An Approach for Non-functional Testing of Code Generators Using System Containers

Mohamed Boussaa, Olivier Barais, Gerson Sunyé, and Benoit Baudry
Inria/IRISA Rennes, France
Email: {mohamed.boussaa, olivier.barais, gerson.sunye, benoit.baudry}@inria.fr

***Abstract*—The intensive use of domain specific languages (DSL) and generative programming techniques provides an elegant engineering solution to deal with the heterogeneity of platforms or technological stacks. However, the use of DSLs also leads to the creation of numerous code generators and compilers. Generally, different optimization techniques must be applied to generate efficient code with respect to memory consumption, execution time, code size, among others. Due to the huge number of optimizations, finding the best optimization sequence for a given platform and a given program is more and more challenging. This paper describes a component-based approach and its support tools for comparing non-functional properties of code generator through monitoring the generated code in a controlled sandboxing environment. This approach provides a fine-grained understanding of resource consumption and analysis of components behavior regarding optimizations. We evaluate the effectiveness of our approach by verifying the optimizations performed by the GCC compiler, a widely used compiler in software engineering community. We also present a number of case studies, in which the tool was successfully used.**

Keywords. *docker, non-functional properties, code generator, testing*.

I. Introduction

In model-driven software engineering, the intensive use of generative programming techniques has become a common practice for software development since it reduces the development and maintenance effort by developing at a higher-level of abstraction through the use of domain-specific languages [1] (DSLs). DSLs, as opposed to general-purpose languages, are software languages that focus on specific problem domains. Thus, the realization of model-driven software development for a specific domain requires the creation of effective code generators and compilers for these DSLs. The use of code generators is needed to transform manually designed models to software artifacts, which can be deployed on different target platforms. This can clearly reduce the effort of software implementation. A code generator is able to translate source code programs represented in a graphical modeling language (model) into general purpose programming languages such as C, Java, C++, etc. In turn, generated code is transformed into machine code (binaries) using a set of specific compilers. These compilers serve as a basis to target different ranges of platforms.

In fact, during the code generation process, different optimizations may be applied for code transformation. Improvement of source program can refer to several different characteristics of the produced code such execution time, memory consumption, code size, among others [2], [3]. For example, embedded systems for which code is generated often have limited resources. Therefore, optimization techniques must be applied whenever possible to generate efficient code with respect to available resources [4]. As general-purpose optimizations, compiler creators[1] usually define fixed and program-independent sequence optimizations. For example, in GCC, we can distinguish optimization levels from O1 to O3. Each optimization level involves a fixed list of compiler optimization options. However, industrial code generators may have a huge number of potential optimization combinations, making it hard and time-consuming for software developers to find the sequence of optimizations that satisfies user key objectives.

In this paper we explore the relationship between runtime execution of optimized code and non-functional properties. We propose a component-based tooled approach to check code generators non-functional properties through the monitoring of generated code in a controlled sandboxing environment. Our approach is based on microservices to automate the deployment and monitoring of different variants of optimized code into a distributed and heterogeneous component-based infrastructure. We assess the effectiveness of our approach by evaluating the optimizations performed by the GCC compiler, a widely used compiler in software engineering community. We also present a number of case studies, in which the tool was successfully used.

The primary contribution of this paper can be summarized as follows: (1) We propose a microservice infrastructure to ensure the deployment and monitoring of generated code regarding resource consumption; (2) We evaluate the effectiveness of our approach by testing the GCC compiler optimizations across two case studies.

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

[1]We consider compilers as a kind of code generators in this paper. Indeed, from the testing point of view, we do not make any differences between testing code generators and testing compilers

II. Motivation

A. Compilers Optimizations

In the past, researchers have shown that the choice of optimization sequences may impact software performance [2], [5]. 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 [2], [6]. Indeed, each one of these optimizations interacts with the code and in turn with all other optimizations in complicated ways. Similarly, code transformations can either create or eliminate opportunities for other transformations and it is quite difficult for users to predict the effectiveness of optimizations on their source code program. As a result, most software engineering programmers that are not familiar with compiler optimizations find difficulties to select effective optimization sequences.

To explore the large optimization space, users have to evaluate the effect of optimizations and optimization combinations, for different target platforms. Thus, finding the optimal optimization options for an input source code is a challenging, very hard, and time-consuming problem. Many approaches [6]–[9] have attempted to solve this optimization selection problem using techniques such as genetic algorithms, iterative compilation, etc.

It is important to notice that performing optimizations to source code can be so expensive at the expense of resource usage and may induce to compiler bugs or crashes. Indeed, in a resource-constrained environment and because of insufficient resources, compiler optimizations can even lead to memory leaks or execution crashes [10]. Thus, a fine-grained understanding of resource consumption and analysis of compilers behavior regarding optimizations becomes necessary to ensure the efficiency of generated code.

B. 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. 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 optimizations sequences.

For instance, version 4.8.4 provides a wide range of command-line optimizations that can be enabled or disabled, including more than 150 options for optimization. Table I summarizes the optimization flags that are enabled by the default optimization levels O1 to O3. We count 76 optimization flags, resulting in a huge space with 2^{76} possible optimization combinations. In our approach, we did not consider some optimization options that are enabled by default, since they do not affect the performance of generated binaries. 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.

TABLE I
Compiler optimization options within standard optimization 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		

III. Novelty Search Compiler Optimizations Exploration

Lots of techniques (random search based, meta-heuristic, constraint programming) can be used to explore the large set of optimization combinations of a modern code generator. In our approach, we study the use of Novelty Search techniques. Our goal of the Novelty Search approach for compiler optimization is to identify the set of compiler optimization options that provides a trade-off with respect to resource utilization.

A. Novelty Search Adaptation

Optimization options are difficult and even impossible to be chosen by programmers or compiler users. Therefore, a tool to help users to choose the best set of options becomes necessary to achieve a compiler optimization with effectiveness.

In our work, we aim at providing a new alternative for choosing effective compiler optimization options. In fact, since

the search space of possible combinations is too large, we aim at using a new search-based technique called Novelty Search [11] to tackle this issue. The idea of this approach is to explore the search space of possible compiler flag options by considering sequences diversity as a single objective. Instead of having a fitness-based selection that maximizes one of the non-functional objectives, we select optimization sequences based on a novelty score showing how different they are compared to all other combinations evaluated so far. We claim that the search toward effective optimization sequences is not straightforward since the interactions between optimizations is too complex and difficult to predict and to define. For instance, in a previous work Chen et al. [5] 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 can not escape. This phenomenon is known as "diversity loss". For example, if the most effective optimization sequence that induces less execution time, lies far from the search space defined by the gradient of the fitness function, then some promising search areas may not be reached. The issue of premature convergence to local optima has been a common problem in evolutionary algorithms. Many methods are proposed to overcome this problem [12], [13]. 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 toward novelty. In the meanwhile, we choose to gather non-functional metrics of explored sequences such as memory consumption. We describe in more details the way we are collecting these non-functional metrics in section 4.

Mainly, NS acts like Genetic Algorithms. 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. In this example, we gathered information about memory consumption. In line 3 & 4, we compile and execute the input program without any optimization (O0). Then, we measure the resulting memory consumption. By doing so, we will be able to compare it to the memory consumption of new generated solutions. The best solution is the one that yields to the lowest memory consumption compared to O0 usage. Before starting the evolutionary process, we generate an initial population with random sequences. Line 6-21 encode the main NS loop, which searches for the best sequence in term 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 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 solution yields to 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. The process is iterated until reaching the maximum number of evaluations.

Algorithm 1: Novelty search algorithm for compiler optimizations exploration

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

1) Optimization Sequence Representation: For our case study, a candidate solution represents all compiler switches that are used in the 4 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. 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.

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

$$distance(S1, S2) = |S1 \triangle S2| \quad (1)$$

where $S1$ et $S2$ are two selected optimization sequences (solutions). In this equation, we calculate the cardinality of the symmetric difference between the two sequences. This distance will be 0 if two optimization sequences are similar and higher than 0 if there is at least one optimization difference. The maximum distance is equal to the total number of input flags.

To measure the sparseness of a solution, we will 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:

$$NM(S) = \frac{1}{k}\sum_{i=1}^{k} distance(S, \mu_i) \quad (2)$$

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

IV. An Infrastructure for Non-functional Testing Using Docker Containers

To allow code generator testing, we need to deploy the test harness, i. e.the produced binaries, on an elastic infrastructure that provides preconfigured virtual server images, storage, and resources that may be provisioned by testers. Monitoring information should also be provided to inform about resource utilization required/needed and to automate the resource management of deployed components. For this purpose, we propose a testing infrastructure based on System Container techniques such as Docker[2] environment.

Docker will automate the deployment of applications inside software containers. It will simplify the creation of highly distributed systems by allowing multiple applications to run autonomously on a server (basically a cloud server). Docker provides a platform as a service (PaaS) style of deployment for software programs. Consequently, we rely on this technology and benefit from all its advantages to:

1) Deploy generated code within Docker containers
2) Automate optimization sequences generation
3) Monitor service containers
4) Gather performance metrics (CPU, Memory, I/O, etc.)

We integrate a collection of Docker components to define the adequate infrastructure for testing and monitoring of code generators. In the following sections, we describe the deployment and testing architecture of generated code within Docker.

A. System Container as Deployment Environment

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 produced binaries by GCC compilers in term of resource usage. Docker is an open source engine that automates the deployment of any application as a lightweight, portable, and self-sufficient container that runs virtually on a host machine. To achieve that, Docker uses the Linux container technology. The main advantages that Docker offers compared to using a full stack virtualization solution is less performance overhead and resource isolation.

Using Docker, we can define preconfigured applications and servers to host. We can also define the way the service should be deployed in the host machine. As properties, we can define the OS where the service has to run, dependencies, etc. Once Docker images are defined, we can instantiate different containers. A simple way to define Docker images is to use Dockerfiles. Docker can build images automatically by reading the instructions from a Dockerfile. Therefore, for our experiments we describe a Dockerfile that defines the target compiler to test, as well the container OS. The same Docker image will be used then, to execute different instances of generated code. Basically, each container deploys an optimized version of the input source code program.

Docker uses as well 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.

Therefore, to run our experiments, each optimized program is executed individually inside an isolated Linux container. By doing so, we ensure that each executed program runs in isolation without being affected by the host machine or any other processes. Moreover, since a container is cheap to create, we are able to create too many containers as long as we have new programs to execute.

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. In fact, containers/programs are running sequentially without defining any constraints on resource utilization for each container. So once execution is done, resources reserved for the container are automatically released to enable spawning next containers.

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

Therefore, the host machine will not suffer too much from the performance trade-offs.

B. 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 non-functional properties.

1) Monitoring Component: This container will provide us 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 using cgroups. For example, we access to live resource consumption of each container available at the cgroup 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 so that, instead of gathering manually metrics located in cgroups file systems, it will extract automatically runtime resource usage statistics relative to running components. 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 generated program by GCC within each container.

To ease the monitoring process, we use google containers called cAdvisor as Container Advisor[3]. It is a tool developed by Google to monitor their infrastructure. cAdvisor Docker image does not need any configuration on the host machine. We have just to run it on our Docker host. It will then have access to the resource usage and performance characteristics of all running containers. This image uses the cgroups mechanism described above to collect, aggregate, process, and export ephemeral real-time information about running containers. Then, it reports all statistics via web UI ($http://localhost:8080$) to view live resource consumption of each container. cAdvisor has been widely in different projects such as Heapster[4] and Google Cloud Platform[5].

However, cAdvisor monitors and aggregates live data over only 60 seconds interval. Therefore, we would like to record all data over time since container's creation. This is useful to run queries and define non-functional metrics from historical data. Thereby, To make gathered data truly valuable for resources usage monitoring, it becomes necessary to log it in a database at runtime. Thus, we link our monitoring component to a back-end database for better understanding of non-functional properties.

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

2) Back-end Database Component: This component represents a times-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. Hence, we define its corresponding ip port into the monitoring component so that, container statistics are sent over TCP port (e.g, 8083) exposed by the database component.

During the execution of generated code, resource usage stats will be continuously sent into this component. When a container is killed, all statistics will be deleted afterward. We choose a time series database because we are collecting time series data that corresponds to the resource utilization profile of generated code execution.

We use InfluxDB[6], 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 average memory usage of container "$generated\_code\_v1$" for each 2s since container has started:

```
select mean(memory_usage) from stats where
container_name='generated_code_v1' group by
time(2s)
```

To give an idea about data stored in InfluxDB. The following table describes the different stored metrics:

Name	Name of the container
Ts	Starting time of the container in epoch time
Network	Stats for network bytes and packets in an out of the container
Disk IO	Disk I/O stats
Memory	Memory usage
CPU	Cumulative CPU usage

TABLE II
RESOURCE USAGE METRICS RECORDED IN INFLUXDB

For instance, we set the back-end container of our component-based infrastructure. It would be nice to pull all pieces together to view resource consumption graphs within a complete dashboard. It is relevant to show performance profiles of memory and CPU consumption for example of our running applications overtime. To do that, we present a front-end visualization component for performance profiling.

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. Thereby, we can perform statistical analysis on this data to detect inconsistencies or performance anomalies. An overview of the monitoring dashboard is shown in Figure 3.

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

To do so, we choose Grafana[7], one of the best time-series metric visualization tools available for Docker. It is considered as a web application running within a container. We run Grafana within a container and we link it to InfluxDB by setting up the data source port 8086 so that it can easily request data from the database. We recall that InfluxDB also provides a web UI to query the database and show graphs. But, Grafana will let us to 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.

C. Wrapping Everything Together: Architecture Overview

To summarize, we present, as shown in Figure 1, an overall overview of the different components involved in our Docker monitoring infrastructure.

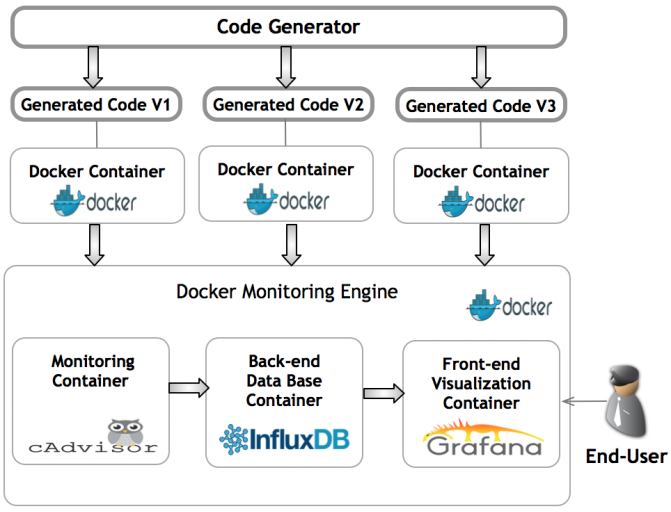

Fig. 1. Overview of the Docker-based testing architecture

Our testing infrastructure will run different jobs within Docker containers. First, we generate and run different versions of code using our target compiler. To do so, we run multiple instances of our preconfigured Docker image that corresponds to specific code generator (e.g, GCC compiler). Each container will execute a specific job. For our case, a job represents a program compiled with new optimization sequence generated by NS. In the meanwhile, we start our runtime testing components (e.g., cAdvisor, InfluxDB and Grafana). The monitoring component collects usage statistics of all running containers and save them at runtime in the time series database component. The visualization component comes later to allow end users to define performance metrics and draw up charts.

V. Evaluation

In this section, we evaluate the implementation of our approach across two case studies. These experiments aim at answering the following research questions:

RQ1: How do standard GCC optimization levels influence on the resource consumption of generated programs? To answer this question, we apply standard optimization options to FFmpeg library. Then, we evaluate the memory footprint and execution time of running FFmpeg command lines and we compare the results. The goal of this initial experiment is to provide an understanding of the performance of generated code by GCC.

RQ2: To what extent can the proposed diversity-based exploration of optimization options impact the resource consumption of generated programs? In a second experiment, we assess our NS approach for automatic optimization sequences generation by comparing the results found when applying standard optimization sequences to new results provided by our approach. The experimental results show that our novelty-based approach can produce optimization sequences with good performance and less resource consumption than standard optimization levels in GCC. In this experiment, we study the correlation between execution time and memory consumption of generated code.

These two case studies also enable to validate the global approach for non functional testing of code generators using system containers.

A. Case Study 1: FFmpeg

In the first experiment, we set up our infrastructure for testing and monitoring of generated code. In this part, we compile the FFmpeg library using standard GCC optimizations(O1, O2, O3, Ofast) and we study the impact of these optimizations on memory consumption and execution time using our Docker-based infrastructure.

1) FFmpeg: Multimedia Encoding Library: FFmpeg[8] is a complete, cross-platform solution to record, convert, and stream audio and video. It is a very fast video and audio converter. It includes libraries of audio/video codecs and a command-line program for transcoding multimedia files. FFmpeg allows different types of multimedia conversion and that depends on the input and output format (video, audio, subtitle, attachment, data). Video/audio processing with FFmpeg usually need high-performance requirements and an important amount of resources in term of memory usage. So, testing GCC on top of this library is very interesting in order to compare resource usage profiles.

2) Docker-based Infrastructure for Monitoring FFmpeg Containers: The goal of this experiment is to compile FFmpeg with standard GCC optimization options and run FFmpeg command examples on top of different versions in order to compare the memory usage profiles and execution time of different variants using our test architecture. An overview of the different components involved in testing and monitoring of FFmpeg containers is shown in Figure 2. First, we compile FFmpeg library with different optimization options (O0, O1, O2, O3 and Ofast) in order to produce 5 variants of FFmpeg. This is done within Docker containers. We configure each

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

[8]https://www.ffmpeg.org/

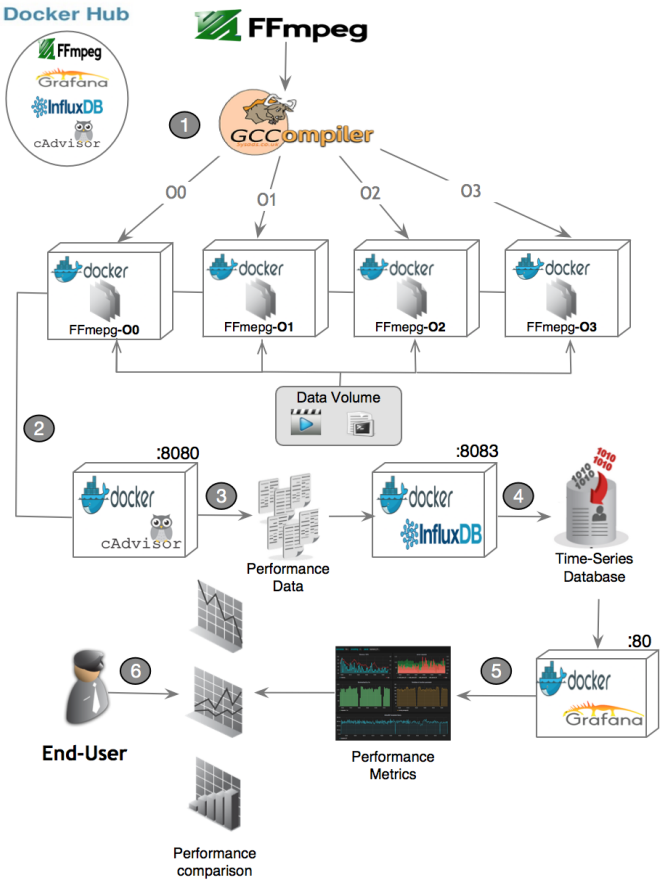

Fig. 2. Overview of the different components involved in testing and monitoring of FFmpeg containers

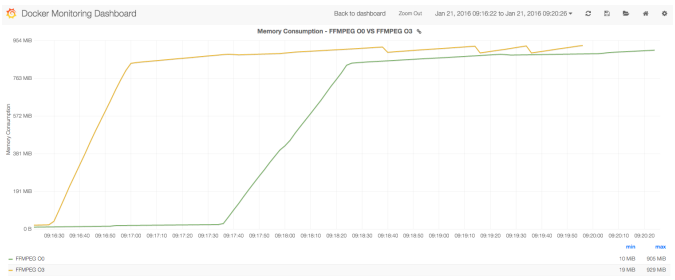

Fig. 3. Snapshot of runtime memory consumption profiles of FFmpeg containers compiled with O0 (no optimization) and O3 options

container to install FFmpeg with a specific configuration and we upload all FFmpeg images in Docker Hub[9]. Docker Hub is a cloud-based registry service for building and shipping application or service containers. We use it for building, saving, and managing all our Docker images.

Afterwards, we execute the same stressload (FFmpeg examples) within each FFmpeg instance container. We choose 15 FFmpeg command examples that cover multiple domains[10] like video conversion, sound extraction, encoding file for iPod or PSP, etc. This list of examples is saved in a script file that should be executed later within FFmpeg containers. The media files needed for encoding are saved in a shared repository. In Docker environment, we call this repository the Data Volume. A data volume is a specially-designated directory within containers that share data with the host machine. Data is persistent and independent of container's life cycle. So, when we run FFmpeg containers we provide a shared volume with the host machine (where the media files are located). As well, the list of FFmpeg commands to execute is mounted in this volume so that, we can execute the same workload each time we run a new container. Before running FFmepg workload on different containers, we run monitoring components (cAdvisor, InfluxDB and Grafana) to start gathering usage statistics. In fact, in this experiment, we choose to gather statistics about memory consumption and execution time of different containers.

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

3) Experimental Results: In this first part of experiments with FFmpeg, we compare a zero-level optimization container (FFmpeg-O0) with a high-level optimization one (FFmpeg-O3) in order to study the impact of optimizations on memory usage and execution time. Figure 3 shows runtime statistics of running two FFmpeg containers O0 and O3 with the same input workload. This chart depicts a snapshot of our visualization component web UI. It presents the memory usage profiles of two components (FFmpeg O0 and O3) started in the same time and running in parallel. Visually, we can see that the execution time of O3 (yellow) is faster than O0 (green) since O3 execution ended before O0. However, we can see that memory usage remains higher than O0 from the beginning to the end of running FFmpeg examples.

To better understand the resource usage of optimized code, we run the same experiment for all FFmpeg containers and we collect the same metrics. Figure 4 presents a comparison of average memory usage and execution time of FFmpeg containers compiled with all standard GCC optimization options. We notice that the memory usage increases as soon as we apply more aggressive optimization. However, when we apply higher optimization level, we clearly improve the execution time compared to O0. For example, O3 and Ofast have the highest memory consumption (100 MBytes more than O0) and best execution time (speedup around 1.8) which can be inconvenient for systems with limited resources.

This results explain that optimizing for execution time (for the case of FFmpeg) is not always efficient regarding memory usage and unfortunately, optimizations may influence negatively the system resources.

[9]https://hub.docker.com/
[10]http://goo.gl/11VLYM

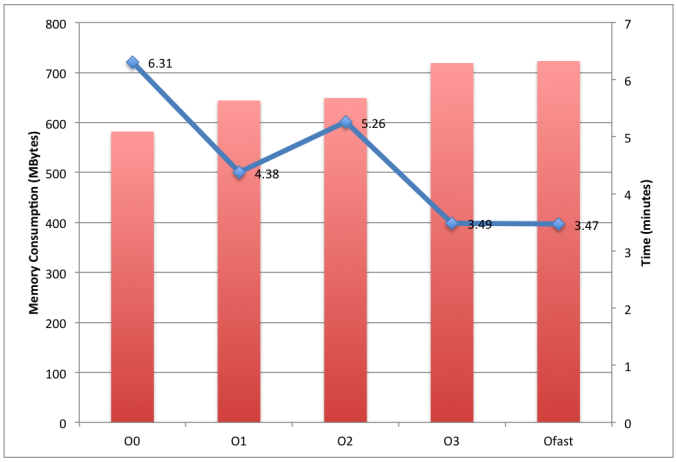

Fig. 4. Comparison of average memory consumption and execution time of FFmpeg containers compiled with standard GCC optimization options

B. Case Study 2: Novelty-based Exploration of Optimization Sequences

In this second case study, we assess the performance of GCC compiler across different benchmark programs. The goal of this experiment is to show that our approach for exploring the search space of optimizations is able to generate efficient code that yields to less resource consumption than GCC default optimizations.

1) Setting Up Infrastructure: For this experiment, we keep the same architecture settings as first case study. However, we provide a more generalized approach for automatic testing of code generators using system containers. So, starting from a list of optimizations defined by the user, an input program and a specific target compiler, we are able to execute and monitor optimized code. Figure 5 shows more details about this process. First, our novelty-based test sequences generator generates a huge amount of diverse optimizations. We compile an input program with these generated sequences using GCC. Then, we execute produced binaries sequentially within isolated Docker containers (Docker image with GCC version 4.8.4 installed above). We keep the same monitoring chain to gather resource usage of running containers. This process is repeated until the end of generated sequences. Finally, end-users (testers) can access to resource consumption statistics through InfluxDB or Grafana Web UI to compare the impact of optimizations on resource consumption. In this example too, we choose to focus on studying the trade-off memory usage/execution time.

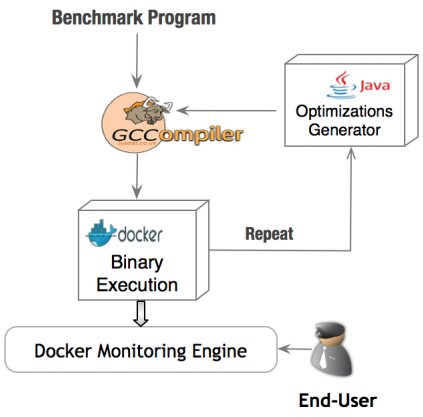

Fig. 5. Overall process of monitoring code generated by GCC

2) Benchmark programs: To explore the impact of compiler optimizations a set of input programs are needed. We run experiments on commonly used benchmarks named Collective Benchmark (cBench) [14]. 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.

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 III
DESCRIPTION OF SELECTED BENCHMARK PROGRAMS

3) Novelty Parameters: NS is implemented as described in Section 3. The first step in the process of selection is to evaluate each individual and compute its novelty score. Novelty is calculated for each organism by taking the mean of its 15 lowest dissimilar optimization sequences (considering all sequences in the current population and in the archive). 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. Standard genetic programming crossover and mutation operators are applied to these novel sequences in order to produce offspring individuals and fulfill the next population. In the meanwhile, individuals that get a score higher than the threshold T they are automatically added to the archive as well. In fact, this threshold is dynamic. Every 150 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 calculations that determine the novelty scores for individuals become more computationally demanding. So to avoid having low accuracy of novelty, we choose to bound the size of the archive. Hence, it follows a

TABLE IV
PARAMETERS OF NS ALGORITHM

Parameter	Value	Parameter	Value
Novelty nearest-k	15	Tournament size	2
Add archive prob.	30	Mutation prob.	0.1
Max archive size	500	Crossover	0.5
Population size	100	Nb generations	100
Individual length	76	Elitism	10
Scaling archive prob.	0.05	Solutions added to archive	3

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 individuals diversity by removing old sequences that may no longer be reachable from the current population.

The parameters of the algorithm 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 [11]. The value that yielded the highest performance scores was chosen. The resulting parameter values are listed in Table 4.

4) Experimental results: The goal of this experiment is to compare novelty-based generated sequences to standard GCC optimizations in term of memory consumption. Figure 6 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).

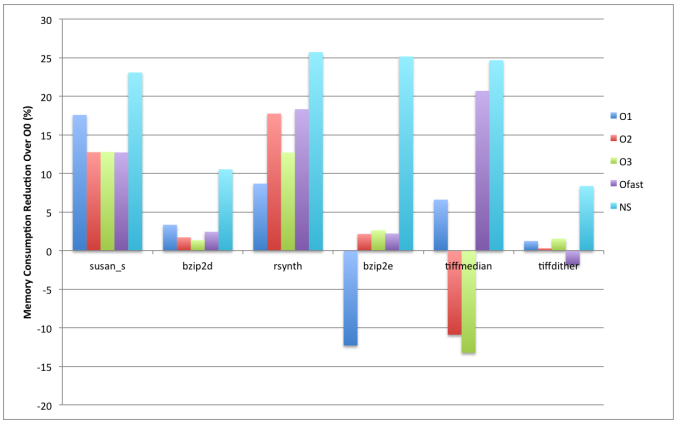

Fig. 6. Evaluating the amount of saved memory after applying standard optimization options compared to best generated optimization using NS

For NS, we select the best sequence that reduces the memory consumption and we compare it to the memory footprint of O0, O1, O2, O3 and Ofast versions. The results show clearly that NS 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 % (like the FFmpeg experiments). 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. Our approach can clearly provide an alternative to catch most relevant optimization sequence regarding resource consumptions.

To study the correlation between execution time and memory consumption of running programs, we present in Figure 7 an evaluation of the speedup. We compare the speedup (according to O0) of the best optimization sequences by NS (gathered in Figure 6) with standard optimization options. The first observation is that optimizations yield to high level of speedup for all benchmark programs (between 1.5 and 4.3). The second observation we can make is that different optimizations do not differ too much in term of execution time. We distinguish that Ofast is slightly more efficient for all programs and NS sequence has almost the same speedup as Ofast. The results of this experiments shows that optimizing for memory usage using NS does not affect programs execution time and we demonstrate that we can find optimizations that reduce memory usage while guaranteeing program performance.

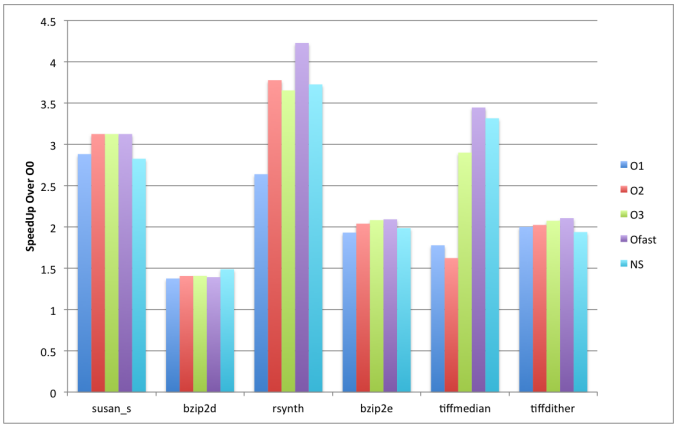

Fig. 7. Evaluating the speedup after applying standard optimization options compared to best generated optimization using NS

VI. RELATED WORK

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 to catch relevant optimizations regrading performance, energy or code size improvements over standard optimization sequences [2], [3], [5]–[9], [15]–[18]. The vast majority of the work on iterative compilation focuses on increasing the speedup of new optimized code compared to standard optimizations. It has been proven that optimizations are highly dependent on target platform and input program. Compared to our proposal, we rather focus on comparing the resource consumption of optimized code.

Novelty Search has never been applied in the filed of iterative compilation. Our work presents the first attempt to

introduce diversity in optimization sequences generation. The idea of NS has been introduced by Lehman et al. [11]. It has been often evaluated in deceptive tasks and especially applied to evolutionary robotics [19], [20] (in the context of neuroevolution). NS can easily be adapted to different research fields. In a previous work [21], NS has been adapted for test data generation where novelty score was calculated as the Manhattan distance between the different vectors representing test data. In our NS adaptation, we are measuring the novelty score using the systematic difference between optimization sequences of GCC.

For code generators testing, Stuermer et al. [22] 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 our approach, we provide a component-based infrastructure to compare non-functional properties of generated code rather than functional ones.

VII. Conclusion and Future Work

In this paper we have described a new approach for testing and monitoring code generators using a component-based infrastructure. We used a set of microservices in order to provide a fine-grained understanding of resource consumption. To validate the approach, we investigated the problem of GCC compiler optimizations through the use of Novelty Search as a search engine. Then, we studied the impact of optimizations on memory consumption and execution time across two case studies. Results showed that our approach is able to automatically find better optimization sequences across different programs.

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 compare our findings using the novelty search approach to multi-objective evolutionary algorithms. Finally, our proposed microservice-based approach for testing can easily be adapted and integrated to new case studies, so we would inspect the behavior of different other code generators and try to find non-functional bugs regarding code generation processes.

Acknowledgment

This work was funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n611337, HEADS project (www.heads-project.eu)

References

[1] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis Lectures on Software Engineering*, vol. 1, no. 1, pp. 1–182, 2012.

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Finding effective compilation sequences," *ACM SIGPLAN Notices*, vol. 39, no. 7, pp. 231–239, 2004.

[3] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*. IEEE, 2006, pp. 12–pp.

[4] M. Nagiub and W. 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*. IEEE, 2013, pp. 69–74.

[5] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu, "Deconstructing iterative optimization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 3, p. 21, 2012.

[6] K. Hoste and L. Eeckhout, "Cole: compiler optimization level exploration," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2008, pp. 165–174.

[7] S. Zhong, Y. Shen, and F. Hao, "Tuning compiler optimization options via simulated annealing," in *Future Information Technology and Management Engineering, 2009. FITME'09. Second International Conference On*. IEEE, 2009, pp. 305–308.

[8] T. Sandran, M. N. B. Zakaria, and A. J. Pal, "A genetic algorithm approach towards compiler flag selection based on compilation and execution duration," in *Computer & Information Science (ICCIS), 2012 International Conference on*, vol. 1. IEEE, 2012, pp. 270–274.

[9] L. G. Martins, R. Nobre, A. C. Delbem, E. Marques, and J. M. 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*. ACM, 2014, pp. 63–72.

[10] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[11] J. Lehman and K. O. Stanley, "Exploiting open-endedness to solve problems through the search for novelty." in *ALIFE*, 2008, pp. 329–336.

[12] W. Banzhaf, F. D. Francone, and P. 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*. Springer, 1996, pp. 300–309.

[13] C. Gathercole and P. Ross, "An adverse interaction between crossover and restricted tree depth in genetic programming," in *Proceedings of the 1st annual conference on genetic programming*. MIT Press, 1996, pp. 291–296.

[14] G. Fursin, "Collective tuning initiative: automating and accelerating development and optimization of computing systems," in *GCC Developers' Summit*, 2009.

[15] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying compiler options to minimize energy consumption for embedded platforms," *The Computer Journal*, vol. 58, no. 1, pp. 95–109, 2015.

[16] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather *et al.*, "Milepost gcc: machine learning based research compiler," in *GCC Summit*, 2008.

[17] S.-C. Lin, C.-K. Chang, and S.-C. Lin, "Automatic selection of gcc optimization options using a gene weighted genetic algorithm," in *Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific*. IEEE, 2008, pp. 1–8.

[18] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 639–652.

[19] S. Risi, C. E. Hughes, and K. O. Stanley, "Evolving plastic neural networks with novelty search," *Adaptive Behavior*, vol. 18, no. 6, pp. 470–491, 2010.

[20] P. Krčah, "Solving deceptive tasks in robot body-brain co-evolution by searching for behavioral novelty," in *Advances in Robotics and Virtual Reality*. Springer, 2012, pp. 167–186.

[21] M. Boussaa, O. Barais, G. Sunyé, and B. Baudry, "A novelty search approach for automatic test data generation," in *8th International Workshop on Search-Based Software Testing SBST@ ICSE 2015*, 2015, p. 4.

[22] I. Stuermer, M. Conrad, H. Doerr, and P. Pepper, "Systematic testing of model-based code generators," *Software Engineering, IEEE Transactions on*, vol. 33, no. 9, pp. 622–634, 2007.