Contents

Contents 1

1 Introduction 1

1.1 Context . . . 1

1.2 Motivation . . . 2

1.3 Scope of the thesis . . . 4

1.4 Challenges . . . 4

1.5 Contributions . . . 6

1.6 Overview of this thesis . . . 7

1.7 Publications . . . 8

I Background and State of the Art 10

2 Background 11

2.1 Diversity in software engineering . . . 11

2.1.1 Software diversity . . . 12

2.1.2 Hardware heterogeneity . . . 14

2.1.3 Matching software diversity to heterogeneous hardware: the marriage 16

2.2 From classical software development to generative programming . . . 18

2.3 An overview of the software development tool chain . . . 19

2.3.1 Automatic code generation 20
2.3.2 Stakeholders and their roles for testing generators 22
2.4 Testing code generators 23
2.4.1 Testing workflow 23
2.4.2 Types of code generators 24
2.4.3 Why testing code generators is complex? 26
2.5 Compilers auto-tuning 27
2.5.1 Code optimizations 27
2.5.2 Why compilers auto-tuning is complex? 29
2.6 Summary: Testing challenges 30

3 State of the art 31

3.1 Compilers auto-tuning techniques 32
3.1.1 Iterative compilation 32
3.1.2 Implementation of iterative compilation system 32
3.1.3 Iterative compilation search techniques 34
3.2 Testing code generators 42
3.2.1 Testing functional properties 42
3.2.2 Testing non-functional properties 44
3.3 Container-based testing techniques 48
3.3.1 Deployment and testing using Docker 49
3.3.2 Runtime monitoring using Docker 51
3.4 Summary & limitations of the state-of-th-art 52

II Contributions 53

4 NOTICE: A tool for auto-tuning compilers 56

4.1 Introduction 57

4.2 Motivation 59
4.2.1 Compiler Optimizations 59
4.2.2 Example: GCC Compiler 60
4.3 Evolutionary Exploration of Compiler Optimizations 62
4.3.1 Novelty Search Adaptation 62
4.3.2 Novelty Search For Multi-objective Optimization 66
4.4 Evaluation 67
4.4.1 Research Questions 67
4.4.2 Experimental Setup 67
4.4.3 Experimental Methodology and Results 71
4.4.4 Discussions 81
4.4.5 Threats to Validity 82
4.4.6 Tool Support Overview 83
4.5 Conclusion 85

5 Automatic non-functional testing of code generators families 87

5.1 Introduction 88
5.2 Motivation 90
5.2.1 Code Generator Families 90
5.2.2 Functional Correctness of a Code Generator Family 91
5.2.3 Performance Evaluation of a Code Generator Family 91
5.2.4 Performance Issues of a Code Generator Family 92
5.3 Approach Overview 93
5.3.1 Non-Functional Testing of a Code Generator Family: a Common Process 93
5.3.2 An Infrastructure for Non-functional Testing Using System Containers 94
5.3.3 Technical Implementation 95
5.4 Evaluation 100

5.4.1 Experimental Setup . . . 100
5.4.2 Experimental Results . . . 104
5.4.3 Threats to Validity . . . 106
5.5 Conclusion . . . 107

6 An infrastructure for resource monitoring based on system containers 111
6.1 Introduction . . . 111
6.2 System Containers as Execution Platforms . . . 111
6.3 Runtime Testing Components . . . 113
6.3.1 Monitoring Component . . . 113
6.3.2 Back-end Database Component . . . 114
6.3.3 Front-end Visualization Component . . . 115

III Conclusion and perspectives 117

7 Conclusion and perspectives 118
7.1 Summary of contributions . . . 118
7.2 Perspectives . . . 118

References 119

Chapter 1

Introduction

1.1 Context

Modern software systems rely nowadays on a highly heterogeneous and dynamic interconnection of platforms and devices that provide a wide diversity of capabilities and services. These heterogeneous services may run in different environments ranging from cloud servers with virtually unlimited resources down to resource-constrained devices with only a few KB of RAM. Effectively developing software artifacts for multiple target platforms and hardware technologies is then becoming increasingly important. As a consequence, we observe in the last years [CE00b], that high-level abstract development received more and more attraction to tame with the runtime heterogeneity of platforms and technological stacks that exist in several domains such as mobile or Internet of Things [BCG11]. Therefore, software developers tend to increasingly use generative programming [CE00b] and model-based techniques [FR07] 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 (DSLs) 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 software platforms and hardware.

On the one hand, code generators are needed to transform the high-level system specifications (e.g., textual or graphical modeling language) into conventional source code programs (e.g., General-purpose Languages GPLs such as Java, C++, etc). Automatic code generation can of course improve the quality and consistency of a program as well the productivity of software development.

On the other hand, compilers are also needed to bridge this gap by taking into account different hardware architectures and properties such as register usage, memory organizations, hardware-specific optimizations, etc. So, compilers are often needed to transform the source code, that was manually written or automatically generated, into machine code (i.e., binaries, executables).

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 whose complex interdependencies pose important challenges. Supposing that a software developer want to generate high-quality source code or executables, he may do it himself by creating his own code generator or compiler, introducing some optimizations, or he could benefit from the work of others by using an off-the-shelf compiler/code generator.

1.2 Motivation

Nowadays, compilers become very user-friendly and highly configurable [FMT+08]. Thus, the generated executables can be easily customized to satisfy the user requirements. Indeed, compilers such as GNU compilers and LLVM provide a large selection of configuration options to control the compiler behavior. For example, different categories of options my be used to help developers to: debug their applications, optimize and tune application performance, select language levels and extensions for compatibility, select the target hardware architecture, and perform many other common tasks that configure the way executables are generated. The huge number of compiler configurations, versions, optimizations and debugging utilities make the task of choosing the best configuration set very difficult and time-consuming. As an example, GCC version 4.8.4 provides a wide range of command-line options that can be enabled or disabled by users, including more than 150 options for optimization. This results in a huge design space with 2^{150} possible optimization combinations that can be enabled by the user. In addition, constructing one single optimization sequence that improves the performance or resource usage for all programs is impossible since the interactions between optimizations is too complex and difficult to define. As well, the optimization's impact is highly dependent on the hardware and the input source code.

This example shows how painful it is for the compiler users to tune compilers (through optimization flags) in order to satisfy different non-functional properties such as execution time, compilation time, code size, etc.

On the contrary, code generators are less configurable than compilers which give less freedom to the users to customize/tune the generated code. This is because code transformations are internally managed by the generator in a very complex way, depending on the nature of the generator (model-to-model, model-to-text, text-to-text transformation rules, etc).

For code generator creators, configuring and testing code generators consists on applying a virtuous cycle known as the *"edit, compile, and test"* cycle. For example, in case of releasing a new generator version, developers may edit the templates and transformation rules that define the code generation process to add new features and settings, then run the generator to create the output files. The output files are then compiled and the generated application is tested. At this point, if they find a problem in the generated code, they alter the templates or the input of the generator and re-generate. This cycle is repeated as long as new changes are applied.

In case of using an off-the-shell code generator during software development (e.g., commercial code generators), engineers need to write the input program in the language supported by the generator (e.g., DSL, Model, GPL, etc). Afterwards, they apply code transformations by generating code to the target programming language. In this case, since the generator is not editable, the quality of the generated code depends only on the efficiency of the selected code generators for the target platforms. If they find any issues with the generated code, the bugs should be reported to the generator creators in order to fix them. For example, this is widely used in the industry by applying the concept "write once, run everywhere" where users can benefit from a family of code generators (e.g., cross-platform code generators [FRSD15]) to generate from the manually written (high-level) code different implementations of the same program in different languages. This technique is very useful to address diverse software platforms and programming languages.

The huge design space of compiler configuration options as well as the complexity of code generators make the activities of design, implementation, and testing very hard and time-consuming [GS15]. From the user's point of view, compilers and code generators are black box components that are used to ease the software production process. The quality of the generated software by either compilers or code generators is directly correlated to the quality of the code generator. As long as the quality of code generators is maintained and improved, the quality of generated software artifacts also improves. Any bug with these generators impacts on the software quality delivered to the market and results in a loss of confidence on the end users. As a consequence, generators testers check the correctness of generated source code or binaries with almost the same, expensive effort as it is needed for manually written code. Testing code generators or correctly tuning compilers is crucial and necessary to guarantee that no errors are incorporated by inappropriate modeling or by the compiler itself. 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 [SCDP07, YCER11] 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.

1.3 Scope of the thesis

In this thesis, we seek to test and evaluate the properties related to the resource usage of generated code. On the one hand, since many different target software platforms can be targeted by the code generator, we provide facilities to the code generator creators and users 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. Consequently, we automatically detect the non-functional inconsistencies caused by some faulty code generators. On the other hand, we provide a mechanism that help compiler users to select the best optimization sets that satisfy specific resource usage requirements for a broad range of programs and hardware architectures.

This thesis addresses two problems: (1) the problem of non-functional testing of code generators and (2) the problem of automatically auto-tuning compilers through the runtime execution and evaluation of the generated code. In particular, it aims at offering effective support for collecting data about resource consumption (e.g., CPU, memory) and detect inconsistencies yielding to an intensive resource usage, as well as an efficient mechanism to help compiler users to choose the best configuration that satisfy specific non-functional requirements and lead to performance improvement.

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

1.4 Challenges

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

- **Oracle problem:** One of the most common challenges in software testing is the oracle problem. A test oracle is the mechanism by which a tester can determine whether a program has failed or not. When talking about the non-functional testing of generators, this problem becomes more challenging because it is quite hard to determine the expected output of a generator under test (e.g., memory consumption of the generated program). Determining whether these non-functional outputs correspond to a generator anomaly or not is also not obvious. That is why testing the generated code becomes very complex when the software user has no precise definition of the oracle he would define. To alleviate the test oracle problem, techniques such as metamorphic testing[1] are widely used to test programs without defining an explicit oracle. Instead, it employs high-level metamorphic relations to verify the outputs automatically. So, which kind of test oracles can we define? How can we automatically detect inconsistencies? All these questions pose important challenges in testing generators.

- **Monitoring code generators/compilers behavior:** For testing the non-functional properties of code generators and compilers, developers generally use to compile, 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 [GS14, DGR04] 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 [PE06]. 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 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, reliability, etc. In order to deliver satisfactory levels of performance on different processor architectures,

[1] https://en.wikipedia.org/wiki/Metamorphic_testing

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 presented in the previous paragraphs.

RQ1. How can we help compiler users to automatically choose the adequate compiler configuration that satisfy specific non-functional requirements?

RQ2. How can we help code generator creators to automatically detect inconsistencies and non-functional errors within code generators?

RQ3. How can we provide efficient support for resource consumption monitoring and management?

1.5 Contributions

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

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

Contribution: automatic detection of inconsistencies within code generators families. In this contribution, we propose an approach for testing code generators families. This approach tries to automatically find real issues in existing code generators. It is based on the intuition that a code generator is often a member of a family of code generators. The availability of multiple generators with comparable functionality enables us to apply the idea of differential testing [McK98] to detect code generator issues. We evaluate our approach by analyzing the performance of Haxe, a popular high-level programming language that involves a set of cross-platform code generators. Experimental results show that our approach is able to detect some performance inconsistencies that reveal real issues in this family of code generators. In particular, we show that we could find two kinds of errors during code transformation: 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 deployment and monitoring of generated code. Finally, we propose a micro-service infrastructure to ensure the deployment and monitoring of different variants of generated code. It also automates the process of code compilation, deployment and execution in order to provide to software developers more facilities to test the generated code. This isolated and sandboxing 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 approach constitutes the playground for testing and evaluating the generated code from either compilers or code generators. This contribution answers mainly *RQ3* but the same infrastructure is used to validate the carried experiments in *RQ1* and *RQ2*.

1.6 Overview of this thesis

The remainder of this thesis is organized as follows:

Chapter 2 first contextualizes this research, situating it in the domain of generative programming. We give a background about the different concepts involved in the field of generative programming as well as an overview of the different aspects of automatic code generation in software development.

Chapter 3 presents the state of the art regarding our approach. This chapter provides a survey of the most used techniques for testing compilers and code generators. We focus more on the non-functional testing aspects. This chapter is divided on two parts. First, we study the previous approaches that have been applied for compiler auto-tuning. Second, we study the different techniques used to test the functional and non-functional properties of code generators.

Chapter 4 resumes the work done related to compiler testing. To do so, we study the impact of compiler optimizations on the generated code. Thus, we present a search-based technique called Novelty search for compiler optimizations exploration. We provide two adaptations of this algorithm: mono and multi objective search. We also show how this technique can easily help compiler users to efficiently generate and evaluate the compiler optimizations. The non-functional metrics we are evaluating are the performance, memory and CPU usage. We evaluate this approach through an empirical study and we discuss the results.

Chapter 5 presents our approach for the non-functional testing of code generators. It shows an adaptation of the idea of differential testing for detecting code generator issues. We report the results of testing multiple generators with comparable functionalities (a code generator family). The non-functional metrics we are evaluating in this section are the performance and memory usage of generated code. We also report the issues we have detected and we propose solutions for code generation improvement.

Chapter 6 shows the testing infrastructure used across all experiments. It shows the usefulness of such architecture, based on system containers, to automatically deploy and execute the generated code by either compilers or code generators. We report the comparison results of using this infrastructure and a non-containerized solution in terms of overhead and we discuss the results.

Chapter 7 draws conclusions and identifies future work and perspectives for testing compilers and code generators.

1.7 Publications

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

- Mohamed Boussaa, Olivier Barais, Benoît Baudry, Gerson Sunyé: **NOTICE: A Framework for Non-functional Testing of Compilers**. In *2016 IEEE Interna-*

tional Conference on Software Quality, Reliability & Security (QRS 2016), Vienna, Austria, August 2016.

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

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

Part I

Background and State of the Art

Chapter 2

Background

In this chapter, the context of this thesis and the general problems it faces are introduced. The objective of this chapter is to give a brief introduction to different domains and concepts in which our work takes place and used throughout this document. This includes generative programming techniques, an overview of the software development tool chain and the main concepts for testing code generators and compilers auto-tuning.

The chapter is structured as follows: In section 2.1, we present the problem of software diversity and hardware heterogeneity induced by the continuous software and hardware innovation. Section 2.2 aims at providing a better understanding of the generative programming concept that is increasingly applied to ease code generation and implementation. In section 2.3, we present the different steps of automatic code generation involved during software development as well the different stakeholders and their roles in testing generators. We highlight then, the main tasks that we are addressing in this thesis to help generators experts/users to efficiently evaluate the generated code and tune compiler configurations. Section 2.4 gives an overview of the different types of code generators used in the literature and we show how testing code generators is complex. Similarly, in Section 2.5, we describe compiler optimizations and we show how complex is compiler tuning by presenting the different challenges that this task is posing.

2.1 Diversity in software engineering

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

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

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

2.1.1 Software diversity

In today's software systems, different system variants are typically developed simultaneously to address a wide range of application contexts and customer requirements [SRC+12]. This variation is referred to *software diversity*. Baudry et al. [BM15b] and Schaefer et al. [SRC+12] have presented an exhaustive overview of the multiple facets of software diversity in software engineering. Software diversity can emerge in different types and dimensions such as diversity of operating systems, languages, data structures, components, execution environments, etc. Like all modern software systems, softwares have to be adapted to address changing requirements over time supporting system evolution, technology and market needs like considering new software platforms, new languages, new customer choices, etc.

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

JHipster[1] is also another concrete example that shows how software diversity is managed in industry production. JHipster is an application generator based on YO generator which provides tools to generate quickly modern web applications using Java stack on the server side (using Spring Boot) and a responsive Web front-end on the client side (with AngularJS and Bootstrap). The generated web application can be quite different from one user to another. It really depends on the options/choices selected by the user to build a configured application. The selected parameter values will configure the way the JHipster code generators will produce code. For example, Figure 2.1 shows a feature model of some configuration examples that the user would select. When building the applications, the user may select the database type he would generate, the Java version, the network protocol, etc. Using this feature model **more than 10k diverse architecture types** of project can be selected which means that 10k program variants may be generated depending on the different criteria.

Whatever configuration selected by the user, the application behavior will not change and the generated application will share a similar architecture and fundamental code-base.

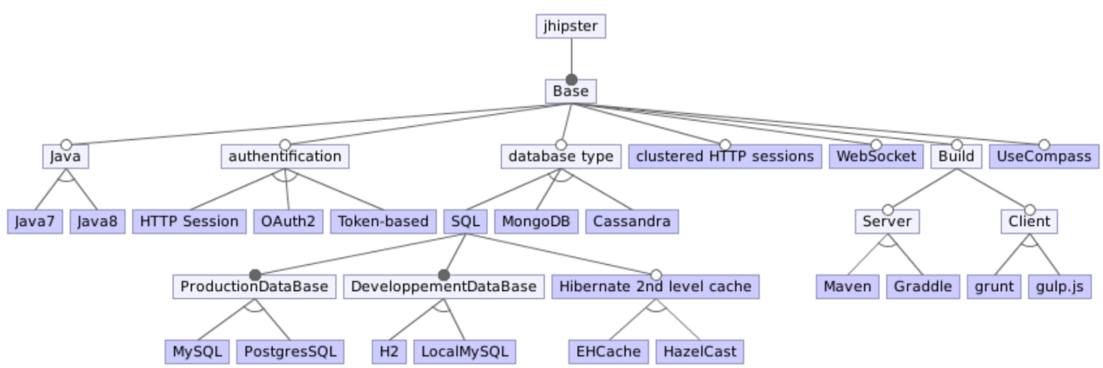

Figure 2.1: Example of JHipster feature model

Accordingly, we propose the following definition of software diversity: ***Software diversity is the generation or implementation of the same program specification in different ways/manners in order to satisfy one or more diversity dimension***

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

such as the diversity of programming languages, execution environments, functionalities, etc.

We define as well the term **"software family"** to categorize these diverse programs that share the same behavior/functionality

2.1.2 Hardware heterogeneity

On the hardware side, modern software systems rely nowadays on a highly heterogeneous and dynamic interconnection of devices that provide a wide diversity of capabilities and services to the end users. These heterogeneous services run in different environments ranging from cloud servers to resource-constrained devices. Hardware heterogeneity comes from the continuous innovation of hardware technologies to support new system configurations and architectural design (e.g., addition of new features, a change in the processor architecture, new hardware is made available, switch to low bandwidth wireless communication, etc). For example, since the early 1970s, the increase in capacity of microprocessors has followed Moore's law for Intel processors. Indeed, we observe that the number of components (transistors) that can be fitted onto a chip doubles every year, increasing the performance and energy efficiency. For instance, Intel Core 2 Duo processor was introduced in 2006 with 291 millions of transistors and 2.93 GHz clock speed. One year later, Intel has introduced the Core 2 Quad processors which came up with 2.66 GHz clock speed and the double number of transistors introduced in 2006 with 582 millions of transistors.

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

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

As shown in Figure 2.2, a compiler is typically divided into two parts, a front end and a back end. The compiler front-end verifies syntax and semantics and analyzes the source code to build an internal representation of the program, called the intermediate representation or IR. For example, the GNU Compiler Collection (GCC) and LLVM support many front ends with languages such as C, C++, Objective-C, Objective-C++, Fortran, Java, Ada, and Go among others. A compiler back end is typically responsible for code

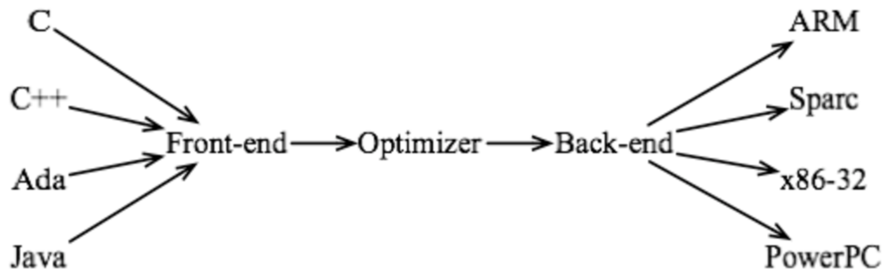

Figure 2.2: Compiler architecture

optimizations and code generation for a particular microprocessor. Today, GCC is able to generate code for approximately **more than 40 different processor architectures**. For example, one important option for compiler flags is *-march*. It tells the compiler what code it should produce for the system's processor architecture (or arch); it tells GCC that it should produce code for a certain kind of CPU. Using *-march=native* enables all the optimization flags that are applicable for the native system's CPU, with all its capabilities, features, instruction sets, and so on. There exits many other optimization options for the target CPU like *–with-arch=i7*, *–with-cpu=corei7*, etc. Generally, each time a new family of processors is released, compiler developers release new compiler version with more sophisticated optimization options for the target platform. For example, old compilers produce only 32-bit programs. These programs still run on new 64-bit computers, but they may not exploit all processor capabilities (e.g. they will not use the new instructions that are offered by x64 CPU architecture). For instance, the current x86-64 assembly language can still perform arithmetic operations on 32-bit registers using instructions like addl, subl, andl, orl, etc, with the l standing for "long", which is 4 bytes/32 bits. 64-bit arthimetic is done with addq, subq, andq, orq, etc, with q standing for "quadword", which is 8 bytes/64 bits.

In short, software developers need to deal with these compiler configurations to truly take advantage of the new chip with more advanced optimizations for the new hardware chip.

2.1.3 Matching software diversity to heterogeneous hardware: the marriage

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

On the one hand, software is facing challenges of a similar magnitude, with major changes in the way software is deployed, is sold, and interacts with hardware. Software diversity, as discussed in section 2.1.1, is driven by software innovation, driving the software development toward highly configurable and complex systems. This complexity is carried by the huge number of software technologies, customer configurations, execution environments, programming languages, etc. This explosion of configurations that software is facing makes the activity of testing and validation very difficult and time consuming. As a consequence, softwares become higher and higher level, managing complexity and gluing lot of pieces together to give programmers the right abstraction for how things really work and how the data is really represented. For example, model-driven software engineering and generative programming techniques such as SPL, DSLs, models, etc, are widely used to provide a new integrated software engineering approach which enables the advanced exploitation of the different dimensions of software diversity.

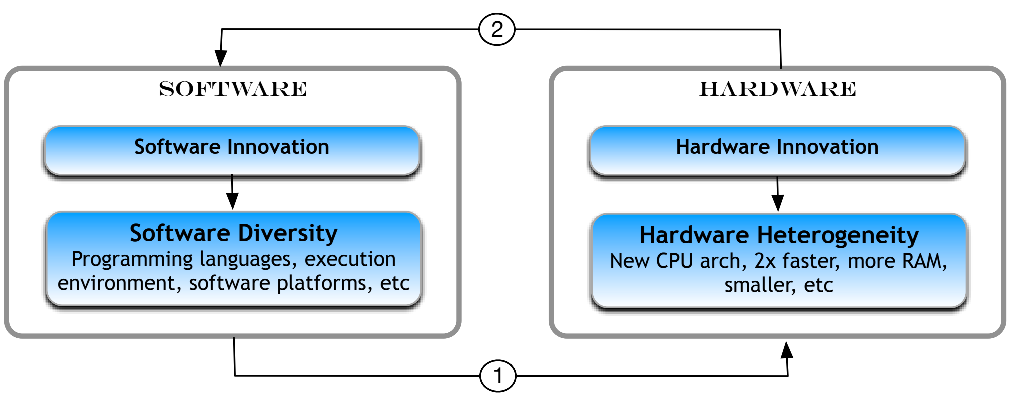

Figure 2.3: Matching software to hardware

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

Matching software to hardware is ensured by providing the adequate software languages and compilers that have to produce efficient code to the target hardware (relation 1 in Figure 2.3). As consequence, people who are writing compilers have to continuously enhance the way the executables are produced by releasing new compiler versions to support new hardware changes (i.e., new optimization flags, instruction sets). For example, Hou et al. [HZG10] have presented SPAP, a container-based programming language for heterogeneous many-core systems. This language allows programmers to write unified programs that are able to run efficiently on heterogeneous processors. SPAP comes with a set of compilers and runtime environments to such hardware processors. Chafi et al. [CDM+10, CSB+11] proposed leveraging domain specific languages (DSLs) to map high-level application code to heterogeneous devices. They showed that the presented DSL can achieve high performance on heterogeneous parallel hardware with no modification required to the source code. They compared this language performance to MATLAB code and they showed that it outperformed it in nearly all cases.

To avoid hardware heterogeneity, software developers use managed languages such as JAVA, Scala, C#, etc to favor software portability. Instead of compiling to native machine instruction set, these languages are compiled into an intermediate language or IL, which is similar to a binary assembly language. These instructions are executed by a JVM, or by .NET's CLR virtual machine, which effectively translates them to native binary instructions specific to the CPU architecture and/or OS of the machine. By using managed code and compiling in this managed execution environment, memory management such as a garbage collector, type safety checking, and destruction of unneeded objects are handled internally within this sandbox runtime environment. Thus, developers focus on the business logic of applications to provide more secure and stable software without taking too much care of the hardware heterogeneity.

In contrast, devices may impose the support of specific programming languages (relation 2 in Figure 2.3). In mobile development for example, Java is needed to implement Android applications and Objective-C is needed to develop iOS products. This means that developers need to create multiple clients in this heterogeneous environment.

2.2 From classical software development to generative programming

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

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

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 target software platform. In principle a software development process can be seen as a mapping between a problem space and a solution space [Cza05] (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 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/program 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.

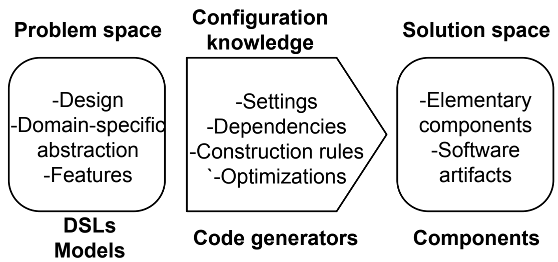

Figure 2.4: Generative programming concept

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

Some commonly benefits of such software engineering process are:

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

Among the main contributions of this thesis is to evaluate the impact of applied configurations during code transformation/optimization (by wether code generators or compilers) on the resource usage requirements.

In the following section, we present a general overview of the complete software development tool chain and the main actors that are involved from design time to runtime.

2.3 An overview of the software development tool chain

The process of generative software development involves many different technologies. In this section, we describe in more details the different activities and stakeholders involved to transform high-level system specifications into executable programs and that from design time to runtime.

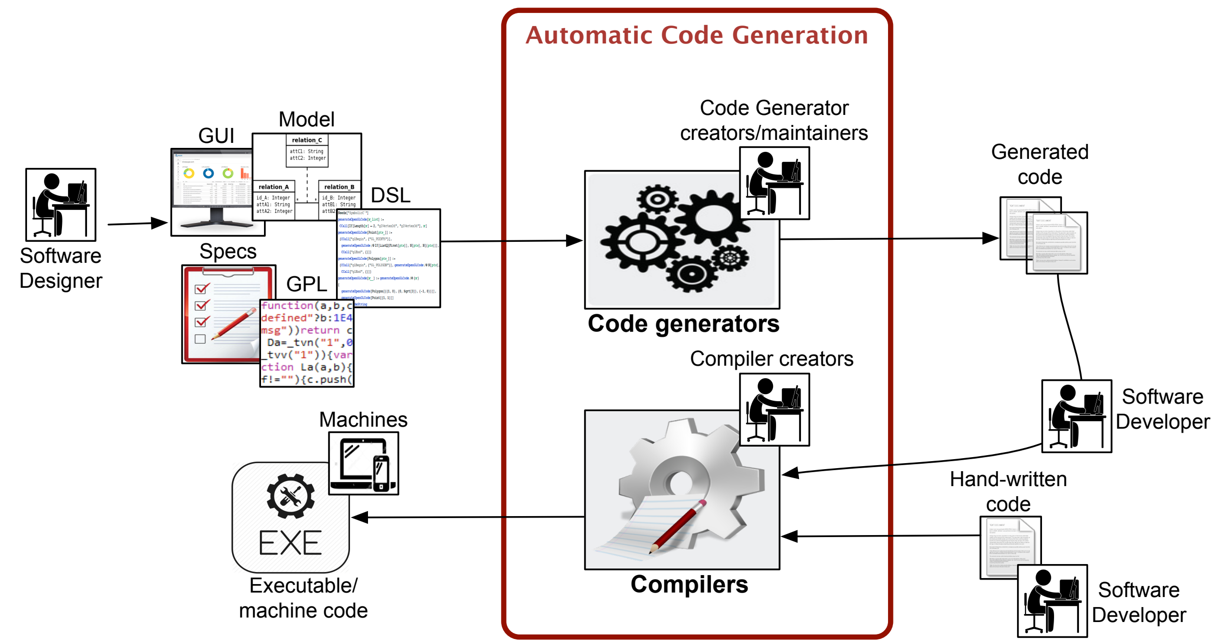

Figure 2.5: Overview of the software development chain

2.3.1 Automatic code generation

Figure 2.2 reviews the different steps of this software development chain. We distinguish four main tasks necessary for ensuring an automatic code generation:

1. ***Software design:*** As part of the generative programming process, the first step consists on representing the system behavior. On the input side we can either use code as the input or an abstract form that represents the design. It depends on the type of the code generator and on the input source program it requires. These programs can range from a formal specification of the system behavior to abstract models that represents the business logic. For example, software designers can define, at design time, softwares behavior using for example Domain-Specific Models (DSMs). A DSM, as an example, 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).

2. ***Code generation:*** Code generation is the technique of building code using programs. The common feature of the generator is to produce code that the software

developer would otherwise write by hand. Generators are generally seen as a black box which requires as input a program and generate as output a source code for a specific target software platform/language. Code generation can build code for one or more target language, once or multiple times. There are different varieties of code generation aspects and it highly depends on the type of the input programs described in the previous step. For example, code generator 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. Thus, the code generation process start by taking the previously defined specification to translate a model to an implementation in a target language. We will see in the next section the different types of code generators.

3. ***Software development:*** Software development may be divided into two main parts. On the one hand, software developers may follow the two previous steps in order to generate automatically code for a specific target software platform. In this case, they use to edit the system specification described in the first step (at a high level) and use to re-generate code each time needed by calling a specific generator. In some cases, generated code can even be edited by the end software developers. This task depends on the complexity of the generated code and it sometimes need software experts that can easily update and maintain the code. However, they may manually implement source code from scratch without using any abstractions or code generation aspects. In this case, they just need to compile and execute the hand-written code in order to test it.

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

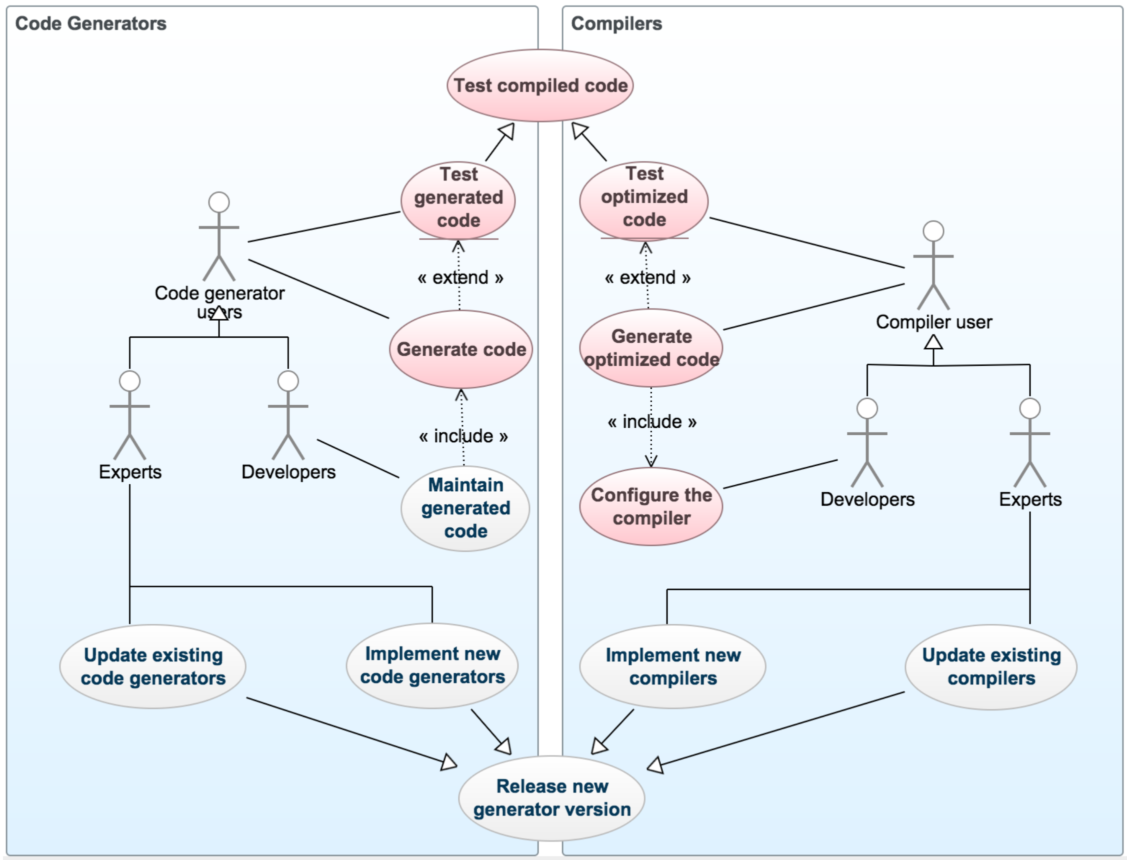

Figure 2.6: Use case diagram of the different actors/roles involved in implementing and testing generators

2.3.2 Stakeholders and their roles for testing generators

Software development involves several stakeholders that play different roles in validating and testing the software development chain described previously. Figure 2.6 depicts a use case diagram that describes these different concerns, actors and roles for testing generators. Basically, we distinguish two stakeholders for code generators and compilers testing: experts and developers. As shown in the bottom of Figure 2.6, experts are the creators/-maintainers of generators that use their expertise and knowledge associated to the software and hardware technologies resulting in efficient code generation. They contribute to the software development community by creating and providing new optimizations and compiler versions updates. For code generators, they may use their knowledge to build new platform-specific code generators or enhance existing ones.

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

The uses cases highlighted in red in Figure 2.6 constitute the main tasks that we are addressing in this thesis. Our main concern is to evaluate the generated code. To do so, we would help code generator users to automatically generate code for different target software platforms and detect code generator inconsistencies by evaluating the resource usage properties. This task may involve both, code generator experts and users. On the other hand, we would help compiler users to auto-tune compilers through the use of optimizations provided by compiler experts. Similarly, this concerns both actors and it consists on testing the impact of these configurations on the performance and resource usage properties.

2.4 Testing code generators

In this thesis, we focus on testing the automatic code generation process (highlighted in red in the left side of figure 2.6). To do so, we introduce in this section some basis about code generators. We give an overview of the different types of code generators and we discuss their complexity which constitute a major obstacle for testing.

2.4.1 Testing workflow

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.

As stated before, the code generation workflow is divided into two levels. It starts by transforming the system design into source code through the use of generators. Afterwards, source code is transformed into executables using compilers. Thus, software developers use

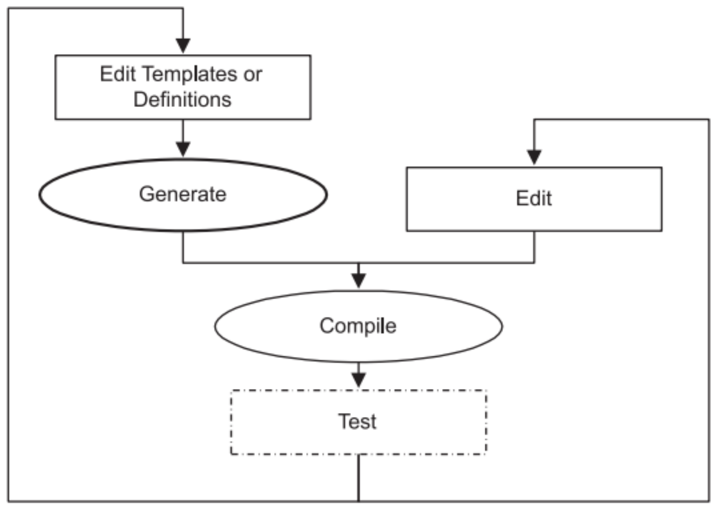

Figure 2.7: Code generation workflow

to generate code, edit it (if needed), compile it and then test it. If changes are applied to compilers or generators, the cycle is repeated. Figure 2.7 presents an overview of this testing cycle. The right-hand side of the figure shows the classic workflow for developing and debugging code which is *edit, compile, and test.*. The user writes or edits an existing code, compiles it using specific compilers, and tests it. Code generation adds a few new workflow elements in the left-hand side of the figure where generator creators edit the templates and definition files (or the generator itself) and then run the generator to create new output files. The output files are then compiled and the application is tested.

2.4.2 Types of code generators

There are many ways to categorize generators. We can differentiate them by their complexity, by usage, or by their input/output. According to [Her03], there are two main categories of automatic code generation: passive or active. Passive code generators build the code only once, then its up to the user to update and maintain the code. The most common use of passive code generators are wizards.

Active code generators, run on code multiple times during the lifecycle. With active code generators, there is code can be edited by the users, and code that should only be modified by the code generator. Active code generators are widely referenced in the literature [PBG05, AE09]. We focus on this thesis on testing this class code generators.

According to the state-of-the-art [Her03, HT00, FB08, BNS13], there are six categories of active code generators:

- **Code munger:** A code munger reads code as input and then builds new code as output. This new code can either be partial or complete depending on the design of the generator. A code munger is the most common form of code generators and are used widely. This kind of generators are often used for automatically generating documentations. A source-to-source compiler, transcompiler or transpiler [2] can also be defined as code mungers. A transcompiler takes a code written in some programming language and translates it to a code written in some other language. **Our contribution related to code generators testing will focus on this kind of generators to validate our approach for automatically detecting inconsistencies.**

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

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

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

- **Mixed code generator:** This model has the same processing flow as the Inline Code Expander, except that the input file is a real source file that can be compiled and run. The generated output file keep the original markup that will denote where the generated code was placed. It enables code generation for multiple small code fragments within a single file or distributed throughout multiple files. Generally, transformation rules are defined using regular expressions.

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

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

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

  Example: ArgoUML and Codegen translate UML class diagrams to general-purpose languages such as C#, Java and C++. They do not generate complete implementations, but it tries to convert the input UML class diagrams into skeleton code that the user can easily edit it.

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

  Examples: Database Access layer, Web client layer, Data export, import, or conversion layers

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

  Example: Matlab is a domain specific math language that makes it easy to represent mathematical operations for example rather than object-oriented languages. Mathematica, ant and SQL languages are other examples.

2.4.3 Why testing code generators is complex?

The complexity of code generators remains on the different code generation models described above. 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 [GS15].

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.

2.5 Compilers auto-tuning

The compiler is a very essential software component in software engineering, responsible for translating user'source code written in general purpose languages into machine code. The key feature of compilers is to bridge source programs written in high-level languages with the underlying hardware architecture.

High-level languages are used to help the software developer to have an easier and simpler way for writing programs. They offer many abstract programming features such as functions, data structures, conditional statements and loops that facilitates software development. Writing code in a high-level programming language may induce significant decrease in performance. Principally, software developers should write understandable, maintainable code without putting too much emphasizes on the performance for example.

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

This process is called program optimization.

2.5.1 Code optimizations

Code optimization within a compiler is the process of transforming a source code program into another functionally equivalent code for the purpose of improving one or more of its non-functional properties.

The most common outcome of optimizations is to minimize the execution time of program execution. Other less common non-functional properties are code size, memory usage and power consumption.

There exist many types of optimizations such as loop unrolling, automatic parallelization, code-block reordering and functions inlining among others. The factors that affect optimizations may include characteristics such as: the number of CPU registers (the more registers, the easier it is to optimize for performance), cache size, CPU architecture, etc.

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

- **Machine-independent optimization:**

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

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

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

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

2.5.2 Why compilers auto-tuning is complex?

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

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

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

We resume, in the following, several issues with optimizing compiler technology which make the activity of compiler tuning very complex:

- **Conflicting objectives:** Compilers usually have to support a variety of conflicting objectives, such as execution time, compilation speed, resource usage and quality of generated code. It is difficult to define a set of optimizations that satisfy all properties.

- **Optimization interactions:** The interaction between optimization phases as well their application order make it difficult to find an optimal sequence.

- **Huge number of optimizations:** The huge number of optimizations is also an issue for the compiler user to choose the best optimization sequence since an exhaustive search is impossible (we count $2^{number\ of\ optimizations}$ possible combination to evaluate).

- **Non universal optimizations:** There is no one universal optimization sequence that will enhance the performance of all programs. Optimization's impact depends on the hardware and on the input program. Thus, constructing an optimization sequence for different programs and hardware architectures becomes very hard and time-consuming.

- **Compiler bugs:** Optimizations may lead to compiler bug and introduce errors in the compiled code. Optimizations must not cause any change in program behavior under any possible condition [LAS14, YCER11].

- **Optimization overhead:** Optimizations should be fast and efficient. They should not delay the overall compiling process.
- **Tuning compilers need expertise:** In case the compiler user has no knowledge and expertise about the compiler technology and its optimizations, it will be quite hard to select the set of optimization sequences to apply.

2.6 Summary: Testing challenges

- **Auto-tuning compilers**: Compilers may have a huge number of potential optimization combinations, making it hard and time-consuming for software developers to find/construct the sequence of optimizations that satisfies user specific key objectives and criteria. It also requires a comprehensive understanding of the available optimizations of the compiler. So, how can we help the compiler user to automatically auto-tune compilers and choose which optimizations he should apply to satisfy a specific non-functional property?
- **Detecting code generator inconsistencies**: Automatic code generation 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 that may themselves contain bugs. Thus, testing code generators becomes very needed. So, how can we automatically detect issues within code generators? Moreover, proving that the generated code is functionally correct is not enough to claim the effectiveness of the code generator under test? What about testing the non-functional properties of automatically generated code?
- **Resource usage monitoring of generated code:** Analyzing the resource usage of optimized or generated code requires a dynamic and adaptive solution that extract efficiently those properties. Due to the software diversity and hardware heterogeneity, monitoring the resource usage of each execution platform becomes challenging and time-consuming. So, how can we ease this process and provide an efficient solution that will help compiler users/experts to evaluate the optimizations and code generator users/experts to test the generated code in terms of non-functional properties?

Chapter 3

State of the art

In this chapter, we present the state-of-the-art of this thesis. We discuss previous efforts in three research areas: (1) Compiler auto-tuning in iterative compilation, (2) code generator testing, and (3) container-based testing.

We first present a brief introduction to the iterative compilation research filed and we identify severals problems that have been investigated in this field. We discuss as well, the different techniques that have been applied to automate the process of auto-tuning compilers. To sum up, we provide a summary table showing the most important research in iterative compilation.

Afterwards, we move to discuss existing techniques related to code generator testing. We start by studying the state-of-the-art approaches related to the functional testing of code generators. In a second stage, since there are few research efforts that investigate the automatic non-functional testing of code generators, we rather focus on studying the oracle problem and the different methods applied to the oracle definition. This will be useful later, to understand our contribution about the non-functional testing of code generators. We end this section by providing a summary table of these approaches.

Finally, we discuss in the last section the container technology as means of automatic software deployment, monitoring, and testing. We compare then, the container's virtualization solution to the classical approaches based on virtual machines. We provide as well, examples of existing solutions in research and industry that opted for this technology to automate software testing and monitoring.

This chapter is structured as follows:

Section 3.1 provides a survey of the most used compiler auto-tuning techniques to construct the best set of optimization options.

In Section 3.2, we review existing techniques for code generators testing and the principal categories of oracles.

Then, in Section 3.3, we discuss the use of virtualization technology for deployment and testing. In particular, we presents some of the approaches that used a container-based solution for automatic testing and monitoring.

Finally, Section 3.4 discusses the limitations of the state of the art.

3.1 Compilers auto-tuning techniques

3.1.1 Iterative compilation

Iterative compilation, *also known by optimization phase selection, adaptive compilation, or feedback directed optimization*, consists in applying software engineering techniques to produce better and more optimized programs by compiling multiple versions of each of them using different optimizations settings. After running these versions on specific hardware machines, the key objective of iterative compilation is to find the best optimizing sequence that leads to the fastest and better code machine code. 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 Software Engineering (SBSE) techniques to guide the search towards relevant optimizations regrading performance, energy consumption, code size, compilation time, etc. Experimental results have been usually compared to standard compiler optimization levels.

It has been proven that optimizations are highly dependent on the target platform and on the input program. 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.

3.1.2 Implementation of iterative compilation system

The implementation of an iterative compilation system consists mainly on applying a sequence of steps to enhance the quality of the generated code. Figure 3.1 shows a general

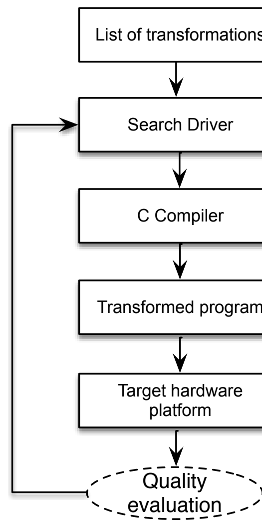

Figure 3.1: Overview of the iterative compilation process

overview of the main steps needed to ensure the implementation of the iterative compilation process.

- **List of transformations:**

  The iterative process starts by defining the optimization space. It represents the list of optimizations that the compiler have to evolve during the search to enhance the software quality.

- **Search driver:**

  It applies a search algorithm or method to efficiently explore the large optimization search space. In fact, it reads the previously defined list of transformations that it needs to examine and decides which transformations have to be applied next using a search algorithm to steer through the optimization space.

- **C Compiler:**

  Once the optimization sequence is defined, the target C compiler is called to compile the input program and also perform initial machine independent optimizations.

- **Transformed program:**

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

– **Target hardware platform:**

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

– **Quality evaluation:**

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

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

3.1.3 Iterative compilation search techniques

In Section 2.5.2 of Chapter 2, we presented several issues with optimizing compilers that make the activity of compiler tuning very complex such as the huge number of optimizations, conflicting objectives, optimization overhead, etc. In this section, we discuss the available tools and approaches dedicated to the automatic search for optimal compiler settings, and give an overview of known approaches that addressed the several compiler optimization challenges. In each subsection, we identify and discuss a particular problem and we present the best known approaches proposed to solve it.

Auto-tuning: a mono objective optimization

Auto-tuning is an area that study of automatic code generation and optimization for different computer architectures. This technique has been used in many optimization scenarios. What all of this prior work on iterative compilation has in common is that it focuses on a single objective function to be optimized. For example, researchers typically focus on speeding up the performance of compiled code which constitutes the major optimization objective for most iterative compilation approaches [PE06, HKW05].

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

For example, ACOVEA [1] as Analysis of Compiler Options via Evolutionary Algorithm, is an open source tool that utilize genetic algorithms to find the best options for compiling programs with the GNU Compiler Collection (GCC) C and C++ compilers. In this context, best solutions define those options that produce the fastest executable program from a given source code. This tool was even included in Gentoo Linux repository to help users to find their best set of optimizations.

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

Most the approaches described above focus on auto-tuning compilers without putting too much emphases on the tuning time. The process of auto-tuning may be too long and can last more than one month [HE08].

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

As a consequence, Pan et al. [PE06] introduce a new algorithm called combined elimination (CE) that was shown to outperform all previous search-based techniques in finding good optimization settings with considerably fewer evaluations. The proposed solution, PEAK, achieves fast tuning speed by measuring a small number of invocations of each code section, instead of the whole-program execution time, as in common solutions. Compared to these solutions PEAK reduces tuning time from 2.19 hours to 5.85 minutes on average, while achieving similar program performance. PEAK improves the performance of SPEC CPU2000 FP benchmarks by 12% on average over GCC O3, the highest optimization level, on a Pentium IV machine.

Escaping local optimum

A common problem in iterative compilation is the local optimum. In fact, the search space of optimizations for a specific program could be very huge and it generally contains many local minima in where the search algorithm could be trapped [BKK+98]. Therefore, researchers in this filed try to build robust techniques and algorithms to avoid such problem. In [BKK+98], Bodin et al. tried to analyze this search space and they found that the optimization space is highly non-linear containing many local minima and some discontinuities. Therefore, techniques based on gradient approaches such as GAs are not applicable. This paper has focused on parameterized transformations. The small area of the transformation space considered in this paper is composed by three parameterized optimizations: loop unrolling (with unroll factors from 1 to 20), loop tiling (with tile sizes from 1 to 100) and padding (from 1 to 100). They focused on compilation time and execution time of the optimized program and used a simulator to target embedded processors. The compiler used is a compiler framework developed to optimise multimedia codes for embedded systems. They analyzed these optimizations on four CPU architectures (UltraSparc, R10000, Pentium Pro, and TriMedia-1000) and the matrix multiplication is selected as the program to optimize. The proposed search algorithm visits a number of points at spaced intervals, applying the appropriate transformation, executing the transformed program and evaluating its worth by measuring the execution time. Those points lying between the current global minimum and the average are added to an ordered queue. Iteratively, such points are removed from the queue and points within the neighboring region are investigated, again at spaced intervals. This process is continued until a specific number of points have been evaluated and the fastest transformed program is reported. They showed that in the case of large transformation spaces, they can achieve within 0.3% of the best possible time by visiting less then 0.25% of the space using a simple algorithm and find the minimum after visiting up to less than 1% of the space.

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

Another way to efficiently explore the large search space in compiler optimization is the Design Space Exploration (DSE) technique based on software code clustering to search for optimization sequences aiming at performance improvements of code fragments (e.g., functions), considering the target processor and the set of optimizations supported by the compiler [MND+14, MNC+16]. They proposed a good sequence of optimizations in application-dependent mode. The DSE is based on a clustering approach for grouping functions with similarities and exploration of a reduced search space resulting from the combination of optimizations previously suggested for the functions in each group. The identification of similarities between functions uses a data mining method that is applied to a symbolic code representation. They compare their approach to the GA and their experimental results reveal that the DSE-based approach achieves a significant reduction in the total exploration time of the search space (20 over a Genetic Algorithm approach) and a performance speedups (41% over the baseline).

Phase ordering problem

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

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

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

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

Several researchers looked at searching for the best sequence of optimizations for a particular program. For example, the work of Cooper et al. [CSS99] adapts the genetic algorithm to solve the optimization phase ordering problem. The focus of this paper is about optimizing for embedded systems and then reducing the code size. They choose 10 program transformations to evolve in Fortran compiler. The solutions generated by this algorithm are compared to solutions found using a fixed optimization sequence and solutions found by testing random optimization sequences. Their technique was successful for reducing the code size by 40% compared to the standard sequence.

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

Evaluating iterative optimization across multiple data sets

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

Conflicting objectives: a multi-objective optimization

The vast majority of the work on iterative compilation focuses on increasing the speedup of new optimized code compared to standard compiler optimization levels. However, they do not put too much emphasis on finding trade-offs between two (or more) non-functional properties [ACG+04, HE08, PE06, PHB15, CFH+12, MND+14, LCL08, MÁCZCA+14].

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

The experimental results using GCC (v4.1.2) show that the automatic construction of optimization levels is feasible in practice, and in addition, yields better optimization levels than GCCs manually derived (-Os, -O1, -O2 and -O3) optimization levels, as well as the optimization levels obtained through random sampling. However, They do not provide a guarantee that the new explored optimization levels selected for SPEC still will be optimal for other applications.

Martinez et al. [MÁCZCA+14] propose an adaptive worst-case execution time WCET-aware compiler framework for automatic search of compiler optimization sequences which yield highly optimized code. Compared to the previously described approaches, authors in this paper focus on generating efficient code for embedded systems. Embedded systems are characterized by both efficiency requirements and critical timing constraints. Properties as average-case performance, power consumption and resource utilization are the main concerns describing the efficiency of a system. Thus, they explore the performance of compiler optimizations with conflicting goals. Besides the objective functions average-case execution time and code size, they consider the WCET, which is a crucial parameter for real-time systems, especially for safety-critical application domains to avoid system failure. Then, they try to find suitable trade-offs between these objectives in order to identify Pareto optimal solutions using stochastic evolutionary multi-objective algorithms. The objective functions try to minimize the WCET-ACET and WCET-Code size properties. They apply three evolutionary multi-objective algorithms (EMO) namely IBEA, NSGA-II and SPEA2 and compare their results to standard levels (O1, O2 and O3). They evolve 30 optimizations within the WCC compiler and performed experiments on top of one single machine shipped with Intel Quad-Core CPU processor. They pick up as well 35 programs from various benchmarks such as DSPstone, MediaBench, MiBench, etc. They find that NSGA-II is the most promising EMO for the given problem. In fact, the discovered optimization sequences significantly outperform standard optimization levels: the highest standard optimization level O3 can be outperformed for the WCET and ACET on average by up to 31.33% and 27.43%, respectively. The same approach performs as well for the WCET-Code size optimization with a 30.6% WCET reduction over O3. However, the code size increases by 133.4%. This is because the WCET and the code size are typical conflicting goals. If a high improvement of one objective function is desired, a significant degradation of the other objective must be accepted.

In [PMV+13], the TACT framework is presented. Compared to previous approaches, TACT is designed primarily for automatic tuning on embedded systems running Linux. Thus, the target CPU architecture for this tool is the ARM architecture (ARM Cortex-A9) and 200 options are used in the GCC compiler for ARM.

TACT supports multiple optimization objectives, so it can tune either for a single optimization parameter, or for two objective functions simultaneously, for example, for performance and code size (or compile time). So, it applies the SPEA2 algorithm and GA for mono-objective optimizations.

The results show how the SPEA2 outperforms the standard GCC levels (O2, O3 and Os) across several open-source popular applications such as C-Ray, Crafty Chess, SciMark, x264 and zlib.

Predicting optimizations: a machine learning optimization

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

Generally, machine learning techniques create mainly in an off line phase a prediction model, which will be used to determine the compiler optimization set that should be used on a test (unseen) program by the online phase. The main advantage of this technique is that it reduces the number of required program evaluations.

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

The proposed infrastructure is based on a machine learning compiler that presents an Interactive Compilation Interface (ICI) and plugins to extract program features (such as the number of instructions in a method, number of branches, etc) and select optimization passes.

The Milepost framework proceeds in two distinct phases: training and deployment. During the training phase, information about the structure of programs (input training programs) is gathered, showing how they behave under different optimization settings. Such information allows machine learning tools to correlate aspects of program structure, or features, with optimizations, building a strategy that predicts good combinations of optimizations. After running an iterative process that evaluates different combinations of optimizations on top of the training programs/features, pedictive models are created to correlate a given set of program features with profitable program transformations. Then, in the deployment phase, the framework analyzes new unseen programs by determining the program features an,d passes them to the new created models to predict the most profitable optimizations to improve execution time or other metrics depending on the users optimization requirements.

GCC was selected as the compiler infrastructure for Milepost as it is currently the most stable and robust open-source compiler. They evolved 100 optimization flags under O1, O2 and O3 levels and compared their results to the O3 level and to the random search.

The experimental results show that it is possible to improve the performance of the MiBench benchmark suite automatically using iterative compilation and machine learning on several platforms, including x86: Intel and AMD, and the ARC configurable core family. Using the machine learning-based framework, they were also able to learn a model that automatically improves the execution time of some individual MiBench programs by a factor of more than 2 while improving the overall MiBench suite by 11% on reconfigurable ARC architecture, without sacrificing code size or compilation time. Furthermore, their approach supports general multi-objective optimization where a user can choose to minimize not only execution time but also code size and compilation time.

Summary: auto-tuning compiler techniques

3.2 Testing code generators

3.2.1 Testing functional properties

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

In case of model-based software development, various approaches have been proposed to verify the model-to-code translation. Verification's purpose is to check that the generated code correctly implements the designed model.

As an example, Conrad et al. [CMKSP10] present an approach called Code Generation Verification (CGV). It represents an automated testing-based approach to assess the equivalence between Simulink models and the generated code.

CGV represents an automated testing-based approach to assess the numerical equivalence between the model used (i.e., Simulink models) and the generated code (i.e., the executables derived from the generated C Code). This procedure is also known as equivalence, comparative, or back-to-back testing approach [Vou90, McK98].

In fact, each individual model-to-code translation is followed by a verification phase to assess that the input Simulink model used for code generation and the output (i.e., the object code derived from the model via code generation and compilation) produce the same numerical results when stimulated with identical inputs.

In their equivalence testing approach, they use to run the model used for code generation using simulation and the generated code with the same input stimuli (test vectors) followed by a numerical comparison of the outputs (result vectors).

Then, they check whether or not the semantics of the model have been preserved during code generation, compilation and linking, by comparing the result vectors, which are the outputs resulting from stimulation with identical test vectors of the model and the generated code.

More precisely, the simulation results should be similar to the execution results. However, when defining the result vector comparison, they tolerate limited differences between both results. They argue that some factors between simulation and execution may cause a small difference between both executions such as limited precision of floating point numbers, target optimized code constructs, quantization effects when using fixed point math and compiler dependent behavior. Thus, they define an application-dependent threshold. So, two result vectors are considered sufficiently similar when their difference is less than or equal to the threshold value.

They illustrate the CGV-based translation validation in the context of embedded automative software by using Simulink and Real-Time Workshop Embedded Coder for verification.

Stuermer et al. [SCDP07] present a systematic test approach for model-based code generators. They investigate the impact of optimization rules for model-based code generation by comparing the output of 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 [JS14], 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. Basically, Genesys realizes back-to-back testing by executing both the source model as well as the generated code on top of different target platforms. Both executions produce traces and execution footprints which are then compared.

3.2.2 Testing non-functional properties

Previous work on non-functional testing of code generators focuses on comparing, as oracle, the non-functional properties of hand-written code to automatically generated code [SP15, RFBJ13]. As an example, Strekelj et al. [ŠLG15] implemented a simple 2D game in both the Haxe programming language 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 the hand-written one.

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

The oracle problem

Most of the work on software testing seeks to automate as much as possible the testing process to make it faster, cheaper, and more reliable.

To this end, automatic program testing involves in selecting suitable inputs as test cases, executing the program and verifying results against expected results. The mechanism against which the software tester verify whether the outputs of the program for the executed test cases are correct or not is called *test oracle*.

For instance, the problem of automatically generating test inputs/data has been the subject of research interest for a long time. It consists in automatically generating test data/cases in order to detect faults/crashes or to maximize the code coverage of the program under test.

Research in software testing has focused on automating many aspects of the testing process such as test data generation, test cases generation and execution, test suites reduction, test cases prioritization, etc. Many techniques, especially search-based testing techniques, have been proposed to solve and automate these processes [ABHPW10].

However, compared to many aspects of test automation, the problem of automating the test oracle is still challenging and less well solved. Only few techniques are available to generate test oracles. In most of the cases, designing and implementing test oracles are still manual and expensive activities. That is because the test oracles are not always available and may be hard to define or too difficult to apply [BM+15a]. This is commonly known as the *"oracle problem"*.

As pointed out in [MK01], the oracle problem has been one of the most difficult tasks in software testing but it is often ignored by researchers in software testing.

For example, let *sine* be a function implementing a specification *S*. Let *D* represent the input domain which represents the input values for which we would calculate the *sine*. Usually, the testing process should verify if the $sine(x) = S(x) \; \forall \; x \in D$. The procedure through which the testing program can check whether $sine(x) = S(x)$ is called an oracle.

Automatically generating *D* (test data) and executing test cases in this example is simple. It consists in generating random numerical values for example. However, it is impossible to do an exhaustive testing to automatically check the expected output.

For instance, only the special input values *0*, $\pi/4$, $\pi/2$, etc., could be the standard test cases since the output of *sine(x)* for these test data values is known and can be automatically checked against the specification. Nevertheless, special values cannot give us enough confidence in the correctness of the program on more complex or random inputs.

Other examples include, testing programs calculating numerical functions or solving complex equations; testing programs that calculate combinatorial problems, testing graphical user interfaces, etc.

Automatically testing generators also implies the oracle problem. When testing compilers, for example, it is quite difficult to automatically verify the equivalence between the source code and object code. This task becomes even more complicated when some optimizations are applied to the machine code.

When testing code generators, the verification of the high level system specification with the generated code is also challenging. Supposing that the *sine* function is defined using a high level language. The generated *sine* function should be evaluated with the same effort as the manual written code in order to verify whether the code generator has introduced bugs or not.

The research community has proposed several approaches [HMSY13, BM+15a] to alleviate the oracle problem. In a recent survey, Harman et al. [HMSY13] classify test oracles in three categories *specified oracles*, *implicit oracles*, and *derived oracles*. We are going to discuss them:

- Specified oracles:

  Specified oracles can be generated from several kinds of specifications, such as algebraic specifications or formal models of the system behavior. Stocks et al. [SC96, RAO92] discuss an approach for deriving test cases and oracles from specification. The idea is that the formal specification of a software can be used as a guide for designing functional tests. Then, test oracles can be associated with individual test templates (test case specifications). Thus, they construct abstract models of expected outputs, called oracle templates. The approach is illustrated with test oracle templates for the Z specification. Specified oracles are effective in identifying errors, but the task of defining and maintaining specifications is very expensive and time consuming. These kinds of oracles are also less adopted in industry.

- Implicit oracles:

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

  Implicit oracles [HMSY13, BM+15a] are easy to obtain at practically no cost. At the same time, implicit oracles are mostly incomplete, since they are not able to identify internal problems and complex failures, but they help to detect, in a black-box way, general errors like system crashes or un-handled exceptions.

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

- Derived oracles:

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

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

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

  Additionally, oracles can be derived from properties of the system under test. For instance, metamorphic testing [CHTZ04] generates test oracles by exploiting metamorphic relations of the system under test. In particular, metamorphic testing exploit the relation between the inputs and outputs of special test cases to derive metamorphic relations defined as test oracles for new test cases.

  A metamorphic testing (MT) method has been proposed to alleviate the oracle problem [CHTZ04]. MT is an automated testing method that employs expected properties of the target functions to test programs without human implication. MT recommends that, given one or more test cases (called source or original test cases) and their expected outcomes, one or more follow-up test cases can be constructed to verify the necessary properties (called Metamorphic Relations MRs) of the system or function to be implemented. For a given problem, usually more than one MR can be identified. It is therefore interesting to select effective MRs that are good at detecting program bugs.

  A recent alternative strategy is equivalence modulo inputs (EMI) testing. Given a well-defined, deterministic program P, EMI testing involves first performing code coverage analysis of P with respect to an input I to identify I-dead statements: statements not covered by I. From P, a series of program variants, P1, P2, ..., Pn can be created, with each Pi obtained by mutating or deleting a subset of the I-dead statements of P. Each Pi should behave identically to P when executed on input I; deviations in behaviour are indicative of compiler bugs. Thus, EMI testing is an

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

example of metamorphic testing the programs are in a metamorphic relationship with one another and with P, with respect to I.

Another type of derived oracles are pseudo-oracles (also known as differential testing, dual coding and N-version programming). The concept of a pseudo-oracle is introduced by Davis and Weyuker [DW81] to address so-called non-testable programs "Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known." A pseudo-oracle is program that is capable of providing the expected outputs and check the correctness of the system by comparing the outputs of multiple independent implementations. It checks the consistency of the results of the different software versions of the systems, when the same functionality is executed. An inconsistency can be detected when one or more versions of the system trigger failures. For example, in compiler testing, different versions of the same program are generated by applying some optimizations. The functionality of the program under test remains the same for all versions. The oracle is this case can be defined as the comparison between the functional outputs between the different versions which should be the same [YCER11].

Summary: oracle definition approaches

3.3 Container-based testing techniques

Using virtual machines (VMs) as a mechanism for deploying and testing applications in the cloud is very useful to run applications workload in heterogeneous and distributed environments. In industry, a number of commercial usage scenarios benefit from virtualization techniques to provide services to the end users. For example, Amazon EC2 makes VMs available to the customers who can use them to run their own computer applications or services on the cloud. Thus, a user can create, launch and terminate new VMs as needed.

However, VMs are known to be very expensive in terms of system resources and performance. In fact, each new VM instance constitutes of a virtual copy of all the hardware of the host machine which adds a lot of resource usage and overhead [Mer14].

Container-based virtualization presents an interesting alternative virtualization technology to virtual machines in the cloud. Containers are an operating-system-level virtualization which imposes little to near zero overhead. Programs in virtual instances use the operating system's system call interface and do not need to be emulated or run in an intermediate virtual machine, as is the case VMs such as VMware, QEMU or Xen. Docker offers the ability to deploy applications and their dependencies into lightweight containers that are very cheap to create and isolated form each other. Processes executing in a Docker container are isolated from processes running on the host OS or in other Docker containers. The Docker solution aims to address the challenges of resource, speed and performance of virtualization in the software development process.

Authors of [SCTF16, SPF+07, Mer14, FFRR15] used to compare the performance of traditional virtual machine solution that isolate VMs at the hardware abstraction layer to container-based operating system technology that isolate VMs at the system call layer. They showed that containers result in better performance than VMs since thay induce less overhead.

The container-based infrastructure has been also applied to the software testing, especially in the cloud [LTC15]. Sun et al. [SWES16] present a tool to test, optimize, and automate cloud resource allocation decisions to meet QoS goals for web applications. Their infrastructure relies on Docker to gather information about the resource usage of deployed web servers.

3.3.1 Deployment and testing using Docker

In software development, the container technology becomes more and more used in order to create a portable, consistent operating environment for development, deployment, and testing.

In the following, we will focus and discuss state of the art approaches that chose the container technology as an efficient infrastructure to solve some testing research problems.

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

Another Docker-based approach is presented in the BenchFlow2 project which focuses on benchmarking BPMN 2.0 engines [FIP15]. This project is dedicated to the performance testing of workflow engines. In this work, Ferme et al. present a framework for automatic and reliable calculation of performance metrics for BPMN 2.0 WfMSs.

According to the authors, benchmarking WfMSs raises many challenges: 1) the system deployment complexity due to the distributed nature of these models execution; 2) the high number of configuration options required to integrate the configuration and the deployment of the system under test, i.e., the WfMS, as part of the performance test definition; 3) the complexity of the execution behaviours that can be expressed by modern modeling and execution languages such as BPMN2.

Therefore, to address these problems, BenchFlow exploits **Docker** as a containerization technology, to enable the automatic deployment and configuration of the WfMS and to ensure that the experimental results can be reproduced. Thus, the WfMSs are automatically deployed and undeployed using Docker. Each component involved in the benchmark are packaged as Docker images to be deployed and executed on different servers connected by a dedicated local network. For each Docker instance, a new instance of the BP models set is executed by the WfE during the experiment.

Thanks to Docker, BenchFlow automatically collects all the data needed to compute performance metrics, and to check the correct execution of the tests (metrics related the RAM/CPU usage and execution time). Their experimental results show that a simple BP model running on two popular open-source WfMSs have uncovered important performance scalability issues.

Hamdy et al. [HIH16] propose Pons, a web based tool for the distribution of pre-release mobile applications for the purpose of manual testing. Pons facilitates building, running, and manually testing of Android applications directly in the browser. Based on Docker technology, this tool gets the developers and end users engaged in testing the applications in one place, alleviates the tester's burden of installing and maintaining testing environments, and provides a platform for developers to rapidly iterate on the software and integrate changes over time. Thus, it speeds up the testing process and reduces its cost.

Pons utilizes Docker by predefining Docker images that contain the required services and tools to build android applications, starting from the operating system up to the software development kit (SDK). A container is then built using one of these images to store the source code of a mobile application at specific moment of history in a sandbox environment. Pons creates then, an android emulator inside the docker container to run the tests. The results are streamed at runtime in the web browser.

3.3.2 Runtime monitoring using Docker

Runtime monitoring is an essential part of cloud computing [ABDDP13]. Like virtual machines before them, containers require a monitoring mechanism. It should provide both historical and timely information about the resource usage from hardware to virtual machine and container level.

To have a true perspective on the performance for containerized environments, container users have to monitor both the host and the containers. There are multiple options for monitoring. Among the popular ways to do that is to monitor each container via the Docker API, or by installing an agent inside each container for detailed visibility inside each container.

The Docker client, for example, already provides a command line tool to inspect containers resource consumption. The command *docker stats*, for example, can be used to get the stats about the running containers at runtime. This will present the CPU utilization for each container, the memory used and total memory available to the container.

Linux Containers rely on cgroups (control groups) which is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. Cgourps do not only track groups of processes, but also expose metrics about CPU, memory, and block I/O usage.

In industry, many commercial solutions are proposed to offer facilities to container users to efficiently monitor their workload inside containers. Most of these solutions rely on automatic cgroups metrics extraction and provide a visual representation of the data shown by the docker stats command. For example, the Datadog[3] and cAdvisor[4] agent uses the native cgroup accounting metrics to gather CPU, memory, network and I/O metrics of the containers. CAdvisor allows to monitor containers that run in the same host machine. As an alternative, Scout[5] is used to aggregate metrics from many hosts and containers in a distributed architecture. It also presents the data over longer time-scales and can create alerts based on those metrics.

Most of these tools provide web-based dashboards to visualize resource consumption at runtime as well as alerting mechanism to that can be triggered if metrics go above or below a configured threshold. Other examples of monitoring docker tools: Sensu Monitoring Framework, Prometheus, Sysdig Cloud, etc. Tools like cAdvisor and Prometheus allow to monitor containers running on one single host machine.

[3] www.datadoghq.com
[4] https://github.com/google/cadvisor
[5] https://scoutapp.com

There are many applications to manage the execution of containers across multiple hosts. For example, Kubernetes[6] is an open source orchestration system for Docker containers. It allows to quickly and efficiently respond to customer demand by deploying applications using multiple hosts and containers on the cloud, scale applications on the fly and optimize the resource usage across multiple hosts. This clustering framework is shipped with a monitoring tool called Hipster[7] that provides a base monitoring platform on Kubernetes. Heapster collects and interprets various signals like resource usage, lifecycle events, etc, and exports cluster metrics via REST endpoints. It supports a pluggable storage backend such as InfluxDB with Grafana and Google Cloud Monitoring.

The Docker monitoring infrastructure has been also used in the academic filed. For instance, [KT15] Kookarinrat et al. have investigated the problem of auto-sharding in NoSQL databases using a container-based infrastructure for runtime monitoring. The auto-sharding technique is used to divide data in the database and distribute it over multiple machines in order to scale it horizontally. The motivation behind this work is that selecting a right key is challenging. It could lead to either an improvement of the performance and capability of a database or to performance issues (i.e., by selecting a wrong key) which could lead to a system halt. For instance, a good shard key should have high degree randomness for write scaling and should contain high locality for range-query reading. Therefore, they analyzed and evaluated such suggested properties by studying how the variation of a shard keys choices could impact the DB performance.

They simulated an environment using Docker containers and measured the read/write performance of variety of keys. Inside each container, they executed write/read queries into the MongoDB database and used docker stats to retrieve automatically information about the memory and CPU usage of inside each container.

They found that a shard key with randomness and good locality could give a decent performance on write and read. However, in case that a shard key has nearly sorted values, combining a small range of random values to it might give acceptable performance for both read and write.

Container monitoring tools discussed above have been used in several other works like in [PHP16, MRA+16].

3.4 Summary & limitations of the state-of-th-art

[6] https://kubernetes.io

[7] https://github.com/kubernetes/heapster

Part II

Contributions

To the reader: summary of contributions

In the rest of this thesis, we present our approaches that contribute to achieve our goal of code generators testing and compilers auto-tuning. Figure 3.2 depicts an overview of how the different contributions we propose are connected to each other and how they contribute to achieve the common goal.‘

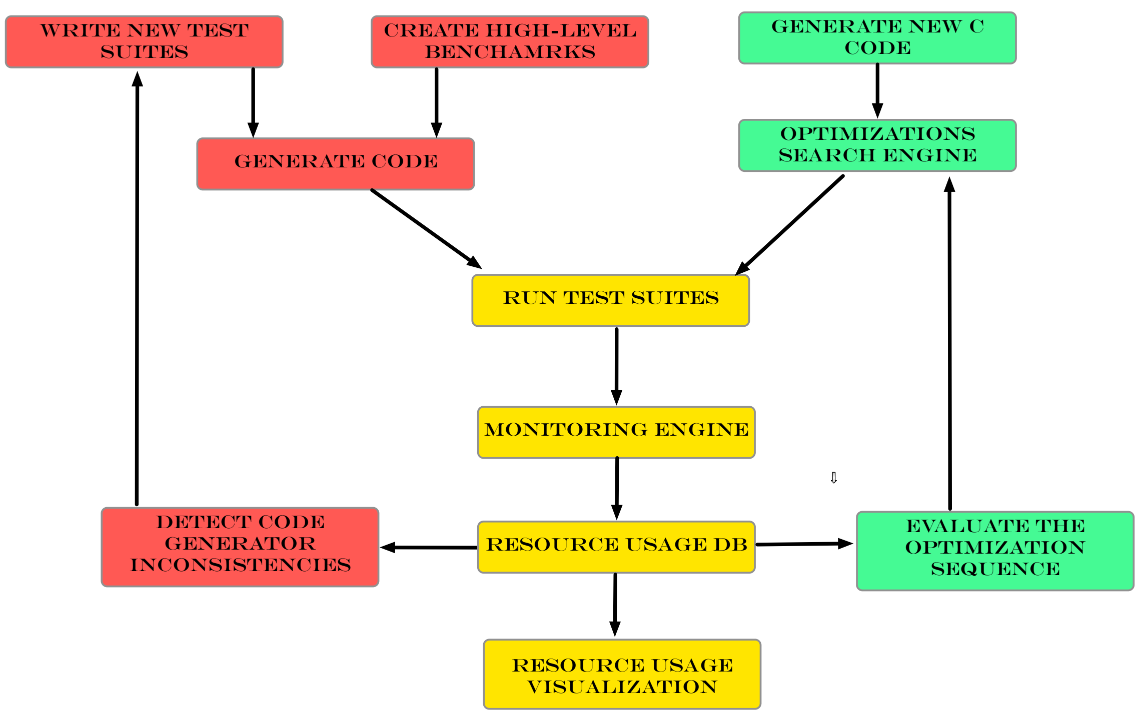

Figure 3.2: Summary of contributions

This thesis makes three contributions:

- **Code generators testing (in red):**

  In this contribution (chapter 4), we propose an approach for testing code generators. To do so, we create high-level benchmarks and test suites using Haxe. Afterwards, we generate automatically source code to five different target languages (Java, C++, C#, PHP and JS). Code execution and runtime monitoring is ensured by the third contribution relative to the code execution and monitoring in a sandbox environment. In this contribution, we rather focus on presenting an automatic way for detecting inconsistencies within code generator families.

- **Compilers auto-testing (in green):**

  As discussed in the state of the art, compilers auto-testing is part of the iterative compilation research field. Thus, we present in chapter 5, an adaptation of the novelty search algorithm for compilers auto-tuning. Our contribution focuses on tuning GCC compilers based on randomly generated C programs. This approach shares the same monitoring infrastructure as the previous contribution in order to evaluate the impact of discovered optimization sequences on resource usage. The outcome of this approach is the best set of optimizations sequences for a giving hardware architecture, a given input program and for a specific resource usage metric.

- **Resource usage monitoring infrastructure (in yellow):**

  We propose in chapter 6, an infrastructure based on micro-services namely Docker in order to automate the software deployment, execution and monitoring. It provides for the first contribution information about the resource usage of generated programs. For the second contribution, it provides as well information about the quality of optimized code in terms of memory and CPU usage. Finally, we provide also in this contribution a mechanism to visualize at runtime the resource usage of running programs.

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

Chapter 4

NOTICE: A tool for auto-tuning compilers

As discussed in the state of the art chapter, auto-tuning compilers is challenging due to the presence of different factors that make this task very complex such as the huge number of optimizations provided by modern compilers, the understanding of the underlying computer architecture and the target application to optimize, the conflicting objectives, etc.

Therefore, we presented and discussed many approaches that have been presented to tackle these problems and help compiler users to efficiently generate code with respect to many non-functional properties such as code size, energy consumption, execution time, etc.

This chapter presents an alternative approach to previous research efforts. We present a tool called NOTICE for automatic compiler auto-tuning. NOTICE is a component-based framework for automatically tuning C compilers through the monitoring of generated code in a controlled sand-boxing environment. It relies on a container-based black-box infrastructure to extract the resource consumption metrics of optimized code. It employs, as well, a set of meta-heuristics (e.g. mono-objective and multi-objective algorithms) to efficiently explore the huge search space of optimizations according to one or many non-functional metrics.

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.

This chapter is organized as follows:

Section 4.1 introduces the context of this work, i.e., auto-tuning compilers, and gives a preview of our main contributions in this field.

Section 4.2 describes the motivation and the challenges behind this work. We present in this section the GCC compiler as a motivation example to better explain the problem. The GCC compiler will also be used by NOTICE to evaluate and validate our approach.

In Section 4.3, the proposed search-based technique, i.e., Novelty Search (NS), is presented. We describe our NS adaptation to the compiler auto-tuning problem. Thus, we present in details our algorithm, the evaluation metrics and the iterative evolutionary process.

In Section 4.4, we conduct an empirical study to evaluate our approach. Thus, we evaluate the implementation of our approach by explaining the design of our experiments, the research questions we set out to answer and the methods we used to answer these questions.

Finally, discussions and concluding remarks are provided in Sections 4.5.

4.1 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) [CHH+16, HE08, LAS14].

In terms of non-functional requirements, improvement of the source code applications can refer to several different non-functional properties of the produced code such as code size, resource or energy consumption, execution time, among others [ACG+04, PE06].

Evaluating the non-functional properties of generated code is 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 [Mol09]. 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) [BSH15,CFH+12,DAH11] 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) [NF13]. Therefore, it is important to construct optimization levels that represent multiple trade-offs between non-functional properties, enabling the software designer to choose among different optimal solutions which best suit the system specifications.

In this chapter, we propose NOTICE (as NOn-functional TestIng of CompilErs), a component-based framework for auto-tuning C compilers. Our approach is based on micro-services to automate the deployment and monitoring of different variants of optimized code. NOTICE is an on-demand tool that employs mono and multi-objective evolutionary search algorithms to construct optimization sequences that satisfy user key objectives (execution time, code size, compilation time, CPU or memory usage, etc.). In this chapter, we make the following contributions:

- We introduce a novel formulation, compared to previous related work, of the compiler optimization problem using Novelty Search [LS08]. NS is applied to tackle the problem of optimizations diversity and then, providing a new way to explore the huge optimization search space.

- We also demonstrate that NOTICE can be used to automatically construct optimization levels that represent optimal trade-offs between multiple non-functional properties. In our approach, we study the relationship between the runtime execution of optimized code and the resource consumption profiles (CPU and memory usage) by providing a fine-grained understanding and analysis of compilers behavior regarding optimizations. Thus, we study the trade-offs execution time/memory usage, etc.

- We conduct an empirical study to evaluate the effectiveness of our approach by verifying the optimizations performed by the GCC compiler. Our experimental results show that NOTICE is able to auto-tune compilers according to user choices (heuristics, objectives, programs, etc.) and construct optimizations that yield to better performance results than standard optimization levels using mono-objective and multi-objective optimization.

4.2 Motivation

4.2.1 Compiler Optimizations

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

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

To explore the large optimization space, users have to evaluate the effect of optimizations according to a specific performance objective (see Figure 4.1). 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 [HE08, MND+14] 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 [YCER11].

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

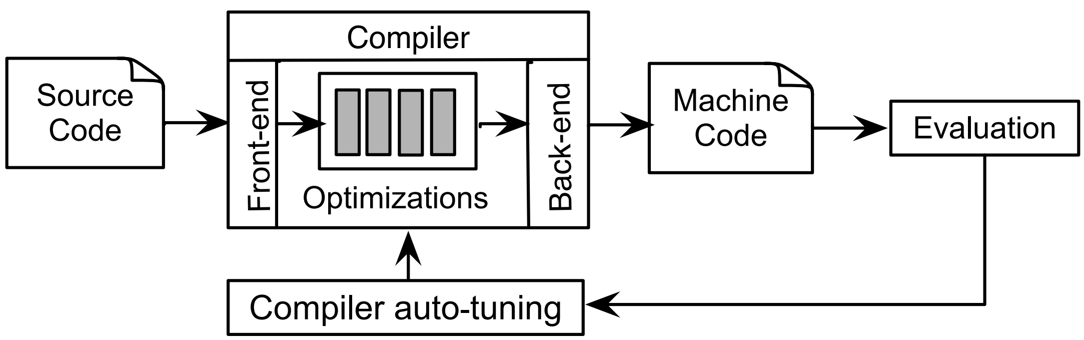

Figure 4.1: Process of compiler optimization exploration

4.2.2 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.

Table 4.1: Compiler optimization options enabled by GCC standard levels

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

As it is shown in Table 4.1, we count 76 optimization flags that are enabled by the four default optimization levels (O1, O2, O3, Ofast).

Each standard level is composed by a number of optimizations. These levels are defined by compiler designers based on their experiences and preliminary experiments. The goal of defining these standard levels is to build general and program independent sequences that represent trade-offs among several non-functional properties.

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

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.

4.3 Evolutionary Exploration of Compiler Optimizations

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

4.3.1 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 [LS08] to tackle this issue. The idea of this technique is to explore the search space of possible compiler flag options by considering sequence diversity as a single objective. Instead of having a fitness-based selection that maximizes one of the non-functional objectives, we select optimization sequences based on a novelty score showing how different they are compared to all other combinations evaluated so far.

NS is a divergent evolutionary algorithm which rewards optimization sequences that diverge from previously discovered ones. Thus, evolution can be viewed as a divergent process compared to the traditional convergent approaches that exert the selection pressure based on fitness values.

Moreover, we claim that the search towards effective optimization sequences is not straightforward since the interactions between optimizations is too complex and difficult to define.

For instance, in a previous work [CFH+12], Chen et al. showed that handful optimizations may lead to higher performance than other techniques of iterative optimization. In fact, the fitness-based search may be trapped into some local optima that cannot escape [BKK+98]. This phenomenon is known as *"diversity loss"*. For example, if the most effective optimization sequence that induces less execution time lies far from the search space defined by the gradient of the fitness function, then some promising search areas may not be reached. The issue of premature convergence to local optima has been a common problem in evolutionary algorithms. Many methods are proposed to overcome this problem [BFN96]. However, all these efforts use a fitness-based selection to guide the search. Considering diversity as the unique objective function to be optimized may be a key solution to this problem.

Therefore, during the evolutionary process, we select optimization sequences that remain in sparse regions of the search space in order to guide the search towards novelty. In the meantime, we choose to gather the non-functional metrics relative to the resource consumption (memory and CPU usage) of optimized code. We describe in more details the way we are collecting these non-functional metrics in section 4.4.

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

Algorithm 1: Novelty search algorithm for compiler optimization exploration

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

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.

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 4.2). 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.

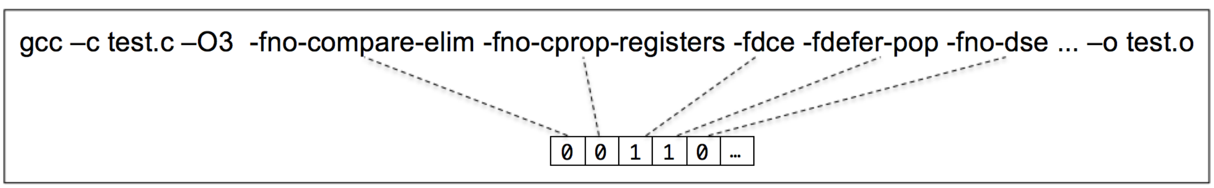

Figure 4.2: Solution representation

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 :

$$distance(S1, S2) = |S1 \bigtriangleup S2| \tag{4.1}$$

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:

$$NM(S) = \frac{1}{k}\sum_{i=1}^{k} distance(S, \mu_i) \tag{4.2}$$

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

4.3.2 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 [LPF+10, DPAM02]. Then, this Pareto non-dominated set is returned as a result. There is typically more than one optimal solution at the end of NS. The maximum size of the final Pareto set cannot exceed the size of the initial population.

4.4 Evaluation

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

4.4.1 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.

4.4.2 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 [YCER11]. Csmith is a tool that can generate random C programs that statically and dynamically conform to the C99 standard. It has been widely used to perform functional testing of compilers [CHH+16, LAS14, NHI13] but not the case for checking non-functional requirements. Csmith can

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

generate C programs that use a much wider range of C features including complex control flow and data structures such as pointers, arrays, and structs.

Csmith programs come with their test suites that explore the structure of generated programs (i.e., high quality code coverage). Yang et al. [YCER11] argue that Csmith is an effective bug-finding tool because it generates tests that explore atypical combinations of C language features. They also argue that larger programs are more effective for functional testing.

Thus, we run Csmith for 24 hours and gathered the largest generated programs. We depicted 111 C programs with an average number of source lines of 12K. 10 programs are used as training set for RQ1, 100 other programs to answer RQ2 and one last program to run RQ4 experiment.

The selected Csmith programs are described in more details at [mbo].

Moreover, we run experiments on commonly used benchmarks named Collective Benchmark (cBench) [Fur09]. It is a collection of open-source sequential programs in C targeting specific areas of the embedded market. It comes with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization. cBench contains more than 20 C programs. The following table describes programs that we have selected from this benchmark to evaluate our approach.

These real world benchmark programs are used to study the influence of compiler optimizations on the resource usage in RQ3 experiments.

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

Table 4.2: Description of selected benchmark programs

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 [CST02]) and two multi-objective optimizations (NS and NSGA-II [DPAM02]). Each initial population/solution of different algorithms is completely random. The stopping criterion is when the maximum number of fitness evaluations is reached. The resulting parameter values are listed in Table 4.3. The same parameter settings are applied to all algorithms under comparison.

NS, which is our main concern in this work, is implemented as described in Section 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 [IJH+13]. The value that yielded the highest performance score was chosen.

Table 4.3: Algorithm parameters

Parameter	**Value**	**Parameter**	**Value**
Novelty nearest-k	15	Tournament size	2
Novelty threshold	30	Mutation prob.	0.1
Max archive size	500	Crossover	0.5
Population size	50	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 4.3 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 HTTP request 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.).

More details about the container-based infrastructure and the technical choices are provided in chapter 5.

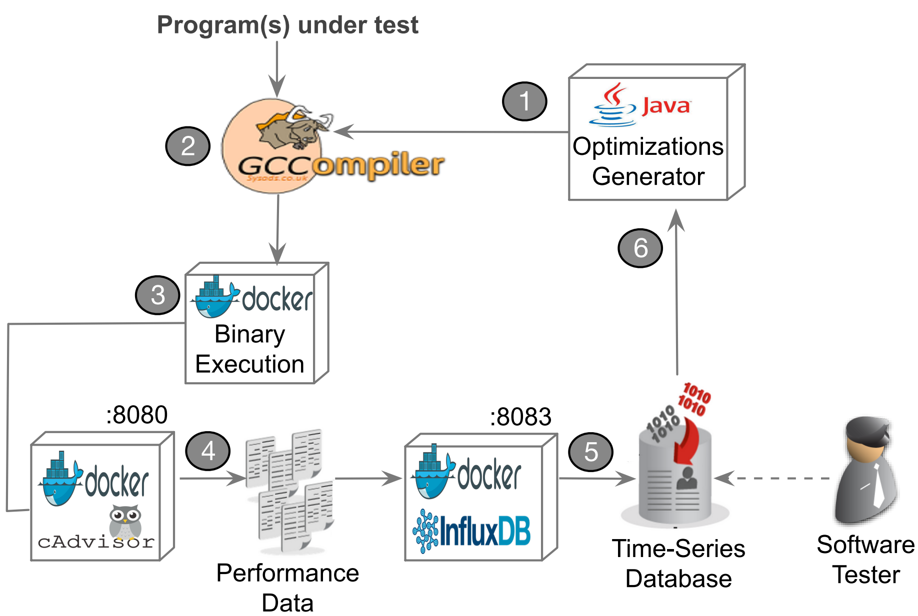

Figure 4.3: NOTICE experimental infrastructure

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.

4.4.3 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.

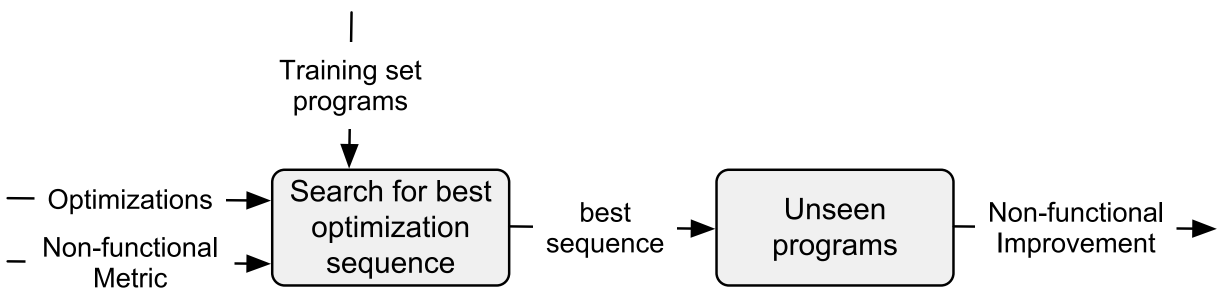

Figure 4.4: Evaluation strategy to answer RQ1 and RQ2

As it is shown on the left side of Figure 4.4, given a list of optimizations and a non-functional objective, we use NOTICE to search for the best optimization sequence across a set of input programs that we call *"the training set"*. This *"training set"* is composed of random Csmith programs (10 programs). We apply then generated sequences to these programs. Therefore, the code quality metric, in this setting, is equal to the average performance improvement (S, MR, or CR) and that, for all programs under test.

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

Results Table 4.4 reports the comparison results of three non-functional properties CR, MR, and S. At the first glance, we can clearly see that all search-based algorithms outper-

Table 4.4: Results of mono-objective optimizations

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

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

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

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

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

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

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 4.4).

Thus, we apply best discovered optimizations in RQ1 to new unseen Csmith (100 new random programs) and we compare then, the performance improvement across these programs (see right side of Figure 4.4). We also apply standard optimizations, O2 and O3, to new Csmith programs in order to compare the performance results. The idea of this experiment is to test whether new generated Csmith programs are sensitive to previously discovered optimizations or not.

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

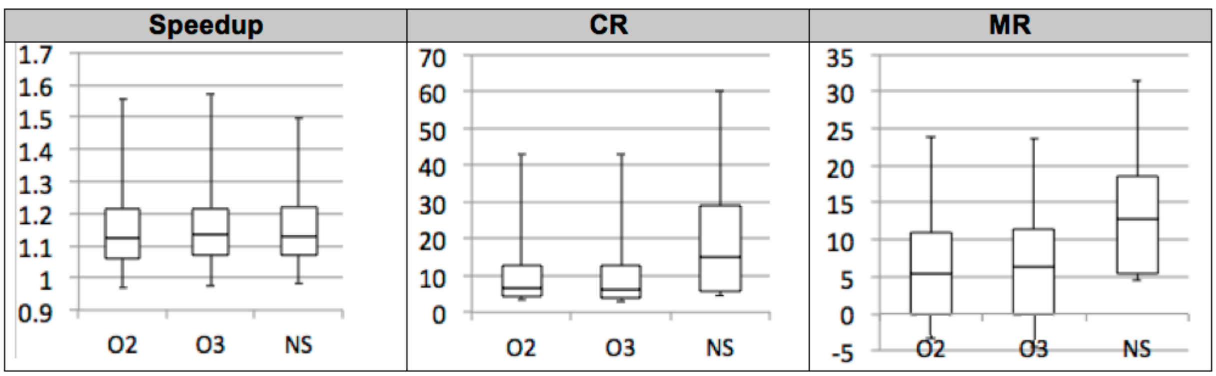

Figure 4.5: Boxplots of the obtained performance results across 100 unseen Csmith programs, for each non-functional property: Speedup (S), memory (MR) and CPU (CR) and for each optimization strategy: O2, O3 and NS

Results Figure 4.5 shows the distribution of memory, CPU and speedup improvement across 100 new Csmith programs. For each non-functional property, we apply O2, O3 and best NS sequences. Speedup results show that the three optimization strategies lead to almost the same distribution with a median value of 1.12 for speedup.

This can be explained by the fact that NS might need more time to find the sequence that best optimizes the execution speed. Meanwhile, O2 and O3 have also the same impact on CR and MR which is almost the same for both levels (CR median value is 8% and around 5% for MR).

However, the impact of applying best generated sequences using NS clearly outperforms O2 and O3 with almost 10% of CPU improvement and 7% of memory improvement.

This proves that NS sequences are efficient and can be used to optimize resource consumption of new Csmith programs. This result also proves that Csmith code generator applies the same rules and structures to generate C code. For this reason, applied optimization sequences always have a positive impact on the non-functional properties.

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

RQ3. Impact of optimizations on resource usage

In this experiment, we evaluate the impact of applying the standard optimization levels and the new discovered sequences on the resource usage. We also study the correlation between speedup and resource consumption of generated code.

The idea of this experiment is to: (1) prove, or not, the usefulness of involving resource usage metrics as key objectives for performance improvement; (2) the need, or not, of multi-objective search strategy to handle the different non-functional requirements such as resource usage and performance properties.

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

Method 1 In this experiment, we use NOTICE to provide an understanding of optimizations impact, in terms of resource consumption, when trying to optimize for execution time.

Thus, we choose one instance of obtained results in RQ1 related to the best speedup improvement (i.e., results obtained in line 1 of Table 4.4) and we study the impact of speedup improvement on memory and CPU consumption. We also compare the resource usage data to standard GCC levels as they were presented in Table 4.4. Improvements are always calculated over the non-optimized version (O0). The following measurements are based on the training set of 10 Csmith programs.

Results 1 Figure 4.6 shows the impact of speedup optimization on resource consumption. For instance, O2 and O3 that led to the best speedup improvement among standard optimization levels in RQ1, present opposite impact on resource usage. Applying O2 induces

-8.4% of MR and -5% of CR. However, applying O3 improves MR and CR respectively by 3.4% and 4.2%. Hence, we note that when applying standard levels, there is no clear correlation between speedup and resource usage since compiler optimizations are generally used to optimize the execution speed and never evaluated to reduce system resources.

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

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

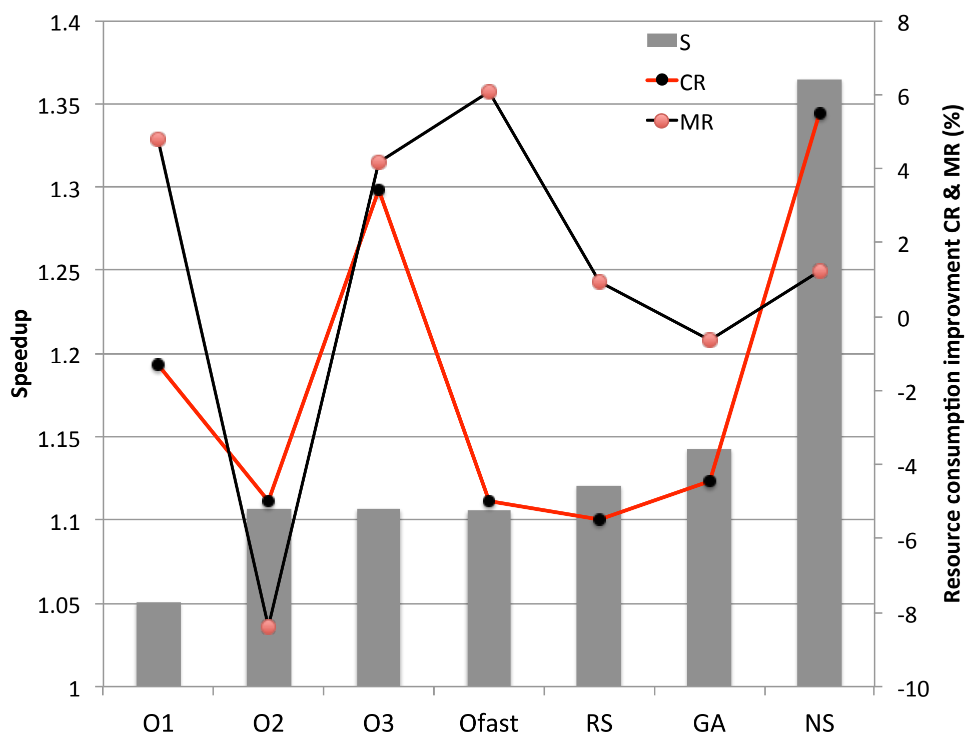

Figure 4.6: Impact of speedup improvement on memory and CPU consumption for each optimization strategy

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

Figure 4.7 shows this comparison across different benchmark programs. It presents the percentage of saved memory of standard and novelty optimizations over O0 level (no optimization applied).

To study the correlation between execution time and memory consumption of running programs, we present in Figure 4.8 an evaluation of the speedup according standard levels. Again, we compare these results to the sequence that had the best memory reduction in Figure 4.7 (i.e., NS solution).

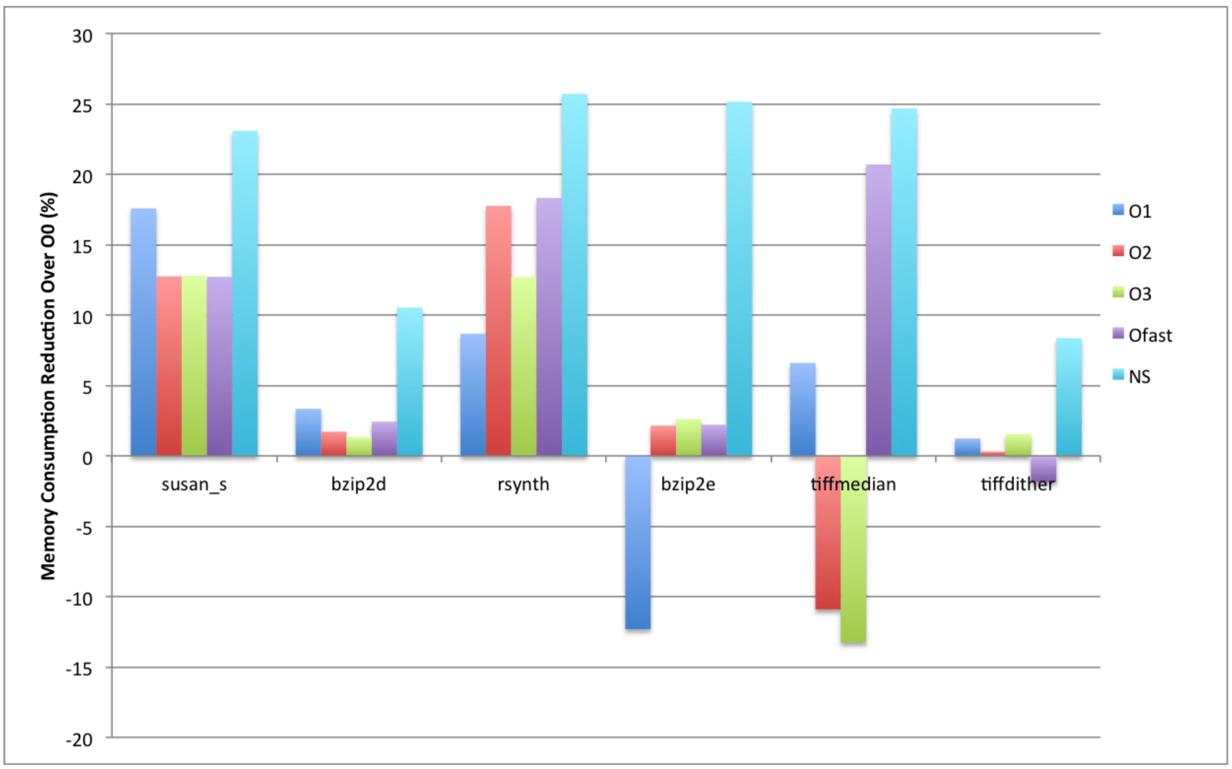

Figure 4.7: Evaluating the amount of saved memory after applying standard optimization options compared to best generated optimization using NS

Results 2 As expected, the results show that NS clearly outperforms standard optimizations for all benchmark programs.

Using NS, we are able to reach a maximum memory consumption reduction of almost 26% for the case rsynth program against a maximum of 18% reduction using Ofast option.

We remark as well that the impact of applying standard optimizations on memory consumption for each program differs from one program to another.

Using O1 for bzip2e and O2, O3 for tiffmedian can even increase the memory consumption by almost 13 %.

This agrees to the idea that standard optimizations does not produce always the same impact results on resource consumption and may be highly dependent on the benchmark and the source code they have been tested on.

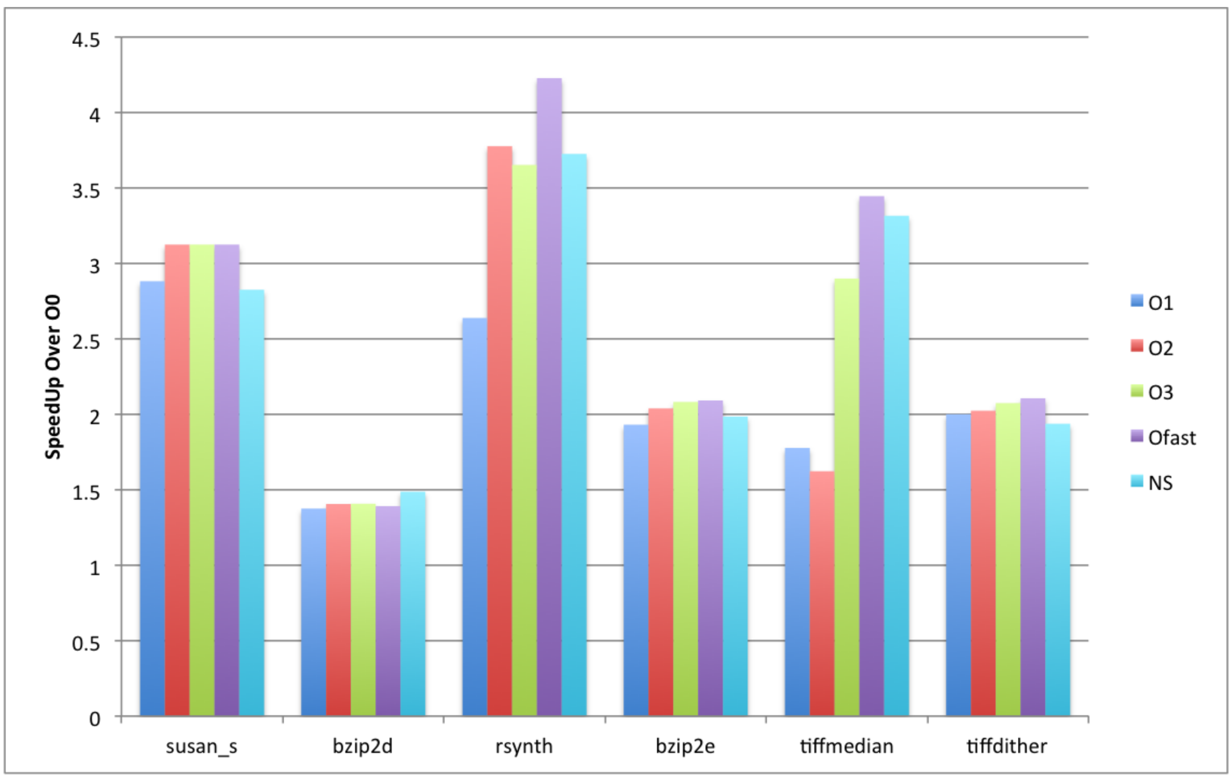

Figure 4.8: Evaluating the speedup after applying standard optimization options compared to best generated optimization using NS

In Figure 4.8, we can see that optimizations yield to high level of speedup for all benchmark programs (between 1.5 and 4.3).

We can also observe that the different optimization levels do not differ too much in term of execution time.

We distinguish that Ofast is more efficient for all programs and NS sequence has almost the same speedup as Ofast.

NS solutions have equal or less performance improvement compared to all standard levels for all benchmarks.

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

RQ4. Trade-offs between non-functional properties

Method Finally, to answer RQ4, we use NOTICE again to find trade-offs between non-functional properties.

In this experiment, we choose to focus on the trade-off <*ExecutionTime–MemoryUsage*>. In addition to our NS adaptation for multi-objective optimization, we implement a commonly used multi-objective approach namely NSGA-II [DPAM02].

We denote our NS adaptation by *NS-II*. We recall that NS-II is not a multi-objective approach as NSGA-II. It uses the same NS algorithm. However, in this experiment, it returns the optimal Pareto front solutions instead of returning one optimal solution relative to one goal.

Apart from that, we apply different optimization strategies to assess our approach. First, we apply standard GCC levels. Second, we apply best generated sequences relative to memory and speedup optimization (the same sequences that we have used in RQ2). Thus, we denote by *NS-MR* the sequence that yields to the best memory improvement MR and *NS-S* to the sequence that leads to the best speedup. This is useful to compare mono-objective solutions to new generated ones.

In this experiment, we assess the efficiency of generated sequences using only one Csmith program.

We evaluate the quality of the obtained Pareto optimal optimization based on raw data values of memory and execution time. Then, we compare qualitatively the results by visual inspection of the Pareto frontiers.

The goal of this experiment is to check whether it exists, or not, a sequence that can reduce both execution time and memory usage.

We report the comparison results of our NS adaptation for optimizations generation to the current state-of-the-art multi-objective approaches namely NSGA-II.

Results Figure 4.9 shows the Pareto optimal solutions that achieved the best performance assessment for the trade-off <*ExecutionTime–MemoryUsage*>. The horizontal axis indicates the memory usage in raw data (in Bytes) as it is collected using NOTICE. In similar fashion, the vertical axis shows the execution time in seconds. Furthermore, the figure shows the impact of applying standard GCC options and best NS sequences on memory and execution time.

Based on these results, we can see that NSGA-II performs better than NS-II. In fact, NSGA-II yields to the best set of solutions that presents the optimal trade-off between the two objectives. Then, it is up to the compiler user to use one solution from this Pareto front that satisfies his non-functional requirements (six solutions for NSGA-II and five for NS-II).

For example, he could choose one solution that maximizes the execution speed in favor of memory reduction.

On the other side, NS-II is capable to generate only one non-dominated solution. For NS-MR, it reduces as expected the memory consumption compared to other optimization levels. The same effect is observed on the execution time when applying the best speedup sequence NS-S. We also note that all standard GCC levels are dominated by our different heuristics NS-II, NSGA-II, NS-S and NS-MR.

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

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.

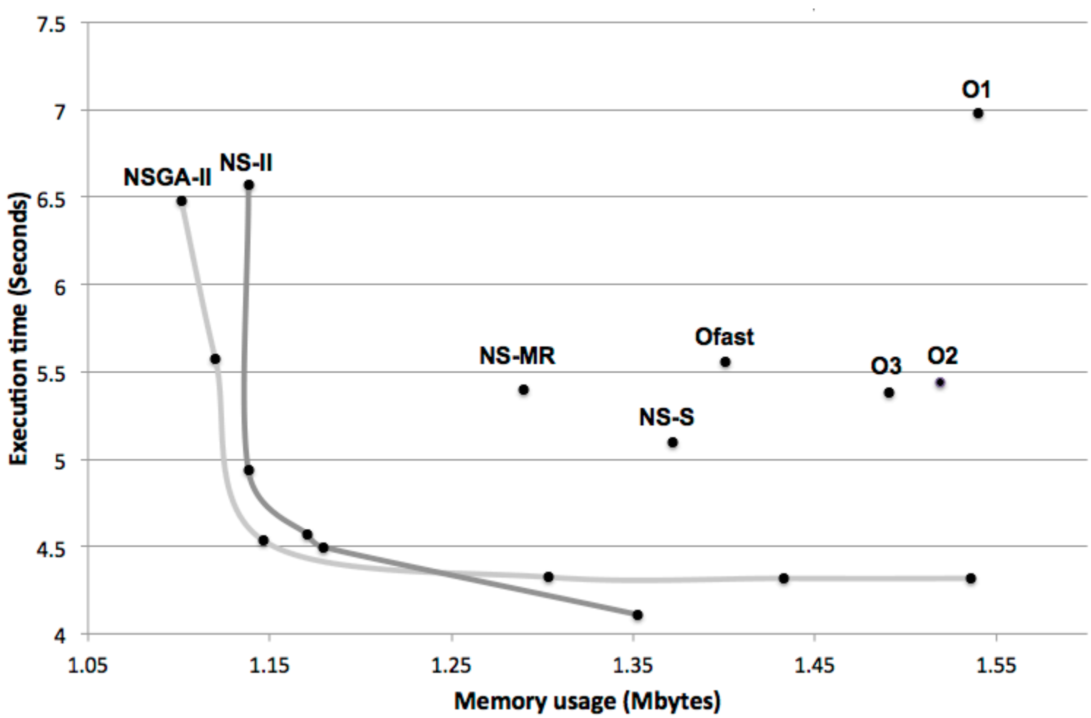

Figure 4.9: Comparison results of obtained Pareto fronts using NSGA-II and NS-II

4.4.4 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 can use NOTICE to evaluate the impact of optimizations on the non-functional properties and build their specific sequences by trying to find trade-offs among these non-functional properties (RQ4).

We would notice that for RQ1, experiments take about 21 days to run all algorithms. This run time might seem long but, it should be noted that this search can be conducted only once, since in RQ2 we showed that best gathered optimizations can be used with unseen programs of the same category as the training set, used to generate optimizations. This has to be proved with other case studies.

Multi-objective search as conducted in RQ4, takes about 48 hours, which we believe is acceptable for practical use. Nevertheless, speeding up the search speed may be an interesting feature for future research.

4.4.5 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 [CHH+16,YCER11]. Csmith programs have been used only for functional testing, but not for non-functional testing. However, we cannot assert that the best discovered set of optimizations can be generalized to industrial applications since optimizations are highly dependent on input programs and on the target architecture. In fact, experiments conducted on RQ1 and RQ2 should be replicated to other case studies to confirm our findings; and build general optimization sequences from other representative training set programs chosen by compiler users.

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

4.4.6 Tool Support Overview

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

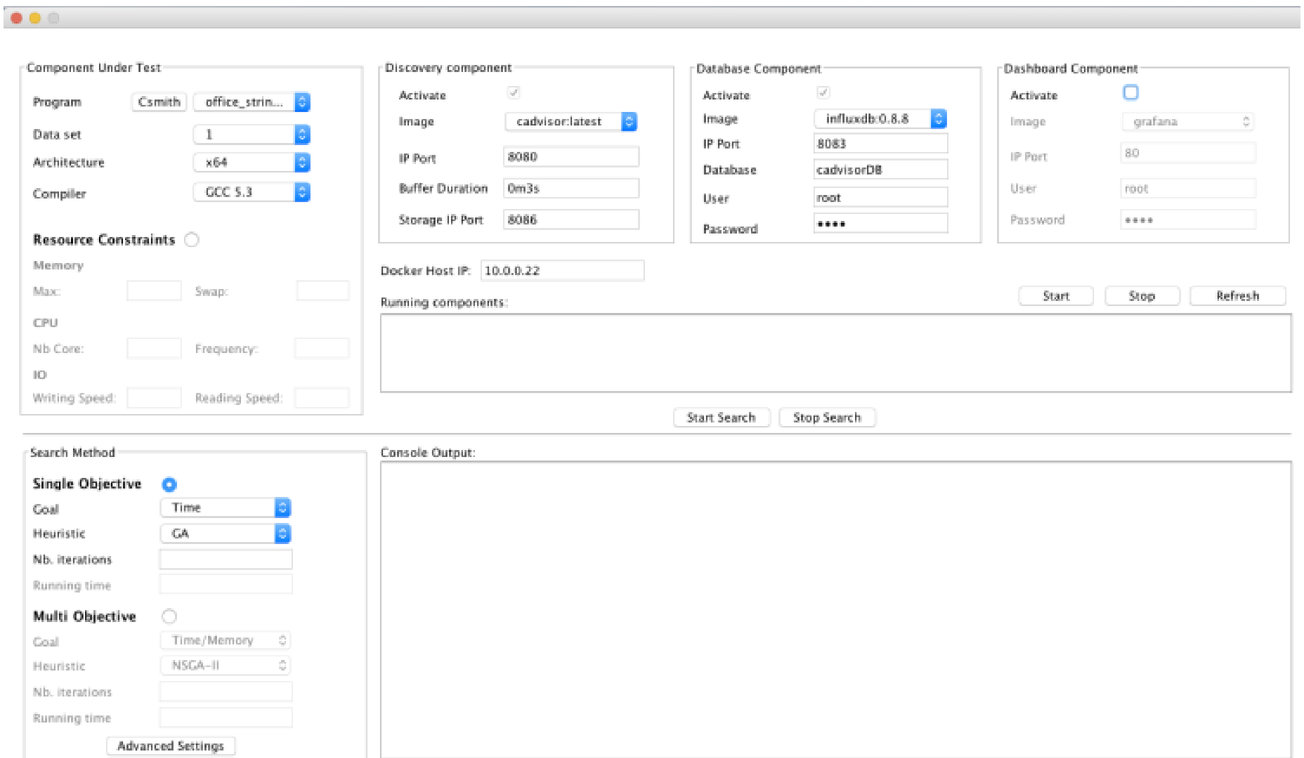

Figure 4.10: Snapshot of NOTICE GUI interface

For instance, NOTICE provides different features to help compiler users to:

- **Select the input program under test:** by generating a new Csmith program or by selecting an existing C program such as Cbench benchmark programs. The generation of a new Csmith program is done randomly.
- **Select datasets:** In case the selected program requires a dataset such as the case for Cbench programs, NOTICE allows the user to choose the dataset for the selected program. We recall that Cbench comes with a set of 20 datasets for each benchamrk program.

- **Select the target computer architecture:** choose the processor architecture where the experiments will be running such as x64, x86, ARM. This is part of our future work since we are running experiments only on native GCC compiler with x64 architecture. We are preparing a QEMU docker image to handle platforms heterogeneity.
- **Define the compiler version:** For now, NOTICE supports all GCC compiler versions from 3.x to 5.x. The process of extracting the target optimizations to evolve is done automatically (i.e., optimizations enabled by O1, O2, O3 and Ofast)
- **Configure the monitoring components:** This refers to the containers needed to extract all the information about the resource consumption. Configuring these components is possible with NOTICE such as image versions, labels, ports, logins, passwords.
- **Choose ip address of the cloud host machine:** NOTICE allows to run experiments remotely thanks to its micro-service infrastructure. Thus, we enable the user to select the ip of the remote machine.
- **Define resource constraints to running container:** In case we would run optimizations under resource constraints, it is possible to define memory and CPU constraints. By default, these option are disabled.
- **Choose the search method:** The user can select either a mono objective or multi-objective search.
- **Choose the meta-heuristic algorithm:** NOTICE supports GA, RS, and NS for mono objective search and NS, RS, and NSGA-II for multi-objective optimization.
- **Choose the number of iterations:** The user can define the number of iterations for each algorithm which corresponds to the number of generated optimization sequences.
- **Choose the search time:** Instead of limiting the number of iterations, the user can fix a limit search time (in hours).
- **Choose the optimization objective:** The goal can be reducing the execution time, memory, CPU, code size, or compilation time. For multi objective search, users can choose trade-offs between these objectives.
- **Edit evolutionary algorithm settings:** Tuning the evolutionary parameters (showed in Table 4.3) such as the population size, the novelty search settings, mutation and crossover probabilities, etc.

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

4.5 Conclusion

Modern compilers come with huge number of optimizations, making complicated for compiler users to find best optimization sequences. Furthermore, auto-tuning compilers to meet user requirements is a difficult task since optimizations may depend on different properties (e.g., platform architecture, software programs, target compiler, optimization objective, etc.).

Hence, compiler users merely use standard optimization levels (O1, O2, O3 and Ofast) to enhance the code quality without taking too much care about the impact of optimizations on system resources.

In this chapter, we have introduced first a novel formulation of the compiler optimization problem based on Novelty Search. The idea of this approach is to drive the search for best optimizations toward novelty.

Our work presents the first attempt to introduce diversity in iterative compilation. Experiments have shown that Novelty Search can be easily applied to mono and multi-objective search problems.

In addition, we have reported the results of an empirical study of our approach compared to different state-of-the-art approaches, and the obtained results have provided evidence to support the claim that Novelty Search is able to generate effective optimizations.

Second, we have presented an automated tool for automatic extraction of non-functional properties of optimized code, called NOTICE. NOTICE applies different heuristics (including Novelty Search) and performs compiler auto-tuning through the monitoring of generated code in a controlled sand-boxing environment. In fact, NOTICE uses a set of micro-services to provide a fine-grained understanding of optimization effects on resource consumption.

We evaluated the effectiveness of our approach by verifying the optimizations performed by GCC compiler. Then, we studied the impact of optimizations on memory consumption and execution time.

Results showed that our approach is able to automatically extract information about memory and CPU consumption. We were also able to find better optimization sequences than standard GCC optimization levels and construct optimizations that present optimal trade-offs between speedup and memory usage.

Chapter 5

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 and technological stacks. The use of domain-specific languages 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 chapter describes a black-box testing approach that automatically detect anomalies in code generators in terms of non-functional properties (i.e., resource usage and performance).

In fact, we adapt the idea of metamorphic testing to the problem of code generators testing. Hence, our approach relies on the definition of high level test oracles (i.e., metamorphic relations) to check the potential inefficient code generator among a family of code generators.

We evaluate our approach by analyzing the performance of Haxe, a popular high-level programming language that involves a set of cross-platform code generators. Experimental results show that our approach is able to detect some performance inconsistencies that reveal real issues in Haxe code generators.

This chapter is organized as follows:

Section 5.1 introduces the context of this work, i.e., the non-functional testing of code generators.

Section 5.2 presents the motivation and background of this work. In particular, we discuss in this section three motivation examples and the problems we are addressing.

Section 5.3 describes the general approach overview and the testing strategy.

In Section 5.4, the evaluation and results of our experiments are discussed. Hence, we provide more details about the experimental settings, the code generators under test, the benchmark used, the evaluation metrics, etc. We discuss then the evaluation results.

Finally, we conclude in Section 5.5.

5.1 Introduction

Generative programming techniques 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.

The main benefit of using generative programming is to reduce the development and maintenance effort, allowing the development at a higher-level of abstraction through the use of Domain-Specific Languages (DSLs) [BCW12] for example.

DSLs, as opposed to general-purpose languages, are high level software languages that focus on specific problem domains. DSLs or models are generally coupled with the use of code generators that will automatically transform the manually designed models to software artifacts, which can be deployed on different target platforms.

However, code generators are known to be very difficult to implement and maintain since they involve a set of complex and heterogeneous technologies [FR07, GS15].

To preserve software reliability and quality, code generators have to respect different requirements. In fact, *non-mature* code generators can generate defective software artifacts which range from uncompilable or semantically dysfunctional code that causes serious damage to the generated software; 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.).

As a matter of fact, these defects (or also anomalies) should be detected and corrected as early as possible in order to ensure the correct behavior of delivered software.

To check the correctness of the code generation process, developers often define (at design or runtime level) a set of test cases that will verify the functional behavior of generated code.

After code generation, test suites are executed within each target platform, which may lead to either a correct behavior (*i.e.*, expected output) or incorrect one (*i.e.*, failures, errors).

However, it is possible to generate code where all tests pass but, having a very poor quality design. In this case, the quality of generated code can negatively influence on the non-functional requirements and cause performance issues [Hun11, RPFD14].

Testing the non-functional properties of code generators is a challenging and time-consuming task because developers need to deploy and run code every time a change is made in order to analyze and verify its non-functional behavior.

This task becomes more tedious when targeting different platforms and software languages. Thus, different platform-specific tools will be needed to track bugs and identify the cause of execution failures [GS14, DGR04].

Currently, there is a lack of automatic solutions that check the non-functional issues such as the properties related to the resource consumption of generated code (Memory or CPU consumption).

In this chapter, we are presenting the following contributions:

- We propose a fully automated black box testing approach for detecting code generator inconsistencies within code generator families. We use metamorphic relations as means of test oracles for our test suites. In this contribution, we focus on detecting anomalies related to performance and resource usage properties.

- We report the results of an empirical study by evaluating the non-functional properties of the Haxe code generators. Haxe is a popular high-level programming language[1] that involves a set of cross-platform code generators able to generate code to different target platforms. The obtained results provide evidence to support the claim that our proposed approach is able to detect code generator issues.

[1] 1442 GitHub stars

5.2 Motivation

5.2.1 Code Generator Families

This paper is based on the intuition that a code generator is often a member of a family of code generators.

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

The availability of multiple generators with comparable functionality enables to apply the idea of differential testing [McK98] to detect code generator issues. As motivating examples for this research, we can cite three approaches that intensively develop and use code generator families:

a. Haxe. Haxe[2] [Das11] 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 target platform. Moreover, Haxe comes with a set of code generators that translate manually-written code (in Haxe language) to different target languages and platforms. This project is popular (more than 1440 stars on GitHub).

b. ThingML. ThingML[3] is a modeling language for embedded and distributed systems [FMSB11]. The idea of ThingML is to develop a practical model-driven software engineering tool-chain which targets resource-constrained embedded systems such as low-power sensors and microcontroller-based devices. ThingML is developed as a domain-specific modeling language which includes concepts to describe both software components and communication protocols. The formalism used is a combination of architecture models, state machines and an imperative action language. The ThingML tool-set provides a code generator families to translate ThingML to C, Java and JavaScript. It includes

[2] http://haxe.org/

[3] http://thingml.org/

a set of variants for the C and JavaScript code generators to target different embedded systems and their constraints. This project is still confidential, but it is a good candidate to represent the modeling community practices.

c. TypeScript. TypeScript[4]is a typed superset of JavaScript that compiles to plain JavaScript [RSF+15]. In fact, it does not compile to only one version of JavaScript. It can transform TypeScript to EcmaScript 3, 5 or 6. It can generate JavaScript that uses different system modules ('none', 'commonjs', 'amd', 'system', 'umd', 'es6', or 'es2015')[5]. This project is popular (more than 12 619 stars on GitHub).

5.2.2 Functional Correctness of a Code Generator Family

A reliable and acceptable way to increase the confidence in the correctness of a code generator family is to validate and check the functionality of generated code, which is a common practice for compiler validation and testing [JS14, SCDP07, SWC05]. 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. Functionally testing a code generator family in this case would be simple. Since the generated programs have the same input program, the oracle can be defined as the comparison between the functional outputs of these programs which should be the same.

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.

5.2.3 Performance Evaluation of a Code Generator Family

Another important aspect of code generators testing is to test the non-functional properties of produced code. Code generators have to respect different requirements to preserve software reliability and quality [DAH11]. In this case, ensuring the code quality of generated code can require examining several non-functional properties such as code size, resource or

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

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

energy consumption, execution time, etc [PE06]. 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, high execution time, etc.). 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 tests to check the consistency of code generators regarding the memory or CPU usage properties. Haxe provides two test cases[6] to benchmark the resulting generated code. One serves to benchmark an example in which object allocations are deliberately (over) used to measure how memory access/GC mixes with numeric processing in different target languages. The second test evaluates the network speed across different target platforms.

5.2.4 Performance Issues of a Code Generator Family

The potential issues that can reveal an inefficient code generator can be resumed as following:

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

The main difficulties with testing the non-functional properties of code generators is that we cannot just observe the execution of produced code, but we have to observe and compare the execution of generated programs with equivalent implementations (i.e., in other languages). Even if there is no explicit oracle to detect inconsistencies, we could benefit from the family of code generators to compare the behavior of several generated programs and detect singular resource consumption profiles that could reveal a code generator bug [Hun11]. Our approach is a black-box testing technique and it does not provide detailed information about the source of the issues, as described above. Nevertheless, we

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

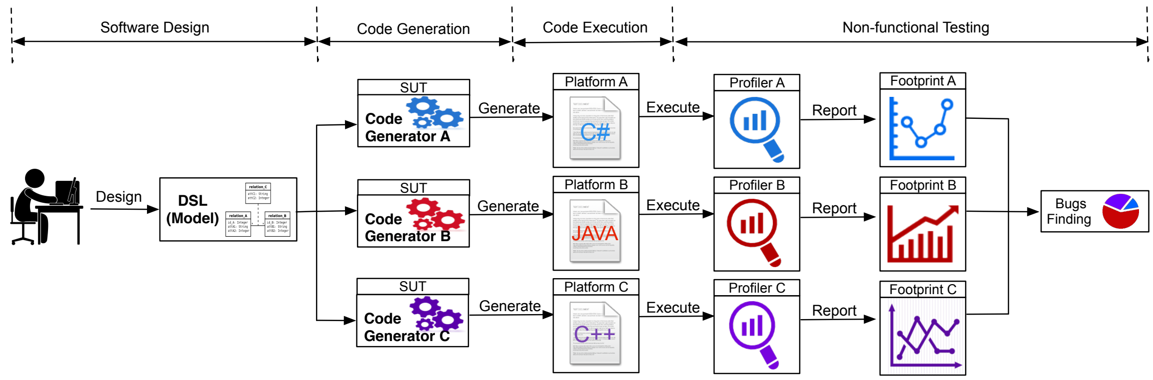

Figure 5.1: An overall overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to runtime: the classical way

rather provide a mechanism to detect these potential issues within a set of code generator families so that, these issues may be investigated and fixed afterwards by code generators/-software maintainers.

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

5.3 Approach Overview

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

Figure 1 summarizes the classical steps that ensure the code generation and non-functional testing of produced code from design time to runtime. 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, the 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 automatically generate code for different languages and platforms. We depict, as an example in Figure 1, three code generators from the same family capable to generate code to three software programming languages (JAVA, C# and C++). The first step is to generate code from the previously designed model. Afterwards, generated software artifacts (e.g., JAVA, C#, C++, etc.) are compiled, deployed and executed across different target platforms (e.g., Android, ARM/Linux, JVM, x86/Linux, etc.). Finally, to perform the non-functional testing of generated code, developers have to collect, visualize and compare information about the performance and efficiency of running code across the different platforms. Therefore, they generally use several platform-specific profilers, trackers, instrumenting and monitoring tools in order to find some inconsistencies or bugs during code execution [GS14, DGR04]. Finding inconsistencies within code generators involves analyzing and inspecting the code and that, for each execution platform. For example, one way to handle that, is to analyze the memory footprint of software execution and find memory leaks [NS07]. Developers can then inspect the generated code and find some fragments of the code-base that have triggered this issue. 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. Compared to this classical testing approach, our proposed work seeks to automate the last three steps: generate code, execute it on top of different platforms, and find code generator issues.

5.3.2 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 provide facilities to the code generator developers 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 ensure these four main steps, we rely on system containers [SPF+07]. It is an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a control host using a single Linux kernel. The Linux kernel provides the cgroups functionality that allows limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) for each container without the need of starting any virtual machines [LYZ]. These containers share the same OS and hardware as the hosting machine and it is very useful to use them in order to create new configurable and isolated instances to run. With container-based virtualization, we reduce the overhead associated with having each guest running a new installed operating system like using virtual machines. This approach can also improve the performance because there is just one operating system taking care of hardware calls.

The rest of this section details the technical choices we have made to synthesize this testing infrastructure.

5.3.3 Technical Implementation

The overall overview of our proposed approach is shown in Figure 2. In the following subsections, we describe the deployment and testing architecture using system containers.

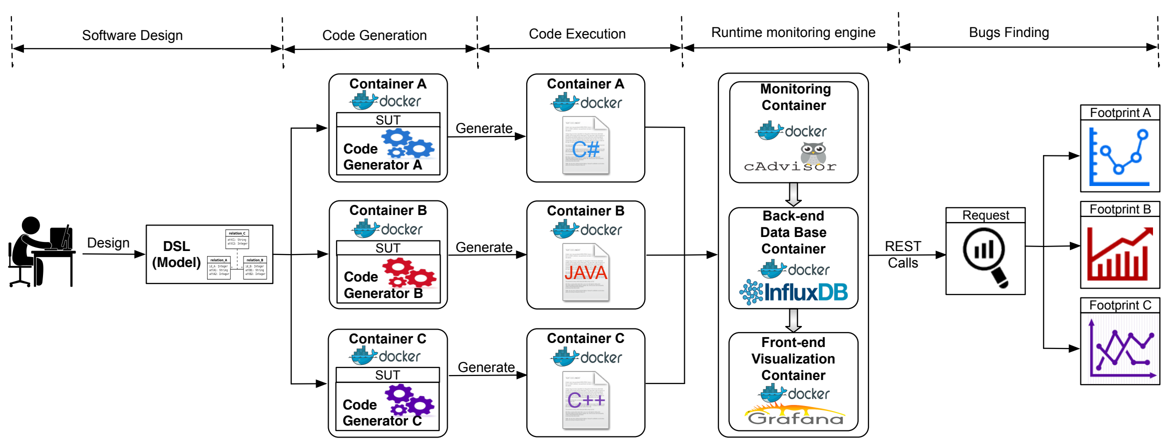

Figure 5.2: A technical overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to runtime.

System Containers as Execution Platforms

Before starting to monitor and test applications, we need to deploy the 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 [Mer14]. Docker[7] 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 preconfigured 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 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 produces code within multiple instances of the preconfigured Docker images (see code generation step in Figure 2). We use the public Docker registry[8] 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 new container instances 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 GB of memory for each container[9]. 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 compared to using a full stack virtualization solution [SCTF16]. Indeed, instrumentation and monitoring tools for memory profiling can induce too much overhead.

[7] `https://www.docker.com`

[8] `https://hub.docker.com/`

[9] p and n can be configured

- Thanks to the use of Dockerfiles, the proposed framework can be configured by software testers in order to define the GUT (*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-constrained 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)

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).

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/cgroup/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[10]. cAdvisor monitors service containers at runtime.

However, cAdvisor monitors and aggregates live data over only 60 seconds interval. Therefore, we need to record all data over time, from the beginning of the container's creation, into a time-series database so that, we can run queries and define non-functional metrics from historical data. To make gathered data truly valuable for resource usage monitoring, we link our monitoring component to a back-end database component.

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 code execution, resource usage stats are continuously sent to this component. When a container is killed, we are able to access 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[11], 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'
```

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:

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

[10] https://github.com/google/cadvisor
[11] https://github.com/influxdata/influxdb

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

resource usage profiles of running programs overtime. To do so, we present a front-end visualization component for performance profiling.

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[12], a time-series visualization tool available for Docker.

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

5.4 Evaluation

So far, we have presented an 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[13].

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

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 GUTs: Java, JS, C++, CS, and PHP code generators. To select cross-platform libraries, we explore github and we use the Haxe library repository[14]. So, we select seven libraries that provide a set of test suites with high code coverage scores.

---

[13] https://testingcodegenerators.wordpress.com/
[14] http://thx-lib.org/

In fact, each Haxe library comes with an API and a set of test suites. These tests, written in Haxe, represent a set of unit tests that covers the different functions of the API. The main task of these tests is to check the correct functional behavior of generated programs. To prepare our benchmark, we remove all the tests that fail to compile to our five targets (i.e., errors, crashes and failures) and we keep only test suites that are functionally correct in order to focus only on the non-functional properties. Moreover, we add manually new test cases to some libraries in order to extend the number of test suites. The number of test suites depends on the number of existing functions within the Haxe library. We use then these test suites to transform functional tests into stress tests. This can be useful to study the impact of this load on the non-functional properties of generated code such as the performance and resource usage. 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 GUT has produced code that is very greedy in terms of resources. Thus, we run each test suite 1K 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 5.2: 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 MB

-*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

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 for specific software platforms. 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. Thus, generated binaries, in the shared volume, are automatically executed within the execution container (Step 2). This execution container is as well an instance of a new Docker image in where we install all the required execution 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 have then access to the resource usage and performance characteristics of all running containers. cAdvisor has been widely used in different projects such as Heapster[15] and Google Cloud Platform[16]. Afterwards, we configure our InfluxDB back-end container in order to record the collected data into a new time-series database (step 4).

[15] https://github.com/kubernetes/heapster

[16] https://cloud.google.com/

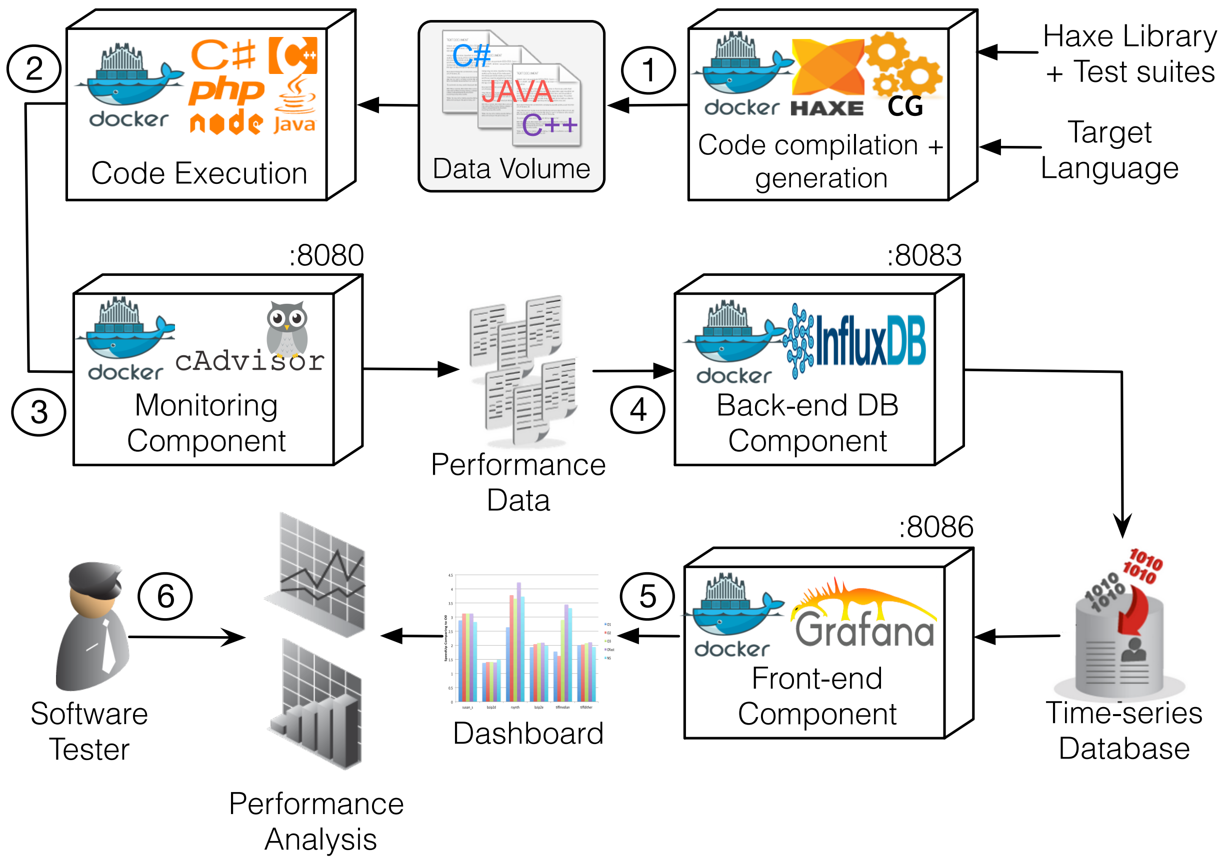

Figure 5.3: Infrastructure settings for running experiments

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 lets 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.

5.4.2 Experimental Results

Evaluation Using the Standard Deviation

We now conduct experiments based on the new created benchmark. The goal of running these experiments is to observe and compare the behavior of generated code in order to detect code generator inconsistencies. Since we are comparing equivalent implementations of the same program written in different languages, we assume that the memory usage and execution time should be more or less the same with a small variation for each test suite. Obviously, we are expecting to get a variation between different executions because we are comparing the execution time and memory usage of test suites that are written in heterogeneous languages and executed using different technologies (e.g., interpreters for PHP, JVM for JAVA, etc.). Therefore, 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 mostly close to 0 - 8 interval. 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 where 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 code behavior for one or more target language, which led to a high performance variation. 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 value. 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 not be the case when the variation is lower than 10 for example. We assume then, that faulty code generators, in identified points, represent a threat for software quality since the generated code has shown symptoms of poor-quality design.

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 in greater depth 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 shows the execution time of each test suite in a specific target language. We also provide 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 a singular behavior regarding the performance with a factor ranging from x40.9 for test suite 3 in benchmark 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 a singular behavior of the PHP code regarding the memory usage for the five last test suites with a factor ranging from x4.1 to x11.5 compared to the JS version. For other test suites versions, the factor varies from x0.8 to x3.7. However, for Color_TS6, C# version consumes higher memory than other languages and even higher than PHP (x2.5 more than JS). These results prove that the PHP code generator is not always effective and it constitutes a performance threat for the generated code. This inconsistency need to be fixed later by code generator creators in order to enhance the code quality of generated code (PHP code for example). Since we are proposing a black-box testing approach, our solution is not able to provide more precise and detailed information about the part of code that has caused these performance issue, which is one of the limitations of this testing infrastructure. Thus, to understand this particular singular performance of the PHP code when applying the test suite core_TS4 for example, we looked (manually) into the PHP code corresponding to this test suite. In fact, we observe the intensive use of *"arrays"* in most of the functions under test. Arrays are known to be slow in PHP and PHP library has introduced much more advanced functions such as $array\_fill$ and specialized abstract types such as *"SplFixedArray"*[17] to overcome this limitation. So, by changing just these two parts in the generated code, we improve the PHP code speed with a factor x5 which is very valuable.

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 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.

5.4.3 Threats to Validity

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

External validity refers to the generalizability of our findings. In this study, we perform experiments on Haxe and on a set of test suite selected from Github and from the Haxe community. For instance, we have no guarantee that these libraries cover all the Haxe language features neither than all the Haxe standard libraries. Consequently, we cannot guarantee that our approach is able to find all the code generators issues unless we develop a more comprehensive test suite. Moreover, the threshold defined to detect the singular performance behavior has a huge impact on the precision and recall of the proposed approach. Experiments should be replicated to other case studies to confirm our findings and try to understand the best heuristic to detect the code generator issues regarding performance (i.e., automatically calculate the threshold values)

Internal validity is concerned with the use of a container-based approach. Even if it exists emulators such as Qemu[18] that allow to reflect the behavior of heterogeneous

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

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

hardware, the chosen infrastructure has not been evaluated to test generated code that target heterogeneous hardware machines. In addition, even though system containers are known to be lightweight and less resource-intensive compared to full-stack virtualization, we would validate the reliability of our approach by comparing it with a non-virtualized approach in order to see the impact of using containers on the accuracy of the results.

5.5 Conclusion

In this paper we have described a new approach for testing and monitoring the code generators families using a container-based infrastructure. We used a set of micro-services in order to provide a fine-grained understanding of resource consumption. To validate our approach, we evaluate a popular family of code generators: HAXE. The evaluation results show that we can find 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.

As a current work, we are discussing with the Haxe community to submit a patch with the first findings. We are also conducting the same evaluation for two other code generators families: ThingML and TypeScript. As a future work, we are going to improve our understanding on the threshold which can provide a best precision for detecting performance issues in code generators. In this paper, we detected inconsistencies related to the execution speed and memory usage. In the future, we seek, using the same testing infrastructure, to detect more code generator inconsistencies related to other non-functional metrics such CPU consumption, etc.

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 5.3: 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 5.4: 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(s)	**Factor**	**Time(s)**	**Factor**	**Time(s)**	**Factor**	**Time(s)**	**Factor**	**Time(s)**	**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	61 777.21	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 5.5: Raw data values of test suites that led to the highest variation in terms of execution time

	JS		JAVA		C++		CS		PHP	
	Memory(Mb)	**Factor**	**Memory(Mb)**	**Factor**	**Memory(Mb)**	**Factor**	**Memory(Mb)**	**Factor**	**Memory(Mb)**	**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 5.6: Raw data values of test suites that led to the highest variation in terms of memory usage

Chapter 6

An infrastructure for resource monitoring based on system containers

6.1 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.

6.2 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 [Mer14]. 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

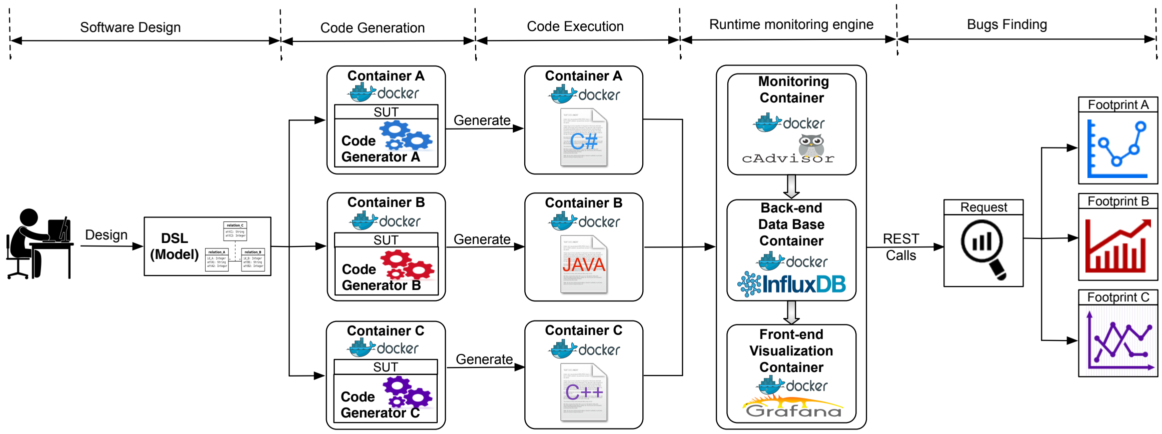

Figure 6.1: A technical overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to runtime.

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.

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

[3] p and n can be cofigured

In short, the main advantages of this approach are:

- The use of containers induces less performance overhead and resource isolation compared to using a full stack virtualization solution [SCTF16]. Indeed, instrumentation and monitoring tools for memory profiling like Valgrind [NS07] can induce too much overhead.
- Thanks to the use of Dockerfiles, 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)

6.3 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).

6.3.1 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/cgroup/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.

6.3.2 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 ex-

[4] https://github.com/google/cadvisor
[5] https://github.com/influxdata/influxdb

ample, the following query reports the maximum memory usage of container "*generated_code_v1*" since its creation:

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

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

6.3.3 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 7

Conclusion and perspectives

7.1 Summary of contributions

7.2 Perspectives

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 [SCDP07]. Thus, we aim to use the same approach to find non-functional issues during the code generation process.

References

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

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

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

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

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

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

[BBSB15] Mohamed Boussaa, Olivier Barais, Gerson Sunyé, and Benoit Baudry. A novelty search approach for automatic test data generation. In *8th*

International Workshop on Search-Based Software Testing SBST@ ICSE 2015, page 4, 2015.

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

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

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

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

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

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

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

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

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

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

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

[CDM+10] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *ACM Sigplan Notices*, volume 45, pages 835–847. ACM, 2010.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[DAH11] Melina Demertzi, Murali Annavaram, and Mary Hall. Analyzing the effects of compiler optimizations on application reliability. In *Workload*

Characterization (IISWC), 2011 IEEE International Symposium on, pages 184–193. IEEE, 2011.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[FRSD15] Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. Runtime code generation and data management for heterogeneous computing in java. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, pages 16–26. ACM, 2015.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Nai16] Nitin Naik. Migrating from virtualization to dockerization in the cloud: Simulation and evaluation of distributed systems. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Environments*

(MESOCA), 2016 IEEE 10th International Symposium on the, pages 1–8. IEEE, 2016.

[NAIT12] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. Random testing of c compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.

[NF13] Mena Nagiub and Wael Farag. Automatic selection of compiler options using genetic techniques for embedded software design. In *Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on*, pages 69–74. IEEE, 2013.

[NHI13] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of c compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, pages 88–93, 2013.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[PB11] Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. In *Software Automatic Tuning*, pages 335–351. Springer, 2011.

[PBG05] Rui Pais, SP Barros, and Luís Gomes. A tool for tailored code generation from petri net models. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 1, pages 8–pp. IEEE, 2005.

[PE06] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[PGL14] Geoffrey Papaux, Daniel Gachet, and Wolfram Luithardt. Processor virtualization on embedded linux systems. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 65–69. IEEE, 2014.

[PHB15] James Pallister, Simon J Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015.

[PHP16] René Peinl, Florian Holzschuher, and Florian Pfitzer. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265–282, 2016.

[PKC11] Eunjung Park, Sameer Kulkarni, and John Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 65–74. ACM, 2011.

[PMV+13] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. *Procedia Computer Science*, 18:1312–1321, 2013.

[PV15] Alireza Pazirandeh and Evelina Vorobyeva. Evaluation of cross-platform tools for mobile development. 2015.

[RAO92] Debra J Richardson, Stephanie Leif Aha, and T Owen O'malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, pages 105–118. ACM, 1992.

[RFBJ13] Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel. Efficient high-level abstractions for web programming. In *ACM SIGPLAN Notices*, volume 49, pages 53–60. ACM, 2013.

[RHS10] Sebastian Risi, Charles E Hughes, and Kenneth O Stanley. Evolving plastic neural networks with novelty search. *Adaptive Behavior*, 18(6):470–491, 2010.

[RPFD14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[RSF+15] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *ACM SIGPLAN Notices*, volume 50, pages 167–180. ACM, 2015.

[SA13] Abdel Salam Sayyad and Hany Ammar. Pareto-optimal search-based software engineering (posbse): A literature survey. In *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2013 2nd International Workshop on*, pages 21–27. IEEE, 2013.

[SC96] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on software Engineering*, 22(11):777–793, 1996.

[SC03] Igno Sturmer and Mirko Conrad. Test suite design for code generation tools. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 286–290. IEEE, 2003.

[SCDP07] Ingo Stuermer, Mirko Conrad, Heiko Doerr, and Peter Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622, 2007.

[SCTF16] Cristian Constantin Spoiala, Alin Calinciuc, Corneliu Octavian Turcu, and Constantin Filote. Performance comparison of a webrtc server on docker versus virtual machine. In *2016 International Conference on Development and Application Systems (DAS)*, pages 295–298. IEEE, 2016.

[SDH+14] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 639–652. ACM, 2014.

[ŠLG15] Domagoj Štrekelj, Hrvoje Leventić, and Irena Galić. Performance overhead of haxe programming language for cross-platform game development. *International Journal of Electrical and Computer Engineering Systems*, 6(1):9–13, 2015.

[SOMA03] Mark Stephenson, Una-May OReilly, Martin C Martin, and Saman Amarasinghe. Genetic programming applied to compiler heuristic optimization. In *European Conference on Genetic Programming*, pages 238–253. Springer, 2003.

[SP15] Stepan Stepasyuk and Yavor Paunov. Evaluating the haxe programming language-performance comparison between haxe and platform-specific languages. 2015.

[SPF+07] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[SRC+12] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.

[SWC05] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–6. ACM, 2005.

[SWES16] Yu Sun, Jules White, Sean Eade, and Douglas C Schmidt. Roar: A qos-oriented modeling framework for automated cloud resource allocation and optimization. *Journal of Systems and Software*, 116:146–161, 2016.

[SZP12] Thayalan Sandran, Mohamed Nordin B Zakaria, and Anindya Jyoti Pal. A genetic algorithm approach towards compiler flag selection based on compilation and execution duration. In *Computer & Information Science (ICCIS), 2012 International Conference on*, volume 1, pages 270–274. IEEE, 2012.

[TCR12] Michele Tartara and Stefano Crespi Reghizzi. Parallel iterative compilation: using mapreduce to speedup machine learning in compilers. In *Proceedings of third international workshop on MapReduce and its Applications Date*, pages 33–40. ACM, 2012.

[TVVA03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 204–215. IEEE, 2003.

[VJ01] Madhavi Valluri and Lizy K John. Is compiling for performancecompiling for power? In *Interaction between Compilers and Computer Architectures*, pages 101–115. Springer, 2001.

[Vou90] Mladen A Vouk. Back-to-back testing. *Information and software technology*, 32(1):34–45, 1990.

[WS90] Deborah Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *ACM SIGPLAN Notices*, volume 25, pages 137–146. ACM, 1990.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[ZSH09] Shengtong Zhong, Yang Shen, and Fei Hao. Tuning compiler optimization options via simulated annealing. In *Future Information Technology and Management Engineering, 2009. FITME'09. Second International Conference On*, pages 305–308. IEEE, 2009.

[ZSP+06] Sergey V Zelenov, Denis V Silakov, Alexander K Petrenko, Mirko Conrad, and Ines Fey. Automatic test generation for model-based code generators. In *ISoLA*, pages 75–81, 2006.