Contents

Contents **1**

Introduction **1**

I Background and State of the Art **6**

1 Background **7**

1.1 From classical software development to generative programming 7

1.2 Software development tool chain overview 9

1.3 Code generators 10

1.3.1 Complexity 11

1.3.2 Software-platform heterogeneity 12

1.4 Compilers 12

1.4.1 Complexity 12

1.4.2 Hardware-platform heterogeneity 13

1.4.3 Problems that still remain 14

2 State of the art **15**

II Contributions **16**

3 Optimizing compiler auto-tuning **17**

3.1 Introduction 17

3.2 Motivation 19

3.2.1 Compiler Optimizations 19

3.2.2 Example: GCC Compiler 21

3.3 Evolutionary Exploration of Compiler Optimizations 21

3.3.1 Novelty Search Adaptation 21

3.3.2 Novelty Search For Multi-objective Optimization 25

3.4 Evaluation 25

3.4.1 Research Questions 26

3.4.2 Experimental Setup 26

3.4.3 Experimental Methodology and Results 29

3.4.4 Discussions 36

3.4.5 Threats to Validity 36

3.5 Conclusion and Future Work 37

4 Automatic non-functional testing of code generators families **39**

4.1 Introduction 39

4.2 Motivation 41

4.2.1 Code Generator Families 41

4.2.2 Functional Correctness of a Code Generator Family 43

4.2.3 Performance Evaluation of a Code Generator Family 43

4.3 Approach Overview 44

4.3.1 Non-Functional Testing of a Code Generator Family: a Common Process 44

4.3.2 An Infrastructure for Non-functional Testing Using System Containers 46

4.4 Evaluation 46
4.4.1 Experimental Setup 47
4.4.2 Experimental Results 50
4.4.3 Threats to Validity 53

5 An infrastructure for resource monitoring based on system containers 57
5.1 Introduction 57
5.2 System Containers as Execution Platforms 57
5.3 Runtime Testing Components 59
5.3.1 Monitoring Component 59
5.3.2 Back-end Database Component 60
5.3.3 Front-end Visualization Component 61

III Conclusion and perspectives 62

6 Conclusion and perspectives 63

References 64

Introduction

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-constraint 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[REF], that high-level abstract development received more and more attraction to tame with the runtime heterogeneity of platforms and technological stacks that exists in several domains such as mobile or Internet of Things. Therefore, software developers tend to increasingly use generative programming and model-based techniques in order to reduce the effort of software development and maintenance by developing at a higher-level of abstraction through the use of domain-specific languages for example. Consequently, the new advances in hardware and platform specifications have paved the way for the creation of multiple code generators and compilers that serve as a basis to target different ranges of platforms. On the one hand, code generators are needed to translate high-level system specifications (e.g., textual or graphical modeling language) into multi-target executable code (e.g., general-purpose languages such as Java, C++, etc). On the other hand, a set of platform-specific compilers are also needed in turn to transform general-purpose languages into machine code (i.e., binaries).

However, code generators as well as compilers are known to be difficult to understand since they involve a set of complex and heterogeneous technologies and configurations which make the activities of design, implementation, and testing very hard and time-consuming [REF]. Thus, their output must be checked with almost the same, expensive effort as is needed for manually written code. Thus, testing code generators and compilers becomes crucial and necessary to guarantee that no errors are incorporated by inappropriate modelling or by the code generator it self. Faulty code generators or compilers 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 [ref] to verify the functional outcome of generated code. However, there is a lack of solutions that pay attention to evaluate the non-functional properties of produced code.

In this thesis, we seek to test the non-functional properties, i.e., resource usage properties, of generated code by either code generators or compilers. On the one hand, since many different target software platforms can be targeted by the code generator, we would help code generator creators to monitor the execution of generated code for different targets and have a deep understanding of its non-functional behavior in terms of resource usage. Thus, we can detect bugs and inconsistencies caused by some faulty code generators. On the other hand, compilers may have a huge number of optimizations and code transformation rules that can be applied during the code generation process. As a consequence, we would help compiler users to select the best optimizations that satisfy specific resource usage requirements for a broad range of programs and hardware architectures.

This thesis addresses the problem of non-functional testing of code generators and compilers. In particular, it aims at offering effective support for collecting data about resource consumption (e.g., CPU, memory), as well as efficient mechanisms to help software developers to choose the best configuration that satisfy specific non-functional requirements and lead to performance improvement.

Challenges

In existing solutions that aim to test code generators and compilers, we find two important challenges. Addressing these challenges, which are described below, is the objective of the present work.

- **Monitoring code generators/compilers behavior:** For testing the non-functional properties of code generators and compilers, developers generally need to deploy and execute generated software artifacts on different execution platforms. Then, they have to collect and compare information about the performance and efficiency of the generated code. Afterwards, they report issues related to the code generation process such as incorrect typing, memory management leaks, etc. Currently, there is a lack of automatic solutions to check the performance issues such as the inefficiency (high memory/CPU consumption) of the generated code. In fact, developers often use manually several platform-specific profilers, debuggers, and monitoring tools [REF] in order to find some inconsistencies or bugs during code execution. Ensuring the quality of generated code in this case can refer to several non-functional properties such as code size, resource or energy consumption, execution time, among others [REF]. Due to the heterogeneity of execution platforms and hardwares, collecting information about the non-functional properties of generated code becomes very hard and time-consuming task since developers have to analyze and verify the generated code for different target platforms using platform-specific tools.

- **Tuning code generators/compilers:** 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, response, 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 challenging and time-consuming problem. Due to the complex interactions and the unknown effect of optimizations, users find difficulties to choose the adequate compiler configuration that satisfy a specific non-functional requirement.

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

RQ1. How can we help users to choose the adequate compiler/code generator configuration that satisfy user non-functional requirements?

RQ2. How can we provide efficient support for resource consumption monitoring and management?

RQ3. How can we automatically detect inconsistencies and non-functional errors within code generators?

Contributions

This thesis establishes two core contributions in the field of non-functional testing of code generators and compilers. These contributions are briefly described in the rest of this section.

Contribution: automatic compiler auto-tuning according to the non-functional requirements. We provide facilities to compiler users to auto-tune compilers according to their non-functional 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: automatic detection of inconsistencies within code generators families. In this contribution, we propose a new approach for testing and monitoring code generators families. This approach try to find automatically real issues in existing code generators. In particular, we show that we could find 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.

Contribution: a microservice-based infrastructure for runtime deployement and monitoring of generated code. We propose a micro-service infrastructure to ensure the deployment and monitoring of different variants of generated code. This isolated and sand-boxing environment is based on system containers, as execution platforms, to provide a fine-grained understanding and analysis of resource usage in terms of CPU and memory. This contribution answers mainly *RQ2* but the same infrastructure is used across all experiments in *RQ1* and *RQ3*.

Overview of this thesis

Publications

- Mohamed Boussaa, Olivier Barais, Benoit Baudry, Gerson Suny: **NOTICE: A Framework for Non-functional Testing of Compilers**. In *2016 IEEE International Conference on Software Quality, Reliability & Security (QRS 2016)*, Vienna, Austria, August 2015.

- Mohamed Boussaa, Olivier Barais, Gerson Suny, Benoit 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 Suny, Benoit 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.

Under Review

Mohamed Boussaa, Olivier Barais, Gerson Suny, Benoit Baudry: **Automatic Non-functional Testing of Code Generators Families**. In *10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017).*

Part I

Background and State of the Art

Chapter 1

Background

In this chapter, we discuss different domains and concepts applied in our proposal, including compilers, code generators, optimizations and non-functional requirements. The objective of this chapter is to give a brief introduction to these concerns, used throughout the thesis. This introduction aims at providing a better understanding of the background and context in which our work takes place, as well as the terminology and concepts presented in the next chapters.

The chapter is structured as follows:

1.1 From classical software development to generative programming

The history of software development shows a continuous increase of complexity in several aspects of the software development process [1]. In fact, 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-constraint devices with only a few KB of RAM. Effectively developing software artifacts for multiple target platforms and hardware technologies is then becoming increasingly important and developing software. Furthermore, the increasing relevance of software in general and the higher demand in quality and performance contribute to the complexity of software development. In comparison to the classical approach where software development was carried out manually, todays modern development requires more automatic and flexible approaches to handle this given complexity. 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. 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* [3]. This paradigm offers the promise of moving from ”one-of-a-kind” software systems to the semi-automated manufacture of wide diversity of software.

Generative software engineering consists on using higher-level programming techniques such as meta-programming, modeling, DSL, etc in order to automatically generate efficient code for the the target software platform. In principle a software development process can be seen as a mapping between a problem space and a solution space [2] (see Figure 2.1).

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, on the other hand, consists of a set of implementation components, which can be composed to create system implementations (for example, the generation of platform-specific software components written using general-purpose languages 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/DSL into specific implementation components) and optimizations (i.e., optimization can be applied during code generation to enhance some of the non-functional properties such as execution speed). It defines also the dependencies and settings among the domain specific concepts and features.

These schema integrates several powerful concepts from Model Driven Engineering (MDE), such as domain-specific languages, feature modeling, generators, components, and software architecture.

Some commonly benefits of such software developing architecture 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

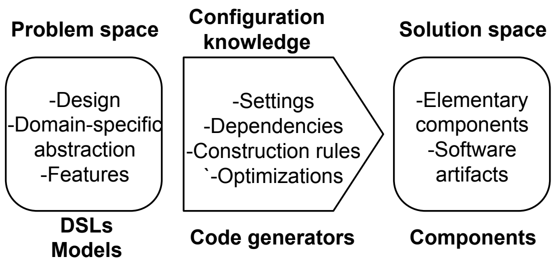


Figure 1.1: Overview of the Docker-based testing architecture

Among the main contributions of this thesis is to verify the correct mapping between the problem space and solution space. In other words, we would evaluate the impact of applied configurations during code transformation on the resource usage requirements.

In the following section, we present a general overview of the abstract software development tool chain and the main actors that are involved.

1.2 Software development tool chain overview

The process of generative software development involves many different technologies. In this section, we describe in more details the different activities and actors involved to transform high-level specification into executable programs and that from design time to runtime. Figure 2.2 review the different steps of this chain. We distinguish four main tasks:

- *Software design:* As part of the generative programming process, the first step consists on representing the system abstraction. Thus, software designers define, at design time, softwares behavior using domain models or 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 need to be modeled in software. These models are specified using a high-level abstract languages (DSLs). Domain-specific languages (DSLs) improve programmer productivity by providing high-level abstractions for the development of applications in a particular domain.

- *Code generation:* Software developers use model-driven techniques in order to generate automatically code. Thus, 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.

  TOFINISH

- *Software development:* TODO

- *Compilation:* TODO

.

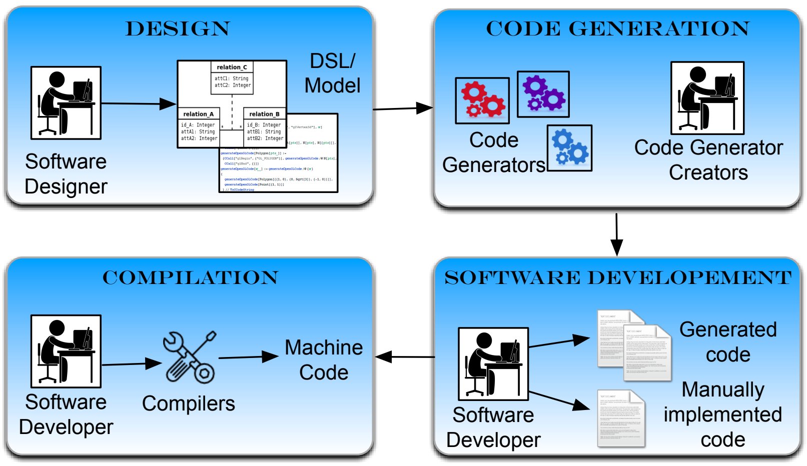


Figure 1.2: Overview of the Docker-based testing architecture

1.3 Code generators

The main goal of generators is to produce software systems from higher-level specifications. Generators bridge the wide gap between the high-level system description and the executable. Generators are based on domain-specific models which define the semantics of the system specification language and also contain the knowledge of how to produce efficient implementations[REF]. We distinguish two major types of code generators: rule-based model-to-model transformation languages (such as ATL) and template-based model-to-text transformation languages (such as Acceleo) to translate high-level system specifications into executable code and scripts.

1.3.1 Complexity

The complexity of code generators remains on the transformation rules and code generation process. In fact, 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 fix it. This task is error prone when done manually. We believe that flexible traceability tools are needed to collect and visualize information about the architecture and operational mechanics of code generators, to reduce the challenges that developers face during their life-cycle.[ref W]

Moreover, the generated code has to meet certain performance requirements (e.g. execution speed, response time, memory consumption, utilization of resources, etc.). The challenge is that the structure of the specification is usually very different from the structure of the implementation: there is no simple one-to-one correspondence between the concepts in the specification and the concepts in the implementation. Efficient implementations are then computed at generation time by applying domain-specific optimizations and replacing, merging, adding, and removing components.

Challenge: Fully automatic program synthesis offers many gains over traditional software development methods. e.g., speed of development, increased adaptability and reliability. But code generators are complex pieces of software themselves that may contain bugs.

- Can you trust the code-generator?
- How can the correctness of the generated code be verified?

1.3.2 Software-platform heterogeneity

The key concept of code generators is to produce code in a general-purpose language, such as Java or C++, that can be compiled and executed. Target execution platforms of the generated code are heterogeneous and diverse. for example, although Android provides Java syntax, it uses its own Google libraries and creates byte code that will not run on the standard JVM (Java Virtual Machine). This means that consumers are carrying devices that support different programming languages and developers will usually need to create multiple clients in this heterogeneous environment.

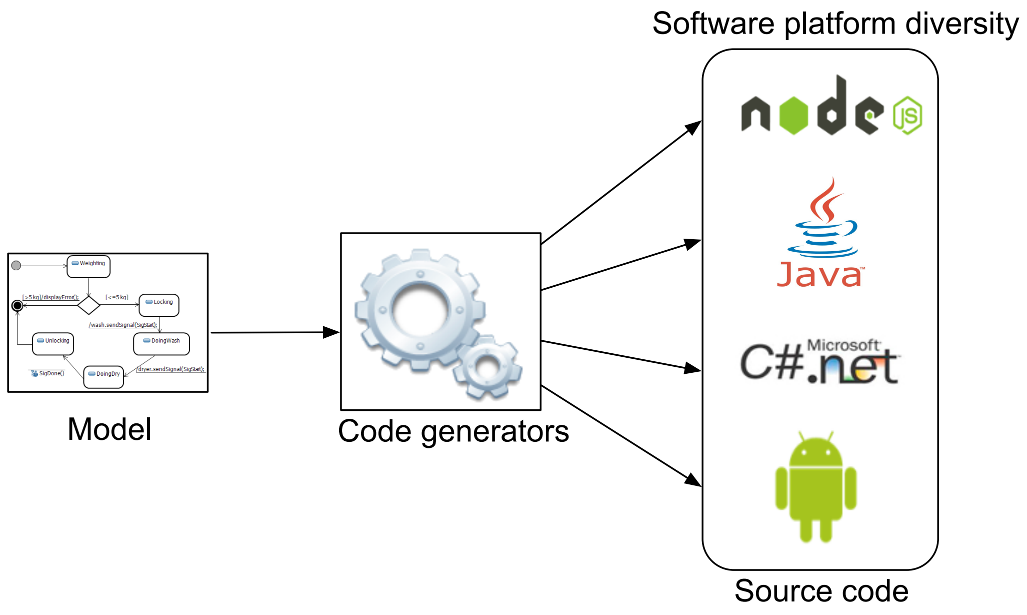


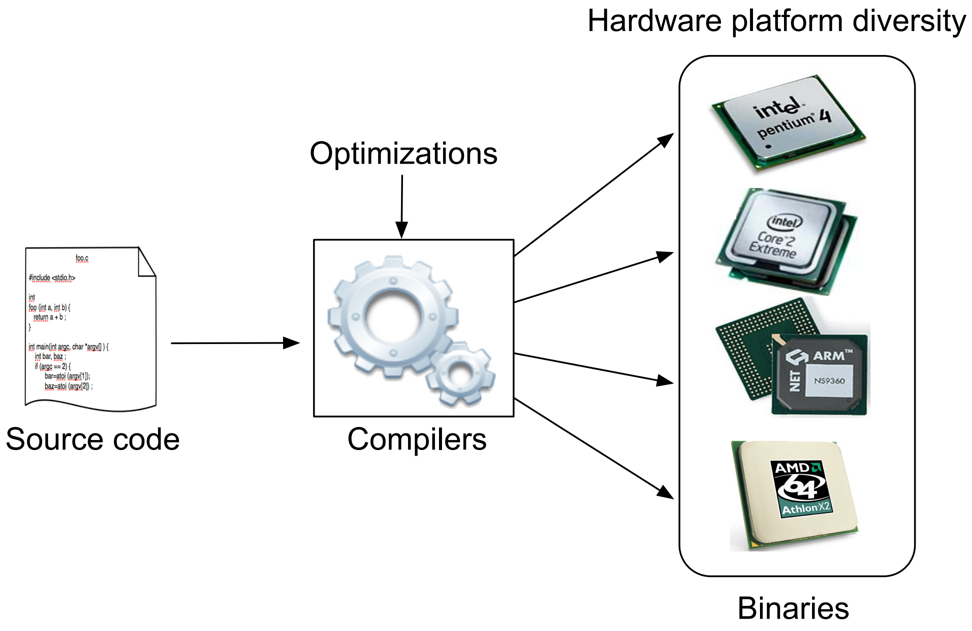
Figure 1.3: Overview of the Docker-based testing architecture

1.4 Compilers

1.4.1 Complexity

Modern compilers implement a number of optimizations. Each optimization tries to improve the performance of certain target applications. Improvement of source code programs in terms of performance can refer to several different non-functional properties of the produced code such as code size, resource or energy consumption, execution time, among others [?, ?]. Testing non-functional properties is more challenging Thus, the determination of optimal settings of compiler optimizations has been identified as a major problem because 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 underlying system architecture, the target application, and the available optimizations of the compiler.

1.4.2 Hardware-platform heterogeneity



```
#include <stdio.h>

int
foo (int a, int b) {
  return a + b ;
}

int main(int argc, char *argv[] ) {
   int bar, baz ;
   if (argc == 2) {
        bar=atoi (argv[1]);
        baz=atoi (argv[2]) ;
```

Figure 1.4: Overview of the Docker-based testing architecture

Generally, software developers use different compilers in order to compile their source code program and execute it on top of a board range target platforms and processors such as arm, intel, amd processors.

Compilers can be classified depending on the platform on which their generated code executes. This is known as the target platform. A native compiler is one which output is intended to directly run on the same type of processor architecture and operating system that the compiler itself runs on. In the counter part, the output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment.

Given the complexity of new emerging processors architecture, rapidly evolving hardware and compiler options, it is not easy to deliver satisfactory levels of performance on modern processors.

Some of the questions that developers have to answer when facing Hardware diversity which optimizations are applied by compiler users in order to satisfy the non-functional properties of a broad range of programs and hardware architectures such as energy consumption, execution time, etc.

1.4.3 Problems that still remain

- Resource usage monitoring of generated code: Due to the software and hardware heterogeneity, monitoring the resource usage of each execution platform is still challenging and time-consuming.
- Auto-tuning compilers:
- Detecting bugs in code generators:

Chapter 2

State of the art

2.1 Introduction

2.2 Testing compilers

Our work is related to iterative compilation research field. 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 using search-based techniques (SBSE) 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. The vast majority of the work on iterative compilation focuses on increasing the speedup of new optimized code compared to standard compiler optimization levels [?, ?, ?, ?, ?, ?, ?, ?]. It has been proven that optimizations are highly dependent on target platform and input program. Compared to our proposal, none of the previous work has studied the impact of compiler optimizations on resource usage. In this work, we rather focus on compiler optimizations related to resource consumption, while bearing in mind the performance improvement.

Novelty Search has never been applied in the field of iterative compilation. Our work presents the first attempt to introduce diversity in the compiler optimization problem. The idea of NS has been introduced by Lehman et al. [?]. It has been often evaluated in deceptive tasks and especially applied to evolutionary robotics [?, ?] (in the context of neuroevolution). NS can be easily adapted to different research fields. In a previous work [?], we have adapted the general idea of NS to the test data generation problem where novelty score was calculated as the Manhattan distance between the different vectors representing the test data. The evaluation metric of generated test suites is the structural coverage of code. In this paper, the evaluation metric represents the non-functional improvements and we are calculating the novelty score as the symmetric difference between optimization sequences.

For multi-objective optimizations, we are not the first to address this problem. New approaches have emerged recently to find trade-offs between non-functional properties [?, ?, ?]. Hoste et al. [?], which is the most related work to our proposal, propose COLE, an automated tool for optimization generation using a multi-objective approach namely SPEA2. In their work, they try to find Pareto optimal optimization levels that present a trade-off between execution and compilation time of generated code. Their experimental results show that the obtained optimization sequences perform better than standard GCC optimization levels. NOTICE provides also a fully automated approach to extract non-functional properties. However, NOTICE differs from COLE because first, our proposed container-based infrastructure is more generic and can be adapted to other case studies (i.e., compilers, code generators, etc.). Second, we provide facilities to compiler users to extract resource usage metrics using our monitoring components. Finally, our empirical study investigates different trade-offs compared to previous work in iterative compilation.

2.3 Testing 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 [?, ?]. As an example, Strekelj et al. [?] implemented a simple 2D game in both the Haxe programming language and the native environment and evaluated the difference in performance between the two versions of code. They showed that the generated code through Haxe has better performance than hand-written code.

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 [?, ?], authors compare the performance of a set of cross-platform code generators and presented the most efficient tools.

The container-based infrastructure has been also applied to the software testing, especially in the cloud [?]. Sun et al. [?] 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 informations about the resource usage of deployed web servers.

Most of the previous work on code generators testing focuses on checking the correct functional behavior of generated code. Stuermer et al. [?] 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 the code execution with the output of the model execution. 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. They have used Simulink as a simulation environment of models. In [?], authors presented 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.

Compared to our proposal, none of the previous work has provided an automatic approach for testing and monitoring the generated code in terms of non-functional properties.

Part II

Contributions

Chapter 3

Optimizing compiler auto-tuning

Generally, compiler users apply different optimizations to generate efficient code with respect to non-functional properties such as energy consumption, execution time, etc. However, due to the huge number of optimizations provided by modern compilers, finding the best optimization sequence for a specific objective and a given program is more and more challenging.

This chapter presents NOTICE, a component-based framework for non-functional testing of compilers through the monitoring of generated code in a controlled sand-boxing environment. 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 user requirements and construct optimizations that yield to better performance results than standard optimization levels. We also demonstrate that NOTICE 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.

3.1 Introduction

Compiler users tend to improve software programs in a safe and profitable way. Modern compilers provide a broad collection of optimizations that can be applied during the code generation process. For functional testing of compilers, software testers generally use to run a set of test suites on different optimized software versions and compare the functional outcome that can be either pass (correct behavior) or fail (incorrect behavior, crashes, or bugs) [?, ?, ?].

For non-functional testing, improvement of source code programs in terms of performance can refer to several different non-functional properties of the produced code such as code size, resource or energy consumption, execution time, among others [?, ?]. Testing non-functional properties is more challenging because 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 underlying system architecture, the target application, and the available optimizations of the compiler.

In some cases, these optimizations may negatively decrease the quality of the software and deteriorate application performance over time [?]. As a consequence, compiler creators usually define fixed and program-independent sequence optimizations, which are based on their experiences and heuristics. For example, in GCC, we can distinguish optimization levels from O1 to O3. Each optimization level involves a fixed list of compiler optimization options and provides different trade-offs in terms of non-functional properties. Nevertheless, there is no guarantee that these optimization levels will perform well on untested architectures or for unseen applications. Thus, it is necessary to detect possible issues caused by source code changes such as performance regressions and help users to validate optimizations that induce performance improvement.

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. Several research efforts try to optimize a single criterion (usually the execution time) [?, ?, ?] and ignore other important non-functional properties, more precisely resource consumption properties (e.g., memory or CPU usage) that must be taken into consideration and can be equally important in relation to the performance. 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) [?]. 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.

In this paper, we propose NOTICE (as NOn-functional TestIng of CompilErs), a component-based framework for non-functional testing of compilers through the monitoring of generated code in a controlled sand-boxing environment. 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 paper, we make the following contributions:

- We introduce a novel formulation of the compiler optimization problem using Novelty Search [?]. We 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.
- We propose a micro-service infrastructure to ensure the deployment and monitoring of different variants of optimized code. In this paper, we focus more on the relationship between runtime execution of optimized code and resource consumption profiles (CPU and memory usage) by providing a fine-grained understanding and analysis of compilers behavior regarding optimizations.
- We also demonstrate that NOTICE can be used to automatically construct optimization levels that represent optimal trade-offs between multiple non-functional properties, such as execution time, memory usage, CPU consumption, etc.

The paper is organized as follows. Section II describes the motivation behind this work. A search-based technique for compiler optimization exploration is presented in Section III. We present in Section IV our infrastructure for non-functional testing using micro-services. The evaluation and results of our experiments are discussed in Section V. Finally, related work, concluding remarks, and future work are provided in Sections VI and VII.

3.2 Motivation

3.2.1 Compiler Optimizations

In the past, researchers have shown that the choice of optimization sequences may influence software performance [?,?]. 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 [?, ?, ?]. 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 [?].

To explore the large optimization space, users have to evaluate the effect of optimizations according to a specific performance objective (see Figure 1). Performance can depend 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. Many approaches [?, ?] have attempted to solve this optimization selection problem using techniques such as Genetic Algorithms (GAs), machine learning techniques, etc.

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 [?]. Thus, it becomes necessary to test the non-functional properties of optimized code and check its behavior regarding optimizations that can lead to performance improvement or regression.

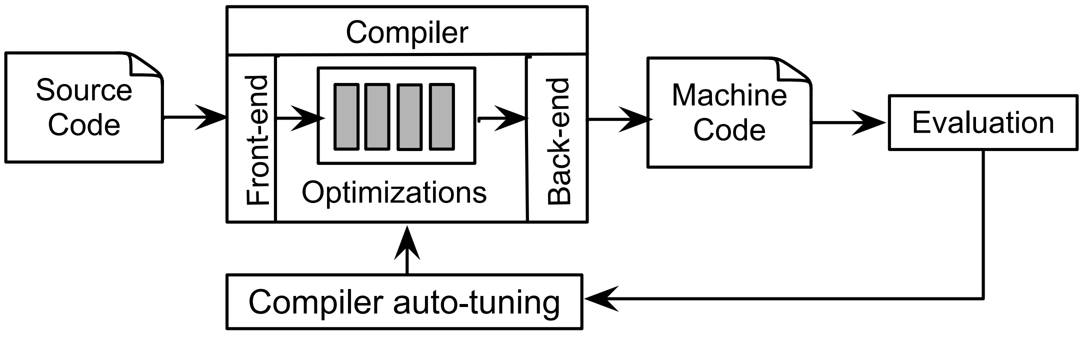


Figure 3.1: Process of compiler optimization exploration

3.2.2 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. For instance, we count 76 optimization flags that are enabled by the four default optimization levels (O1, O2, O3, Ofast). In fact, O1 reduces 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 eight 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. The full list of optimizations is available here [?]. 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.

3.3 Evolutionary Exploration of Compiler Optimizations

Many techniques (meta-heuristics, constraint programming, etc.) can be used to explore the large set of optimization combinations of modern compilers. In our approach, we 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.

3.3.1 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 [?] 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. 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 [?], 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. 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. Many methods are proposed to overcome this problem [?]. 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 non-functional metrics of explored sequences such as memory consumption. We describe in more details the way we are collecting these non-functional metrics in section 4.

Generally, NS acts like GAs (Example of GA use in [?]). 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. Algorithm 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. Line 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.

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: $initialize\_parameters(\mathcal{L}, \mathcal{T}, \mathcal{N}, \mathcal{K})$
2: $create\_archive(\mathcal{L})$
3: $generated\_code \leftarrow compile(\text{"-O0"}, \mathcal{C})$
4: $minimum\_usage \leftarrow execute(generated\_code)$
5: $population \leftarrow random\_sequences(\mathcal{O})$
6: repeat
7:     for $sequence \in population$ do
8:         $generated\_code \leftarrow compile(sequence, \mathcal{C})$
9:         $memory\_usage \leftarrow execute(generated\_code)$
10:        $novelty\_metric(sequence) \leftarrow distFromKnearest(archive, population, \mathcal{K})$
11:        if $novelty\_metric > \mathcal{T}$ then
12:            $archive \leftarrow archive \cup sequence$
13:        end if
14:        if $memory\_usage < minimum\_usage$ then
15:            $best\_sequence \leftarrow sequence$
16:            $minimum\_usage \leftarrow memory\_usage$
17:        end if
18:    end for
19:    $new\_population \leftarrow generate\_new\_population(population)$
20:    $generation \leftarrow generation + 1$
21: until $generation = \mathcal{N}$
22: return $best\_sequence$
```

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 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.

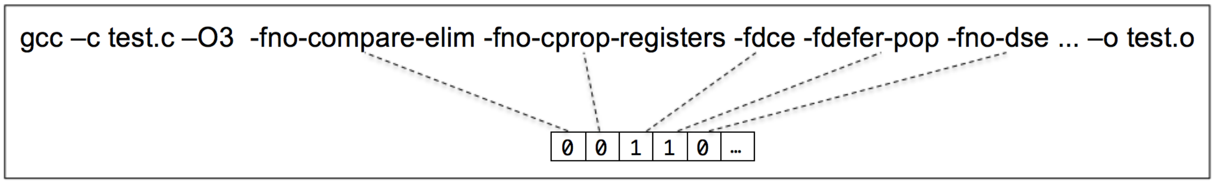

gcc –c test.c –O3 -fno-compare-elim -fno-cprop-registers -fdce -fdefer-pop -fno-dse ... –o test.o
0
0
1
1
0
…

Figure 3.2: Solution representation

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. 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 :

$$distance(S1, S2) = |S1 \bigtriangleup S2| \tag{3.1}$$

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:

$$NM(S) = \frac{1}{k}\sum_{i=1}^{k} distance(S, \mu_i) \tag{3.2}$$

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

3.3.2 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 [**?**, **?**]. 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.

3.4 Evaluation

So far, we have presented a sound procedure and automated component-based framework for extracting the non-functional properties of generated code. 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].

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

3.4.1 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?*

To answer these questions, we conduct several experiments using NOTICE to validate our global approach for non-functional testing of compilers using system containers.

3.4.2 Experimental Setup

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 [?]. 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 [?, ?, ?] 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. Authors 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. Selected Csmith programs are described in more details at [?].

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 Search (RS), NS, and GA [?]) and two multi-objective optimizations (NS and NSGA-II [?]). 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 2. 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 3. 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 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. For each parameter, a set of values was tested. The parameter values chosen are the mostly used in the literature [?]. The value that yielded the highest performance score was chosen.

Table 3.1: 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	Nb generations	100
Individual length	76	Elitism	10
Scaling archive prob.	0.05	Solutions added to archive	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.

Setting up Infrastructure

To answer the previous research questions, we configure NOTICE to run different experiments. Figure 4 shows a big picture of the testing and monitoring infrastructure considered in these experiments. 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 generate a new optimized code and deploy the output binary within a new instance of our preconfigured Docker image (step 2). While executing the optimized code inside the container, we collect runtime performance data (step 4) and record it in a new time-series database using our InfluxDB back-end container (step 5). Next, NOTICE accesses remotely to stored data in InfluxDB using REST API calls and assigns new performance values to the current solution (step 6). The choice of performance metrics depends on experiment objectives (Memory improvement, speedup, etc.).

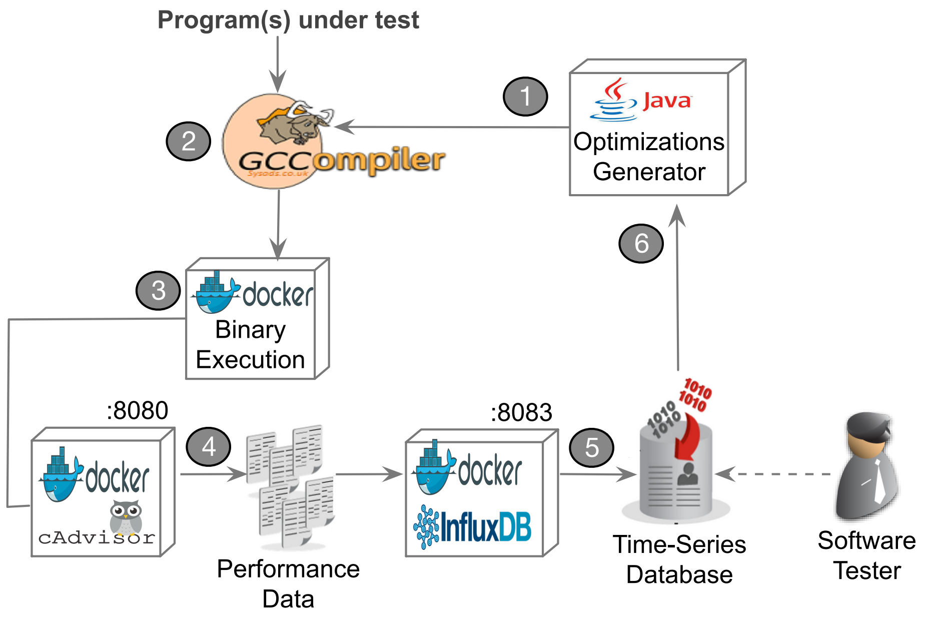


Figure 3.3: NOTICE experimental infrastructure

To obtain comparable and reproducible results, we use the same hardware across all experiments: 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.

3.4.3 Experimental Methodology and Results

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

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.

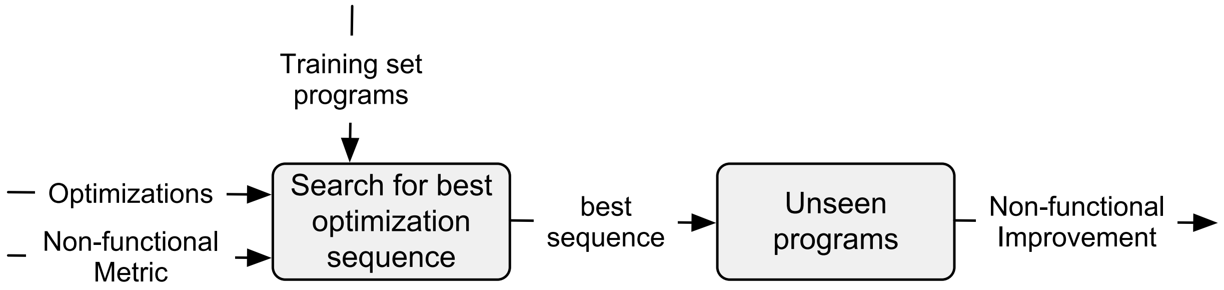


Figure 3.4: Evaluation strategy to answer RQ1 and RQ2

As it is shown on the left side of Figure 5, 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.

Table 3.2: 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

Results Table 3 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 outperform 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.

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 3). 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). 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.

Results Figure 6 shows the distribution of memory, CPU and speedup improvement across 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

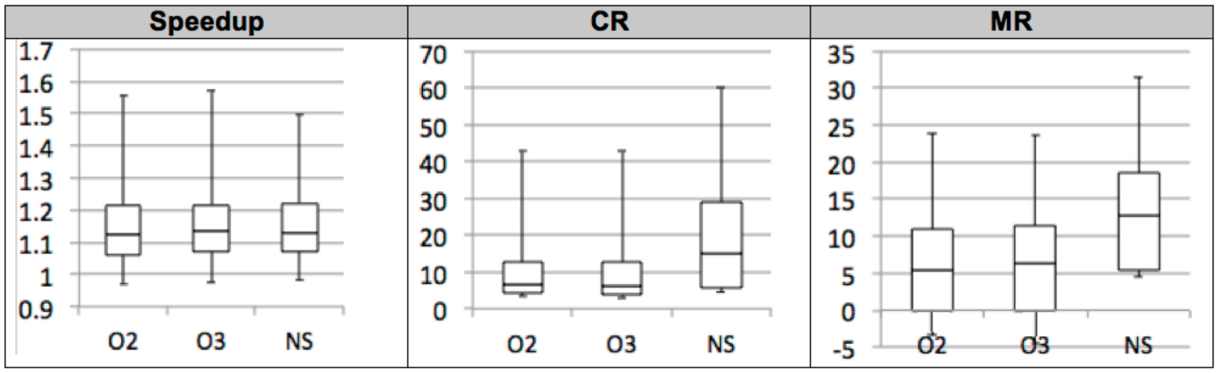


Figure 3.5: Boxplots of the obtained performance results across 100 unseen Csmith programs, for each non-functional property: Speedup (S), memory (MR) and CPU (CR) and for each optimization strategy: O2, O3 and NS

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.

RQ3. Impact of optimizations on resource usage

Method In this experiment, we use NOTICE to provide an understanding of optimizations behavior, 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 3) and we study the impact of speedup improvement on memory and CPU consumption. We also compare resource usage data to standard GCC levels as they were presented in Table 3. Improvements are always calculated over the non-optimized version. 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 both resource usage and performance properties.

Results Figure 7 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.

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

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 [**?**]. 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.

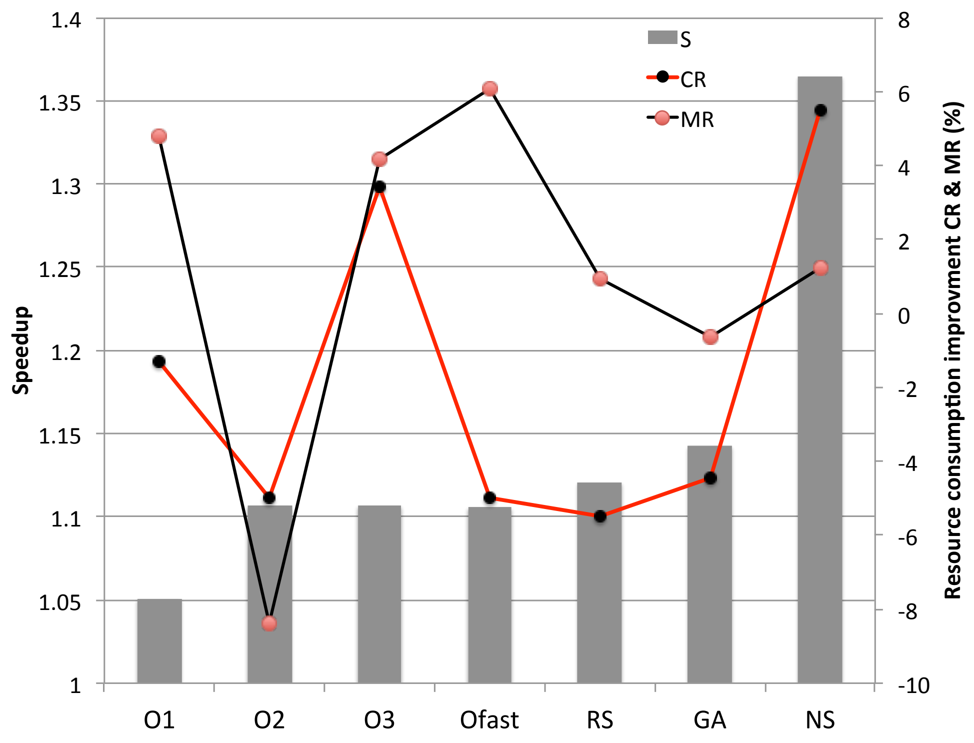


Figure 3.6: Impact of speedup improvement on memory and CPU consumption for each optimization strategy

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.

Results Figure 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 on 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.

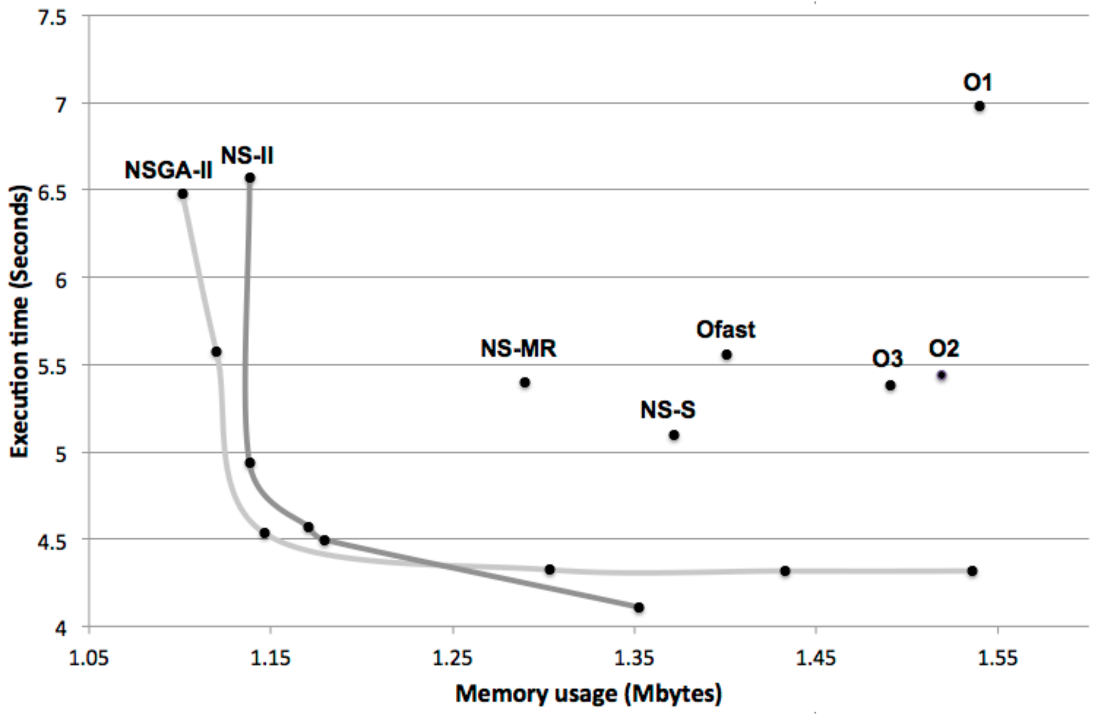


Figure 3.7: Comparison results of obtained Pareto fronts using NSGA-II and NS-II

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 for 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.

3.4.4 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 also 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 would use NOTICE to test 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. As an alternative, it would be great to test model-based code generators. In the same fashion as Csmith, code generators apply to same rules to generate new software programs. Thus, we can use NOTICE to define general-purpose optimizations from a set of generated code artifacts. 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.

3.4.5 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 [**?**, **?**]. 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 [**?**, **?**] 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.

3.5 Conclusion and Future Work

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 paper, 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 tool for automatic extraction of non-functional properties of optimized code, called NOTICE. NOTICE applies different heuristics (including Novelty Search) and performs non-functional testing of compilers 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. 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.

As a future work, we plan to explore more trade-offs among resource usage metrics e. g., the correlation between CPU consumption and platform 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 model-based code generators since different optimizations can be performed to generate code from models [**?**]. Thus, we aim to use the same approach to find non-functional issues during the code generation process.

Chapter 4

Automatic non-functional testing of code generators families

The intensive use of generative programming techniques provides an elegant engineering solution to deal with the heterogeneity of platforms or technological stacks. The use of Domain Specifics Language, for example, leads to the creation of numerous code generators that automatically translate high-level system specifications into multi-target executable code. Producing correct and efficient code generator is complex and error-prone. Although software designers provide generally high-level test suites to verify the functional outcome of generated code, it remains challenging and tedious to verify the behavior of produced code in terms of non-functional properties. This paper describes a practical approach based on a runtime monitoring infrastructure to automatically check potential inefficient code generators. This infrastructure, based on system containers as execution platforms, allows code-generator developers to evaluate the generated code performance. 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 that target different platforms. Experimental results show that our approach is able to detect some performance inconsistencies among Haxe generated code that reveal real issues in Haxe code generators.

4.1 Introduction

The intensive use of generative programming techniques has become a common practice for software development to tame the runtime platform heterogeneity that exists in several domains such as mobile or Internet of Things development. Generative programming techniques reduce the development and maintenance effort by developing at a higher-level of abstraction through the use of domain-specific languages [?] (DSLs) for example. A code generator can be used to transform source code programs/models represented in a graphical/textual language to general purpose programming languages such as C, Java, C++, PHP, JavaScript etc. In turn, generated code is transformed into machine code (binaries) using a set of specific compilers.

However, code generators are known to be difficult to understand since they involve a set of complex and heterogeneous technologies which make the activities of design, implementation, and testing very hard and time-consuming [?, ?]. Code generators have to respect different requirements which preserve software reliability and quality. In fact, 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.).

In order to check the correctness of the code generation process, developers often define (at design time) a set of test cases that verify the functional behavior of the generated code. After code generation, test suites are executed within each target platform. This may lead to either a correct behavior (i. e., expected output) or a incorrect one (i. e., failures, errors). Nevertheless, for performance testing of code generators, developers need to deploy and execute software artifacts on different execution platforms. Then, they have to collect and compare information about the performance and efficiency of the generated code. Finally, they report issues related to the code generation process such as incorrect typing, memory management leaks, etc. Currently there is a lack of automatic solutions to check the performance issues such as the low efficiency (huge memory/CPU consumption) of the generated code. In fact, developers often use manually several platform-specific profilers, trackers, and monitoring tools [?, ?] in order to find some inconsistencies or bugs during code execution. Ensuring the code quality of the generated code can refer to several non-functional properties such as code size, resource or energy consumption, execution time, among others [?]. Testing the non-functional properties of code generators remains a challenging and time-consuming task because developers have to analyze and verify the generated code for different target platforms using platform-specific tools.

This paper is based on the intuition that a code generator is often a member of a code generators family[1] Thus, we can automatically compare the performance between different versions of generated code (coming from the same source program). Based on this com-

[1]A code generator family is a set of code generators that takes the same input language and targets different target platforms.

parison, we can automatically detect singular resource consumptions that could reveal a code generator bug. As a result, we propose an approach to automatically compare and detect inconsistencies between code generators. This approach provides a runtime monitoring infrastructure based on system containers, as execution platforms, to compare the generated code performance. We evaluate our approach by analyzing the non-functional properties of Haxe code generators. Haxe is a popular high-level programming language[2] that involves a family of cross-platform code generators able to generate code to different target platforms. Our experimental results show that our approach is able to detect performance inconsistencies that reveal real issues in Haxe code generators.

The contributions of this paper are the following:

- We propose a fully automated micro-service infrastructure to ensure the deployment and monitoring of generated code. This paper focuses on the relationship between runtime execution of generated code and resource consumption profiles (memory usage).
- We also report the results of an empirical study by evaluating the non-functional properties of the Haxe code generators. The obtained results provide evidence to support the claim that our proposed infrastructure is effective.

The paper is organized as follows. Section 2 describes the motivations behind this work by discussing three examples of code generator families. Section 3 presents an overview of our approach to automatically perform non-functional tests of code generator families. In particular, we present our infrastructure for non-functional testing of code generators using micro-services. The evaluation and results of our experiments are discussed in Section 4. Finally, related work, concluding remarks, and future work are provided in Sections 5 and 6.

4.2 Motivation

4.2.1 Code Generator Families

We can cite three approaches that intensively develop and use code generators:

[2] 1442 GitHub stars

a. Haxe. Haxe[3] [?] 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 and object-oriented programming paradigms. It has a common type hierarchy, making certain API available on every targeted platform. Moreover, Haxe comes with a set of code generators 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. Compilers ensure the correctness of user code in terms of syntax and type safety. Haxe comes also with a set of standard libraries that can be used on all supported targets and platform-specific libraries for each of them. One of the main usage of Haxe is to develop Cross-Platform Games or Cross-Platform libraries that can run on mobile, on the Web or on a Desktop. This project is popular (more than 1440 stars on GitHub).

b. ThingML. ThingML[4] is a modeling language for embedded and distributed systems [?]. 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 sensor 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 toolset provides a code generators 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 system and their constraints. This project is still confidential but it is a good candidate to represent the modeling community practices.

c. TypeScript. TypeScript[5]is a typed superset of JavaScript that compiles to plain JavaScript [?]. 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')[6]. This project is popular (more than 12619 stars on GitHub).

[3]`http://haxe.org/`

[4]`http://thingml.org/`

[5]`https://www.typescriptlang.org/`

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

4.2.2 Functional Correctness of a Code Generator Family

A reliable and acceptable way to increase the confidence in the correctness of code generators is to validate and check the functionality of generated code, which is a common practice for compiler validation and testing. Therefore, developers try to check the syntactic and semantic correctness of the generated code by means of different techniques such as static analysis, test suites, etc., and ensure that the code is behaving correctly. In model-based testing for example [**?**, **?**], testing code generators focuses on testing the generated code against its design. Thus, the model and the generated code are executed in parallel, by means of simulations, with the same set of test suites. Afterwards, the two outputs are compared with respect to certain acceptance criteria. Test cases, in this case, can be designed to maximize the code or model coverage [**?**]. 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.

4.2.3 Performance Evaluation of a Code Generator Family

Code generators have to respect different requirements to preserve software reliability and quality [**?**]. A non-efficient code generator might generate defective software artifacts (code smells) that violates common software engineering practices. Thus, poor-quality code can affect system reliability and performance (e.g., high resource usage, low execution speed, etc.). Thus, another important aspect of code generator's testing is to test the non-functional properties of produced code. Proving that the generated code is functionally correct is not enough to claim the effectiveness of the code generator under test. In looking at the three motivating examples, we can observe that ThingML and TypeScript do not provide any specific test to check the consistency of code generators regarding the memory or CPU usage. Haxe provides two test cases [7] 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 mainly bench the network speed for each target platform.

However, we believe that we can detect inefficient code generator, based on existing benchmarks among technical environments comparing the behavior of generated code of a large set of programs [**?**]. The kind of errors we are trying to track can be resumed as following:

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

- the lack of use of a **specific function that exists in the standard library** of the targeted language that can speed or reduce the memory consumption of the resulting program.
- the lack of use of **a specific type that exists in the standard library** of the targeted language that can speed or reduce the memory consumption of the resulting program.
- the lack of use of **a specific language feature in a targeted language** that can speed or reduce the memory consumption of the resulting program.

The main difficulties is the fact that, for testing the non functional properties of code generators such as performance, we cannot just observe the execution of the code generator but we have to observe and compare the execution of the generated program. Even if there is no exact oracle to detect inconsistencies, we could benefit from the family of code generators to compare the behavior of several programs generated from the same source and run on top of different technical stacks.

Next section discusses the common process used by developers to automatically test the performance of generated code. We illustrate also how we can benefit from the code generators families to identify suspect singular behaviors.

4.3 Approach Overview

4.3.1 Non-Functional Testing of a Code Generator Family: a Common Process

Figure 1 summarizes the classical steps to ensure the code generation and non-functional testing of produced code from design time to run time. We distinguish 4 major steps: the software design using high-level system specifications, code generation by means of code generators, code execution, and non-functional testing of generated code.

In the first step, software developers have to define, at design time, software's 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 generate automatically code that targets different languages and platforms. We depict, in Figure 1, three code generators capable to generate code in three software programming languages (JAVA, C# and C++). In this step, code generators transform the previously designed model to produce, as a consequence, software artifacts for the target platform.

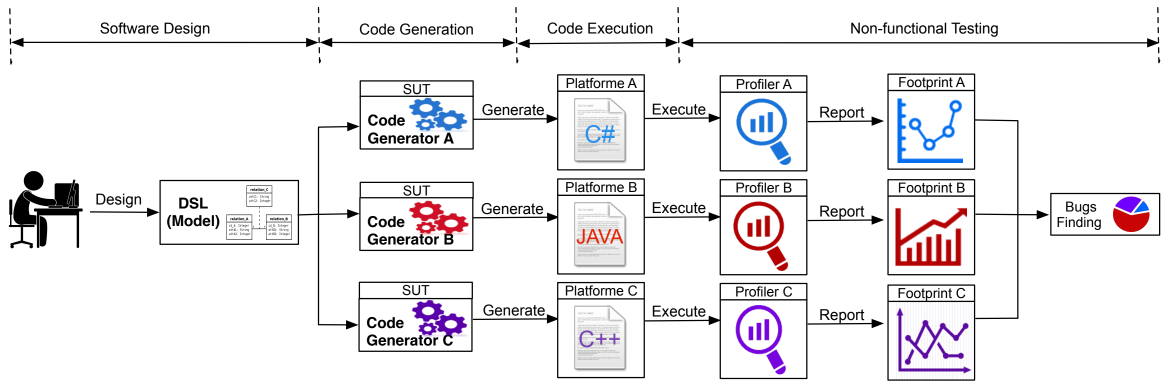


Figure 4.1: An overall overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to run time: the classical way

In the next step, 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.). Thus, several code compilers are needed to transform source code to machine code (binaries) in order to get executed.

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 [?, ?]. Ensuring the code quality of generated code can refer to several non-functional properties such as code size, resource or energy consumption, execution time, among others [?]. Finding inconsistencies within code generators involves analyzing and inspecting the code and that, for each execution platform. For example, one of the methods to handle that is to analyze the memory footprint of software execution and find memory leaks. Developers can then inspect the generated code and find some parts of the code-base that have triggered this issue. Therefore, software testers generally use to report statistics about the performance of generated code in order to fix, refactor, and optimize the code generation process. Our approach aims to automate the three last steps: generate code for a set of platforms, execute code on top of different platforms, monitor and compare the execution.

4.3.2 An Infrastructure for Non-functional Testing Using System Containers

To assess the performance/non-functional properties of generated code many system configurations (i.e., execution environments) must be considered. Running 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 to compiler user facilities to ensure the deployment and monitoring of generated code in different environment settings.

Consequently, our 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 performance metrics (CPU, Memory, etc.)

To get these four main steps, we rely on system containers [**?**]. Thus, instead of configuring all code generators under test (GUTs) within the same host machine, we wrap each GUT within a system container. Afterwards, a new instance of the container is created to enable the execution of generated code in an isolated and configured environment. Meanwhile, we start our runtime testing components. A monitoring component collects usage statistics of all running containers and save them at runtime in a time series database component. Thus, we can compare later informations about the resource usage of generated programs and detect inconsistencies within code generators.

4.4 Evaluation

So far, we have presented a procedure and automated component-based framework for extracting the performance properties of generated code. In this section, we evaluate the implementation of our approach by explaining 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[8].

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

4.4.1 Experimental Setup

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 comes with a set of compilers that translate manually-written code (in Haxe language) to different target languages and platforms.

The process of code transformation and generation can be described as following: Haxe compilers analyzes 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 target platform/language.

Haxe offers the option of choosing which platform to target for each program using a command-line tool. Moreover, some optimizations and debugging information can be enabled through CLI but in our experiments, we did not turned on any further options.

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 proposed benchmark is composed of a collection of cross-platform libraries that can be compiled to different targets. In these experiments, we consider five Haxe code generators to test: Java, JS, C++, CS, and PHP code generators. To select cross-platform libraries, we explore github and we use the Haxe library repository[9]. We select seven libraries that have the best code coverage score.

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 once generated code is executed within the target platform. 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.

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

We use then, these test suites then, to generate a load and stress the target library. This can be useful to study the impact of this load on the resource usage of the system. For example, if one test suite consumes a lot of resources for a specific target, then this could be explained by the fact that the code generator has produced code that is very greedy in terms of resources.

Thus, we run each test suite 1000 times to get comparable values in terms of resource usage. Table 2 describes the Haxe libraries that we have selected in this benchmark to evaluate our approach.

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	3	Library for lightweight promises and futures
Culture	4	Localization library for Haxe
Math	3	Generation of random values

Table 4.1: Description of selected benchmark libraries

Evaluation Metrics Used

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

-*Memory usage*: It corresponds to the maximum memory consumption of the running container under test. Memory usage is measured in Mbytes.

-*Execution time*: Program execution time is measured in seconds.

We recall that our tool 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).

Setting up Infrastructure

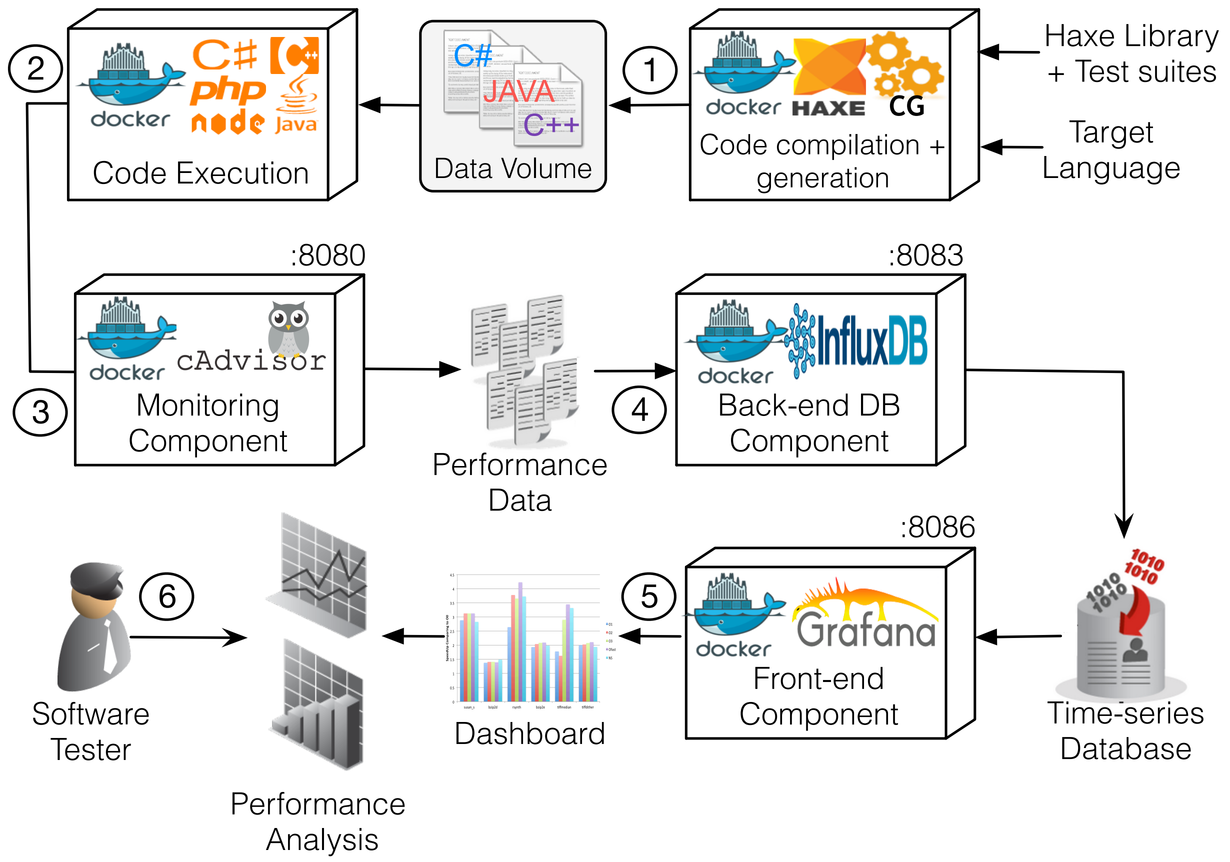


Figure 4.2: Infrastructure settings for running experiments

To assess our approach, we configure our previously proposed container-based infrastructure in order to run experiments on the Haxe case study. Figure 3 shows a big picture of the testing and monitoring infrastructure considered in these experiments.

First, we create a new Docker image in where we install the Haxe code generators and compilers (through the configuration file "Dockerfile"). Then a new instance of that image is created. It takes as an input the Haxe library we would to test and the list of test suites (step 1). It produces as an output the source code and binaries that have to be executed. These files are saved in a shared repository. In Docker environment, this repository is called "data volume". A data volume is a specially-designated directory within containers that shares data with the host machine. So, when we execute the generated test suites, we provide a shared volume with the host machine so that, binaries can be executed in the execution container (Step 2). In fact, for the code execution we created, as well, a new Docker image in where we install all execution tools and environments such as php interpreter, NodeJS, etc.

In the meantime, while running test suites inside the container, we collect runtime resource usage data using cAdvisor (step 3). The cAdvisor Docker image does not need any configuration on the host machine. We have just to run it on our host machine. It will then have access to resource usage and performance characteristics of all running containers. This image uses the Cgroups mechanism described previously to collect, aggregate, process, and export ephemeral real-time information about running containers. Then, it reports all statistics via web UI to view live resource consumption of each container. cAdvisor has been widely used in different projects such as Heapster[10] and Google Cloud Platform[11]. In this experiment, we choose to gather information about the memory usage of running container. Afterwards, we record these data into a new time-series database using our InfluxDB back-end container (step 4).

Next, we run Grafana and we link it to InfluxDB. Grafana can request data from the database. We recall that InfluxDB also provides a web UI to query the database and show graphs (step 5). But, Grafana let us display live results over time in much pretty looking graphs. Same as InfluxDB, we use SQL queries to extract non-functional metrics from the database for visualization and analysis (step 6). In our experiment, we are gathering the maximum memory usage values without presenting the graphs of resource usage profiles.

To obtain comparable and reproducible results, we use the same hardware across all experiments: a farm of 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. We reserve one core and 4 GB of memory for each running container.

4.4.2 Experimental Results

Evaluation using the standard deviation

We now conduct experiments based on the Haxe benchmark. We run each test suite 1K times and we report the execution time and memory usage across the different target languages: Java, JS, C++, CS, and PHP. The goal of running these experiments is to observe and compare the behavior of generated code regarding the testing load. We recall, as

[10]https://github.com/kubernetes/heapster
[11]https://cloud.google.com/

mentioned in the motivation, that we are not using any oracle function to detect inconsistencies. However, we rely on the comparison results across different targets to define code generator inconsistencies. Thus, we use, as a quality metric, the standard deviation to quantify the amount of variation among execution traces (i.e., memory usage or execution time) and that for the five target languages. We recall that the formula of standard deviation is the square root of the variance. Thus, we are calculating this variance as the squared differences from the mean. Our data values in our experiment represent the obtained values in five languages. So, for each test suite we are taking the mean of these five values in order to calculate the variance. A low standard deviation of a test suite execution, indicates that the data points (execution time or memory usage data) tend to be close to the mean which we consider as an acceptable behavior. On the other hand, a high standard deviation indicates that one or more data points are spread out over a wider range of values which can be more likely interpreted as a code generator inconsistency.

In Table 3, we report the comparison results of running the benchmark in terms of execution speed. At the first glance, we can clearly see that all standard deviations are more mostly close to 0 - 8 interval. It is completely normal to get such small deviations, because we are comparing the execution time of test suites that are written in heterogeneous languages and executed using different technologies (e.g., interpreters for PHP, JVM for JAVA, etc.). So, it is expected to get a small deviation between the execution times after running the test suite in different languages. However, we remark in the same table, that there are some variation points where the deviation is relatively high. We count 8 test suites where the deviation is higher than 60 (highlighted in gray). We choose this value (i.e., standard deviation = 60) as a threshold to designate the points where the variation is extremely high. Thus, we consider values higher than 60 as a potential possibility that a non-functional bug could occur. These variations can be explained by the fact that the execution speed 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 behavior of code for one or more target language. We provide later better explanation in order to detect the faulty code generators.

Similarly, Table 4 resumes the comparison results of test suites execution regarding 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 for each language. Nevertheless, we can recognize some points where the variation is extremely high. Thus, we choose a threshold value equal to 400 and we highlighted, in gray, the points that exceed this threshold. Thus, we detect 6 test suites where the variation is extremely 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 are so greedy in terms of resources. This may be not the case when the variation is lower than 10 for example. We assume then that the faulty code generator, in this case, represents a threat for software quality since it can generate a code that is very resource consuming.

The inconsistencies we are trying to find here are more related to the incorrect memory utilization patterns produced by the faulty code generator. Such inconsistencies may come from an inadequate type usage, high resource instantiation, etc.

Analysis

Now that we have observed the non-functional behavior of test suites execution in different languages, we can analyze the extreme points we have detected in previous tables to observe more in deep the source of such deviation. For that reason, we present in Table 5 and 6 the raw data values of these extreme test suites in terms of execution time and memory usage.

Table 5 is showing the execution time of each test suite in a specific target language. We provide also factors of execution times among test suites running in different languages by taking as a baseline the JS version. We can clearly see that the PHP code has can have a singular behavior regarding the performance with a factor ranging from x40.9 for test suite 3 in benchamrk Format (Format_TS3) to x481.7 for Math_TS1. We remark also that running Core_TS4 takes 61777 seconds (almost 17 hours) compared to a 416 seconds (around 6 minutes) in JAVA which is a very large gap. The highest factor detected for other languages ranges from x0.3 to x5.4 which is not negligible but it represents a small deviation compared to PHP version. 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 more likely to be a code generator inconsistency that led to such performance regression.

Meanwhile, we gathered information about the points that led to the highest standard deviation in terms of memory usage. Table 6 shows these results. Again, we can identify singular behavior regarding the performance. For Color_TS6, C# version consumes the highest memory (x2.5 more than JS). For other test suites versions, the factor varies from x0.8 to x3.7.

Focusing particularly on the singular performance of PHP code in core TS4, 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 many advanced functions such as *array_fill* and specialized abstract types such as *"SplFixedArray"*[12] to overcome this limitation. By changing just these two parts in the code generator, we improve the PHP code speed with a factor x5.

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. In contrast, selecting carefully the adequate types and functions to generate code by code generators can lead to performance improvement. We can observe the same kind of error in the C++ program during one test suite execution (Color_TS6) which consumes too much memory. The types used in the code generator are not the best ones.

4.4.3 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 Haxe and a set of test suite selected from Github and from the Haxe community. We have no guarantee that these libraries covers all the Haxe language features neither than all the Haxe standard libraries. Consequently, we cannot guarantee that the approach is able to find all the code generators issues. In particular, the threshold to detect singular 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 and try to understand the best heuristic to detect the code generator issues regarding performance.

Internal validity is concerned with the use of a container-based approach. Even if it exists emulator such as Qemu that allows to reflect the behavior of heterogeneous hardware, the chosen infrastructure has not been evaluated to test generated code that target heterogeneous hardware machine. In such a case, the provided compiler for a dedicated hardware can provide specific optimizations that lead to a large number of false positives.

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

Benchmark	TestSuite	Std_dev	TestSuite	Std_dev	TestSuite	Std_dev
Color	**TS1**	0.55	**TS8**	0.24	**TS15**	0.73
	TS2	0.29	**TS9**	0.22	**TS16**	0.12
	TS3	0.34	**TS10**	0.10	**TS17**	0.31
	TS4	2.51	**TS11**	0.17	**TS18**	0.34
	TS5	1.53	**TS12**	0.28	**TS19**	120.61
	TS6	43.50	**TS13**	0.33		
	TS7	0.50	**TS14**	1.88		
Core	**TS1**	0.35	**TS18**	0.16	**TS35**	1.30
	TS2	0.07	**TS19**	0.60	**TS36**	1.13
	TS3	0.30	**TS20**	5.79	**TS37**	2.02
	TS4	27299.89	**TS21**	0.47	**TS38**	0.26
	TS5	6.12	**TS22**	2.74	**TS39**	0.16
	TS6	21.90	**TS23**	2.14	**TS40**	8.12
	TS7	0.41	**TS24**	3.79	**TS41**	5.45
	TS8	0.28	**TS25**	0.19	**TS42**	0.11
	TS9	0.78	**TS26**	0.13	**TS43**	1.41
	TS10	1.82	**TS27**	5.59	**TS44**	1.56
	TS11	180.68	**TS28**	1.71	**TS45**	0.11
	TS12	185.02	**TS29**	0.26	**TS46**	1.04
	TS13	128.78	**TS30**	0.44	**TS47**	0.23
	TS14	0.71	**TS31**	1.71	**TS48**	1.34
	TS15	0.12	**TS32**	2.42	**TS49**	1.86
	TS16	0.65	**TS33**	8.29	**TS50**	1.28
	TS17	0.26	**TS34**	5.25	**TS51**	3.53
Hxmath	**TS1**	31.65	**TS3**	30.34	**TS5**	0.40
	TS2	4.27	**TS4**	0.25	**TS6**	0.87
Format	**TS1**	0.28	**TS3**	95.36	**TS4**	1.49
	TS2	64.94				
Promise	**TS1**	0.29	**TS2**	13.21	**TS3**	1.21
Culture	**TS1**	0.13	**TS3**	0.13	**TS4**	1.40
	TS2	0.10				
Math	**TS1**	642.85	**TS2**	28.32	**TS3**	24.40

Table 4.2: The comparison results of running each test suite across five target languages: the metric used is the standard deviation between execution times

Benchmark	TestSuite	Std_dev	TestSuite	Std_dev	TestSuite	Std_dev
Color	TS1	10.19	TS8	1.23	TS15	14.44
	TS2	1.17	TS9	1.95	TS16	1.13
	TS3	0.89	TS10	1.27	TS17	0.72
	TS4	30.34	TS11	0.57	TS18	0.97
	TS5	31.79	TS12	1.11	TS19	777.32
	TS6	593.05	TS13	0.46		
	TS7	12.14	TS14	45.90		
Core	TS1	1.40	TS18	1.00	TS35	14.13
	TS2	1.17	TS19	20.37	TS36	32.41
	TS3	0.60	TS20	128.23	TS37	22.72
	TS4	403.15	TS21	24.38	TS38	2.19
	TS5	41.95	TS22	76.24	TS39	0.26
	TS6	203.55	TS23	18.82	TS40	126.29
	TS7	19.69	TS24	72.01	TS41	31.01
	TS8	0.78	TS25	0.21	TS42	0.93
	TS9	30.41	TS26	2.30	TS43	50.36
	TS10	57.19	TS27	101.53	TS44	12.56
	TS11	68.92	TS28	43.67	TS45	0.91
	TS12	74.19	TS29	0.90	TS46	27.28
	TS13	263.99	TS30	4.02	TS47	1.10
	TS14	19.89	TS31	52.35	TS48	15.40
	TS15	0.30	TS32	134.75	TS49	37.01
	TS16	28.29	TS33	82.66	TS50	23.29
	TS17	1.16	TS34	89.57	TS51	1.28
Hxmath	TS1	444.18	TS3	425.65	TS5	17.69
	TS2	154.80	TS4	0.96	TS6	46.13
Format	TS1	0.74	TS3	255.36	TS4	8.40
	TS2	106.87				
Promise	TS1	0.30	TS2	58.76	TS3	20.04
Culture	TS1	1.28	TS3	0.58	TS4	15.69
	TS2	4.51				
Math	TS1	1041.53	TS2	234.93	TS3	281.12

Table 4.3: The comparison results of running each test suite across five target languages: the metric used is the standard deviation between memory consumptions

	JS		JAVA		C++		CS		PHP	
	Time	**Factor**	**Time**	**Factor**	**Time**	**Factor**	**Time**	**Factor**	**Time**	**Factor**
Color_TS19	4.52	x 1.0	8.61	x 1.9	10.73	x 2.4	14.99	x 3.3	279.27	x61.8
Core_TS4	665.78	x 1.0	416.85	x 0.6	699.11	x 1.1	1161.29	x 1.7	61777.21	x92.8
Core_TS11	4.27	x 1.0	1.80	x 0.4	1.57	x 0.4	5.71	x 1.3	407.33	x95.4
Core_TS12	4.71	x 1.0	2.06	x 0.4	1.60	x 0.3	5.36	x 1.1	417.14	x88.6
Core_TS13	6.26	x 1.0	5.91	x 0.9	11.04	x 1.8	14.14	x 2.3	297.21	x47.5
Format_TS2	2.31	x 1.0	2.10	x 0.9	1.81	x 0.8	6.08	x 2.6	148.24	x64.1
Format_TS3	5.40	x 1.0	5.03	x 0.9	7.67	x 1.4	12.38	x 2.3	220.76	x40.9
Math_TS1	3.01	x 1.0	12.51	x 4.2	16.30	x 5.4	14.14	x 4.7	1448.90	x481.7

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

	JS		JAVA		C++		CS		PHP	
	Memory	**Factor**	**Memory**	**Factor**	**Memory**	**Factor**	**Memory**	**Factor**	**Memory**	**Factor**
Color_TS6	900.70	x 1.0	1362.55	x 1.5	2275.49	x 2.5	1283.31	x 1.4	758.79	x 0.8
Color_TS19	253.01	x 1.0	819.92	x 3.2	923.99	x 3.7	327.61	x 1.3	2189.86	x 8.7
Core_TS4	303.09	x 1.0	768.22	x 2.5	618.42	x 2	235.75	x 0.8	1237.15	x 4.1
Hxmath_TS1	104.00	x 1.0	335.50	x 3.2	296.43	x 2.9	156.41	x 1.5	1192.98	x11.5
Hxmath_TS3	111.68	x 1.0	389.73	x 3.5	273.12	x 2.4	136.49	x 1.2	1146.05	x10.3
Math_TS1	493.66	x 1.0	831.44	x 1.7	1492.97	x 3	806.33	x 1.6	3088.15	x 6.3

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

Chapter 5

An infrastructure for resource monitoring based on system containers

5.1 Introduction

The general overview of the technical implementation is shown in Figure 2. In the following subsections, we describe the deployment and testing architecture of generated code using system containers.

5.2 System Containers as Execution Platforms

Before starting to monitor and test applications, we have to deploy generated code on different components to ease containers provisioning and profiling. We aim to use Docker Linux containers to monitor the execution of different generated artifacts in terms of resource usage [?]. Docker[1] is an engine that automates the deployment of any application as a lightweight, portable, and self-sufficient container that runs virtually on a host machine. Using Docker, we can define pre-configured applications and servers to host as virtual images. We can also define the way the service should be deployed in the host machine using configuration files called Docker files. In fact, instead of configuring all code generators

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

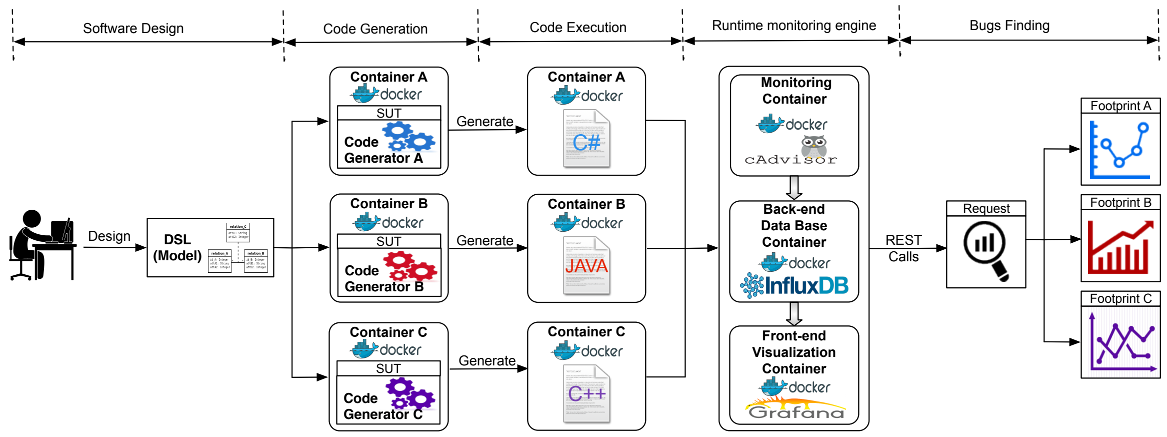


Figure 5.1: A technical overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to runtime.

under test (GUTs) within the same host machine (as shown in Figure 1), our tool wrap 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 produce code within multiple instances of preconfigured Docker images (see code generation step in Figure 2). We use the public Docker registry[2] for saving, and managing all our Docker images. We can then instantiate different containers from these Docker images.

Next, each generated code is executed individually inside an isolated Linux container (see code execution step in Figure 2). 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. 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 provide a bare-metal installation of docker and docker swarm. On a single machine, containers/softwares are running sequentially and we pin p cores and n Gbytes of memory for each container[3]. 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.

In short, the main advantages of this approach are:

- The use of containers induces less performance overhead and resource isolation com-

[2] https://hub.docker.com/
[3] p and n can be cofigured

pared to using a full stack virtualization solution [?]. Indeed, instrumentation and monitoring tools for memory profiling like Valgrind [?] can induce too much overhead.

- Thanks to the use of Dockerfiles, the proposed framework can be configured by software testers in order to define the code generators under test (e. g., code generator version, dependencies, etc.), the host IP and OS, the DSL design, the optimization options, etc. Thus, we can use the same configured Docker image to execute different instances of generated code. 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)

5.3 Runtime Testing Components

In order to test our running applications within Docker containers, we aim to use a set of Docker components to ease the extraction of resource usage information (see runtime monitoring engine in Figure 2).

5.3.1 Monitoring Component

This container provides an understanding of the resource usage and performance characteristics of our running containers. Generally, Docker containers rely 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 runtime performance metrics stored in Cgroups files. For example, we access live resource consumption of each container available at the Cgroups file system via stats found in *"/sys/fs/c-*

group/cpu/docker/(longid)/" (for CPU consumption) and *"/sys/fs/cgroup/memory/docker/(longid)/"* (for stats related to memory consumption). This component will automate 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 generated code that is running within a container). We note that resource usage information is collected in raw data. This process may induce a little overhead because it does very fine-grained accounting of resource usage on running container. Fortunately, this may not affect the gathered performance values since we run only one version of generated code within each container. To ease the monitoring process, we integrate cAdvisor, a Container Advisor[4]. cAdvisor monitors service containers at runtime.

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 code-generator testers to run queries and define non-functional metrics from historical data. Thereby, to make gathered data truly valuable for resource usage monitoring, we link our monitoring component to a back-end database component.

5.3.2 Back-end Database Component

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 the execution of generated code, 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[5], 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_v*1" since its creation:

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

[4] https://github.com/google/cadvisor

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

To give an idea about the data gathered by the monitoring component and stored in the time-series database, we describe in Table 1 these collected metrics:

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

Table 5.1: Resource usage metrics recorded in InfluxDB

Apart from that, our framework provides also information about the size of generated binaries and the compilation time needed to produce code. For instance, resource usage statistics are collected and stored using these two components. It is relevant to show resource usage profiles of running programs overtime. To do so, we present a front-end visualization component for performance profiling.

5.3.3 Front-end Visualization Component

Once we gather and store resource usage data, the next step is visualizing them. That is the role of the visualization component. 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 profiles of resource consumption of running components through web UI. Thereby, we can compare visually the profiles of resource consumption among containers. Moreover, we 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 (see bugs finding step in Figure 2). As a visualization component, we use Grafana[6], a time-series visualization tool available for Docker.

[6] https://github.com/grafana/grafana

Part III

Conclusion and perspectives

Chapter 6

Conclusion and perspectives

References

[1] 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.

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

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