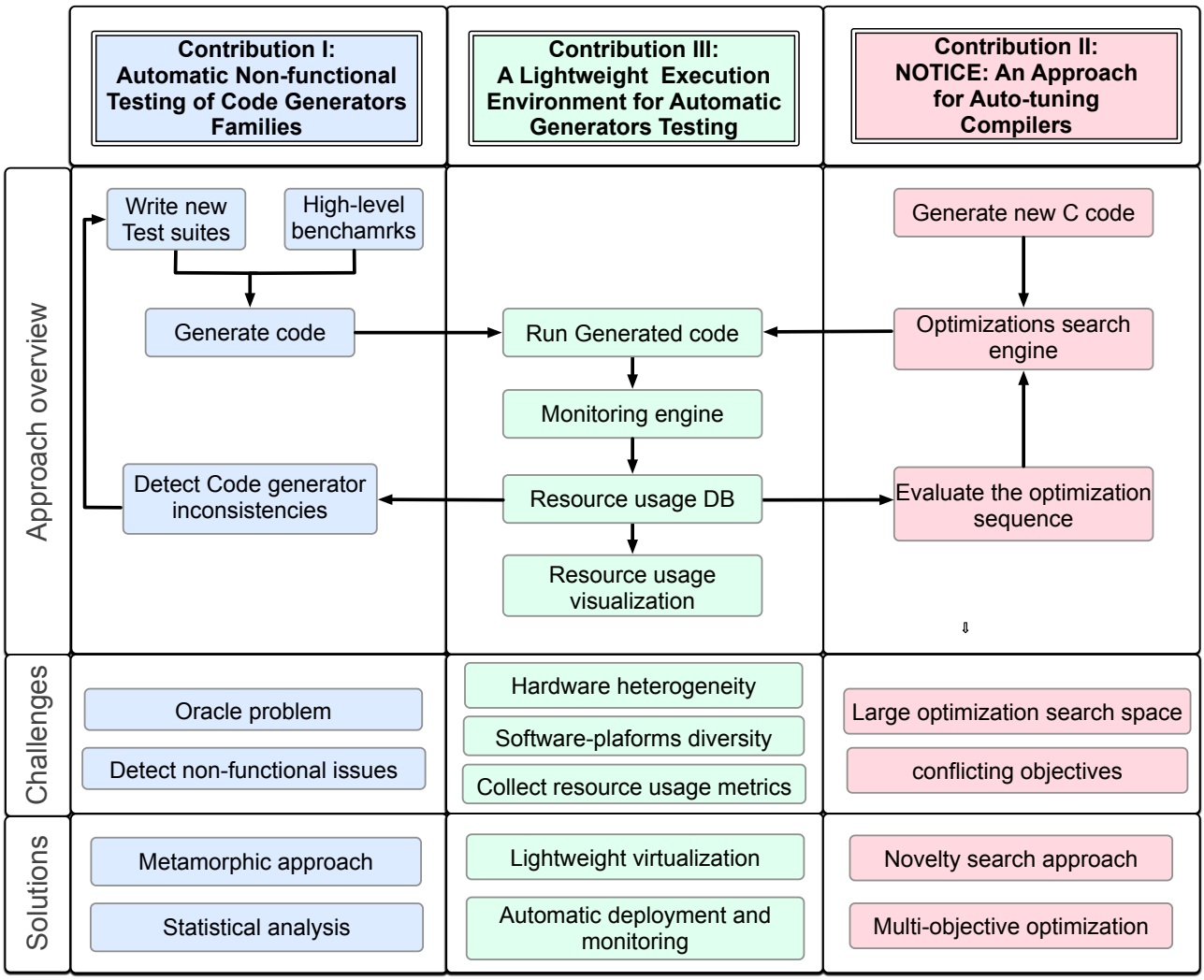


Figure 3.3: Summary of contributions

present in Chapter 5, an adaptation of the Novelty Search algorithm to the problem of compilers auto-tuning. Our contribution focuses on tuning GCC compilers based on randomly generated C programs. This approach shares the same monitoring infrastructure as the previous contribution in order to evaluate the impact of discovered optimization sequences on resource usage. The outcome of this approach is the best set of optimization options for a given hardware architecture, for a given input program, and for a specific non-functional property (*i.e.*, execution time, CPU

consumption, and memory usage). We also conduct a multi-objective optimization to tackle the problem of conflicting objectives (*e.g.*, memory usage and execution time).

- **Contribution III: A lightweight execution environment for automatic generators testing (in green):**

  We propose in Chapter 6, a common infrastructure used by both previous contributions to gather information about the quality of generated code in terms of memory and CPU usage. It serves as a lightweight execution environment to easily run tests across different software and hardware settings. In particular, it is based on microservices, namely Docker, in order to automate software deployment, execution and monitoring. Finally, we provide a mechanism to visualize at runtime the resource consumption of running programs. This contribution addresses the limitations of the state of the art, namely the lack of solutions that handle the software and hardware diversity when testing/tuning generators.

The validation of each contribution is presented in the corresponding chapter. Different experiments are used to illustrate the characteristics of each solution we present.

Chapter 4

Automatic non-functional testing of code generator families

Generative software development has paved the way to the creation of numerous code generators that automatically translate high-level system specifications into multi-target executable code. To preserve software reliability and quality, generated code has to be tested with the same effort as for the manually written code. Any issue with code generators should be detected and corrected as early as possible in order to ensure the correct behavior of delivered software. We presented in Chapter 3 several approaches that address the automatic functional testing of code generators. However, automatically testing the non-functional properties of generated code remains a challenging task that has not been addressed yet.

This chapter describes a testing approach that automatically detects anomalies in code generators in terms of non-functional properties (*i.e.*, resource usage and performance). In fact, we adapt the idea of metamorphic testing to the problem of code generators testing. Hence, our approach relies on the definition of high-level test oracles (*i.e.*, metamorphic relations) to check the potential inefficient code generator among a family of code generators. Moreover, we apply different statistical methods in order to automate the inconsistencies detection. We evaluate our approach by analyzing the performance of Haxe, a popular high-level programming language that involves a set of cross-platform code generators. Experimental results show that our approach is able to detect some performance inconsistencies that reveal real issues in Haxe code generators.

This chapter is organized as follows:

Section 4.1 presents the motivations behind this work. In particular, we discuss three

motivation examples and the challenges we are facing.

Section 4.2 presents the traditional process for non-functional testing of a code generator family.

Section 4.3 describes the general approach overview and the testing strategy.

In Section 4.4, the evaluation and results of our experiments are discussed. Hence, we provide more details about the experimental settings, the code generators under test, the benchmark used, the evaluation metrics, etc. We discuss then the evaluation results.

Finally, we conclude in Section 4.5.

4.1 Context and motivations

4.1.1 Code generator families

Today, different customizable code generators can be used to easily and efficiently generate code for different software platforms, programming languages, operating systems, etc. This work is based on the intuition that a code generator is often a member of a family of code generators [CB08].

Definition (Code generator family). *We define a code generator family as a set of code generators that takes as input the same language/model and generate code for different target software platforms.*

For example, this concept is widely used in industry when applying the *"write once, run everywhere"* paradigm. Users can benefit from a family of code generators (*e.g.*, cross-platform code generators [FRSD15]) to generate from the manually written (high-level) code, different implementations of the same program in different languages. This technique is very useful to address diverse software platforms and programming languages.

As motivating examples for this research work, we can cite three approaches that intensively develop and use code generator families:

a. Haxe. Haxe[1] [Das11] is an open source toolkit for cross-platform development which compiles to a number of different programming platforms, including JavaScript, Flash,

[1] http://haxe.org/

PHP, C++, C#, and Java. Haxe involves many features: the Haxe language, multi-platform compilers, and different native libraries. The Haxe language is a high-level programming language which is strictly typed. This language supports both, functional and object-oriented programming paradigms. It has a common type hierarchy, making certain API available on every target platform. Moreover, Haxe comes with a set of code generators that translate manually-written code (in Haxe language) to different target languages and platforms. This project is popular (more than 1440 stars on GitHub).

b. ThingML. ThingML[2] is a modeling language for embedded and distributed systems [FMSB11]. The idea of ThingML is to develop a practical model-driven software engineering tool-chain which targets resource-constrained embedded systems such as low-power sensors and microcontroller-based devices. ThingML is developed as a domain-specific modeling language which includes concepts to describe both software components and communication protocols. The formalism used is a combination of architecture models, state machines and an imperative action language. The ThingML tool-set provides a code generator family to translate ThingML to C, Java and JavaScript. It includes a set of variants for the C and JavaScript code generators to target different embedded systems and their constraints. This project is still confidential, but it is a good candidate to represent the modeling community practices.

c. TypeScript. TypeScript[3]is a typed superset of JavaScript that compiles to plain JavaScript [RSF+15]. In fact, it does not compile to only one version of JavaScript. It can transform TypeScript to EcmaScript 3, 5 or 6. It can generate JavaScript that uses different system modules ('none', 'commonjs', 'amd', 'system', 'umd', 'es6', or 'es2015')[4]. This project is popular (more than 22 478 stars on GitHub).

Functional testing of a code generator family is simple. Since the produced programs are generated from the same high-level program, the oracle can be defined as a comparison between their functional outputs, which should be the same. In fact, based on the three sample projects presented above, we remark that all GitHub code repositories of the corresponding projects use unit tests to check the correctness of code generators.

In terms of non-functional tests, we observe that ThingML and TypeScript do not provide any specific tests to check the consistency of code generators regarding the memory or

[2] http://thingml.org/

[3] https://www.typescriptlang.org/

[4] Each of this variation point can target different code generators (function *emitES6Module* vs *emitUMDModule* in emitter.ts for example).

CPU usage properties. Haxe provides two test cases[5] to benchmark the resulting generated code. One serves to benchmark an example in which object allocations are deliberately (over) used to measure how memory access/GC mixes with numeric processing in different target languages. The second test evaluates the network speed across different target platforms.

4.1.2 Issues when testing a code generator family

The main difficulties when testing the resource usage properties of code generators is that we cannot just observe the execution of produced code, but we have to observe and compare the execution of generated programs with equivalent (or reference) implementations (*i.e.*, in other languages). Even if there is no explicit oracle to detect inconsistencies for a single code generator, we could benefit from the family of code generators to compare the behavior of several generated programs and detect singular resource consumption profiles that could reveal a code generator inconsistency [Hun11].

As a consequence, we define a code generator inconsistency as:

Definition (**code generator inconsistency**). *A generated code that exhibits an unexpected behavior in terms of performance or resource usage compared to all equivalent implementations in the same code generator family.*

Next section discusses the common process used by developers to test the non-functional properties of generated code. We also illustrate how we can benefit from code generator families to identify unexpected behaviors.

4.2 The traditional process for non-functional testing of a code generator family

A reliable and acceptable way to increase confidence in code generators is to validate and check the functional behavior of generated code, which is a common practice in generators testing [JS14, SCDP07, SWC05]. However, proving that the generated code is functionally correct is not enough to claim the effectiveness of the code generator under test.

[5] https://github.com/HaxeFoundation/haxe/tree/development/tests/benchs

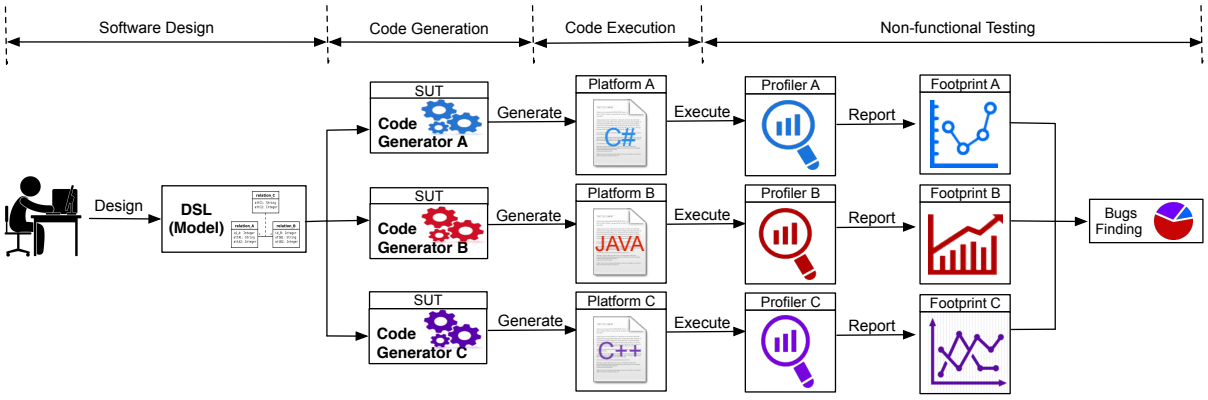


Figure 4.1: An overall overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to runtime: the classical way

In fact, code generators have to respect different requirements to preserve software reliability and quality [DAH11]. In this case, ensuring the quality of generated code requires examining several non-functional properties such as code size, resource or energy consumption, execution time, etc [PE06]. Figure 4.1 summarizes the classical steps required to ensure the code generation and non-functional testing of a code generator family. We distinguish four major steps: software design using high-level system specifications, code generation, code execution, and non-functional testing of generated code.

In the first step, software developers have to define, at design time, the software behavior using a high-level abstract language (DSLs, models, program, etc). Afterwards, developers can use platform-specific code generators to ease the software development and automatically generate code for different languages and platforms. We depict, as an example in Figure 4.1, three code generators from the same family capable to generate code to three software programming languages (Java, C# and C++). The first step is to generate code from the previously designed model. Afterwards, generated software artifacts (*e.g.*, Java, C#, C++, etc.) are compiled, deployed and executed across different target platforms (*e.g.*, Android, ARM/Linux, JVM, x86/Linux, etc.). Finally, to perform the non-functional testing of generated code, developers have to collect, visualize and compare information about the performance and efficiency of running code across the different platforms. Therefore, they generally use several platform-specific profilers, trackers, instrumenting and monitoring tools in order to find some inconsistencies or bugs during code execution [GS14, DGR04]. Finding inconsistencies within code generators involves analyzing and inspecting the code and that, for each execution platform. For example, one way

to handle that, is to analyze the memory footprint of software execution and find memory leaks [NS07]. Developers can then inspect the generated code and find some fragments of the code-base that have triggered this issue. Then, they report this information in order to fix, refactor, and optimize the code generation process. Compared to this classical (and manual) testing approach, our proposed work seeks to automate the last three steps: the code generation and execution on top of different software platforms, and the detection of non-functional issues.

4.3 Approach overview

Our contributions in this work are divided into two parts:

- First, we describe our testing infrastructure. This contribution addresses the problem of software diversity, as discussed in Chapter 2.
- Second, we present a methodology for automatic detection of inconsistencies in code generator families. This approach addresses the oracle problem when testing the non-functional properties.

4.3.1 An infrastructure for non-functional testing using system containers

In this contribution, we focus on evaluating the non-functional properties related to the resource usage and performance of generated code. To do so, many system configurations (*i.e.*, execution environments, libraries, compilers, etc.) must be taken into account to efficiently generate and test code.

However, tuning different applications (*i.e.*, generated code) with different configurations on one single machine is complex. A single system has limited resources and this can lead to performance regressions. Moreover, each execution environment comes with a collection of appropriate tools such as compilers, code generators, debuggers, profilers, etc. Therefore, we need to deploy the test harness, *i.e.*, the produced binaries, on an elastic infrastructure that provides facilities to the code generator developers to ensure the deployment and monitoring of generated code in different environment settings. Consequently, the testing infrastructure provides support to automatically:

1. Deploy the generated code, its dependencies, and its execution environments

2. Execute the produced binaries in an isolated environment
3. Monitor the execution
4. Gather resource usage metrics (CPU, Memory, etc.)

To ensure these four main steps, we rely on system containers [SPF+07] as a dynamic and configurable execution environment for running and evaluating the generated programs in terms of resource usage.

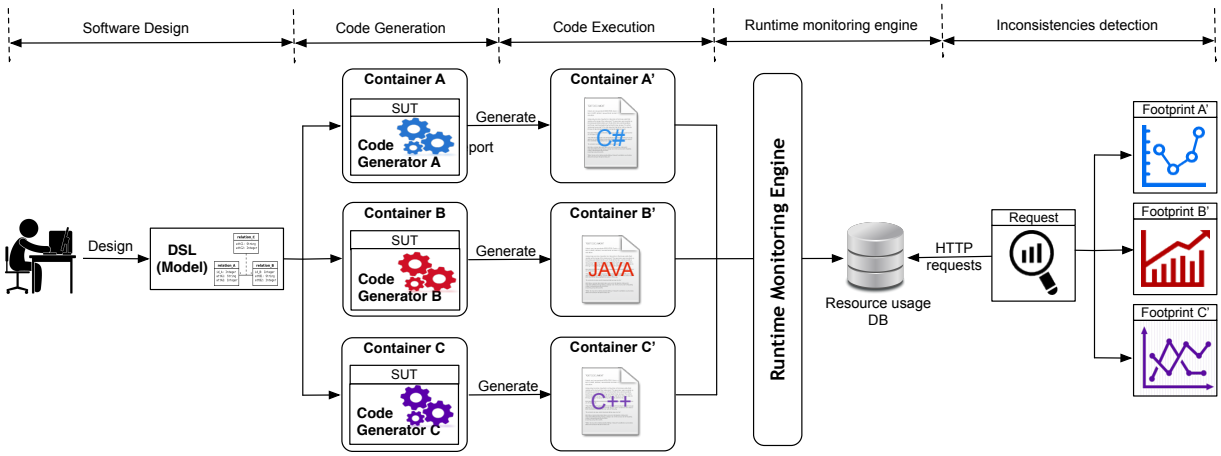


Figure 4.2: A technical overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to runtime.

Figure 4.2 shows the container-based infrastructure used for testing code generators. Compared to the classical method presented in Figure 4.1, we add the following features:

– *At the code generation level*: Code generators are configured inside different containers in order to generate code for the target platform.

– *At the code execution level*: Libraries, compilers and different dependencies are configured in different containers in order to execute the generated code. For each target platform, a new instance is created.

– *At the non-functional testing level*: We add a runtime monitoring engine (based on containers) in order to extract and save the resource usage metrics of all running containers. Finally, inconsistencies detection involves the analysis of the resource usage data extracted from the database in order to find issues with the generated code. This step is discussed in details in the next section.

Chapter 6 provides more details about the technical choices we have made to synthesize this testing infrastructure.

4.3.2 A metamorphic testing approach for automatic detection of code generator inconsistencies

We discussed in Section 3.1.2.2 several approaches proposed by the software testing community in order to alleviate the oracle problem. Among the attractive approaches that can be applied to test code generators, we distinguish the metamorphic testing approach (derived oracles). In the following, we describe the basic concept of metamorphic testing and our adaptation of this method for the non-functional testing of code generator families.

4.3.2.1 Basic concept of metamorphic testing

In this section, we shall introduce the basic concepts of metamorphic testing (MT), proposed by Chen *et al.* [CCY98]. The idea of MT is to derive test oracles from the relation between test cases outputs instead of reasoning about the relation between test inputs and outputs.

MT recommends that, given one or more test cases (called "source test cases", "original test cases", or "successful test cases") and their expected outcomes (obtained through multiple executions of the target program under test), one or more follow-up test cases can be constructed to verify the necessary properties (called Metamorphic Relations MRs) of the system or function to be implemented. In this case, the generation of the follow-up test cases and verification of the test results require the respect of the MR.

The classical example of MT is that of a program that computes the *sin* function. A useful metamorphic relation for *sin* functions is $sin(x) = sin(\pi - x)$. Thus, even though the expected value for the source test case $sin(50)$ for example in not known, a follow-up test case can be constructed to verify the MR defined earlier. In this case, the follow-up test case is $sin(\pi - 50)$ which must produce an output value that is equal to the one produced by the original test case $sin(50)$. If this property is violated, then a failure is immediately detected. MT generates follow-up test cases as long as the metamorphic relations are respected. This is an example of a metamorphic relation: an input transformation that can be used to generate new test cases from existing test data, and an output relation MR, that compares the outputs produced by a pair of test cases. MR can be any properties involving the inputs and outputs of two or more executions of the target program such as equalities, inequalities, convergence constraints, and many others.

Because MT checks the relation among several executions rather than the correctness of individual outputs, it can be used to fully automate the testing process without any manual intervention. However, constructing the metamorphic relations is typically a manual task that demands thorough knowledge of the program under test. It also depends on the application context and domain. The effectiveness of metamorphic testing is highly dependent on the identified metamorphic relations, and designing effective metamorphic relations is thus a critical step when applying metamorphic testing.

We describe in the next section our adaptation of MT to the problem of non-functional testing of code generator families.

4.3.2.2 Adaptation of the MT approach to detect code generator inconsistencies

In general, MT can be applied to any problem in which a necessary property involving multiple executions of the target function can be formulated. Some examples of successful applications are presented in [ZHT+04]. We note that MT is recently applied to compilers testing [DL16, TWZS10, LAS14].

To apply MT, there are four basic steps to follow:

1. Find the properties of the system under test: the system should be investigated manually in order to find intended MRs defining the relation between inputs and outputs. This is based on the source test cases.

2. Generate/select test inputs that satisfy the MR: this means that new follow-up test cases must be generated or selected in order to verify their outputs using the MR.

3. Execute the system with the inputs and get outputs: original and follow-up test cases are executed in order to gather their outputs.

4. Check whether these outputs satisfy the MR, and if not, report failures.

We develop now these four points in details to show how we can adapt the MT approach to the code generator testing problem.

4.3.2.3 Metamorphic relation

Step 1 consists in identifying the necessary properties of the program under test and represent them as metamorphic relations. As already stated, a metamorphic relation is a relation derived from different system executions. We use the MR definition as presented in [TWZS10, CCL+06]:

Definition (Metamorphic relation). *Let $(x_1, x_2, ..., x_k)$ be a series of inputs to a function f, where $k \geqslant 1$, and $f(x_1, x_2, ..., x_k)$ be the corresponding series of results. Suppose $(f(x_{i1}), f(x_{i2}), ..., f(x_{im}))$ is a subseries, possibly an empty subseries, of $(f(x_1), f(x_2),..., f(x_k))$. Let $(x_{k+1}, x_{k+2}, ..., x_n)$ be another series of inputs to f, where $n \geqslant k+1$, and $(f(x_{k+1}), f(x_{k+2}),..., f(x_{xn}))$ be the corresponding series of results. Suppose, further, that there exists relations $r(x_1, x_2,..., x_k, f(x_{i1}), f(x_{i2}),...., f(x_{im}), x_{k+1}, x_{k+2},..., x_n))$ and $r'(x_1, x_2,..., x_n, f(x_1, f(x_2,..., f(x_n))$ such that r' must be true whenever r is satisfied. We say that*

$$\boldsymbol{MR} = (x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n) \mid$$

$$r(x_1, x_2, ..., x_k, f(x_{i1}), f(x_{i2}), ..., f(x_{im}), x_{k+1}, x_{k+2}, ..., x_n))$$

$$\Rightarrow r'(x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n))$$

is a metamorphic relation. When there is no ambiguity, we simply write the metamorphic relation as

$$\boldsymbol{MR}\text{: if } r(x_1, x_2, ..., x_k, f(x_{i1}), f(x_{i2}), ..., f(x_{im}), x_{k+1}, x_{k+2}, ..., x_n))$$

$$\text{then } r'(x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n)).$$

Furthermore, $x_1, x_2, ..., x_k$ are known as source test cases and $x_{k+1}, x_{k+2}, ..., x_{xn}$ are known as follow-up test cases

A code generator family can be seen as a function: $C : I \rightarrow P$, where I is the domain of valid high-level source programs and P is the domain of the target programs that are generated by the different code generators of the same family. The property of a code generator family implies that the generated programs P share the same behavior as it is specified in I.

The availability of multiple generators with comparable functionality allows us to adapt the MT in order to detect non-functional inconsistencies. In fact, if we can find out proper relation R (see Equation 4.1) of the non-functional behavior, we can get the metamorphic relation and conduct MT for testing code generator families. Let $f(P(t_i))$ be a function that calculates the non-functional output (such as execution time or memory usage) of the

input test suite (t_i), running on a generated program (P). Since we have different program versions generated in the same family, we denote by ($P_1(t_i)$, $P_2(t_i)$,..., $P_n(t_i)$) the set of generated programs. The corresponding outputs would be ($f(P_1)$, $f(P_2)$,..., $f(P_n)$). Thus, our MR looks like this:

$$R(P_1(t_i), P_2(t_i), ..., P_n(t_i)) \Rightarrow R(f(P_1(t_i)), f(P_2(t_i)), ..., f(P_n(t_i))) \tag{4.1}$$

On the one hand, we use the following equation $P_1(t_i) \equiv P_2(t_i)$ to denote *the functional equivalence relation* between two generated programs P_1 and P_2 from the same family. This means that the generated programs P_1 and P_2 have the same behavioral design, and for any test suite t_i, they have the same functional output. If this relation is not satisfied, then there is at least one faulty code generator that produced incorrect code. In this work, we focus on the non-functional testing, so we ensure that this relation is ensured by excluding all the programs that do not exhibit the same behavior.

On the other hand, since we are comparing equivalent implementations of the same program written in different languages, we assume that the memory usage and execution time should be more or less the same with a small variation for each test suite across the different versions. Obviously, we are expecting to get a variation between different executions because we are comparing the execution time and memory usage of test suites that are written in different languages and executed using different technologies (*e.g.*, interpreters for PHP, JVM for Java, etc.). This observation is also based on initial experiments, where we evaluate the resource usage/execution time of several test suites across a set of equivalent versions generated using a code generator family (presented in details in the evaluation, Section 4.4). As a consequence, we use the notation $\Delta\{f(P_1(t_i)), f(P_2(t_i))\}$ to designate the variation of memory usage or execution time of test suite execution t_i across two versions of generated code P_1 and P_2 written in different languages. We suppose that this variation should not exceed a certain threshold value T, otherwise, we raise a code generator inconsistency. Based on this intuition, the MR can be represented as:

$$P_1(t_i) \equiv P_2(t_i) \equiv ... \equiv P_n(t_i) \Rightarrow \Delta\{f(P_1(t_i)), f(P_2(t_i)), ..., f(P_n(t_i))\} < T \quad (n \geqslant 2) \tag{4.2}$$

This MR is equivalent to say that: **if** a set of functionally equivalent programs are generated using the same code generator family (($P_1(t_i)$, $P_2(t_i)$,...,$P_n(t_i)$), and with the same input test suite t_i, **then** the comparison of their non-functional outputs ($f(P_1(t_i))$, $f(P_2(t_i))$,..., $f(P_n(t_i))$) should be the same while taking into account a tolerance interval defined by the variation Δ that shall not exceed a specific threshold value T.

The generated code that violates this metamorphic property represents an inconsistency and its corresponding code generator is considered as defective.

4.3.2.4 Metamorphic testing

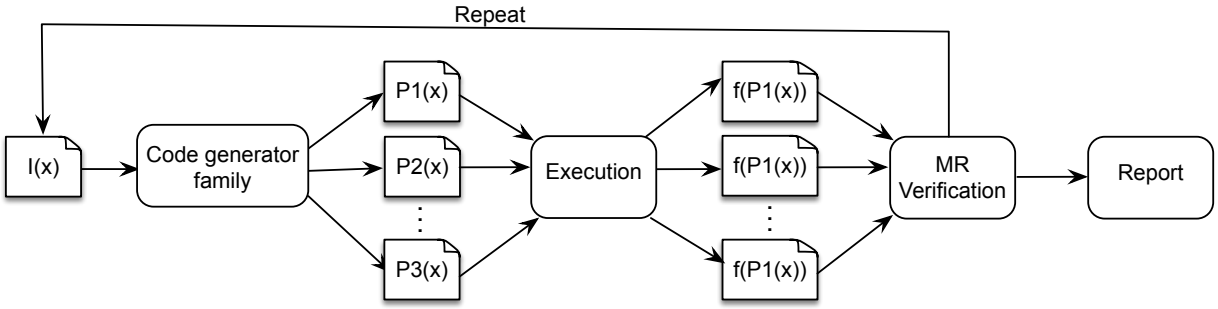


Figure 4.3: The metamorphic testing approach for automatic detection of code generator inconsistencies

So far, we have defined the MR necessary for inconsistencies detection. We describe now our automatic metamorphic testing approach based on this relation (Steps 2, 3, and 4). Figure 4.3 shows the approach overview. The code generator family takes the same input program I and generate a set of equivalent test programs (P_1, P_2,...,P_n). This corresponds to Step 2. In our MT adaptation, follow-up test cases represent the equivalent test programs that are automatically generated using a code generator family. Test suites are also generated automatically since we suppose that they are already defined at design time. In fact, the same test suite (test cases + input data values) is passed to all generated programs. Then, generated programs and their corresponding test suites are executed (Step 3). Afterwards, we measure the memory usage or execution time of these generated programs ($f(P_1(t_i))$, $f(P_2(t_i))$,..., $f(P_n(t_i))$). Finally, the execution results are compared and verified using the MR defined earlier (Step 4). In this process, inconsistencies will be reported when one of the follow-up equivalent test program violates the MR.

4.3.2.5 Variation threshold

One of the questions that may be raised when applying our MT approach is how can we find the right variation threshold T from which an inconsistency is detected? Answering this question is very important to prove the effectiveness of our MT approach. To do so,

we conduct a statistical analysis of our non-functional data in order to find an accurate threshold value T. Before that, the non-functional outputs need to be prepared to make them suitable for the statistical methods employed by our methodology. Thus, we describe first our process for data preparation.

Data preparation

As depicted in Table 4.1, each program comes with a set of test suites (t_1, t_2,..., t_m). Evaluating a test suite requires the calculation of the memory usage or execution time $f(P_1(t_i))$, $f(P_2(t_i))$,..., $f(P_n(t_i))$ where $(1 \leq i \leq m)$ for all target software platforms. Thus, obtained results represent a matrix where columns indicate the non-functional value (raw data) for each target software platform and rows indicate the corresponding test suite.

	Target platform 1	Target platform 2	...	Target platform n
t_1	$f(P_1(t_1))$	$f(P_2(t_1))$	...	$f(P_n(t_1))$
t_2	$f(P_1(t_2))$	$f(P_2(t_2))$	...	$f(P_n(t_2))$
...	...	...	...	...
t_m	$f(P_1(t_m))$	$f(P_2(t_m))$	...	$f(P_n(t_m))$

Table 4.1: Results of test suites execution

The non-functional data should be converted into a format that can be understood by our statistical methods. One way to compare these non-functional outputs is to study the factor differences. In other words, we would evaluate for each target platform the number of times (the factor) that a test suite takes to run compared to a reference execution. The reference execution corresponds to the minimum obtained non-functional value of t_i execution across the n target platforms. The resulting factor is the ratio between the actual non-functional value and the minimum value obtained among the n versions. The following equation is applied for each cell in order to transform our data:

$$F(f(P_j(t_i))) = \frac{f(P_j(t_i))}{Min(f(P_1(t_i)), ..., f(P_n(t_i)))} \tag{4.3}$$

The reference execution will automatically get a score value $F = 1$. The maximum value is the one leading to the maximum deviation from the reference execution. For example, let P_1 be the generated program in Java. If the execution time needed to run t_1 yields to the minimum value $f(P_1(t_1))$ compared to other versions, then $f(P_1(t_1))$ will get a factor value F equal to 1 and the other versions will be divided by $f(P_1(t_1))$ to get the corresponding factor values compared to Java.

Statistical analysis

In our MT approach, an inconsistency is a resource usage/performance variation that exceeds a specific threshold value T. We propose the use of two variation analysis methods [MHH13]: principal components analysis (PCA) and range charts (R-chart). Table 4.2 gives an overview of these two statistical methods. The key objective of these methods is to evaluate the memory usage and performance variation, and consequently defining an appropriate T value for our MR.

Technique	**Method**
R-chart	Define T as a variation between an upper and lower control limit
PCA	A cutoff value of the PC score distances defines the T

Table 4.2: Variation analysis approaches

R-Chart

In this approach, the variation evaluation between the different versions is determined by comparing the non-functional measurements based on a statistical quality control technique called *R-Chart* or *range chart* [MHH13]. R-Chart is used to analyze the variation within processes. It is designed to detect changes in variation over time and to evaluate the consistency of process variation. R-Chart uses control limits (LCL and UCL) to represent the limits of variation that should be expected from a process. LCL denotes the Lower Control Limit and UCL denotes the Upper Control Limit.

When a process is within the controlled limits, any variation is normal. It is said that the process is **in control**. Outside limit variations, however, it is considered as deviation and the R-chart is considered as **out of control** which means that the process variation is not stable. Thus, it tells that there is an inconsistency leading to this high variation deviation (see Figure 4.4).

In our case, a process represents the n non-functional outputs obtained after the execution of a test suite t_i. As we defined the MR, the variation within a single process has to be lower than a threshold T. In our settings, this variation must be between the LCL and UCL.

Therefore, for each test suite we calculate the range R corresponding to the difference between the maximum and minimum non-functional outputs across all target platforms.

$$R(t_i) = Max(f(P_1(t_i)), ..., f(P_n(t_i))) - Min(f(P_1(t_i)), ..., f(P_n(t_i))) \tag{4.4}$$

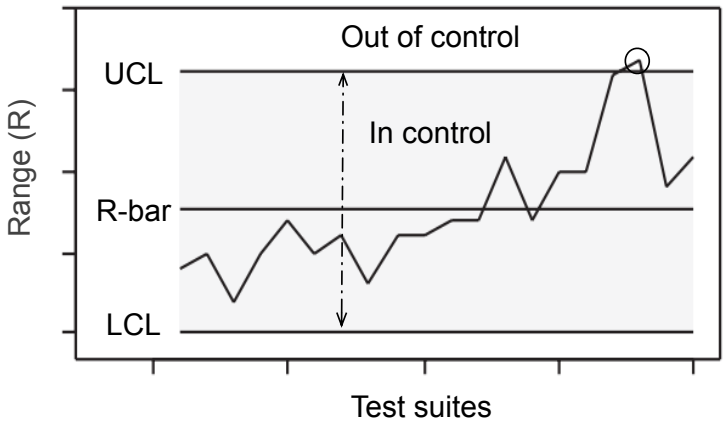


Figure 4.4: The R-Chart process

R quantifies the variation results when running the same test suite t_i across different program versions. To determine whether the variation is in control or not, we need to determine the control limits values. UCL and LCL reflect the actual amount of variation that is observed. Both metrics are a function of R-bar ($\bar{R}$). $\bar{R}$ is the average of R for all test suites. The UCL and LCL are calculated as follows:

$$\begin{aligned} UCL &= D_4\bar{R} \\ LCL &= D_3\bar{R} \end{aligned} \tag{4.5}$$

where D_4, D_3, are control chart constants that depend on the number of variables inside each process (see constants values[6]).

For example, for a family composed of less than 7 code generators, the D_3 value is equal to 0, and as a consequence LCL = 0. In this case, the UCL represents the threshold T value from which we detect a high variation deviation, leading to an inconsistency. As we stated earlier, the UCL is a function of $\bar{R}$, and $\bar{R}$ is a function of range differences. So, the UCL value (or T) is sensitive to new test suites. So, when a new test suite is executed, the T value is updated and the variation is evaluated with the new threshold value.

We present in the following an alternative statistical approach to analyze the variation of all our data.

[6] http://www.bessegato.com.br/UFJF/resources/table_of_control_chart_constants_old.pdf

PCA

With a large number of program versions, the matrix of non-functional data (Table 4.1) may be too large to study and interpret the variation properly. There would be too many pairwise correlations between the different versions to consider and the variation is impossible to display (graphically) when test suites are executed in more than three target software platforms. With 12 variables, for example, there will be more than 200 three-dimensional scatter plots to be designed to study the variation and correlations. To interpret the data in a more meaningful form, it is therefore necessary to reduce the number of variables composing our data.

Principal Component Analysis[7] (PCA) is a multivariate statistical approach that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called Principal Components (PCs). It can be applied when data are collected on a large number of variables from a single observation. Thus, we apply the PCA approach to our case study because our dimension space as it is presented in Table 4.1, is composed of a set of processes (test suites) where n variables (*e.g.*, target programming languages) are composing each observation. The variability within our model is correlated to these n variables representing the test suites running on n target platforms.

The main objective of applying PCA is to reduce the dimensionality of the original data and explain the maximum amount of variance with the fewest number of principal components. To do so, PCA is concerned with summarizing the variance-covariance matrix. It involves computing the eigenvectors and eigenvalues of the variance-covariance matrix. The eigenvectors are used to project the data from n dimensions down to a lower dimensional representation. The eigenvalues give the variance of the data in the direction of the eigenvector. The first eigenvector is the vector which defines the direction of maximum variance in the data. The first principal component is calculated such that it accounts for the greatest possible variance in the data set. The second principal component is calculated in the same way, with the condition that it is uncorrelated with (*i.e.*, perpendicular to) the first principal component and that it accounts for the next highest variance. The eigenvector associated with the largest eigenvalue has the same direction as the first principal component. The eigenvector associated with the second largest eigenvalue determines the direction of the second principal component. PCA uses many data transformations and statistical concepts. We are not interested in studying all the mathematical aspects of PCA. Thus, we use an existing R package[8] to transform and reduce our data into two PCs in order to visualize the variation of all our data points in a 2-dimensional space.

[7] https://en.wikipedia.org/wiki/Principal_component_analysis
[8] http://factominer.free.fr/

Our intuition behind the PCA approach is to conduct a general and complete analysis of variation in order to find extreme variation points at the boundaries of the multivariate data. These extreme points represent, from a statistical perspective, *outliers*. Following our MT approach, these points correspond to the inconsistencies (or deviations) we would detect. Outliers have an important influence over the PCs. An outlier is defined as an observation which does not follow the model followed by the majority of the data. One way to detect outliers is to use a metric called Score Distance (SD). SD measures the dispersion of the observations within the PCA space. It thus measures how far an observation lies from the rest of the data within the PCA subspace. SD measures the statistical distance from a PC score to the center of the scores. For an observation x_i the score distance is defined as:

$$SD_i = \sqrt{\sum_{j=1}^{a} \frac{t_{ij}^2}{\lambda_j}} \tag{4.6}$$

where a is the number of PCs forming the PCA space, t_{ij} are the elements of the score matrix obtained after running PCA, and λ_j is the variance of the j^{th} PC which corresponds to the j^{th} eigenvalue. In order to find the outliers, we compute the 97.5%-Quantile Q of the Chi-square distribution as a cutoff value of the SD ($\sqrt{\chi^2_{a,0.975}}$). It corresponds to a confidence ellipse that covers 97.5% of the data points. According to the table of the Chi-Square distribution[9], this value is equal to $\sqrt{7.38} = 2.71$. Any sample whose SD is larger than the cutoff value, is identified as an outlier (or inconsistency). This cut-off value represents the variation threshold T we would define for our MR using the PCA approach.

We move now to present the evaluation of our approach.

4.4 Evaluation

So far, we have presented an automated approach for detecting inconsistencies within code generator families. So, we shape our goal as this research question:

RQ1: ***How effective is our metamorphic testing approach for automatically detecting inconsistencies in code generator families?***

To answer this question, we evaluate the implementation of our approach by explaining

[9] https://store.fmi.uni-sofia.bg/fmi/statist/education/Virtual_Labs/tables/tables3.html

the design of our empirical study and the different methods we used to assess the effectiveness of our approach. The experimental material is available for replication purposes[10].

4.4.1 Experimental setup

4.4.1.1 Code generators under test: Haxe compilers

In order to test the applicability of our approach, we conduct experiments on a popular high-level programming language called Haxe and its code generators. Haxe is an open source toolkit for cross-platform development which compiles to a number of different programming platforms, including JavaScript, Flash, PHP, C++, C# and Java. Haxe involves many features: the Haxe language, multi-platform compilers, and different native libraries. The Haxe language is a high-level programming language which is strictly typed. This language supports both functional programming and object-oriented programming paradigms. It has a common type hierarchy, making certain API available on every targeted platform. Haxe comes with a set of compilers that translate manually-written code (in Haxe language) to different target languages and platforms. Haxe code can be compiled for applications running on desktop, mobile and web platforms. It comes also with a set of standard libraries that can be used on all supported targets and platform-specific libraries for each of them.

The process of code transformation and generation can be described as following: Haxe compilers analyze the source code written in Haxe language. Then, the code is checked and parsed into a typed structure, resulting in a typed abstract syntax tree (AST). This AST is optimized and transformed afterwards to produce source code for the target platform/language. Haxe offers the option of choosing which platform to target for each program using command-line options. Moreover, some optimizations and debugging information can be enabled through command-line interface, but in our experiments, we did not turn on any further options.

The Haxe code generators constitute the code generator family we would evaluate in this work.

4.4.1.2 Cross-platform benchmark

One way to prove the effectiveness of our approach is to create benchmarks. Thus, we use the Haxe language and its code generators to build a cross-platform benchmark. The

[10] https://testingcodegenerators.wordpress.com/

proposed benchmark is composed of a collection of cross-platform libraries that can be compiled to different targets. In these experiments, we consider a code generator family composed of five target Haxe compilers: Java, JS, C++, CS, and PHP code generators. To select cross-platform libraries, we explore github and we use the Haxe library repository[11]. So, we select seven libraries that provide a set of test suites with high code coverage scores.

In fact, each Haxe library comes with an API and a set of test suites. These tests, written in Haxe, represent a set of unit tests that covers the different functions of the API. The main task of these tests is to check the correct functional behavior of generated programs. To prepare our benchmark, we remove all the tests that fail to compile to our five targets (*i.e.*, errors, crashes and failures) and we keep only test suites that are functionally correct in order to focus only on the non-functional properties. Moreover, we add manually new test cases to some libraries in order to extend the number of test suites. The number of test suites depends on the number of existing functions within the Haxe library.

Library	**#TestSuites**	**Description**
Color	19	Color conversion from/to any color space
Core	51	Provides extensions to many types
Hxmath	6	A 2D/3D math library
Format	4	Format library such as dates, number formats
Promise	5	Library for lightweight promises and futures
Culture	5	Localization library for Haxe
Math	5	Generation of random values

Table 4.3: Description of selected benchmark libraries

We use then these test suites to transform functional tests into stress tests. This can be useful to study the impact of this load on the resource usage properties of the five target versions. We run each test suite 1K times to get comparable values in terms of resource usage. Table 4.3 describes the Haxe libraries that we have selected in this benchmark to evaluate our approach and the number of test suites used per benchmark. In total, we have 95 test suites to execute across all benchmark programs.

4.4.1.3 Evaluation metrics used

We evaluate the efficiency of generated code using the following non-functional metrics:

[11] http://thx-lib.org/

-*Memory usage*: It corresponds to the maximum memory consumption of the running test suite. Memory usage is measured in MB

-*Execution time*: The execution time of test suites is measured in seconds.

We recall that our testing infrastructure is able to evaluate other non-functional properties of generated code such as code generation time, compilation time, code size, CPU usage. We choose to focus, in this experiment, on the performance (*i.e.*, execution time) and resource usage (*i.e.*, memory usage). Collecting resource usage metrics is ensured by our monitoring infrastructure, presented in Chapter 6.

4.4.1.4 Setting up infrastructure

To assess our approach, we configure our previously proposed container-based infrastructure in order to run experiments on the Haxe case study. Figure 4.5 shows a big picture of the testing infrastructure considered in these experiments.

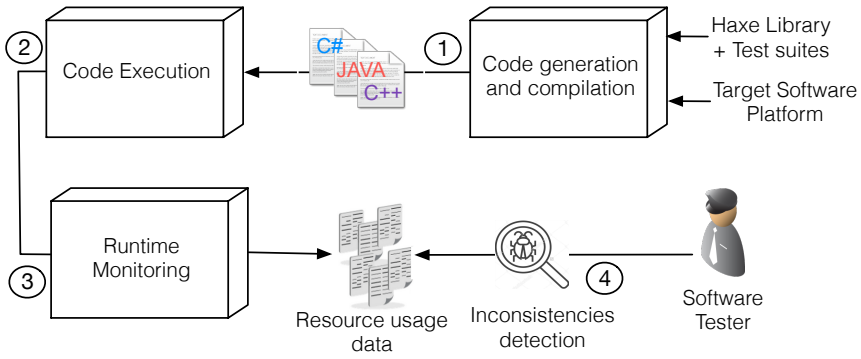


Figure 4.5: Infrastructure settings for running experiments

First, a first component is created in where we install the Haxe code generators and compilers. It takes as an input the Haxe library we would evaluate and the list of test suites (Step 1). It produces as an output the source code files relative to the target software platforms. Afterwards, generated files are compiled (if needed) and automatically executed within the execution container (Step 2). This component is a pre-configured container instance where we install the required execution environments such as PHP interpreter, node (for JS), mono (for C#), etc. In the meantime, while running test suites inside the container, we collect runtime resource usage data (Step 3). Chapter 6 presents more details about the monitoring engine. Finally, in Step 4, we analyze the non-functional data in order to detect code generator inconsistencies.

4.4.2 Experimental methodology and results

In the following paragraphs, we report the methodology we used to answer **RQ1** and the results of our experiments.

4.4.2.1 Method

We now conduct experiments based on the new created benchmark libraries. The goal of running these experiments is to observe and compare the behavior of generated code using the defined MR in order to detect code generator inconsistencies.

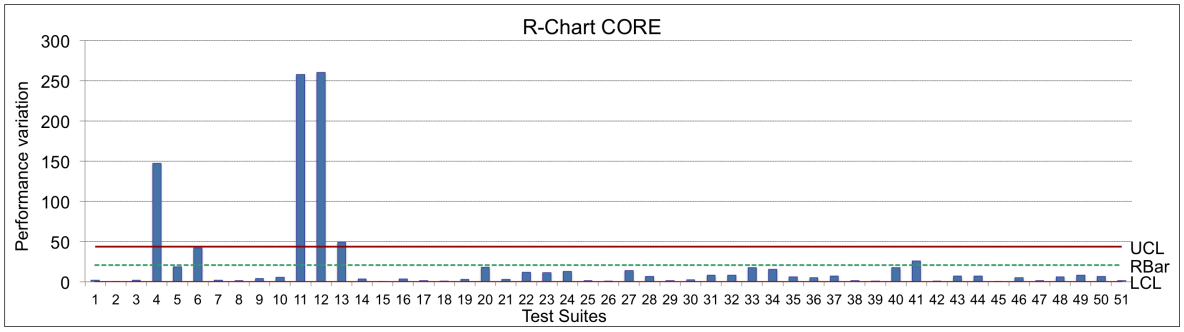
Therefore, we set up, first, our container-based infrastructure as it is presented in Section 4.3.1 in order to generate, execute, and collect the memory usage of our test suites. Afterwards, we prepare and normalize the gathered data to make it valuable for the statistical analysis. Then, we conduct the R-chart and PCA analysis as described in Section 4.3.2.5 in order to analyze the performance and resource usage variations. This will lead us to define an appropriate formula of the MR, used to automatically detect inconsistencies within code generator families (Section 4.3.2.4). Finally, we report the inconsistencies we have detected.

4.4.2.2 Results

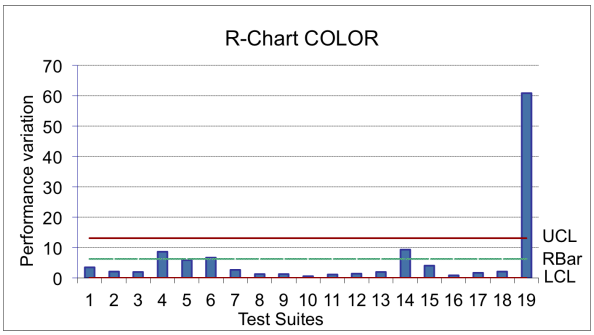
R-chart results

The results of R-charts for the seven benchmark programs relative to the performance and resource usage variations are reported in Figures 4.6 and 4.7. In Figure 4.6, we report the performance variation corresponding to the range difference R between the maximum and minimum execution time of each test suite across the five targets (Java, JS, C++, C#, and PHP). Data is normalized, dividing all values by the minimum execution time per test suite. The LCL for our experiments is always equal to 0 because the D_3 constant value as defined in Equation 4.5, is equal to zero according to the R-chart constants table[12]. In fact, the D_3 constant changes depending on the number of subgroups. In our experiments, our data record is composed of five subgroups corresponding to the five target programming languages. The central line (in green) corresponds to *R-bar*. This value changes from one benchmark to another depending on the average of R for all test suites in the benchmark.

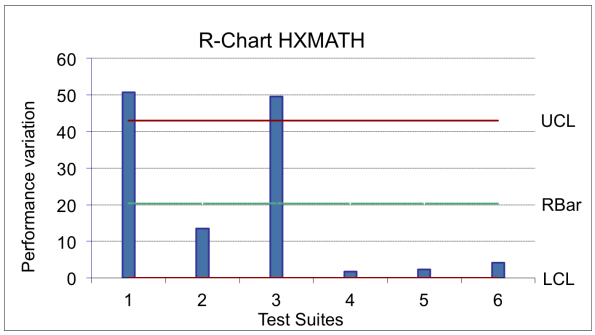
[12] http://www.bessegato.com.br/UFJF/resources/table_of_control_chart_constants_old.pdf



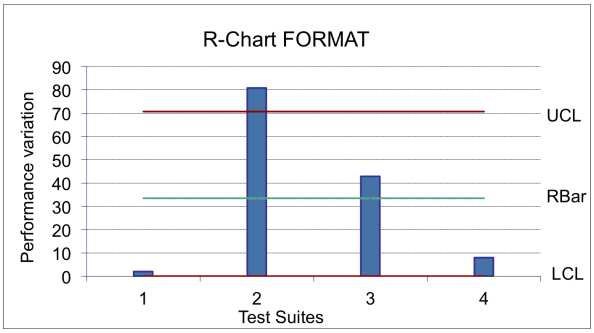
(a) R-chart of the Core benchmark program



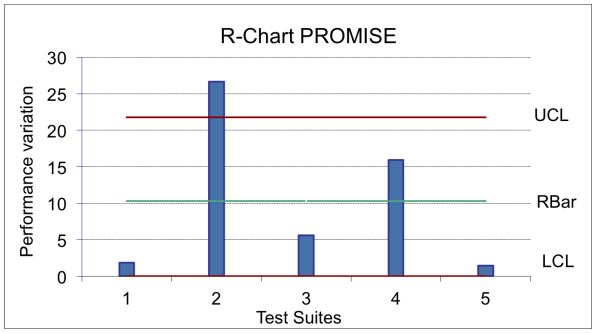
(b) R-chart of the Color benchmark program



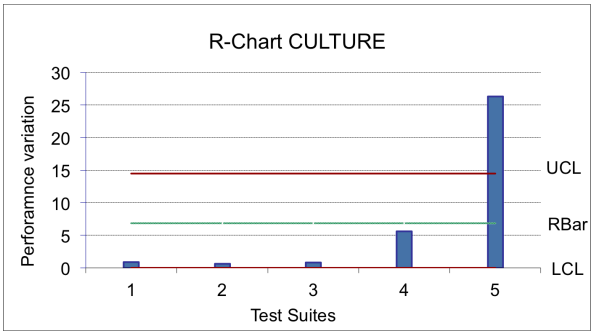
(c) R-chart of the Hxmath benchmark program



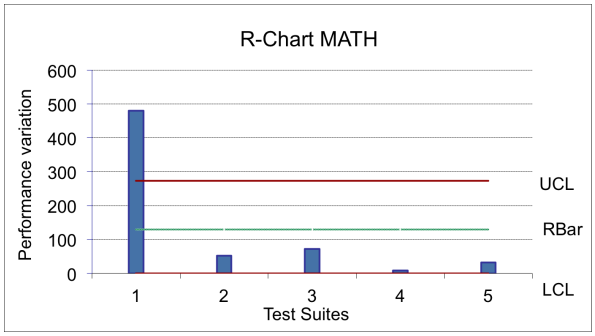
(d) R-chart of the Format benchmark program



(e) R-chart of the Promise benchmark program

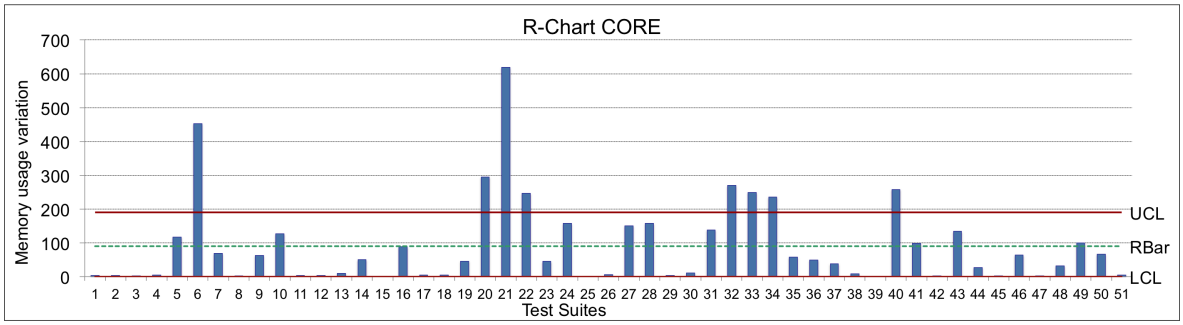


(f) R-chart of the Culture benchmark program

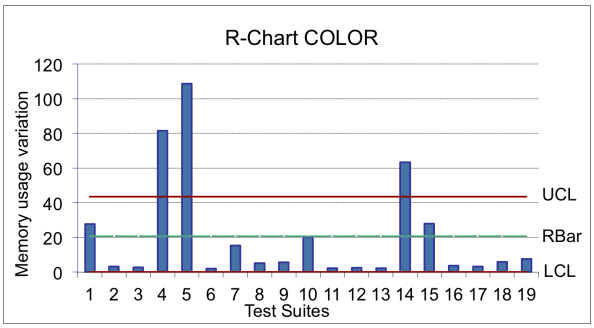


(g) R-chart of the Math benchmark program

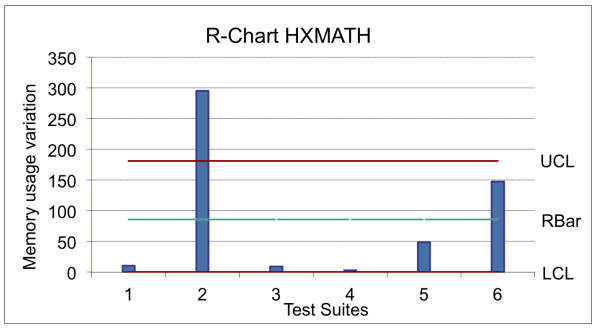
Figure 4.6: Performance variation of test suites across the different Haxe benchmarks



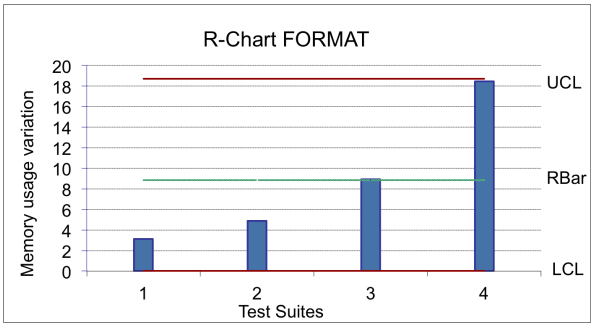
(a) R-chart of the Core benchmark program



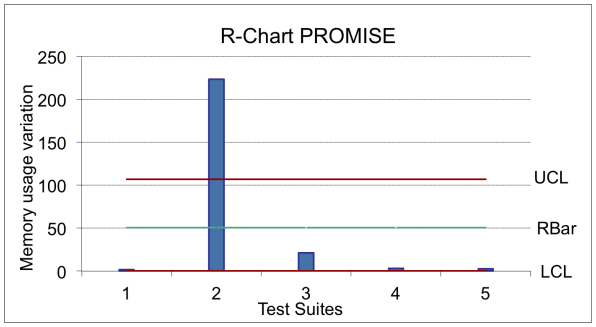
(b) R-chart of the Color benchmark program



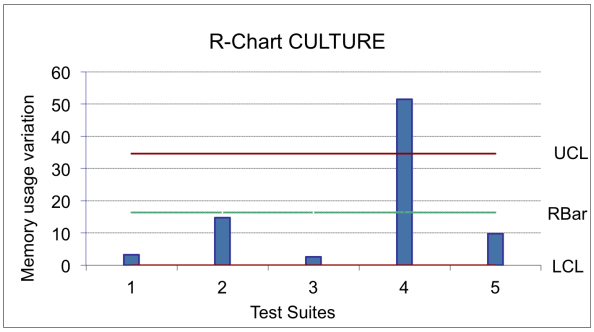
(c) R-chart of the Hxmath benchmark program



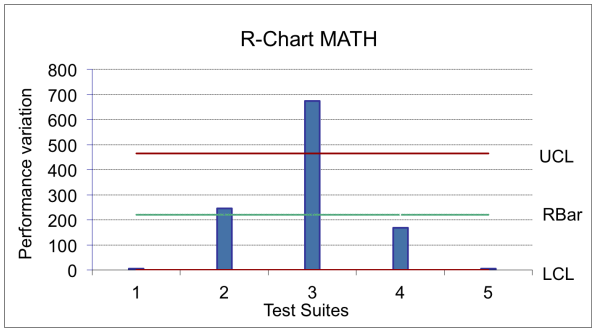
(d) R-chart of the Format benchmark program



(e) R-chart of the Promise benchmark program



(f) R-chart of the Culture benchmark program



(g) R-chart of the Math benchmark program

Figure 4.7: Memory usage variation of test suites across the different Haxe benchmarks

As a consequence, UCL, which is a function of *R-bar*, changes as long as we add new test suites to the experiments. UCL is equal to $D_4 * \bar{R}$ where $D_4 = 2.114$ according to the R-chart constants table. We recall that the average variation *R-bar* and the threshold value UCL change dynamically by adding new test suites to the corresponding benchmark. We note as well that these parameter values are appropriate to each benchmark program. We made the threshold values specific for each benchmark because we believe that the variation is highly dependent on the application domain and on the program under test. The R-charts used for visualizing the memory usage variation follow the same concept as we have just been describing for performance variation.

Results in Figure 4.6, show that most of the performance variations are in the interval $[0 - UCL]$, which corresponds to *in-control* variation zone as it is described in Section 4.3.2.5. However, we remark for several test suites that the performance variation becomes relatively high (higher than the UCL value of the corresponding benchmark program). We detect 11 among 95 performance deviations lying in the *out of control* variation zone. For the other test suites, the variation is even less than the total average variations *R-bar*. There are only 7 test suites among the remaining 84 ones where the variation lies in the interval $[\bar{R} - UCL]$. This variation is high but we are not detecting it as a performance deviation because according to the R-chart, variation in this zone is still *in-control*. The 11 performance deviations we have detected can be explained by the fact that the execution time of one or more test suites varies considerably from one language to another. This argues the idea that the code generator has produced a suspect code behavior, which led to a high performance variation. We provide later better explanation about the faulty code generators.

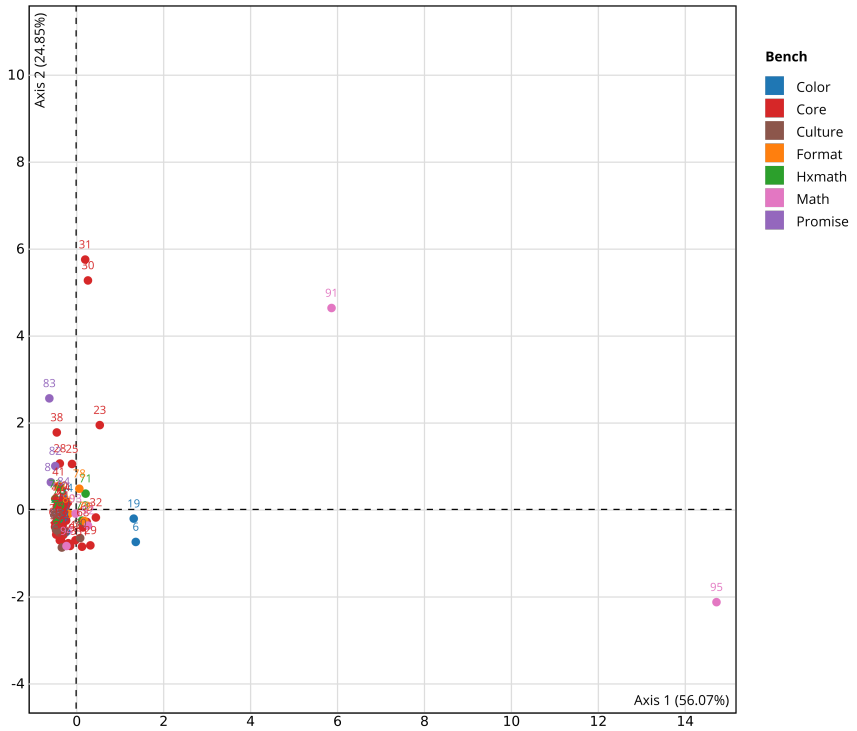
Similarly, Figure 4.7 resumes the comparison results of test suites execution regarding the memory usage. The variation in this experiment are more important than previous results. This can be argued by the fact that the memory utilization and allocation patterns are different from one language to another. Nevertheless, we can recognize some points where the variation is extremely high. Thus, we detect 15 among 95 test suites that exceed the corresponding UCL value. When the variation is below UCL, we detect 14 among the 80 remaining test suites where the variation lies in the interval $[\bar{R} - UCL]$, which is relatively high. One of the reasons that caused this variation may occur when the test suite executes some parts of the code (in a specific language) that strangely consume a lot of resources. This may not be the case when the variation is lower than the $\bar{R}$ for example.

To resume, we have detected 11 extreme performance variations and 15 extreme memory usage variations among the 95 executed test suites. We assume then, that faulty code generators, in identified points, represent a threat for software quality since the generated code has shown symptoms of poor-quality design.

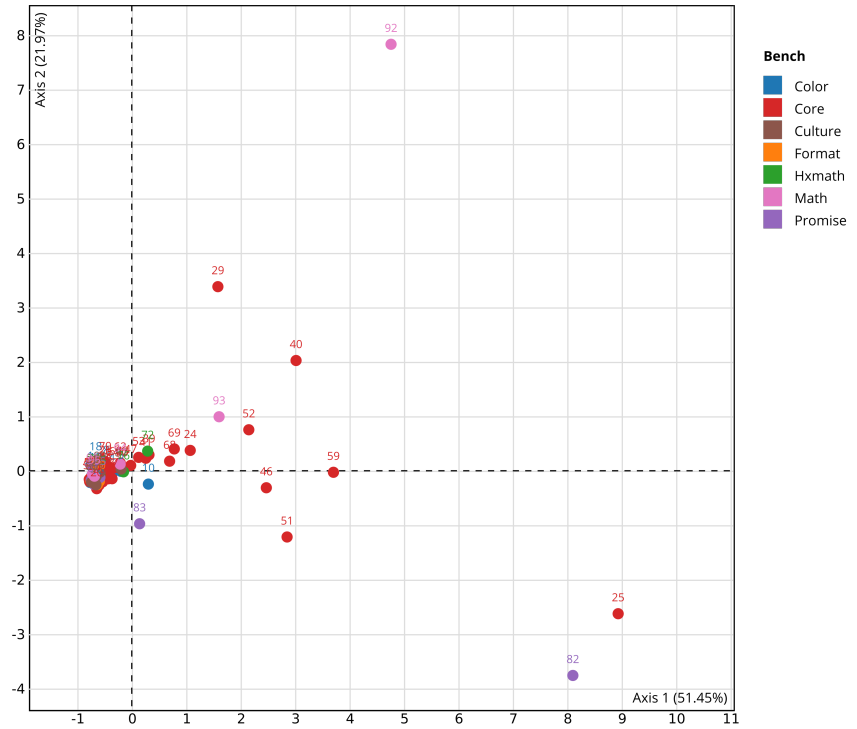
PCA results

We apply the PCA approach as an alternative to the R-chart approach. Figure 4.8 shows the dispersion of our data points in the PC subspace. PC1 et PC2 represent the directions of our two first principal components, having the highest orthogonal variations. Our data points represent the performance variation (Figure 4.8(a)) and the memory usage variation (Figure 4.8(b)) of the 95 test suites we have executed. Variation points are colored according to the benchmark program they belong to (displayed in the figure legend). At the first glance, we can clearly see that the variation points are situated in the same area except some points that lie far from this data cluster. In Figure 4.8(a), the pink points corresponding to the Math benchmark show visually the largest variation. The three Core test suites (in red) which are identified as performance deviations in R-chat, show also a deviation in the PCA scatter plot. Points 91 relative to the Math benchmark is deviating from the cloud point. However, in the R-chart diagram, it is not detected as a performance deviation (see the test suite 3 of Figure 4.6(g)). In fact, this test suite takes more than 80 times to run. Compared to other test suites, the performance variation does not exceed 80. In effect, PCA performs a complete analysis of the whole data we have collected in all benchmarks. Thus, variations are displayed with respect to all test suites variations in all benchmarks. The variation evaluation is not limited within the benchmark program as we used to do using R-charts. We report the same results in Figure 4.8(b) about the memory usage variation in the PCA.

To confirm this observation, we present in Figure 4.9, the results of our outliers detection approach. We identify 4 inconsistencies (or outliers) in each diagnostic plot. Inconsistencies in Figure 4.8(a) are relative to the performance deviations. Points 31 and 32 correspond to the test suites 12 and 11 in benchmark Core of Figure 4.6(a). Points 91 and 95 correspond to the test suites 3 and 1 in benchmark Math of Figure 4.6(g). For memory usage variation, we detect points 25, 29, 82, and 92 which corresponds relatively to the test suites 21 and 6 of benchmark Core, 2 of benchmark Promise, and 3 of benchmark Math. We can clearly see that this technique helps to identify extreme-value outliers, which are mostly covered by the R-chart approach. We used 97.5%-Quantile of the Chi-square distribution to define the cutoff value which is commonly used in the literature [ELB+08, HRV09]. If we decrease this value we will be able to detect more variation points.

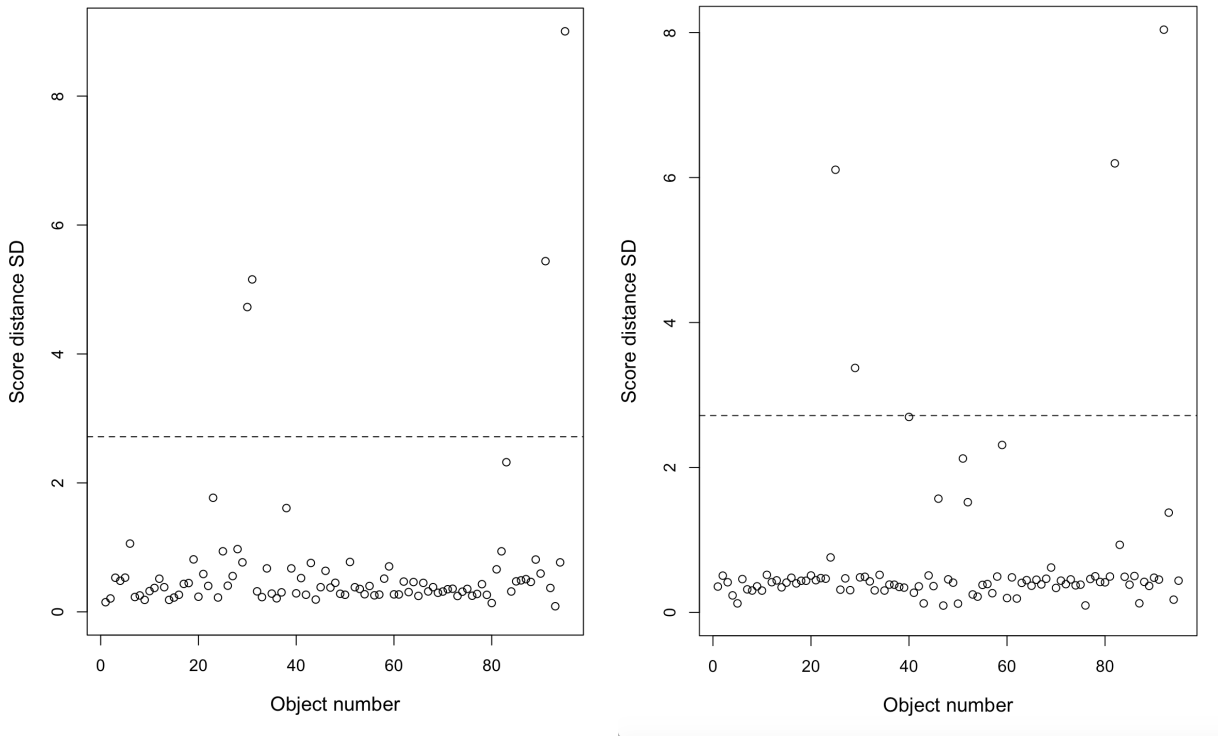


(a) Test suites relative to the execution times



(b) Test suites relative to the memory consumptions

Figure 4.8: PCAs showing the dispersion of our data over the PC subspace



(a) Performance deviations

(b) Memory usage deviations

Figure 4.9: Diagnostic plots using score distance SD. The vertical lines indicate critical values separating regular observations from outliers (97.5%)

Detected inconsistencies

Now that we have observed the performance and memory usage variations of test suites execution, we can analyze the extreme points we have detected using R-chart to observe in greater depth the source of such deviation. For that reason, we present in Table 4.4 and 4.5 the raw data values of these test suites leading to an extreme variation in terms of execution time and memory usage. We report the inconsistencies gathered from the first approach, R-chart.

Table 4.4 shows the execution time factor of each test suite execution in a specific target language. This factor is scaled with respect to the lowest execution time among the five

Benchamrk	Test Suite	Java	JS	CPP	CS	PHP	UCL(R)	Defective CG
Color	**TS19**	1.90	1	2.37	3.31	61.84	13.08	PHP
Core	**TS4**	1	1.59	1.67	2.78	148.20	43.62	PHP
		1.14	2.71	1	3.63	258.94		PHP
		1.28	2.94	1	3.36	261.36		PHP
		1	1.05	1.86	2.39	50.30		PHP
Hxmath	**TS1**	2.38	1.43	1	2.82	51.72	42.97	PHP
		2.14	1.10	1	2.25	50.56		PHP
Format	**TS2**	1.16	1.27	1	3.35	81.85	70.66	PHP
Promise	**TS2**	1.52	1.85	1	1.51	27.67	21.76	PHP
Culture	**TS5**	1.62	1	1.27	2.02	27.29	14.47	PHP
Math	**TS1**	4.15	1	5.41	4.70	481.68	273.24	PHP

Table 4.4: Raw data values of test suites that led to the highest variation in terms of execution time

targets. We also report the UCL defined per benchmark. In the last column, we report the code generator that caused such large deviation. To do so, we designate by *defective CG*, the code generator that led to a performance variation higher than the UCL value.

We can clearly see that the PHP code has a singular behavior regarding the performance with a factor ranging from x27.29 for test suite 5 in benchmark Culture (Culture_TS5) to x481.7 for Math_TS1. For example, if Math_TS1 takes 1 minute to run in JS, the same test suite in PHP will take around 8 hours to run which is a very large gap. The highest factor detected for other languages is x5.41 which is not negligible but it represents a small deviation compared to PHP deviations. While it is true that we are comparing different versions of generated code, it was expected to get some variations while running test cases in terms of execution time. However, in the case of PHP code generator, it is far to be a simple variation, but it is a code generator inconsistency that led to such performance regression.

Meanwhile, we gather information about the points that led to the highest variation in terms of memory usage. Table 4.5 shows these results. Again, we can identify a singular behavior of the PHP code regarding the memory usage with a factor ranging from x52.47 to x675. For other test suites versions, the factor varies from x1 to x160.84. We observe as well a singular behavior of the Java code for Core_TS6, Core_TS32, and Promise_TS2, yielding to a variation higher than the UCL. These results prove that the PHP and Java code generators are not always effective and they constitute a threat for the generated software in terms of memory usage.

Benchmark	Test suite	Java	JS	CPP	CS	PHP	UCL	Defective CG
Color	**TS4**	1	2.29	1.47	3.59	82.46	43.53	PHP
	TS5	1	3.08	1.83	4.53	109.69		PHP
	TS14	1	1.32	1.00	2.03	64.45		PHP
Core	**TS6**	250.77	71.71	1	69.90	454.15	190.03	PHP & Java
	TS20	2.31	1.34	1	3.27	296.10		PHP
	TS21	11.90	1	14.63	36.18	620.22		PHP
	TS22	1	2.70	1.74	4.69	247.32		PHP
	TS32	270.78	2.27	1	5.61	153.37		Java
	TS33	1.82	1.12	1	54.19	250.35		PHP
	TS34	1	1.17	1.48	3.90	236.97		PHP
	TS40	160.84	1.10	1	49.43	259.20		PHP
Hxmath	**TS2**	1	1.16	1.91	2.82	296.16	181.11	PHP
Promise	**TS2**	214.53	92.45	1	57.68	224.41	106.82	PHP & Java
Culture	**TS4**	2.75	1.01	2.52	1	52.47	34.63	PHP
Math	**TS3**	1.29	1	1.72	3.60	675.00	464.80	PHP

Table 4.5: Raw data values of test suites that led to the highest variation in terms of memory usage

To give more insights about the source of this issue, we provide in the following further analysis of these inconsistencies.

4.4.2.3 Analysis

These inconsistencies need to be fixed by code generator creators in order to enhance the quality of generated code (PHP code for example). Since we are proposing a black-box testing approach, our solution is not able to provide more precise and detailed information about the part of code causing these performance issues, which is one of the limitations of our testing approach.

Therefore, to understand this unexpected behavior of the PHP code when applying the test suite Core_TS4 for example, we looked (manually) into the PHP code corresponding to this test suite. In fact, we observe the intensive use of *"arrays"* in most of the functions under test. Arrays are known to be slow in PHP and PHP library has introduced much more advanced functions such as $array\_fill$ and specialized abstract types such as *"SplFixedArray"*[13] to overcome this limitation. So, by just changing these two parts in the

[13] http://php.net/manual/fr/class.splfixedarray.php

generated code, we improve the PHP code speed with a factor x5 which is very valuable. We also reduce the memory usage of this test suite by a factor of x2.

In short, the lack of use of specific types, in native PHP standard library, by the PHP code generator such as *SplFixedArray* shows a real impact on the non-functional behavior of generated code. Obviously, the types used during code generation are not the best ones. In contrast, selecting carefully the adequate types and functions to generate code can lead to performance improvement.

4.4.3 Threats to validity

We resume, in the following paragraphs, external and internal threats that can be raised:

External validity refers to the generalizability of our findings. In this study, we perform experiments on Haxe and on a set of test suite selected from Github and from the Haxe community. For instance, we have no guarantee that these libraries cover all Haxe language features. Consequently, we cannot guarantee that our approach is able to find all the code generators issues unless we develop a more comprehensive test suite. Moreover, the threshold defined to detect the singular performance behavior has a huge impact on the precision and recall of the proposed approach. Experiments should be replicated to other case studies to confirm our findings.

Internal validity is concerned with the use of a container-based approach. Even if it exists emulators such as Qemu[14] that allow to reflect the behavior of heterogeneous hardware, the chosen infrastructure has not been evaluated to test generated code that target heterogeneous hardware machines. In addition, even though system containers are known to be lightweight and less resource-intensive compared to full-stack virtualization, we would validate the reliability of our approach by comparing it with a non-virtualized approach in order to see the impact of using containers on the accuracy of the results.

[14] https://goo.gl/SxKG1e

4.5 Conclusion

In this work we have described a metamorphic testing approach for automatic detection of code generator inconsistencies. Our approach is based on the intuition that a code generator is often a member of a family of code generators. Therefore, we benefit from the existence of multiple generators with comparable functionality (*i.e.*, code generator families) to apply the idea of metamorphic testing, defining high-level test oracles (*i.e.*, metamorphic relations) as test oracles. We define the metamorphic relation as a comparison between the variations of performance and resource usage of code, generated from the same code generator family. Any variation that exceeds a specific threshold value is automatically detected as an anomaly. We apply two statistical methods (*i.e.*, principal component analysis and range-charts) in order to automate the inconsistencies detection. We evaluate our approach by analyzing the performance of Haxe, a popular high-level programming language that involves a set of cross-platform code generators. We evaluate the properties related to the resource usage and performance for five different target software platforms. We run a bench of test suites across 7 Haxe benchmark libraries in order to verify the metamorphic relation (*i.e.*, the performance and memory usage variation) for each of them. Experimental results show that our approach is able to detect, among 95 executed test suites, 11 performance and 15 memory usage inconsistencies, violating the metamorphic relation. We answered RQ1, showing that our approach can automatically detect real issues in code generator families. In particular, we show that we could find at least two kinds of errors: the lack of use of a specific function and an abstract type that exist in the standard library of the target language which can reduce the memory usage/execution time of the resulting program. These issues need to be investigated and fixed by code generator maintainers/experts.

Chapter 5

NOTICE: An approach for auto-tuning compilers

Ensuring the code quality during software development is very important in software engineering. It provides facilities to the software developers to maintain, test, and debug their source code. When it comes to the compiler level, ensuring the quality of generated code highly depends on the applied configurations (*i.e.*, optimizations) to the compiler.

However, as discussed in Chapter 2, compiler tuning is challenging because of the huge number of potential optimization combinations, making it hard and time-consuming for software developers to find/construct the sequence of optimizations that satisfies user specific non-functional requirements. It also requires a comprehensive understanding of the underlying system architecture, the target application, and the available compiler optimizations. We also note that when trying to optimize software performance, many non-functional properties and design constraints must be involved and satisfied simultaneously to better optimize code. Sometimes, improving program execution time can result in a high resource usage which may decrease system performance. For example, embedded systems for which code is generated often have limited resources. Thus, optimization techniques must be applied whenever possible to generate efficient code and improve performance (in terms of execution time) with respect to available resources (CPU or memory usage) [NF13]. Therefore, it is important to construct optimization levels that represent multiple trade-offs between non-functional properties, enabling the software designer to choose among different optimal solutions which best suit the system specifications.

As discussed in the state of the art chapter, there are many approaches that address these optimization issues in order to help users configuring (or tuning) compilers with

respect to many non-functional properties such as code size, energy consumption, execution time, etc.

This chapter presents an alternative approach to previous research efforts. We present NOTICE, an approach for automatically tuning compilers. Our approach is based on micro-services to automate the deployment and monitoring of different variants of optimized code. NOTICE is an on-demand tool that employs mono and multi-objective evolutionary search algorithms to construct optimization sequences that satisfy user key objectives (execution time, code size, compilation time, CPU or memory usage, etc.). In this chapter, we make the following contributions:

- We introduce a novel formulation, compared to previous related work, of the compiler optimization problem using Novelty Search [LS08]. NS is applied to tackle the problem of optimizations diversity and then, providing a new way to explore the huge optimization search space.
- We also demonstrate that NOTICE can be used to automatically construct optimization levels that represent optimal trade-offs between multiple non-functional properties. In our approach, we study the relationship between the runtime execution of optimized code and the resource consumption profiles (CPU and memory usage) by providing a fine-grained understanding and analysis of compilers behavior regarding optimizations. Thus, we study the trade-offs execution time/memory usage, etc.
- We conduct an empirical study to evaluate the effectiveness of our approach by verifying the optimizations performed by the GCC compiler. Our experimental results show that NOTICE is able to auto-tune compilers according to user choices (heuristics, objectives, programs, etc.) and construct optimizations that yield to better performance results than standard optimization levels using mono-objective and multi-objective optimizations. We also demonstrate that NOTICE can be used to automatically construct optimization levels that represent optimal trade-offs between the speedup and memory usage.

This chapter is organized as follows:

Section 5.1 describes the motivation and the challenges behind this work. We present in this section the GCC compiler as a motivation example to better explain the problem. The GCC compiler will also be used by NOTICE to evaluate and validate our approach.

In Section 5.2, the proposed search-based technique, *i.e.*, Novelty Search (NS), is presented. We describe our NS adaptation to the compiler auto-tuning problem. Thus, we

present in details the algorithm, the evaluation metrics and the iterative evolutionary process.

In Section 5.3, we conduct an empirical study to evaluate our approach. Thus, we evaluate the implementation of our approach by explaining the design of our experiments, the research questions we set out to answer and the methods we used to answer these questions.

Finally, discussions and concluding remarks are provided in Section 5.4.

5.1 Motivation

In the past, researchers have shown that the choice of optimization sequences may influence software performance [ACG+04, CFH+12]. As a consequence, software-performance optimization becomes a key objective for both software industries and developers, which are often willing to pay additional costs to meet specific performance goals, especially for resource-constrained systems.

Universal and predefined sequences, *e.g.*, O1 to O3 in GCC, may not always produce good performance results and may be highly dependent on the benchmark and the source code they have been tested on [HE08, CHE+10, EAC15]. Indeed, each one of these optimizations interacts with the code and in turn, with all other optimizations in complicated ways. Similarly, code transformations can either create or eliminate opportunities for other transformations and it is quite difficult for users to predict the effectiveness of optimizations on their source code program. As a result, most software engineering programmers that are not familiar with compiler optimizations find difficulties to select effective optimization sequences [ACG+04].

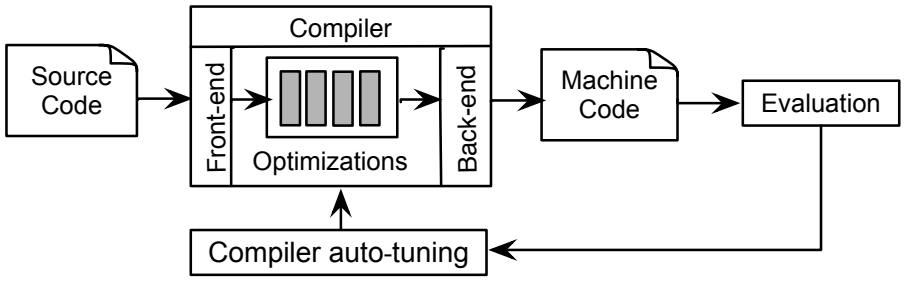


Figure 5.1: Process of compiler optimization exploration

To explore the large optimization space, users have to evaluate the effect of optimizations according to a specific non-functional objective (see Figure 5.1). Optimizations influence on different properties such as execution time, compilation time, resource consumption, code size, etc. Thus, finding the optimal optimization combination for an input source code is a challenging and time-consuming problem.

It is important to notice that performing optimizations to source code can be so expensive at resource usage that it may induce compiler bugs or crashes. Indeed, in a resource-constrained environment and because of insufficient resources, compiler optimizations can lead to memory leaks or execution crashes [YCER11]. Thus, it becomes necessary to test the non-functional properties of optimized code and check its behavior regarding optimizations which can lead to performance improvement or regression.

Example: GCC compiler

The GNU Compiler Collection, GCC, is a very popular collection of programming compilers, available for different platforms. GCC exposes its various optimizations via a number of flags that can be turned on or off through command-line compiler switches.

For instance, version 4.8.4 provides a wide range of command-line options that can be enabled or disabled, including more than 150 options for optimization. The diversity of available optimization options makes the design space for optimization level very huge, increasing the need for heuristics to explore the search space of feasible optimization sequences. As it is shown in Table 5.1, we count 76 optimization flags that are enabled by the four default optimization levels (O1, O2, O3, Ofast). Each standard level is composed by a number of optimizations. These levels are defined by compiler designers based on their experiences and preliminary experiments. The goal of defining these standard levels is to build general and program independent sequences that represent trade-offs among several non-functional properties.

For instance, O1 enables the optimization flags that reduce the code size and execution time without performing any optimization that reduces the compilation time. It turns on 32 flags. O2 increases the compilation time and reduces the execution time of generated code. It turns on all optimization flags specified by O1 plus 35 other options. O3 is more aggressive level which enables all O2 options plus 8 more optimizations. Finally, Ofast is the most aggressive level which enables optimizations that are not valid for all standard-compliant programs. It turns on all O3 optimizations plus one more aggressive optimization. This results in a huge space with 2^{76} possible optimization combinations.

Table 5.1: Compiler optimization options enabled by GCC standard levels

Level	Optimization option	Level	Optimization option
O1	-fauto-inc-dec	O2	-fthread-jumps
	-fcompare-elim		-falign-functions
	-fcprop-registers		-falign-jumps
	-fdce		-falign-loops
	-fdefer-pop		-falign-labels
	-fdelayed-branch		-fcaller-saves
	-fdse		-fcrossjumping
	-fguess-branch-probability		-fcse-follow-jumps
	-fif-conversion2		-fcse-skip-blocks
	-fif-conversion		-fdelete-null-pointer-checks
	-fipa-pure-const		-fdevirtualize
	-fipa-profile		-fexpensive-optimizations
	-fipa-reference		-fgcse
	-fmerge-constants		-fgcse-lm
	-fsplit-wide-types		-fhoist-adjacent-loads
	-ftree-bit-ccp		-finline-small-functions
	-ftree-builtin-call-dce		-findirect-inlining
	-ftree-ccp		-fipa-sra
	-ftree-ch		-foptimize-sibling-calls
	-ftree-copyrename		-fpartial-inlining
	-ftree-dce		-fpeephole2
	-ftree-dominator-opts		-fregmove
	-ftree-dse		-freorder-blocks
	-ftree-forwprop		-freorder-functions
	-ftree-fre		-frerun-cse-after-loop
	-ftree-phiprop		-fsched-interblock
	-ftree-slsr		-fsched-spec
	-ftree-sra		-fschedule-insns
	-ftree-pta		-fschedule-insns2
	-ftree-ter		-fstrict-aliasing
	-funit-at-a-time		-fstrict-overflow
			-ftree-switch-conversion
O3	-finline-functions		-ftree-tail-merge
	-funswitch-loops		-ftree-pre
	-fpredictive-commoning		-ftree-vrp
	-fgcse-after-reload		
	-ftree-vectorize		
	-fvect-cost-model		
	-ftree-partial-pre		
	-fipa-cp-clone		
Ofast	-ffast-math		

Optimization flags in GCC can be turned off by using *"-fno-"*+flag instead of *"-f"*+flag in the beginning of each optimization. We use this technique to play with compiler switches.

5.2 Evolutionary exploration of compiler optimizations

Many techniques (meta-heuristics, random search, etc.) can be used to explore the large set of optimization combinations of modern compilers. In our approach, we particularly study the use of the Novelty Search (NS) technique to identify the set of compiler optimization options that optimize the non-functional properties of code.

5.2.1 Novelty search adaptation

In this work, we aim at providing a new alternative for choosing effective compiler optimization options compared to the state of the art approaches. In fact, since the search space of possible combinations is too large, we aim at using a new search-based technique called Novelty Search [LS08] to tackle this issue. The idea of this technique is to explore the search space of possible compiler flag options by considering sequence diversity as a single objective. Instead of having a fitness-based selection that maximizes one of the non-functional objectives, we select optimization sequences based on a novelty score showing how different they are compared to all other combinations evaluated so far. NS is a divergent evolutionary algorithm which rewards optimization sequences that diverge from previously discovered ones. Thus, evolution can be viewed as a divergent process compared to the traditional convergent approaches such GAs that exert the selection pressure based on fitness values.

Moreover, we claim that the search towards effective optimization sequences is not straightforward since the interactions between optimizations is too complex and difficult to define. For instance, in a previous work [CFH+12], Chen *et al.* showed that handful optimizations may lead to higher performance than other techniques of iterative optimization. In fact, the fitness-based search may be trapped into some local optima that cannot escape [BKK+98]. This phenomenon is known as *"diversity loss"*. For example, if the most effective optimization sequence that induces less execution time lies far from the search space defined by the gradient of the fitness function, then some promising search areas may not be reached. The issue of premature convergence to local optima has been a common problem in evolutionary algorithms. As discussed in the state of the art, many methods are proposed to overcome this problem [BFN96]. However, all these efforts use a

fitness-based selection to guide the search. Considering diversity as the unique objective function to be optimized may be a key solution to this problem.

Therefore, during the evolutionary process, we select optimization sequences that remain in sparse regions of the search space in order to guide the search towards novelty. In the meantime, we choose to gather the non-functional metrics relative to the resource consumption (memory and CPU usage) of optimized code.

Algorithm 1: Novelty search algorithm for compiler optimization exploration

Require: Optimization options $\mathcal{O}$
Require: Program $\mathcal{C}$
Require: Novelty threshold $\mathcal{T}$
Require: Limit $\mathcal{L}$
Require: Nearest neighbors $\mathcal{K}$
Require: Number of evaluations $\mathcal{N}$
Ensure: Best optimization sequence *best_sequence*

```
1:  initialize_parameters(L, T, N, K)
2:  create_archive(L)
3:  generated_code ← compile("-O0", C)
4:  minimum_usage ← execute(generated_code)
5:  population ← random_sequences(O)
6:  repeat
7:      for sequence ∈ population do
8:          generated_code ← compile(sequence, C)
9:          memory_usage ← execute(generated_code)
10:         novelty_metric(sequence) ← distFromKnearest(archive, population, K)
11:         if novelty_metric > T then
12:             archive ← archive ∪ sequence
13:         end if
14:         if memory_usage < minimum_usage then
15:             best_sequence ← sequence
16:             minimum_usage ← memory_usage
17:         end if
18:     end for
19:     new_population ← generate_new_population(population)
20:     generation ← generation + 1
21: until generation = N
22: return best_sequence
```

Generally, NS acts like GAs (Example of GA use in [CST02]). However, NS needs extra changes. First, a new novelty metric is required to replace the fitness function. Then, an archive must be added to the algorithm, which is a kind of a database that remembers individuals that were highly novel when they were discovered in past generations. Algo-

rithm 1 describes the overall idea of our NS adaptation. The algorithm takes as input a source code program and a list of optimizations.

We initialize first the novelty parameters and create a new archive with limit size L (lines 1 & 2). In this example, we gather information about memory consumption. In lines 3 & 4, we compile and execute the input program without any optimization (O0). Then, we measure the resulting memory consumption. By doing so, we will be able to compare it to the memory consumption of new generated solutions. The best solution is the one that yields to the lowest memory consumption compared to O0 usage. Before starting the evolutionary process, we generate an initial population with random sequences. Lines 6-21 encode the main NS loop, which searches for the best sequence in terms of memory consumption. For each sequence in the population, we compile the input program, execute it and evaluate the solution by calculating the average distance from its k-nearest neighbors. Sequences that get a novelty metric higher than the novelty threshold T are added to the archive. T defines the threshold for how novel a sequence has to be before it is added to the archive. In the meantime, we check if the optimization sequence yields to the lowest memory consumption so that, we can consider it as the best solution. Finally, genetic operators (mutation and crossover) are applied afterwards to fulfill the next population. This process is iterated until reaching the maximum number of evaluations.

5.2.1.1 Optimization sequence representation

For our case study, a candidate solution represents all compiler switches that are used in the four standard optimization levels (O1, O2, O3 and Ofast). Thereby, we represent this solution as a vector where each dimension is a compiler flag. The variables that represent compiler options are represented as genes in a chromosome. Thus, a solution represents the CFLAGS value used by GCC to compile programs. A solution has always the same size, which corresponds to the total number of involved flags. However, during the evolutionary process, these flags are turned on or off depending on the mutation and crossover operators (see example in Figure 5.2). As well, we keep the same order of invoking compiler flags since that does not affect the optimization process and it is handled internally by GCC.

5.2.1.2 Novelty metric

The Novelty metric expresses the sparseness of an input optimization sequence. It measures its distance to all other sequences in the current population and to all sequences that were discovered in the past (*i.e.*, sequences in the archive). We can quantify the sparseness of a solution as the average distance to the k-nearest neighbors.

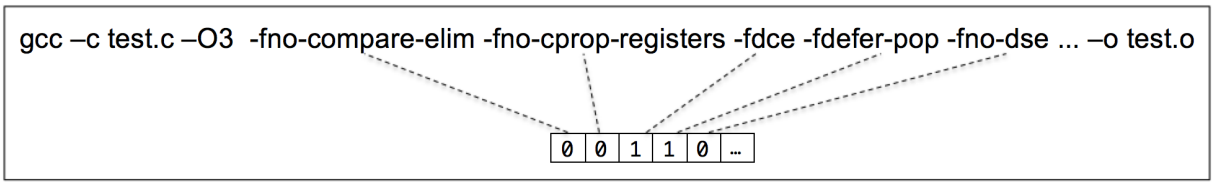


Figure 5.2: Solution representation

If the average distance to a given point's nearest neighbors is large then it belongs to a sparse area and will get a high novelty score. Otherwise, if the average distance is small so it belongs certainly to a dense region then it will get a low novelty score. The distance between two sequences is computed as the total number of symmetric differences among optimization sequences. Formally, we define this distance as follows :

$$distance(S1, S2) = |S1 \triangle S2| \tag{5.1}$$

where $S1$ and $S2$ are two selected optimization sequences (solutions). The distance value is equal to 0 if the two optimization sequences are similar and higher than 0 if there is at least one optimization difference. The maximum distance value is equal to the total number of input flags.

To measure the sparseness of a solution, we use the previously defined distance to compute the average distance of a sequence to its k-nearest neighbors. In this context, we define the novelty metric of a particular solution as follows:

$$NM(S) = \frac{1}{k}\sum_{i=1}^{k} distance(S, \mu_i) \tag{5.2}$$

where μ_i is the i^{th} nearest neighbor of the solution S within the population and the archive of novel individuals.

5.2.2 Novelty search for multi-objective optimization

A multi-objective approach provides a trade-off between two objectives where the developers can select their desired solution from the Pareto-optimal front. The idea of this approach is to use multi-objective algorithms to find trade-offs between non-functional properties of generated code such as <*ExecutionTime–MemoryUsage*>. The correlations

we are trying to investigate are more related to the trade-offs between resource consumption and execution time.

For instance, NS can be easily adapted to multi-objective problems. In this adaptation, the SBSE formulation remains the same as described in Algorithm 1. However, in order to evaluate the new discovered solutions, we have to consider two main objectives and add the non-dominated solutions to the Pareto non-dominated set. We apply the Pareto dominance relation to find solutions that are not Pareto dominated by any other solution discovered so far, like in NSGA-II [LPF+10, DPAM02]. Then, this Pareto non-dominated set is returned as a result. There is typically more than one optimal solution at the end of NS. The maximum size of the final Pareto set cannot exceed the size of the initial population.

5.3 Evaluation

So far, we have presented a sound procedure for auto-tuning compilers through the use of NS. In this section, we evaluate the implementation of our approach by explaining the design of our empirical study; the research questions we set out to answer and different methods we used to answer these questions. The experimental material is available for replication purposes[1].

5.3.1 Research questions

Our experiments aim at answering the following research questions:

RQ1: Mono-objective SBSE Validation. *How does the proposed diversity-based exploration of optimization sequences perform compared to other mono-objective algorithms in terms of memory and CPU consumption, execution time, etc.?*

RQ2: Sensitivity. *How sensitive are input programs to compiler optimization options?*

RQ3: Impact of optimizations on resource consumption. *How compiler optimizations impact on the non-functional properties of generated programs?*

RQ4: Trade-offs between non-functional properties. *How can multi-objective approaches be useful to find trade-offs between non-functional properties?*

[1] https://noticegcc.wordpress.com/

To answer these questions, we conduct several experiments using NOTICE to validate our global approach for compiler auto-tuning.

5.3.2 Experimental setup

5.3.2.1 Programs used in the empirical study

To explore the impact of compiler optimizations a set of input programs are needed. To this end, we use a random C program generator called Csmith [YCER11]. Csmith is a tool that can generate random C programs that statically and dynamically conform to the C99 standard. It has been widely used to perform functional testing of compilers [CHH+16, LAS14, NHI13] but not the case for checking non-functional requirements. Csmith can generate C programs that use a much wider range of C features including complex control flow and data structures such as pointers, arrays, and structs. Csmith programs come with their test suites that explore the structure of generated programs (*i.e.*, high quality code coverage). Yang *et al.* [YCER11] argue that Csmith is an effective bug-finding tool because it generates tests that explore atypical combinations of C language features. They also argue that larger programs are more effective for functional testing. Thus, we run Csmith for 24 hours and gathered the largest generated programs. We depicted 111 C programs with an average number of source lines of 12K. 10 programs are used as training set for RQ1, 100 other programs to answer RQ2 and one last program to run RQ4 experiment. The selected Csmith programs are described in more details at [mbo].

Moreover, we run experiments on commonly used benchmarks named Collective Benchmark (cBench) [Fur09]. It is a collection of open-source sequential programs in C targeting specific areas of the embedded market. It comes with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization. cBench contains more than 20 C programs. Table 5.2 describes the programs we have selected from this benchmark to evaluate our approach. These real world benchmark programs are used to study the influence of compiler optimizations on the resource usage in RQ3 experiments.

5.3.2.2 Parameters tuning

An important aspect for meta-heuristic search algorithms lies in the parameters tuning and selection, which are necessary to ensure not only fair comparison, but also for potential replication. NOTICE implements three mono-objective search algorithms (Random

Program	Source lines	Description
automative_susan_s	1376	Image recognition package
bzip2e	5125	Compress any file source code
bzip2d	5125	Decompress zipped files
office_rsynth	4111	Text to speech program produced by integrating various pieces of code
consumer_tiffmedian	15870	Apply the median cut algorithm to data in a TIFF file
consumer_tiffdither	15399	Convert a grey-scale image to bi-level

Table 5.2: Description of selected benchmark programs

Search (RS), NS, and GA [CST02]) and two multi-objective optimizations (NS and NSGA-II [DPAM02]). Each initial population/solution of different algorithms is completely random. The stopping criterion is when the maximum number of fitness evaluations is reached. The resulting parameter values are listed in Table 5.3. The same parameter settings are applied to all algorithms under comparison.

NS, which is our main concern in this work, is implemented as described in Section 5.2.1. During the evolutionary process, each solution is evaluated using the novelty metric. Novelty is calculated for each solution by taking the mean of its 15 nearest optimization sequences in terms of similarity (considering all sequences in the current population and in the archive). Initially, the archive is empty. Novelty distance is normalized in the range [0-100]. Then, to create next populations, an elite of the 10 most novel organisms is copied unchanged, after which the rest of the new population is created by tournament selection according to novelty (tournament size = 2). Standard genetic programming crossover and mutation operators are applied to these novel sequences in order to produce offspring individuals and fulfill the next population (crossover = 0.5, mutation = 0.1). In the meantime, individuals that get a score higher than 30 (threshold T), they are automatically added to the archive as well. In fact, this threshold is dynamic. Every 200 evaluations, we check how many individuals have been copied into the archive. If this number is below 3, the threshold is increased by multiplying it by 0.95, whereas if solutions added to archive are above 3, the threshold is decreased by multiplying it by 1.05. Moreover, as the size of the archive grows, the nearest-neighbor calculation that determines the novelty scores for

Table 5.3: Algorithm parameters

Parameter	Value	Parameter	Value
Novelty nearest-k	15	Tournament size	2
Novelty threshold	30	Mutation prob.	0.1
Max archive size	500	Crossover	0.5
Population size	50	Elitism	10
Individual length	76	Scaling archive prob.	0.05

individuals becomes more computationally demanding. So, to avoid having low accuracy of novelty, we choose to limit the size of the archive (archive size = 500). Hence, it follows a first-in first-out data structure which means that when a new solution gets added, the oldest solution in the novelty archive will be discarded. Thus, we ensure individual diversity by removing old sequences that may no longer be reachable from the current population.

Algorithm parameters were tuned individually in preliminary experiments. The parameter values chosen are the mostly used in the literature [IJH+13, BBSB15]. The value that yielded the highest performance score was chosen.

5.3.2.3 Evaluation metrics used

For mono-objective algorithms, we use to evaluate solutions using the following metrics:

-*Memory Consumption Reduction (MR)*: corresponds to the percentage ratio of memory usage reduction of running container over the baseline. The baseline in our experiments is O0 level, which means a non-optimized code. Larger values for this metric mean better performance. Memory usage is measured in bytes.

-*CPU Consumption Reduction (CR)*: corresponds to the percentage ratio of CPU usage reduction over the baseline. Larger values for this metric mean better performance. The CPU consumption is measured in seconds, as the CPU time.

-*Speedup (S)*: corresponds to the percentage improvement in execution speed of an optimized code compared to the execution time of the baseline version. Program execution time is measured in seconds.

5.3.2.4 Setting up infrastructure

To answer the previous research questions, we configure NOTICE to run different experiments. Figure 5.3 shows a big picture of the testing and monitoring infrastructure

considered in these experiments.

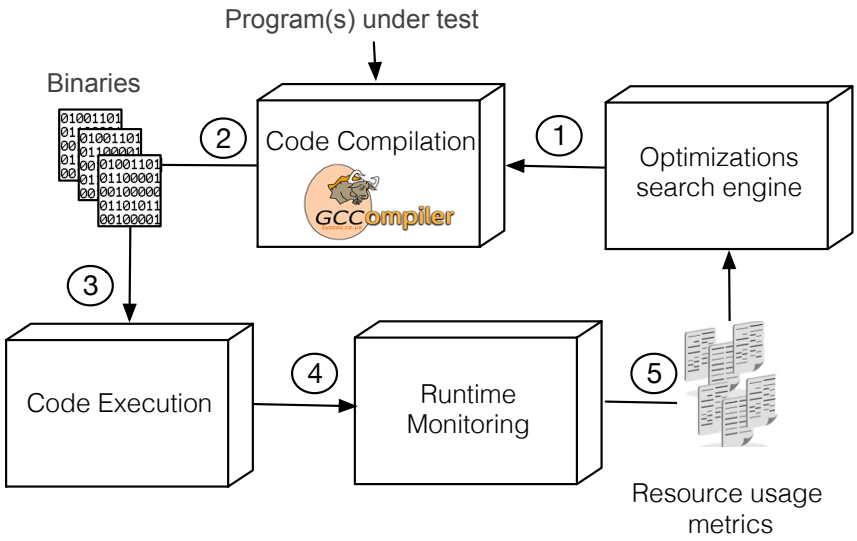


Figure 5.3: NOTICE experimental infrastructure

First, a meta-heuristic (mono or multi-objective) is applied to generate specific optimization sequences for the GCC compiler (step 1). During all experiments, we use GCC 4.8.4, as it is introduced in the motivation section, although it is possible to choose another compiler version using NOTICE since the process of optimizations extraction is done automatically. Then, we compile the input source code program using the set of generated optimizations (step 2). Afterwards, we execute the optimized code within a new container instance (step 3). While running the optimized code, we collect resource usage metrics (step 4). More details about the runtime monitoring engine are provided in Chapter 6. Finally, NOTICE evaluates the set of optimizations, giving a performance/resource usage score to the current solution (step 5). The choice of non-functional metrics depends on experiment objectives (Memory improvement, speedup, trade-offs, etc.).

5.3.3 Experimental methodology and results

In the following paragraphs, we report the methodology and results of our experiments.

5.3.3.1 RQ1. Mono-objective SBSE validation

Method To answer the first research question RQ1, we conduct a mono-objective search for compiler optimization exploration in order to evaluate the non-functional properties of optimized code. Thus, we generate optimization sequences using three search-based techniques (RS, GA, and NS) and compare their performance results to standard GCC optimization levels (O1, O2, O3, and Ofast).

In this experiment, we choose to optimize for execution time (S), memory usage (MR), and CPU consumption (CR). Each non-functional property is improved separately and independently of other metrics. We recall that other properties can be also optimized using NOTICE (*e.g.*, code size, compilation time, etc.), but in this experiment, we focus only on three properties.

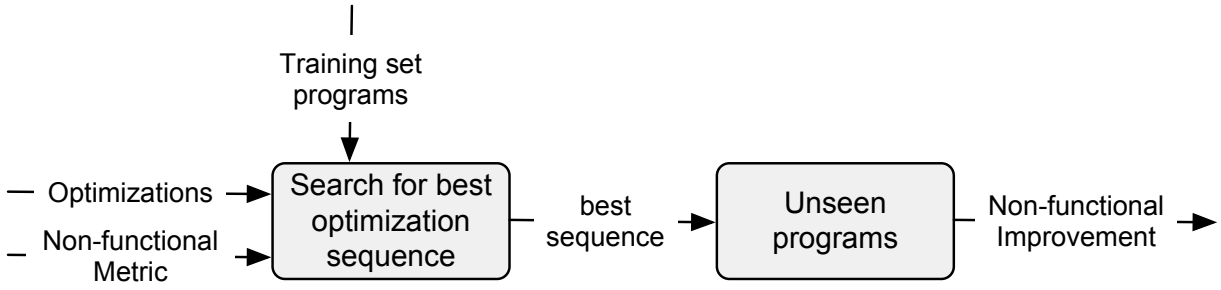


Figure 5.4: Evaluation strategy to answer RQ1 and RQ2

As it is shown on the left side of Figure 5.4, given a list of optimizations and a non-functional objective, we use NOTICE to search for the best optimization sequence across a set of input programs that we call *"the training set"*. This *"training set"* is composed of random Csmith programs (10 programs). We apply then generated sequences to these programs. Therefore, the code quality metric, in this setting, is equal to the average performance improvement (S, MR, or CR) and that, for all programs under test.

To summarize, in this experiment we aim to: (1) compare the performance of our proposed diversity-based exploration of optimization sequences (NS) to GA and RS; and (2) demonstrate that NOTICE is able to find the optimal solution relative to the input training set.

Results Table 5.4 reports the comparison results of three non-functional properties CR, MR, and S. At the first glance, we can clearly see that all search-based algorithms outper-

Table 5.4: Results of mono-objective optimizations

	O1	**O2**	**O3**	**Ofast**	**RS**	**GA**	**NS**
S	1.051	1.107	1.107	1.103	1.121	1.143	1.365
MR(%)	4.8	-8.4	4.2	6.1	10.70	15.2	15.6
CR(%)	-1.3	-5	3.4	-5	18.2	22.2	23.5

form standard GCC levels with minimum improvement of 10% for memory usage and 18% for CPU time (when applying RS).

Our proposed NS approach has the best improvement results for three metrics with 1.365 of speedup, 15.6% of memory reduction and 23.5% of CPU time reduction across all programs under test. NS is clearly better than GA in terms of speedup. However, for MR and CR, NS is slightly better than GA with 0.4% improvement for MR and 1.3% for CR. RS has almost the lowest optimization performance but is still better than standard GCC levels.

We remark as well that applying standard optimizations has an impact on the execution time with a speedup of 1.107 for O2 and O3. Ofast has the same impact as O2 and O3 for the execution speed. However, the impact of GCC levels on resource consumption is not always efficient. O2, for example, increases resource consumption compared to O0 (-8.4% for MR and -5% for CR).

This can be explained by the fact that standard GCC levels apply some aggressive optimizations that increase the performance of generated code and deteriorate system resources.

Key findings for RQ1.
– Best discovered optimization sequences using mono-objective search techniques always provide better results than standard GCC optimization levels.
– Novelty Search is a good candidate to improve code in terms of non-functional properties since it is able to discover optimization combinations that outperform RS and GA.

5.3.3.2 RQ2. Sensitivity

Method Another interesting experiment is to test the sensitivity of input programs to compiler optimizations and evaluate the general applicability of best optimal optimization sets, previously discovered in RQ1. These sequences correspond to the best generated sequences using NS for the three non-functional properties S, MR and CR (*i.e.*, sequences obtained in column 8 of Table 5.4).

Thus, we apply best discovered optimizations in RQ1 to new unseen Csmith (100 new random programs) and we compare then, the performance improvement across these programs (see right side of Figure 5.4). We also apply standard optimizations, O2 and O3, to new Csmith programs in order to compare the performance results. The idea of this experiment is to test whether new generated Csmith programs are sensitive to previously discovered optimizations or not.

If so, this will be useful for compiler users and researchers to use NOTICE in order to build general optimization sequences from their representative *training set* programs.

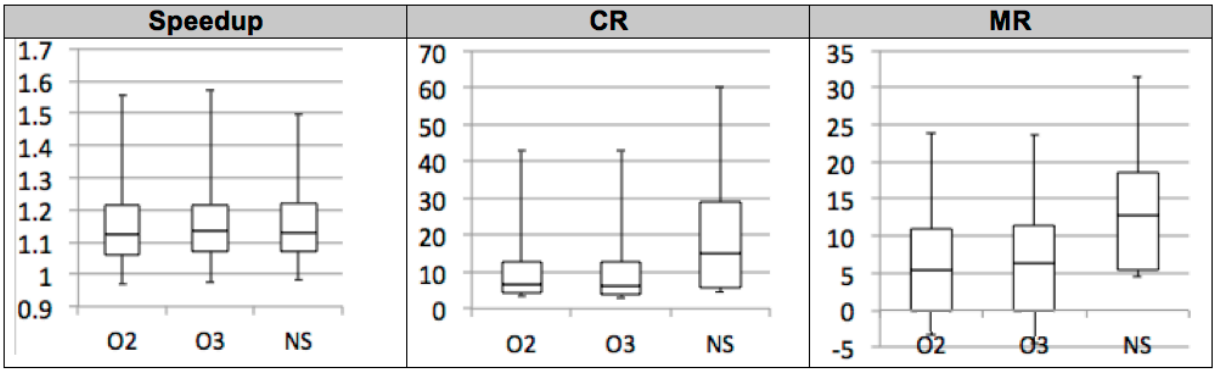


Figure 5.5: Boxplots of the obtained performance results across 100 unseen Csmith programs, for each non-functional property: Speedup (S), memory (MR) and CPU (CR) and for each optimization strategy: O2, O3 and NS

Results Figure 5.5 shows the distribution of memory, CPU and speedup improvement across 100 new Csmith programs. For each non-functional property, we apply O2, O3 and best NS sequences. Speedup results show that the three optimization strategies lead to almost the same distribution with a median value of 1.12 for speedup. This can be explained by the fact that NS might need more time to find the sequence that best optimizes the execution speed. Meanwhile, O2 and O3 have also the same impact on CR and MR which is almost the same for both levels (CR median value is 8% and around 5% for MR).

However, the impact of applying best generated sequences using NS clearly outperforms O2 and O3 with almost 10% of CPU improvement and 7% of memory improvement. This proves that NS sequences are efficient and can be used to optimize resource consumption of new Csmith programs. This result also proves that Csmith code generator applies the same rules and structures to generate C code. For this reason, applied optimization sequences always have a positive impact on the non-functional properties.

Key findings for RQ2.
– It is possible to build general optimization sequences that perform better than standard optimization levels
– Best discovered sequences in RQ1 can be mostly used to improve the memory and CPU consumption of Csmith programs. To answer RQ2, Csmith programs are sensitive to compiler optimizations.

5.3.3.3 RQ3. Impact of optimizations on resource usage

In this experiment, we evaluate the impact of applying the standard optimization levels and the new discovered sequences on the resource usage. We also study the correlation between speedup and resource consumption of generated code. The idea of this experiment is to: (1) prove, or not, the usefulness of involving resource usage metrics as key objectives for performance improvement; (2) the need, or not, of multi-objective search strategy to handle the different non-functional requirements such as resource usage and performance properties.

In the following, we describe two methods to run experiments. The first is based on Csmith programs and the second is based on Cbench programs.

Method 1 In this experiment, we use NOTICE to provide an understanding of optimizations impact, in terms of resource consumption, when trying to optimize for execution time. Thus, we choose one instance of obtained results in RQ1 related to the best speedup improvement (*i.e.*, results obtained in line 1 of Table 5.4) and we study the impact of speedup improvement on memory and CPU consumption. We also compare the resource usage data to standard GCC levels as they were presented in Table 5.4. Improvements are always calculated over the non-optimized version (O0). The following measurements are based on the training set of 10 Csmith programs.

Results 1 Figure 5.6 shows the impact of speedup optimization on resource consumption. For instance, O2 and O3 that led to the best speedup improvement among standard optimization levels in RQ1, present opposite impact on resource usage. Applying O2 induces -8.4% of MR and -5% of CR. However, applying O3 improves MR and CR respectively by 3.4% and 4.2%. Hence, we note that when applying standard levels, there is no clear correlation between speedup and resource usage since compiler optimizations are generally used to optimize the execution speed and never evaluated to reduce system resources.

On the other hand, the outcome of applying different mono-objective algorithms for speedup optimization also proves that resource consumption is always in conflict with execution speed. The highest MR and CR is reached using NS with respectively 1.2% and 5.4%. This improvement is considerably low compared to scores reached when we have applied resource usage metrics as key objectives in RQ1 (*i.e.*, 15.6% for MR and 23.5% for CR). Furthermore, we note that generated sequences using RS and GA have a high impact on system resources since all resource usage values are worse than the baseline.

These results agree to the idea that compiler optimizations do not put too much emphasis on the trade-off between execution time and resource consumption.

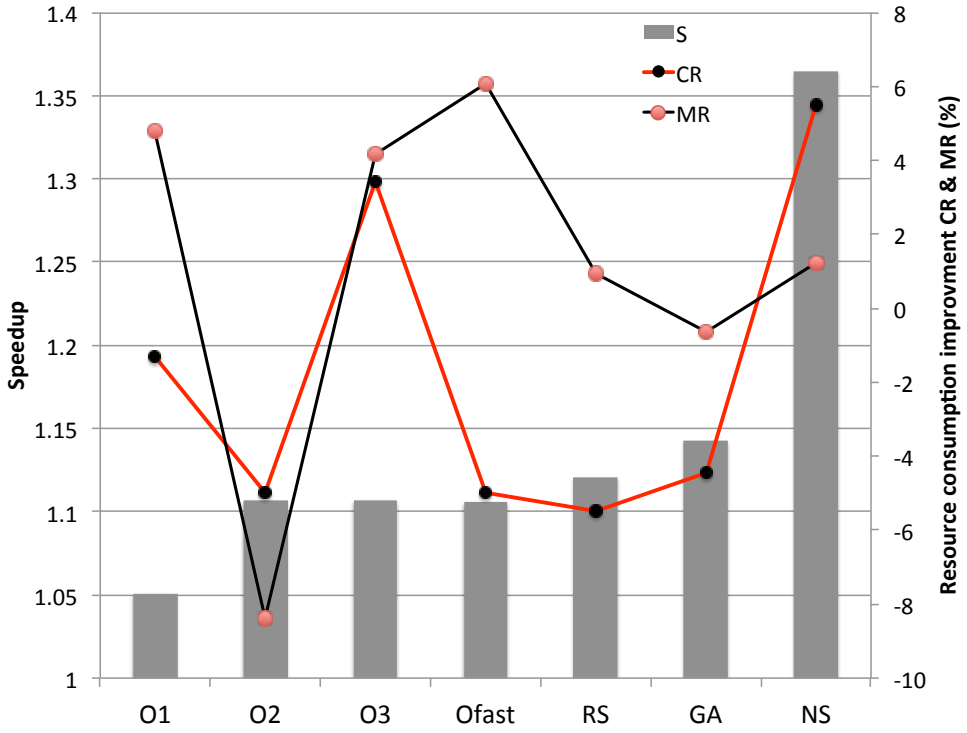


Figure 5.6: Impact of speedup improvement on memory and CPU consumption for each optimization strategy

Method 2 Now, we study the impact of applying standard levels (01, O2, O3, Ofast) on the memory usage of 5 different Cbench programs. We compare the results with solutions generated using NOTICE, having the best memory consumption reduction (*i.e.*, generated

by NS). Figure 5.7 shows this comparison across different benchmark programs. It presents the percentage of saved memory by standard and novelty optimizations over O0 level (no optimization applied).

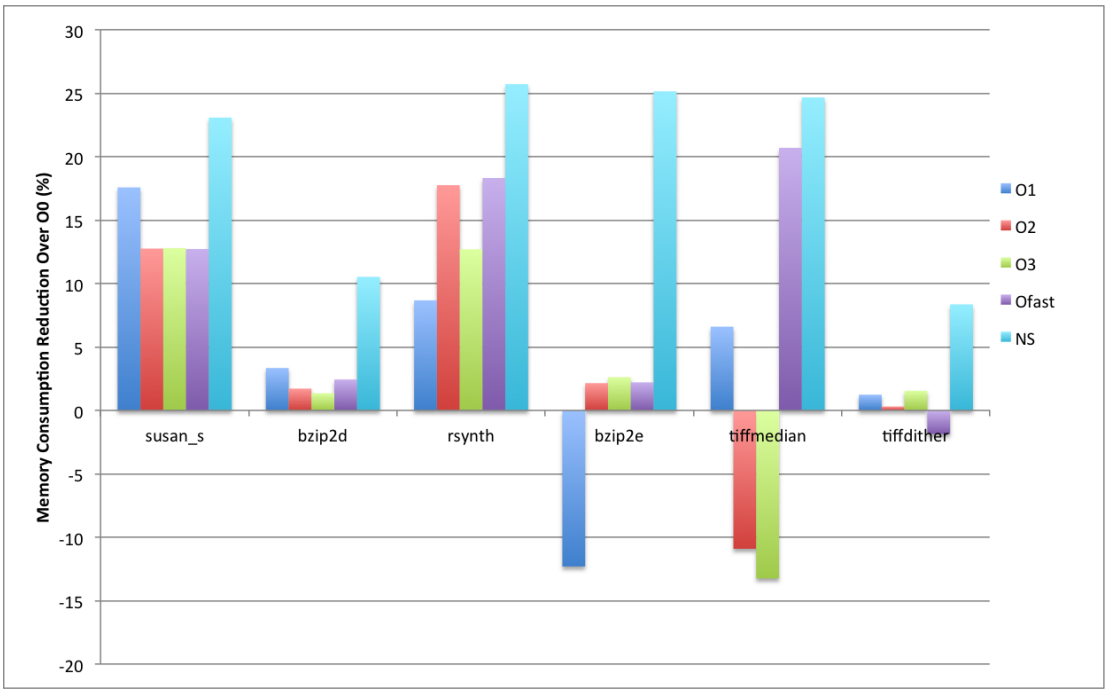


Figure 5.7: Evaluating the amount of saved memory after applying standard optimization options compared to best generated optimization using NS

Results 2 As expected, the results show that NS clearly outperforms standard optimizations for all benchmark programs. Using NS, we are able to reach a maximum memory consumption reduction of almost 26% for the case rsynth program against a maximum of 18% reduction using Ofast option. We remark as well that the impact of applying standard optimizations on memory consumption for each program differs from one program to another. Using O1 for bzip2e and O2, O3 for tiffmedian increase the memory consumption by almost 13 %. This agrees to the idea that standard optimizations does not produce always the same impact results on resource consumption and may be highly dependent on the benchmark and the source code they have been tested on.

Key findings for RQ3.
– Optimizing software performance can induce undesirable effects on system resources.
– A trade-off is needed to find a correlation between both software performance and resource usage.

5.3.3.4 RQ4. Trade-offs between non-functional properties

Method Finally, to answer RQ4, we use NOTICE again to find trade-offs between non-functional properties. In this experiment, we choose to focus on the trade-off <*ExecutionTime–MemoryUsage*>. In addition to our NS adaptation for multi-objective optimization, we implement a commonly used multi-objective approach namely NSGA-II [DPAM02]. We denote our NS adaptation by *NS-II*. We recall that NS-II is not a multi-objective approach as NSGA-II. It uses the same NS algorithm. However, in this experiment, it returns the optimal Pareto front solutions instead of returning one optimal solution relative to one goal. Apart from that, we apply different optimization strategies to assess our approach. First, we apply standard GCC levels. Second, we apply best generated sequences relative to memory and speedup optimization (the same sequences that we have used in RQ2). Thus, we denote by *NS-MR* the sequence that yields to the best memory improvement MR and *NS-S* to the sequence that leads to the best speedup. This is useful to compare mono-objective solutions to new generated ones. In this experiment, we assess the efficiency of generated sequences using only one Csmith program. We evaluate the quality of the obtained Pareto optimal optimization based on raw data values of memory and execution time. Then, we compare qualitatively the results by visual inspection of the Pareto frontiers. The goal of this experiment is to check whether it exists, or not, a sequence that can reduce both execution time and memory usage. We report the comparison results of our NS adaptation for optimizations generation to the current state-of-the-art multi-objective approaches namely NSGA-II.

Results Figure 5.8 shows the Pareto optimal solutions that achieved the best performance assessment for the trade-off <*ExecutionTime–MemoryUsage*>. The horizontal axis indicates the memory usage in raw data (in Bytes) as it is collected using NOTICE. In similar fashion, the vertical axis shows the execution time in seconds. Furthermore, the figure shows the impact of applying standard GCC options and best NS sequences on memory and execution time.

Based on these results, we can see that NSGA-II performs better than NS-II. In fact, NSGA-II yields to the best set of solutions that presents the optimal trade-off between the

two objectives. Then, it is up to the compiler user to use one solution from this Pareto front that satisfies his non-functional requirements (six solutions for NSGA-II and five for NS-II). For example, he could choose one solution that maximizes the execution speed in favor of memory reduction. On the other side, NS-II is capable to generate only one non-dominated solution. For NS-MR, it reduces as expected the memory consumption compared to other optimization levels. The same effect is observed on the execution time when applying the best speedup sequence NS-S. We also note that all standard GCC levels are dominated by our different heuristics NS-II, NSGA-II, NS-S and NS-MR.

This agrees to the claim that standard compiler levels do not present a suitable trade-off between execution time and memory usage.

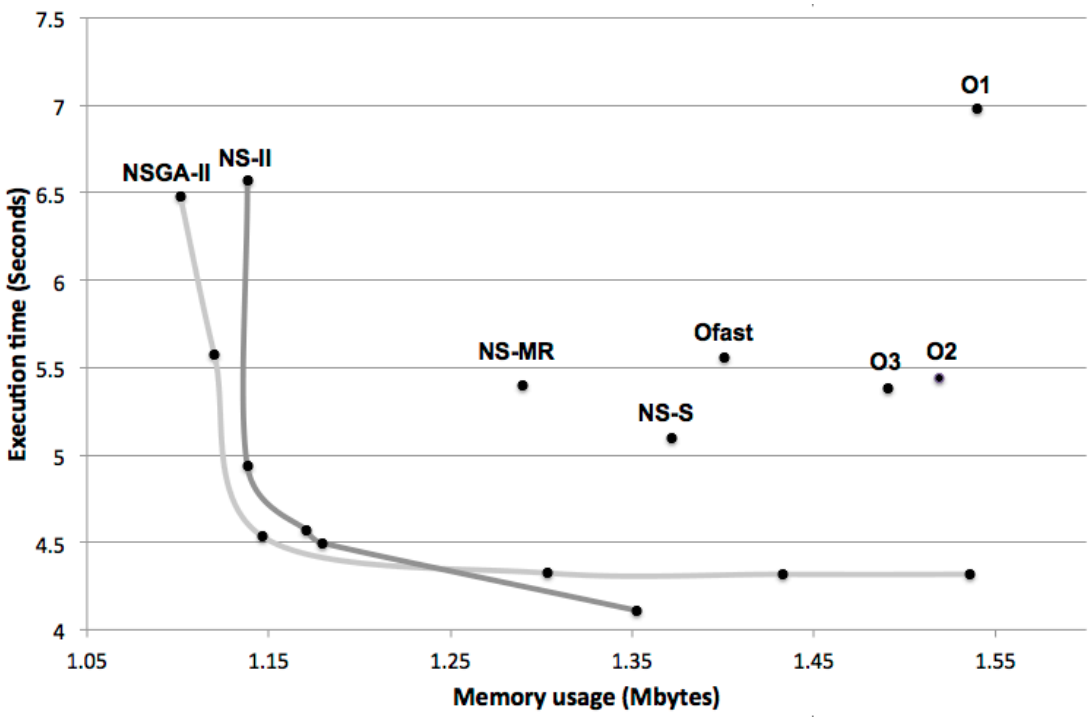


Figure 5.8: Comparison results of obtained Pareto fronts using NSGA-II and NS-II

Key findings for RQ4.
– NOTICE is able to construct optimization levels that represent optimal trade-offs between non-functional properties.
– NS is more effective when it is applied to mono-objective search.
– NSGA-II performs better than our NS adaptation for multi-objective optimization. However, NS-II performs clearly better than standard GCC optimizations and previously discovered sequences in RQ1.

5.3.4 Discussions

Through these experiments, we showed that NOTICE is able to provide facilities to compiler users to test the non-functional properties of generated code. It provides a support to search for the best optimization sequences through mono-objective and multi-objective search algorithms. NOTICE infrastructure has shown its capability and scalability to satisfy user requirements and key objectives in order to produce efficient code in terms of non-functional properties.

During all experiments, standard optimization levels have been fairly outperformed by our different heuristics. Moreover, we have also shown (in RQ1 and RQ3) that optimizing for performance may be, in some cases, greedy in terms of resource usage. For example, the impact of standard optimization levels on resource usage is not always efficient even though it leads to performance improvement. Thus, compiler users can use NOTICE to evaluate the impact of optimizations on the non-functional properties and build their specific sequences by trying to find trade-offs among these non-functional properties (RQ4).

We would notice that for RQ1, experiments take about 21 days to run all algorithms. This run time might seem long but, it should be noted that this search can be conducted only once, since in RQ2 we showed that best gathered optimizations can be used with unseen programs of the same category as the training set, used to generate optimizations. This has to be proved with other case studies. Multi-objective search as conducted in RQ4, takes about 48 hours, which we believe is acceptable for practical use. Nevertheless, speeding up the search speed may be an interesting feature for future research.

5.3.5 Threats to validity

Any automated approach has limitations. We resume, in the following paragraphs, external and internal threats that can be raised:

External validity refers to the generalizability of our findings. In this study, we perform experiments on random programs using Csmith and we use iterative compilation techniques to produce best optimization sequences. We believe that the use of Csmith programs as input programs is very relevant because compilers have been widely tested across Csmith programs [CHH+16,YCER11]. Csmith programs have been used only for functional testing, but not for non-functional testing. However, we cannot assert that the best discovered set of optimizations can be generalized to industrial applications since optimizations are highly dependent on input programs and on the target architecture. In fact, experiments conducted on RQ1 and RQ2 should be replicated to other case studies to confirm our findings; and build general optimization sequences from other representative training set programs chosen by compiler users.

Internal validity is concerned with the causal relationship between the treatment and the outcome. Meta-heuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. Are we providing a statistically sound method or it is just a random result? Due to time constraints, we run all experiments only once. Following the state-of-the-art approaches in iterative compilation, previous research efforts [HE08, MÁCZCA+14] did not provide statistical tests to prove the effectiveness of their approaches. This is because experiments are time-consuming. However, we can deal with these internal threats to validity by performing at least five independent simulation runs for each problem instance.

5.3.6 Tool support overview

NOTICE provides also a GUI interface. The goal of this tool support is to help users to easily use NOTICE and finely auto-tune GCC compilers. This tool has been used to answer all previous research questions.

As shown in Figure 5.9, NOTICE provides different features to help compiler users to:

- **Select the input program under test:** by generating a new Csmith program or by selecting an existing C program such as Cbench benchmark programs. The generation of a new Csmith program is done randomly.

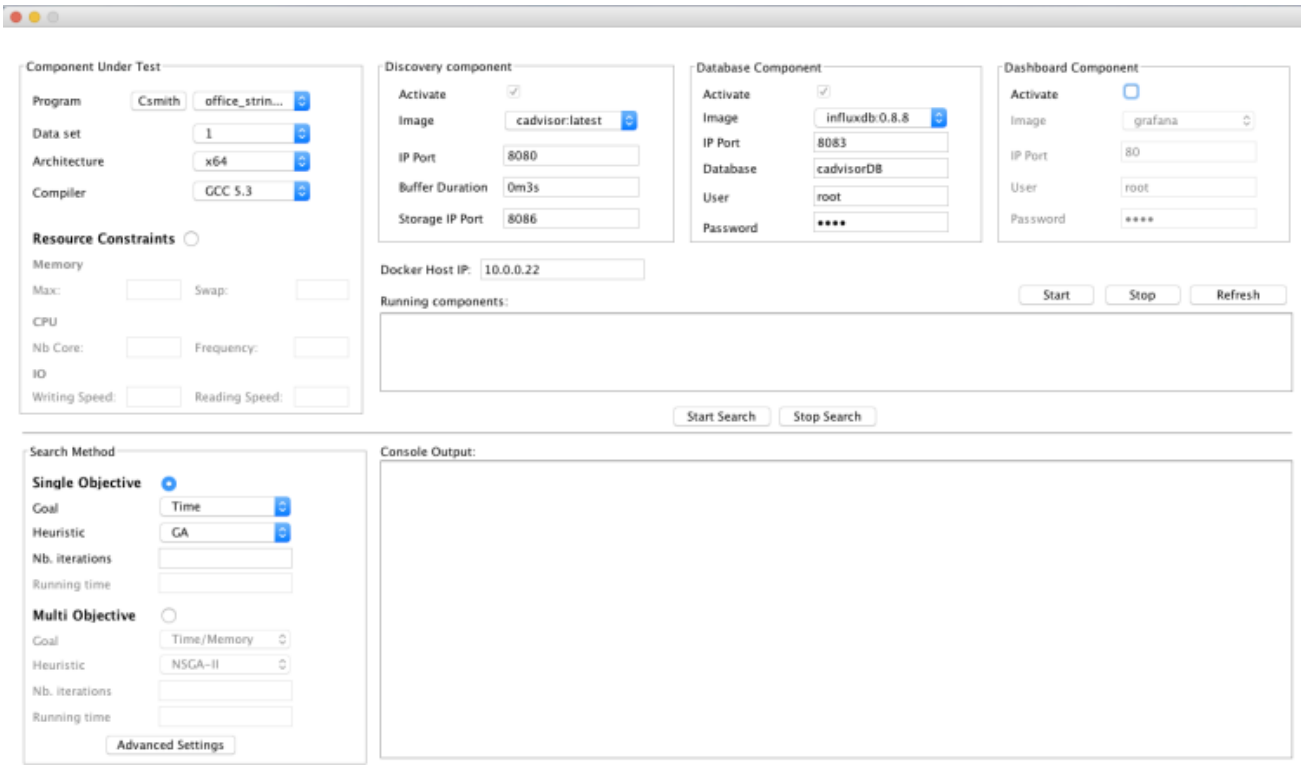


Figure 5.9: Snapshot of NOTICE GUI interface

- **Select datasets:** In case the selected program requires a dataset such as the case for Cbench programs, NOTICE allows the user to choose the dataset for the selected program. We recall that Cbench comes with a set of 20 datasets for each benchamrk program.
- **Select the target computer architecture:** choose the processor architecture where the experiments will be running such as x64, x86, ARM. In our experiments, we use native GCC compiler of the host machine, with x64 architecture.
- **Define the compiler version:** For now, NOTICE supports all GCC compiler versions from 3.x to 5.x. The process of extracting the target optimizations to evolve is done automatically (*i.e.*, optimizations enabled by O1, O2, O3 and Ofast)
- **Configure the monitoring components:** This refers to the containers needed to extract all the information about the resource consumption. Configuring these

components is possible with NOTICE such as image versions, labels, ports, logins, passwords.

- **Choose ip address of the cloud host machine:** NOTICE allows to run experiments remotely thanks to its micro-service infrastructure. Thus, we enable the user to select the ip of the remote machine.
- **Define resource constraints to running container:** In case we would run optimizations under resource constraints, it is possible to define memory and CPU constraints. By default, these option are disabled.
- **Choose the search method:** The user can select either a mono objective or multi-objective search.
- **Choose the meta-heuristic algorithm:** NOTICE supports GA, RS, and NS for mono objective search and NS, RS, and NSGA-II for multi-objective optimization.
- **Choose the number of iterations:** The user can define the number of iterations for each algorithm which corresponds to the number of generated optimization sequences.
- **Choose the search time:** Instead of limiting the number of iterations, the user can fix a limited tuning time (in hours).
- **Choose the tuning goals:** The goal can be reducing the execution time, memory, CPU, code size, or compilation time. For multi objective search, users can choose trade-offs between these objectives.
- **Edit evolutionary algorithm settings:** Tuning the evolutionary parameters (showed in Table 5.3) such as the population size, the novelty search settings, mutation and crossover probabilities, etc.

The console output (*i.e.*, the execution result of this tool) displays at the end, the comparison results of standard optimization levels to the new discovered solutions.

5.4 Conclusion

Modern compilers come with huge number of optimizations, making complicated for compiler users to find best optimization sequences. Furthermore, auto-tuning compilers to meet user requirements is a difficult task since optimizations may depend on different properties (*e.g.*, platform architecture, software programs, target compiler, optimization objective, etc.). Hence, compiler users merely use standard optimization levels (O1, O2, O3 and Ofast) to enhance the code quality without taking too much care about the impact of optimizations on system resources.

In this chapter, we have introduced first a novel formulation of the compiler optimization problem based on Novelty Search. The idea of this approach is to drive the search for best optimizations toward novelty. Our work presents the first attempt to introduce diversity in iterative compilation. Experiments have shown that Novelty Search can be easily applied to mono and multi-objective search problems. In addition, we have reported the results of an empirical study of our approach compared to different state-of-the-art approaches, and the obtained results have provided evidence to support the claim that Novelty Search is able to generate effective optimizations. Second, we have presented an automated approach for automatic extraction of non-functional properties of optimized code, called NOTICE. NOTICE applies different heuristics (including Novelty Search) and performs compiler auto-tuning through the monitoring of generated code in a controlled sand-boxing environment. In fact, NOTICE uses a set of micro-services to provide a fine-grained understanding of optimization effects on resource consumption. We evaluated the effectiveness of our approach by verifying the optimizations performed by GCC compiler. Then, we studied the impact of optimizations on memory consumption and execution time. Results showed that our approach is able to automatically extract information about memory and CPU consumption. We were also able to find better optimization sequences than standard GCC optimization levels and construct optimizations that present optimal trade-offs between speedup and memory usage.

Chapter 6

A lightweight execution environment for automatic generators testing

6.1 Introduction

Software platforms diversity and hardware heterogeneity, as discussed in Chapter 2, constitutes a major obstacle for software testing. In fact, running tests requires many environment configurations and settings in order to test the whole application. For example, testing a web application requires the installation of the Maven dependencies, web server, libraries, etc. When software developers upgrades the web server version for example, they need to rebuild the application and run the same integration tests in order to check that no errors have been incorporated. Thus, testing applications using different execution environments and system settings becomes very time consuming and tedious.

For instance, as we discussed in Chapter 4 and 5, to evaluate the automatically generated code (by either code generators or compilers), we use to generate code, compile it, and then run test cases. To do so, different system configurations were required to ensure these steps such as installing the generator version (GCC or Haxe versions), install interpreters, compilers, Maven dependencies, etc.

One way to test these configurable generators is to use the virtualization technology. For instance, an alternative method leverages container-based system virtualization (*e.g.*, Docker, as discussed in Section 3.3) to automate the code generation, deployment, and testing inside pre-configured software containers. This technology enables to mimic the execution environment settings and reproduce the tests in isolated and highly configurable system containers.

When it comes to evaluate the resource consumptions of automatically generated code, this technology becomes very valuable because it allows a fine-grained resource management and isolation. Moreover, it facilitates resource usage extraction and limitation of programs running inside containers.

This chapter presents a technical description of this lightweight runtime environment and its benefit for automating the non-functional testing of generated code. This infrastructure is used in our two first contributions as means to run tests in a configurable execution environment and to efficiently collect resource consumption metrics.

This chapter is organized as follows:

Section 6.2 introduces system containers as a lightweight execution environment. We show the benefit of using this technology to automate software testing.

Section 6.3 describes the runtime monitoring components required to collect resource usage metrics.

Section 6.4 shows the adaptation of this sandboxing infrastructure to the generator case study as it is used in Chapters 4 and 5.

Finally, we conclude in Section 6.5.

6.2 System containers as a lightweight execution environment

System containers are operating system-level virtualization method that allows running multiple isolated Linux systems on a control host using a single Linux kernel. Containers share the same OS and hardware as the hosting machine and it is very useful to use them in order to create new configurable and isolated instances to run. Container-based virtualization reduces the overhead associated with having each guest running a new installed operating system such the case for virtual machines. This approach can also improve the performance because there is just one operating system taking care of hardware calls. The Linux kernel provides the control groups[1] (Cgroups) functionality that allows the limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) inside containers so that, one container does not starve the others in terms of resources.

For instance, Docker[2] is a popular container-based technology that automates the deployment of any application as a lightweight, portable, and self-sufficient container, running

[1] https://fr.wikipedia.org/wiki/Cgroups
[2] https://www.docker.com

virtually on a host machine [Mer14]. Today, Docker is one of the most popular infrastructure technology adopted in cloud computing [PHP16]. For example, in 2015, Docker had about 3% market share, and by 2017 it is running on 15% of the hosts[3]. Using Docker, it is possible to define pre-configured applications and servers to host as virtual images. It also defines the way the service should be deployed in the host machine using configuration files called Dockerfiles. Moreover, we can enable some configuration options to control and limit resources. For example, we can provide option flags to limit how much memory or CPU usage each service is allowed to consume, associate CPU cores to each service, etc.

In short, the main advantages of this micro-services approach are:

- The use of containers induces less performance overhead compared to using a full stack virtualization solution [SCTF16]. Indeed, instrumentation and monitoring tools for memory profiling like Valgrind [NS07] can induce too much overhead.
- Thanks to the use of Dockerfiles, it is possible to easily configure the execution environment in order to build and customize applications using numerous settings (*e.g.*, generator version, dependencies, host IP and OS, optimization options, etc.). Thus, we can use the same configured Docker image to execute different instances of the same application. For hardware architecture, containers share the same platform architecture as the host machine (*e.g.*, x86, x64, ARM, etc.).
- Docker uses Linux control groups (Cgroups) to group processes running in the container. This allows us to manage the resources of a group of processes, which is very valuable. This approach increases the flexibility when we want to manage resources, since we can manage every group individually. For example, if we would evaluate the non-functional requirements of generated code within a resource-constraint environment, we can request and limit resources within the execution container according to the needs.
- Although containers run in isolation, they can share data with the host machine and other running containers. Thus, non-functional data relative to resource consumption can be gathered and managed by other containers (*i.e.*, for storage purpose, visualization)

[3] https://www.datadoghq.com/docker-adoption/

6.3 Runtime Monitoring Engine

In order to monitor the applications (*i.e.*, tests) running within containers, we aim to use a set of Docker components to ease the extraction of resource usage information.

6.3.1 Monitoring Container

First, a monitoring component is needed to collect the resource usage and performance characteristics of running containers. As discussed earlier, Docker relies on Cgroups file systems to expose a lot of metrics about accumulated CPU cycles, memory, block I/O usage, etc. Therefore, our monitoring component automates the extraction of these runtime performance metrics stored in Cgroups files. Among the popular ways to do that is to monitor each container via the Docker API, or by installing an agent for detailed visibility inside each container. The Docker client already provides a command-line tool to inspect containers' resource consumption. The command *docker stats*, for example, can be used to get the stats about the running containers at runtime. If we want to do that manually, we can access to live resource consumption of each container available at the Cgroups file system via stats found in *"/sys/fs/cgroup/cpu/docker/(longid)/"* (for CPU consumption) and *"/sys/fs/cgroup/memory/docker/(longid)/"* (for stats related to memory consumption). Our monitoring component automates the process of service discovery and metrics aggregation for each new container. Thus, instead of gathering manually metrics located in Cgroups file systems, it extracts automatically the runtime resource usage statistics relative to the running component (*i.e.*, the executed test suite within a container). We note that resource usage information is collected in raw data. This process may induce a little overhead because it performs a very fine-grained accounting of resource usage on running container. Fortunately, this may not affect the gathered data since we run only one test suite or application within each container. To ease the monitoring process, we integrate cAdvisor, a Container Advisor[4]. cAdvisor monitors service containers at runtime as described above. It has been widely used in different projects such as Heapster[5] and Google Cloud Platform[6].

However, cAdvisor monitors and aggregates live data over only 60 seconds interval. Therefore, we record all data over time, since container's creation, in a time-series database. It allows the end users to run queries and define non-functional metrics from historical data.

[4] https://github.com/google/cadvisor
[5] https://github.com/kubernetes/heapster
[6] https://cloud.google.com/

Thereby, to make the gathered data truly valuable for resource usage monitoring, we link our monitoring component to a back-end database container.

6.3.2 Back-end Database Container

This component represents a time-series database back-end. It is plugged with the previously described monitoring component to save the non-functional data for long-term retention, analytics and visualization. During application execution, resource usage stats are continuously sent to this component. When a container is killed, we are able to access to its relative resource usage metrics through the database. We choose a time series database because we are collecting time series data that correspond to the resource utilization profiles of programs execution.

We use InfluxDB[7], an open source distributed time-series database as a back-end to record data. InfluxDB allows the user to execute SQL-like queries on the database. For example, the following query reports the maximum memory usage of container *"generated_code_v1"* since its creation:

```
select max (memory_usage) from stats
where container_name='generated_code_v1'
```

To give an idea about the data gathered by the monitoring component and stored in the time-series database, we describe in Table 6.1 these collected metrics:

Metric	**Description**
Name	Container Name
T	Elapsed time since container's creation
Network	Stats for network bytes and packets
Disk IO	Disk I/O stats
Memory	Memory usage
CPU	CPU usage

Table 6.1: Resource usage metrics recorded in InfluxDB

Apart from that, we provide information about the application size (*e.g.*, size of generated binaries) and the compilation time required to produce code. For instance, resource

[7] https://github.com/influxdata/influxdb

usage statistics are collected and stored using the previously described components. It is relevant to show resource usage profiles of running programs overtime. To do so, we present a front-end visualization container for resource usage profiling.

6.3.3 Front-end Visualization Container

Once we gather and store resource usage data, the next step is visualizing them. That is the role of the visualization container. It will be the endpoint component that we use to visualize the recorded data. Therefore, we provide a dashboard to run queries and view different resource consumption profiles of running components, through a Web UI. Thereby, we can compare visually the profiles of resource consumption among containers. Moreover, we can use this component to export the data currently being viewed into static CSV document. So, we can perform statistical analysis on this data to detect inconsistencies or performance anomalies. As a visualization component, we use Grafana[8], a time-series visualization tool available for Docker. Grafana lets us display live results over time in much pretty looking graphs. Same as InfluxDB, we use SQL queries to extract the non-functional data from the database for visualization and analysis.

6.4 The generator case study

We present now an adaptation of this micro-service infrastructure to the generator case study, as applied in Chapters 4 and 5. We recall that we used containers as means for running different variants of optimized code in Chapter 5, and for running a bench of test suites across different software platforms in Chapter 4. The runtime monitoring components, presented in this chapter, are used in both contributions to evaluate the resource usage of generated code. An overview of the micro-service and technical solutions applied for the generator case study are shown in Figure 6.1. In the following, we describe in details the infrastructure settings. The experimental material is also available online[9][10].

Code generation Before starting to monitor and test applications, we have to deploy the generated code (by compilers or code generators) on different Docker containers. Thus, instead of configuring all generators under test (GUTs) within the same host machine, we

[8] https://github.com/grafana/grafana
[9] https://testingcodegenerators.wordpress.com/
[10] https://noticegcc.wordpress.com/

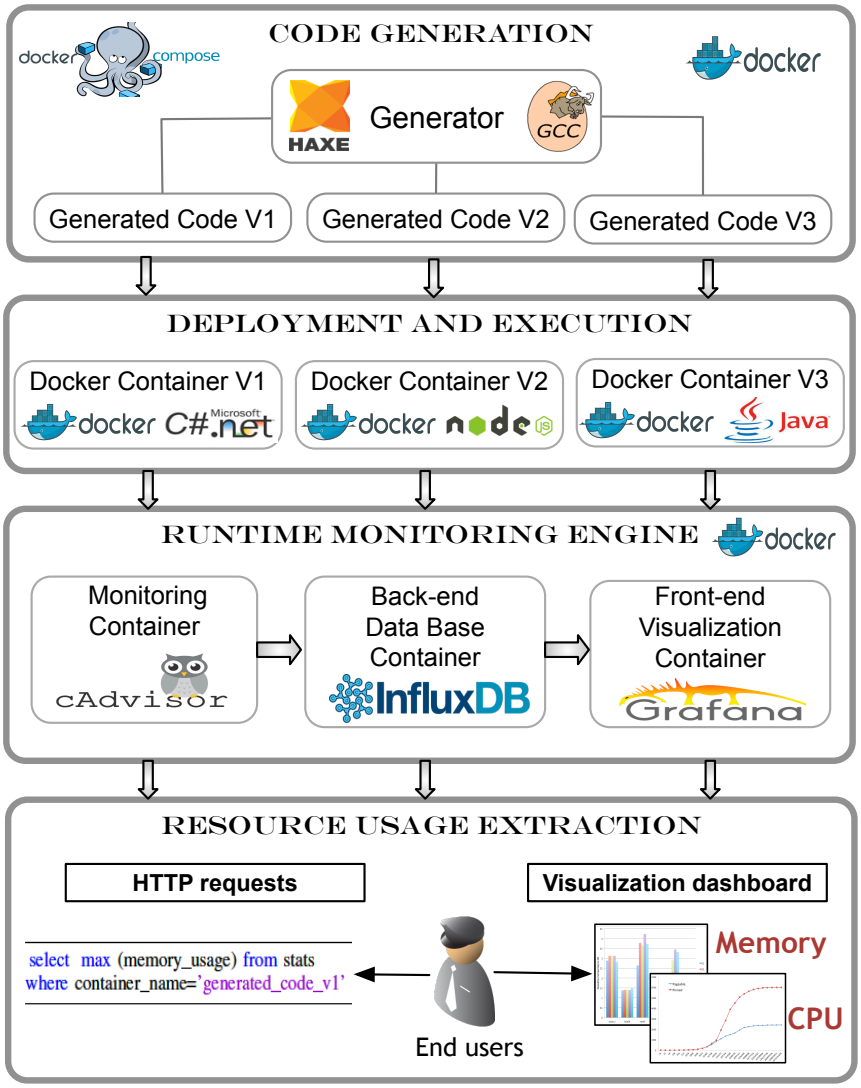


Figure 6.1: Our container-based infrastructure for automatic generators testing

deploy each GUT within a container. To do so, we create a new configuration image for each GUT (*i.e.*, the Docker image) where we install all the libraries, compilers, and dependencies needed to ensure the code generation and compilation. Thereby, the GUT produces code within multiple instances of pre-configured Docker images. So, we use Dockerfiles to configure all these settings. We use the public Docker registry[11] (a cloud-

[11] https://hub.docker.com/

based registry service) to save and manage all our Docker images. Once code generation is done, the generated output files are saved in a shared repository. In Docker environment, this repository is called *data volume*. It is a specially-designated directory within containers that shares data with the host machine and with other running containers.

Deployment and execution Next, generated code (in the *data volume*) is executed individually inside an isolated Docker container. By doing so, we ensure that each executed program runs in isolation without being affected by the host machine or any other processes. Moreover, since a container is cheap to create, we are able to create too many containers as long as we have new programs to execute (*e.g.*, new optimized code, test suite for a specific software platform, etc.). Since each program execution requires a new container to be created, it is crucial to remove and kill containers that have finished their job to eliminate the load on the system. We run the experiment on top of a private data-center that provides a bare-metal installation of Docker. On a single machine, containers are running sequentially and we pin p cores and n Gbytes of memory for each container[12]. Once the execution is done, resources reserved for the container are automatically released to enable spawning next containers. Therefore, the host machine will not suffer too much from performance trade-offs.

Runtime monitoring While running containers, we run the three monitoring containers described above in order to monitor the running workload. To do so, we use Docker Compose[13] to run all containers simultaneously. The concept of Docker Compose is similiar to Dockerfiles. It uses a configuration file to run and link multi-container Docker services. We set up the Compose file so that we run all services and particularly to map running containers to cAdvisor and InfluxDB, using Docker ports and network links in order to stream resource usage data.

Resource usage extraction The end users have two ways to extract the information about the resource usage of generated code. It is possible to directly request the remote time series database via HTTP requests, executing SQL-like queries like the example presented in Section 6.3.2. They can request different metrics such as CPU, memory usage, disk writing speed, etc. An alternative solution is to visualize the resource consumption of generated code within the web-based dashboard provided by Grafana. The visualization

[12] p and n can be cofigured
[13] https://docs.docker.com/compose/overview/

tool is not used when auto-tuning and testing generators because we just needed to extract the CPU or memory usage for each test or optimized code.

We use the same hardware across all experiments in Chapters 4 and 5: an AMD A10-7700K APU Radeon(TM) R7 Graphics processor with 4 CPU cores (2.0 GHz), running Linux with a 64 bit kernel and 16 GB of system memory.

Limitations. *We would notice that this testing infrastructure can be generalized and adapted to other case studies other than generators. Using system containers, any software application/generated code can be easily executed and monitored using Docker. However, among the limitations of this micro-services infrastructure is that containers require Linux kernel to run. Running Docker engine on macOS or Windows requires a virtual machine to deploy small Linux-based OS that has Docker pre-installed. Collecting the resource usage of these windows containers for example, may induce an overhead due to the use of Docker within virtual machines, which may affect the accuracy of gathered resource usage data. In addition, resource isolation in Docker has some limitations, especially for disk I/O metrics. It has been recently proven [XDOR+15], that the isolation layer does not properly isolate the shared resources when running disk-intensive workloads within containers, causing some performance interferences.*

6.5 Conclusion

We presented in this chapter, the technical details of the infrastructure used to collect the non-functional metrics (e.g., memory and CPU consumptions) of automatically generated code (by either compilers or code generators). This solution offers effective support for automatically deploying, executing, and testing the generated code in different environment settings. The same monitoring infrastructure is used to evaluate quality of generated code in the two first contributions of this thesis. The experiments conducted in Chapters 4 and 5 showed the usefulness of this infrastructure for tuning and testing generators.

Part III

Conclusion and Perspectives

Chapter 7

Conclusion and perspectives

In this chapter, we first summarize all the contributions of this thesis, recalling the challenges and how we addressed each of them. Next and finally, we discuss some perspectives for future work.

7.1 Summary of contributions

Generative software development has paved the way for the creation of multiple generators that serve as a basis for automatically generating code to a broad range of software and hardware platforms. With full automatic code generation, users are able to rapidly synthesize software artifacts for various software platforms. In addition, they can easily customize the generated code for the target hardware platform since modern generators (*i.e.*, C compilers) become highly configurable, offering numerous configuration options that the user can apply. The quality of automatically generated software is highly correlated to the configuration settings as well as the generator itself. Therefore, we have highlighted, throughout this thesis, the challenges that we face when testing and auto-tuning configurable generators.

In reviewing the state of the art, we identified numerous approaches for testing generators. However, few of them evaluate the non-functional properties of automatically generated code, namely the performance and resource usage properties. The main issue we have identified when testing the non-functional properties is the oracle problem, since there is no a clear definition of how the oracle might be defined when it comes to the test of the performance and resource usage properties. Similarly, research in auto-tuning

generators, especially compilers, has been studied for decades, proposing different solutions for exploring the large optimization search space. However, they do not exploit the recent advances in search-based software engineering to effectively find the best configuration set.

From a software engineering point of view, this thesis contributes to improve the quality and reliability of generators. In particular, we provide a mechanism that helps generator creators/maintainers to efficiently test their created code generators, and thus provide evidence to the end users of the quality of generated code. We also provide facilities to the generator users to effectively auto-tune exiting generators in order to produce a high-quality code.

The contributions are summarized as follows:

Our first contribution addresses the problem of non-functional testing of generators. In particular, we tackle the oracle problem in the domain of code generators testing. Thus, we propose an approach for automatically detecting inconsistencies in code generators in terms of non-functional properties (*i.e.*, resource usage and performance). Our approach is based on the intuition that a code generator is often a member of a family of code generators. Therefore, we benefit from the existence of multiple generators with comparable functionality (*i.e.*, code generator families) to apply the idea of metamorphic testing [ZHT+04], defining high-level test oracles (*i.e.*, metamorphic relations) as test oracles. We define the metamorphic relation as a comparison between the variations of performance and resource usage of code, generated from the same code generator family. Any variation that exceeds a specific threshold value is automatically detected as an anomaly. We apply two statistical methods (*i.e.*, principal component analysis and range-charts) in order to automate the inconsistencies detection. We evaluate our approach by analyzing the performance of Haxe, a popular high-level programming language that involves a set of cross-platform code generators. We evaluate the properties related to the resource usage and performance for five different target software platforms. We run a bench of test suites across 7 Haxe benchmark libraries in order to verify the metamorphic relation (*i.e.*, the performance and memory usage variation) for each of them. Experimental results show that our approach is able to detect, among 95 executed test suites, 11 performance and 15 memory usage inconsistencies, violating the metamorphic relation. These results show that our approach can automatically detect real issues in code generator families.

The second contribution addresses the problem of generators auto-tuning. Particularly, we are interested in auto-tuning compilers because of the large number of configuration options (*i.e.*, optimizations) they offer to control the quality of the generated code. In this context, we exploit the recent advances in search-based software engineering in order to provide an effective approach to tune compilers (*i.e.*, through optimizations) according to

user's non-functional requirements (*i.e.*, performance and resource usage). Our approach, called NOTICE, applies a novel formulation, compared to previous related work, of the compiler optimization problem using the Novelty Search algorithm [LS08]. Novelty Search is applied to tackle the problem of optimizations diversity and then, providing a new way to explore the huge optimization search space. In fact, since the search space of possible combinations is multi-modal [BKK+98] and too large, we apply this technique is to explore the search space of possible compiler optimization options by considering sequence diversity as a single objective. We conduct an empirical study to evaluate the effectiveness of our approach by verifying the optimizations performed by the GCC compiler. Our experimental results show that NOTICE is able to auto-tune compilers according to user choices (heuristics, objectives, programs, etc.) and construct optimizations that yield to better performance results than standard optimization levels and classical genetic algorithms. We also demonstrate that NOTICE can be used to automatically construct optimization levels that represent optimal trade-offs between the speedup and memory usage using multi-objective algorithms.

Evaluating the resource usage of automatically generated code is complex because of the diversity of software and hardware platforms that exist in the market. To handle this problem, we present in the third contribution the technical details of the infrastructure used to collect the non-functional metrics (e.g., memory and CPU consumptions) of automatically generated code (by either compilers or code generators). In fact, we benefit from the recent advances in lightweight system virtualization, in particular container-based virtualization, in order to offer effective support for automatically deploying, executing, and monitoring the generated code in heterogeneous environment. The same monitoring infrastructure is used to evaluate the experiments conducted in the two first contributions.

7.2 Perspectives

The work presented in this thesis represents a step towards proving support to evaluate configurable generators. In the reminder of this chapter we will describe possible improvements and extensions to the contributions of this thesis.

Tracking the source of code generator inconsistencies In Chapter 4, we present a black-box testing approach that identifies the presence of potential issues within code generator families. However, we do not provide detailed information about the source of the issues. We investigate the generated code manually in order to fix the bug. As a future

work, we believe that a traceability method can be applied to track the inconsistency, at the source code level. Thus, we can help code generator maintainers to easily identify the source of the bugs (*e.g.*, parts of the code that affect the software performance), and fix the issues.

Improving the efficiency of our auto-tuning approach We plan to explore more trade-offs among resource usage metrics *e.g.*, the correlation between CPU consumption and speedup. Using the Docker technology, we can easily run containers on different host machines, in the cloud. As a future work, we want to evaluate our approach across different hardware architectures. We also intend to provide more facilities to NOTICE users in order to test optimizations performed by modern compilers such as Clang, LLVM, etc. Finally, NOTICE can be easily adapted and integrated to new case studies. As an example, we would inspect the behavior of code generators since different optimizations can be performed to generate code from models [SCDP07]. The same approach can be applied as long as the code generator accept many configuration options.

Speed up the time required to tune and test generators Throughout the experiments conducted in this thesis, we use to run containers sequentially. This can take too much time. In particular, the tuning time grows exponentially as long as we have new configurations to evaluate. In order to reduce the time needed to run and monitor multiple versions of generated code (e.g., optimized versions), we intend to deploy tests on many nodes in the cloud using multiple containers in parallel. Doing so, we will be able to accelerate the testing process. Solutions as Docker Swarm[1] already exist and can be applied to manage clusters of containers running in parallel in the cloud.

Interact with generator experts in order to improve our testing approach Few months after running our experiments, the Haxe community has released a new version of the PHP code generator[2] with many performance improvements, especially for arrays. As we expected, they introduced advanced functions (*array_fill*[3]) for arrays initialization in order to improve their performance. Actually, we are discussing with the Haxe community in order to expand our testing approach, introducing new target software platforms to test and creating new benchmark programs with large number of test suites. Finally, we

[1] https://docs.docker.com/engine/swarm/

[2] https://github.com/HaxeFoundation/haxe/releases/tag/3.4.0

[3] https://github.com/HaxeFoundation/haxe/blob/f375ec955b41550546e494e9f79a5deefa1b96ac/std/php7/Global.hx#L226

may evaluate the impact of the new code generator improvements (*i.e.*, running the same experiments with new code generator versions) and check if the fixes have eliminated the previously identified inconsistencies.

Test amplification in the context of generators testing When testing generators, we write test suites manually using high-level benchmarks. Writing test suites for each benchmark is time-consuming. Hence, inconsistencies detection is limited to the parts of code executed by our test suites. An emerging testing field called *Test Amplification* is recently presented by Danglot *et al.* [DVPY+17], to improve the efficiency of test suites. It consists in modifying or creating new tests from existing ones so as to improve an engineering goal such as improving coverage, improving properties of the application under test, crash reproduction, fault localization, etc. This technique is useful in agile software development (*i.e.*, DevOps) to synthesize new tests according to the changes made during code deployment and execution. Test amplification is the core idea of the European project STAMP[4] which aims to develop test amplification tools to increase the levels of automation in software testing. In this context, we can adapt our testing approach to generate or create new test suites to detect more inconsistencies. Based on the new changes made within code generators, new test suites have to be generated efficiently to track the non-functional failures.

Leveraging the metamorphic approach for testing large-scale distributed systems In a component-based architecture, distributed and heterogeneous applications involve a set of configurable nodes that communicate with each other. Efficiently testing the different configurations of distributed systems is quite challenging, especially for the non-functional properties. It requires the generation of new configurations of the same distributed application and then run tests across all configurations. An idea is to leverage metamorphic testing in order to evaluate the configuration changes at runtime. A good metamorphic relation can be defined as a comparison between the non-functional properties of a new system configuration and a reference one. We can benefit from this technique to evaluate the non-functional properties of distributed systems such as scalability, security, network efficiency, etc. For instance, we plan to use Kevoree[5], a multi-platform distributed model tool, that facilitates application deployment within a highly configurable distributed environment.

[4] https://stamp.ow2.org
[5] http://kevoree.org/

References

[ABB+14] Mathieu Acher, Olivier Barais, Benoit Baudry, Arnaud Blouin, Johann Bourcier, Benoit Combemale, Jean-Marc Jézéquel, and Noël Plouzeau. Software diversity: Challenges to handle the imposed, opportunities to harness the chosen. In *GDR GPL*, 2014.

[ABB+15] Simon Allier, Olivier Barais, Benoit Baudry, Johann Bourcier, Erwan Daubert, Franck Fleurey, Martin Monperrus, Hui Song, and Maxime Tricoire. Multitier diversification in web-based software applications. *IEEE Software*, 32(1):83–90, 2015.

[ABDDP13] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[ABHPW10] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[ACG+04] Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. *ACM SIGPLAN Notices*, 39(7):231–239, 2004.

[AE09] N Amanquah and OT Eporwei. Rapid application development for mobile terminals. In *2009 2nd International Conference on Adaptive Science & Technology (ICAST)*, pages 410–417. IEEE, 2009.

[Ajw07] Nora Ajwad. Evaluation of automatic code generation tools. *MSc Theses*, 2007.

[BBGM11] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 427–430. IEEE, 2011.

[BBSB15] Mohamed Boussaa, Olivier Barais, Gerson Sunyé, and Benoit Baudry. A novelty search approach for automatic test data generation. In *8th International Workshop on Search-Based Software Testing SBST@ ICSE 2015*, page 4, 2015.

[BCG11] Tobias Betz, Lawrence Cabac, and Matthias Güttler. Improving the development tool chain in the context of petri net-based software development. In *PNSE*, pages 167–178. Citeseer, 2011.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[BDF08] Nurlida Basir, Ewen Denney, and Bernd Fischer. Constructing a safety case for automatically generated code from formal program verification information. In *International Conference on Computer Safety, Reliability, and Security*, pages 249–262. Springer, 2008.

[BFN96] Wolfgang Banzhaf, Frank D Francone, and Peter Nordin. The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets. In *Parallel Problem Solving from NaturePPSN IV*, pages 300–309. Springer, 1996.

[BKK+98] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[BM08] Alexandre Bragança and Ricardo J Machado. Transformation patterns for multi-staged model driven software development. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 329–338. IEEE, 2008.

[BM+15a] Earl T Barr, Mark , Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2015.

[BM15b] Benoit Baudry and Martin Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Computing Surveys (CSUR)*, 48(1):16, 2015.

[BNS13] Andrejs Bajovs, Oksana Nikiforova, and Janis Sejans. Code generation from uml model: State of the art and practical implications. *Applied Computer Systems*, 14(1):9–18, 2013.

[BR04] Andrew Burnard and Land Rover. Verifying and validating automatically generated code. In *Proc. of International Automotive Conference (IAC)*. Citeseer, 2004.

[BSH15] Prathibha A Ballal, H Sarojadevi, and PS Harsha. Compiler optimization: A genetic algorithm approach. *International Journal of Computer Applications*, 112(10), 2015.

[BY07] Guy Bashkansky and Yaakov Yaari. Black box approach for selecting optimization options using budget-limited genetic algorithms. In *Workshop Proceedings*, page 27, 2007.

[CB08] Wonseok Chae and Matthias Blume. Building a family of compilers. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 307–316. IEEE, 2008.

[CBGM12] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 225–236. IEEE Computer Society, 2012.

[CCL+06] WK Chan, Tsong Yueh Chen, Heng Lu, TH Tse, and Stephen S Yau. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering*, 16(05):677–703, 2006.

[CCY98] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[CDM+10] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language

virtualization for heterogeneous parallel computing. In *ACM Sigplan Notices*, volume 45, pages 835–847. ACM, 2010.

[CE00a] Krzysztof Czarnecki and Ulrich W Eisenecker. Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, page 15, 2000.

[CE00b] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CFH+12] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):21, 2012.

[CGH+06] Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. *The Journal of Supercomputing*, 36(2):135–151, 2006.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[CHE+10] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 datasets. In *ACM Sigplan Notices*, volume 45, pages 448–459. ACM, 2010.

[CHH+16] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[CHTZ04] Tsong Yueh Chen, DH Huang, TH Tse, and Zhi Quan Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583. Polytechnic University of Madrid, 2004.

[CMKSP10] T Erkkinen Conrad, T Maier-Komor, G Sandmann, and M Pomeroy. Code generation verification–assessing numerical equivalence between simulink models and generated code. In *4th Conference Simulation and Testing in Algorithm and Software Development for Automobile Electronics*, 2010.

[CO05] John Cavazos and Michael FP O'Boyle. Automatic tuning of inlining heuristics. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 14–14, 2005.

[Con09] Mirko Conrad. Testing-based translation validation of generated code in the context of iec 61508. *Formal Methods in System Design*, 35(3):389–401, 2009.

[CSB+11] Hassan Chafi, Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Anand R Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. *ACM SIGPLAN Notices*, 46(8):35–46, 2011.

[CSS99] Keith D Cooper, Philip J Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *ACM SIGPLAN Notices*, volume 34, pages 1–9. ACM, 1999.

[CST02] Keith D Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 2002.

[Cza05] Krzysztof Czarnecki. Overview of generative software development. In *Unconventional Programming Paradigms*, pages 326–341. Springer, 2005.

[DA13] Charalampos Doukas and Fabio Antonelli. Compose: Building smart & context-aware mobile applications utilizing iot technologies. In *Global Information Infrastructure Symposium, 2013*, pages 1–6. IEEE, 2013.

[DAH11] Melina Demertzi, Murali Annavaram, and Mary Hall. Analyzing the effects of compiler optimizations on application reliability. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 184–193. IEEE, 2011.

[Das11] Benjamin Dasnois. *HaXe 2 Beginner's Guide*. Packt Publishing Ltd, 2011.

[Deb01] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[DF05] Ewen Denney and Bernd Fischer. Certifiable program generation. In *International Conference on Generative Programming and Component Engineering*, pages 17–28. Springer, 2005.

[DGR04] Nelly Delgado, Ann Q Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on software Engineering*, 30(12):859–872, 2004.

[DJB+09] Christophe Dubach, Timothy M Jones, Edwin V Bonilla, Grigori Fursin, and Michael FP O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 78–88. ACM, 2009.

[DL16] Alastair F Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing*, pages 44–47. ACM, 2016.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[DVPY+17] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Martin Monperrus, and Benoit Baudry. The emerging field of test amplification: A survey. *arXiv preprint arXiv:1705.10692*, 2017.

[DW81] Martin D Davis and Elaine J Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM'81 Conference*, pages 254–257. ACM, 1981.

[EAC15] Rodrigo D Escobar, Alekya R Angula, and Mark Corsi. Evaluation of gcc optimization parameters. *Revista Ingenierias USBmed*, 3(2):31–39, 2015.

[ECGN00] Michael D Ernst, Adam Czeisler, William G Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*, pages 449–458. ACM, 2000.

[ELB+08] David P Enot, Wanchang Lin, Manfred Beckmann, David Parker, David P Overy, and John Draper. Preprocessing, classification modeling and feature selection using flow injection electrospray mass spectrometry metabolite fingerprint data. *Nature Protocols*, 3(3):446–470, 2008.

[FB08] Kresimir Fertalj and Mario Brcic. A source code generator based on uml specification. *International journal of computers and communications*, (1):10–19, 2008.

[FFRR15] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172. IEEE, 2015.

[FIP15] Vincenzo Ferme, Ana Ivanchikj, and Cesare Pautasso. A framework for benchmarking bpmn 2.0 workflow management systems. In *International Conference on Business Process Management*, pages 251–259. Springer, 2015.

[FKM+11] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[FMSB11] Franck Fleurey, Brice Morin, Arnor Solberg, and Olivier Barais. Mde to manage communications with and between resource-constrained systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2011.

[FMT+08] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. Milepost gcc: machine learning based research compiler. In *GCC Summit*, 2008.

[FOK02] GG Fursin, Michael FP OBoyle, and Peter MW Knijnenburg. Evaluating iterative compilation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 362–376. Springer, 2002.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[FRSD15] Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. Runtime code generation and data management for heteroge-

neous computing in java. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, pages 16–26. ACM, 2015.

[FS08] Ines Fey and Ingo Stürmer. Code generation for safety-critical systems–open questions and possible solutions. *SAE international journal of passenger cars-electronic and electrical systems*, 1(2008-01-0385):150–155, 2008.

[Fur09] Grigori Fursin. Collective tuning initiative: automating and accelerating development and optimization of computing systems. In *GCC Developers' Summit*, 2009.

[GPB15] Ahmad Nauman Ghazi, Kai Petersen, and Jürgen Börstler. Heterogeneous systems testing techniques: An exploratory survey. In *International Conference on Software Quality*, pages 67–85. Springer, 2015.

[GR96] Chris Gathercole and Peter Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In *Proceedings of the 1st annual conference on genetic programming*, pages 291–296. MIT Press, 1996.

[GS14] Victor Guana and Eleni Stroulia. Chaintracker, a model-transformation trace analysis tool for code-generation environments. In *ICMT*, pages 146–153. Springer, 2014.

[GS15] Victor Guana and Eleni Stroulia. How do developers solve software-engineering tasks on model-based code generators? an empirical study design. In *First International Workshop on Human Factors in Modeling (HuFaMo 2015). CEUR-WS*, pages 33–38, 2015.

[HE08] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174. ACM, 2008.

[He10] Liqiang He. Computer architecture education in multicore era: Is the time to change. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 9, pages 724–728. IEEE, 2010.

[Her03] Jack Herrington. *Code generation in action*. Manning Publications Co., 2003.

[HGE10] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. Automated just-in-time compiler tuning. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 62–72. ACM, 2010.

[HIH16] Abeer Hamdy, Osman Ibrahim, and Ahmed Hazem. A web based framework for pre-release testing of mobile applications. In *MATEC Web of Conferences*, volume 76, page 04041. EDP Sciences, 2016.

[HKW05] Masayo Haneda, Peter MW Knijnenburg, and Harry AG Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 123–132. IEEE, 2005.

[HMSY13] Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013.

[HRV09] Mia Hubert, Peter Rousseeuw, and Tim Verdonck. Robust pca for skewed data and its outlier map. *Computational Statistics & Data Analysis*, 53(6):2264–2274, 2009.

[HSD11] Gustavo Hartmann, Geoff Stead, and Asi DeGani. Cross-platform mobile development. *Mobile Learning Environment, Cambridge*, pages 1–18, 2011.

[HT00] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[Hun11] Robert Hundt. Loop recognition in c++/java/go/scala. In *Proceedings of Scala Days 2011*, June 2011.

[HZG10] Qiming Hou, Kun Zhou, and Baining Guo. Spap: A programming language for heterogeneous many-core systems. Technical report, Technical report, Zhejiang University Graphics and Parallel Systems Lab, 2010.

[IJH+13] Benjamin Inden, Yaochu Jin, Robert Haschke, Helge Ritter, and Bernhard Sendhoff. An examination of different fitness and novelty based selection methods for the evolution of neural networks. *Soft Computing*, 17(5):753–767, 2013.

[IRH09] Mostafa EA Ibrahim, Markus Rupp, and SE-D Habib. Compiler-based optimizations impact on embedded software power consumption. In *Circuits and Systems and TAISA Conference, 2009. NEWCAS-TAISA'09. Joint IEEE North-East Workshop on*, pages 1–4. IEEE, 2009.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *International Conference on Model Driven Engineering Languages and Systems*, pages 128–138. Springer, 2005.

[JK13] Michael R Jantz and Prasad A Kulkarni. Performance potential of optimization phase selection during dynamic jit compilation. *ACM SIGPLAN Notices*, 48(7):131–142, 2013.

[JS14] Sven Jörges and Bernhard Steffen. Back-to-back testing of model-based code generators. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 425–444. Springer, 2014.

[KJB+09] Tim Kapteijns, Slinger Jansen, Sjaak Brinkkemper, Henry Houët, and Rick Barendse. A comparative case study of model driven development vs traditional development: The tortoise or the hare. *From code centric to model centric software engineering: Practices, Implications and ROI*, 22, 2009.

[KKO02] Peter MW Knijnenburg, Toru Kisuki, and Michael FP OBoyle. Iterative compilation. In *Embedded processor design challenges*, pages 171–187. Springer, 2002.

[KKS98] Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230–239. IEEE, 1998.

[Krč12] Peter Krčah. Solving deceptive tasks in robot body-brain co-evolution by searching for behavioral novelty. In *Advances in Robotics and Virtual Reality*, pages 167–186. Springer, 2012.

[KT15] Pakorn Kookarinrat and Yaowadee Temtanapat. Analysis of range-based key properties for sharded cluster of mongodb. In *Information Science and Security (ICISS), 2015 2nd International Conference on*, pages 1–4. IEEE, 2015.

[KVI02] Mahmut Kandemir, N Vijaykrishnan, and Mary Jane Irwin. Compiler optimizations for low power systems. In *Power aware computing*, pages 191–210. Springer, 2002.

[KWTD06] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. Exhaustive optimization phase order space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 306–318. IEEE Computer Society, 2006.

[KWTD09] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1):1, 2009.

[KZM+03] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *ACM SIGPLAN Notices*, volume 38, pages 12–23. ACM, 2003.

[LAS14] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Notices*, volume 49, pages 216–226. ACM, 2014.

[LCL08] San-Chih Lin, Chi-Kuang Chang, and San-Chih Lin. Automatic selection of gcc optimization options using a gene weighted genetic algorithm. In *Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific*, pages 1–8. IEEE, 2008.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[LPF+10] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Multi-objective exploration of compiler optimizations for real-time systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pages 115–122. IEEE, 2010.

[LS08] Joel Lehman and Kenneth O Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336, 2008.

[LTC15] Li Li, Tony Tang, and Wu Chou. A rest service framework for fine-grained resource management in container-based cloud. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 645–652. IEEE, 2015.

[LYZ] Ruici Luo, Wei Ye, and Shikun Zhang. Towards a deployment system for cloud applications.

[MÁCZCA+14] Antonio Martínez-Álvarez, Jorge Calvo-Zaragoza, Sergio Cuenca-Asensi, Andrés Ortiz, and Antonio Jimeno-Morenilla. Multi-objective adaptive evolutionary strategy for tuning compilations. *Neurocomputing*, 123:381–389, 2014.

[mbo] Notice settings. https://noticegcc.wordpress.com/.

[McK98] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[Mer14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[MFS90] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[MHC14] Paul Marinescu, Petr Hosek, and Cristian Cadar. Covrig: a framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 93–104. ACM, 2014.

[MHH13] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1012–1021. IEEE Press, 2013.

[MJL+06] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, 2, 2006.

[MK01] LI Manolache and Derrick G. Kourie. Software testing using model programs. *Software: Practice and Experience*, 31(13):1211–1236, 2001.

[MNC+16] Luiz GA Martins, Ricardo Nobre, João MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):8, 2016.

[MND+14] Luiz GA Martins, Ricardo Nobre, Alexandre CB Delbem, Eduardo Marques, and João MP Cardoso. Exploration of compiler optimization sequences using clustering-based selection. In *Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 63–72. ACM, 2014.

[Mol09] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. " O'Reilly Media, Inc.", 2009.

[MPP07] Leonardo Mariani, Sofia Papagiannakis, and Mauro Pezze. Compatibility and regression testing of cots-component-based software. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 85–95. IEEE, 2007.

[MRA+16] Víctor Medel, Omer Rana, Unai Arronategui, et al. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 257–262. ACM, 2016.

[Nai16] Nitin Naik. Migrating from virtualization to dockerization in the cloud: Simulation and evaluation of distributed systems. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA), 2016 IEEE 10th International Symposium on the*, pages 1–8. IEEE, 2016.

[NAIT12] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. Random testing of c compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.

[NF13] Mena Nagiub and Wael Farag. Automatic selection of compiler options using genetic techniques for embedded software design. In *Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on*, pages 69–74. IEEE, 2013.

[NHI13] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of c compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, pages 88–93, 2013.

[NRZ06] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. *ACM SIGPLAN Notices*, 41(6):132–143, 2006.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[PB11] Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. In *Software Automatic Tuning*, pages 335–351. Springer, 2011.

[PBG05] Rui Pais, SP Barros, and Luís Gomes. A tool for tailored code generation from petri net models. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 1, pages 8–pp. IEEE, 2005.

[PCC+16] Matthew Patrick, Andrew P Craig, Nik J Cunniffe, Matthew Parry, and Christopher A Gilligan. Testing stochastic software using pseudo-oracles. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 235–246. ACM, 2016.

[PE04] Zhelong Pan and Rudolf Eigenmann. Rating compiler optimizations for automatic performance tuning. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 14. IEEE Computer Society, 2004.

[PE06] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[PGL14] Geoffrey Papaux, Daniel Gachet, and Wolfram Luithardt. Processor virtualization on embedded linux systems. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 65–69. IEEE, 2014.

[PHB15] James Pallister, Simon J Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015.

[PHP16] René Peinl, Florian Holzschuher, and Florian Pfitzer. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265–282, 2016.

[PKC11] Eunjung Park, Sameer Kulkarni, and John Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 65–74. ACM, 2011.

[PMV+13] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. *Procedia Computer Science*, 18:1312–1321, 2013.

[PV15] Alireza Pazirandeh and Evelina Vorobyeva. Evaluation of cross-platform tools for mobile development. 2015.

[RAO92] Debra J Richardson, Stephanie Leif Aha, and T Owen O'malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, pages 105–118. ACM, 1992.

[RFBJ13] Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel. Efficient high-level abstractions for web programming. In *ACM SIGPLAN Notices*, volume 49, pages 53–60. ACM, 2013.

[RHS10] Sebastian Risi, Charles E Hughes, and Kenneth O Stanley. Evolving plastic neural networks with novelty search. *Adaptive Behavior*, 18(6):470–491, 2010.

[RPFD14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[RSF+15] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *ACM SIGPLAN Notices*, volume 50, pages 167–180. ACM, 2015.

[RT06] Filippo Ricca and Paolo Tonella. Detecting anomaly and failure in web applications. *IEEE MultiMedia*, 13(2):44–51, 2006.

[SA13] Abdel Salam Sayyad and Hany Ammar. Pareto-optimal search-based software engineering (posbse): A literature survey. In *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2013 2nd International Workshop on*, pages 21–27. IEEE, 2013.

[SC96] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on software Engineering*, 22(11):777–793, 1996.

[SC03] Igno Sturmer and Mirko Conrad. Test suite design for code generation tools. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 286–290. IEEE, 2003.

[SCDP07] Ingo Stuermer, Mirko Conrad, Heiko Doerr, and Peter Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622, 2007.

[SCTF16] Cristian Constantin Spoiala, Alin Calinciuc, Corneliu Octavian Turcu, and Constantin Filote. Performance comparison of a webrtc server on docker versus virtual machine. In *2016 International Conference on Development and Application Systems (DAS)*, pages 295–298. IEEE, 2016.

[SDH+14] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 639–652. ACM, 2014.

[SH09] Hesham Shokry and Mike Hinchey. Model-based verification of embedded software. *IEEE Computer*, 42(4):53–59, 2009.

[ŠLG15] Domagoj Štrekelj, Hrvoje Leventić, and Irena Galić. Performance overhead of haxe programming language for cross-platform game development. *International Journal of Electrical and Computer Engineering Systems*, 6(1):9–13, 2015.

[SOMA03] Mark Stephenson, Una-May OReilly, Martin C Martin, and Saman Amarasinghe. Genetic programming applied to compiler heuristic optimization. In *European Conference on Genetic Programming*, pages 238–253. Springer, 2003.

[SP15] Stepan Stepasyuk and Yavor Paunov. Evaluating the haxe programming language-performance comparison between haxe and platform-specific languages. 2015.

[SPF+07] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[SRC+12] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.

[SWC05] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–6. ACM, 2005.

[SWES16] Yu Sun, Jules White, Sean Eade, and Douglas C Schmidt. Roar: A qos-oriented modeling framework for automated cloud resource allocation and optimization. *Journal of Systems and Software*, 116:146–161, 2016.

[SZP12] Thayalan Sandran, Mohamed Nordin B Zakaria, and Anindya Jyoti Pal. A genetic algorithm approach towards compiler flag selection based on compilation and execution duration. In *Computer & Information Science (ICCIS), 2012 International Conference on*, volume 1, pages 270–274. IEEE, 2012.

[TCR12] Michele Tartara and Stefano Crespi Reghizzi. Parallel iterative compilation: using mapreduce to speedup machine learning in compilers. In *Proceedings of third international workshop on MapReduce and its Applications Date*, pages 33–40. ACM, 2012.

[TVVA03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In *Code Gen-*

eration and Optimization, 2003. CGO 2003. International Symposium on, pages 204–215. IEEE, 2003.

[TWZS10] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 270–279. IEEE, 2010.

[VJ01] Madhavi Valluri and Lizy K John. Is compiling for performancecompiling for power? In *Interaction between Compilers and Computer Architectures*, pages 101–115. Springer, 2001.

[Vou90] Mladen A Vouk. Back-to-back testing. *Information and software technology*, 32(1):34–45, 1990.

[WS90] Deborah Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *ACM SIGPLAN Notices*, volume 25, pages 137–146. ACM, 1990.

[XDOR+15] Miguel G Xavier, Israel C De Oliveira, Fabio D Rossi, Robson D Dos Passos, Kassiano J Matteussi, and César AF De Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 253–260. IEEE, 2015.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[ZHT+04] Zhi Quan Zhou, DH Huang, TH Tse, Zongyuan Yang, Haitao Huang, and TY Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, pages 346–351, 2004.

[ZSH09] Shengtong Zhong, Yang Shen, and Fei Hao. Tuning compiler optimization options via simulated annealing. In *Future Information Technology and Management Engineering, 2009. FITME'09. Second International Conference On*, pages 305–308. IEEE, 2009.

[ZSP+06] Sergey V Zelenov, Denis V Silakov, Alexander K Petrenko, Mirko Conrad, and Ines Fey. Automatic test generation for model-based code generators. In *ISoLA*, pages 75–81, 2006.

List of Figures

List of Tables

Acronyms

Cgroups Control Groups

CR CPU Consumption Reduction

DSL Domain-Specific Language

GA Genetic Algorithm

GP Generative Programming

GPL General-Purpose Language

GUT Generator Under Test

LCL Lower Control Limit

MR Metamorphic Relation

MT Metamorphic Testing

NM Novelty Metric

NS Novelty Search

NSGA-II Non-dominated Sorting Genetic Algorithm

PCA Principal Component Analysis

QoS Quality of Service

R Range

RS Random Search

S Speedup

SBSE Search-Based Software Engineering

SD Score Distance

UCL Upper Control Limit

Abstract

Generative software development has paved the way for the creation of multiple generators (code generators and compilers) that serve as a basis for automatically producing code to a broad range of software and hardware platforms. With full automatic code generation, users are able to rapidly synthesize software artifacts for various software platforms. In addition, they can easily customize the generated code for the target hardware platform since modern generators (i.e., C compilers) become highly configurable, offering numerous configuration options that the user can apply. Consequently, the quality of generated software becomes highly correlated to the configuration settings as well as to the generator itself.

In this context, it is crucial to verify the correct behavior of generators. Numerous approaches have been proposed to verify the functional outcome of generated code but few of them evaluate the non-functional properties of automatically generated code, namely the performance and resource usage properties.

This thesis addresses three problems:

Non-functional testing of generators: We benefit from the existence of multiple code generators with comparable functionality (i.e., code generator families) to automatically test the generated code. We leverage the metamorphic testing approach to detect non-functional inconsistencies in code generator families by defining metamorphic relations as test oracles. We define the metamorphic relation as a comparison between the variations of performance and resource usage of code, generated from the same code generator family. We evaluate our approach by analyzing the performance of HAXE, a popular code generator family. Experimental results show that our approach is able to automatically detect several inconsistencies that reveal real issues in this family of code generators.

Generators auto-tuning: We exploit the recent advances in search-based software engineering in order to provide an effective approach to tune generators (i.e., through optimizations) according to user's non-functional requirements (i.e., performance and resource usage). We also demonstrate that our approach can be used to automatically construct optimization levels that represent optimal trade-offs between multiple non-functional properties such as execution time and resource usage requirements. We evaluate our approach by verifying the optimizations performed by the GCC compiler. Our experimental results show that our approach is able to auto-tune compilers and construct optimizations that yield to better performance results than standard optimization levels.

Handling the diversity of software and hardware platforms in software testing: Running tests and evaluating the resource usage in heterogeneous environments is tedious. To handle this problem, we benefit from the recent advances in lightweight system virtualization, in particular container-based virtualization, in order to offer effective support for automatically deploying, executing, and monitoring code in heterogeneous environment, and collect non-functional metrics (e.g., memory and CPU consumption). This testing infrastructure serves as a basis for evaluating the experiments conducted in the two first contributions.