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

Mohamed Boussaa, Olivier Barais, Gerson Sunye 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, disk writing speed, 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 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 (DSLs). DSLs, as opposed to general-purpose languages, are software languages that focus on a specific problem domain. 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, disk writing speed, among others. 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 resource. As general-purpose optimizations, compiler creators 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, compilers may have a huge number of optimization combinations that can be applied and it very hard and time-consuming for software developer to find the optimal sequence of optimization options that satisfies user key objective.

In this paper we want to explore the relationship between runtime execution of optimized code and resource consumption like memory consumption. We propose a component-based tooled approach to check non-functional properties of compilers through monitoring of generated code in a controlled sandboxing environment. Our approach is based on Docker containers to automate the deployment and monitoring of different variants of optimized code into a distributed and heterogeneous component-based infrastructure. We evaluate the effectiveness of our approach by means of testing optimizations performed by the GCC compiler, a widely used compiler in software engineering community. We present as well a number of case studies, in which the tool was successfully used.

The primary contribution of this paper can be summarized as follows: (1) The paper proposes a docker-based infrastructure to ensure the deployment and monitoring of generated code regarding resource consumption; (3) We evaluate the effectiveness of our approach by testing GCC compiler across two case studies.

The paper is organized as follows: section 2 describes the motivation. A novelty search technique for compiler optimizations exploration is presented in Section 3. We present in section 4 our infrastructure for non-functional testing using docker containers. The evaluation and results of our experiments across two two case studies are discussed in Section 5. Finally, concluding remarks and future work are provided in Section 6.

II. Motivation

A. Compilers optimizations

By the past, researchers have shown that the choice of optimization sequences can have an effect on software performance[ref]. As a consequence, optimizing software programs for performance becomes key objective for industries and software developers because they are often willing to pay additional costs to meet specific performance goals especially for resource-constrained systems.

Universal and predefined sequences (O1 to O3) may not produce always good performance results and may be highly dependent on the benchmark and the source code they have been tested on [2] [3]. In addition, most software engineering programmers are not familiar with compiler optimizations and they don't have the capability of selecting effective optimization sequences. Furthermore, each of these optimizations interacts with the code and in turn with all other optimizations in complicated ways. A code transformation can create opportunities for other transformations. Similarly, a transformation can eliminate opportunities for other transformations. Thus, interactions are too complex and it is quite difficult for users to predict the effectiveness of optimizations on their source code program

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

We note also that performing optimizations to source code may be so expensive at the expense of resource usage and may induce to compiler bugs or crashes. With the increasing of resource usage, it is important to evaluate the compiler behavior. Indeed, in resource-constrained environment, compiler optimizations may lead to memory leaks or execution bottlenecks.

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

We choose GCC compiler as a motivating example in order to explain how we would study the impact of compiler optimizations using a component-based infrastructure for testing and monitoring. The compiler optimization level design space is very huge which needs a heuristic to explore the search space of feasible optimizations sequences. In next section, we present a search based technique called Novelty Search for automatic generation of optimization sequences.

GCC exposes its various optimizations via a number of flags that can be turned on or off through command-line compiler switches. For example, the GNU Compiler Collection (GCC) 4.8.4 compiler provides a wide range of command-line optimizations that we can enable or disable (it includes more than 150 options for optimization). We count 76 optimization options that are appearing in the default standard optimization levels O1-O3 of GCC. 10 options are enabled by default and the remaining options are defined within O1, O2, O3 and Ofast levels. This results in a huge space with 2^{76} possible optimization combinations. We would note that in GCC there are some optimization options that are enabled by default. In our approach, we didn't involve these optimizations since they don't affect either the performance or size 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 So, our goal is to investigate different combinations of optimization sequences using Novelty Search (NS) in order to explore as much as possible the search space.

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

The goal of the Novelty Search approach for compiler optimization, introduced by Lehman and Stanley in 2008 [22],

is to identify a set of compiler optimization levels that provide 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. So a tool to help users to choose the best set of options becomes necessary to achieve a compiler optimization with effectiveness and efficiently.

There have been many previous works that have investigated this problem by using different techniques like search based or machine learning techniques, among others[ref]. Some of the works focused on optimizing compilers in term of execution time. Some others focused on reducing the energy consumption of running programs on hardware machines.

In this work, we aim to provide a new alternative for choosing effective compiler optimization options. In fact, since the search space of possible combinations is too large, we aim to use a new search based technique called Novelty Search to tackle this issue. The idea of this approach is to explore the search space of possible compiler flag options without regard to any objective. In fact, instead of having a fitness-based selection that aim to optimize either execution time or energy consumption, we select optimization sequences based on a novelty score showing how different they are compared to all other combinations evaluated so far. This makes the tool fitness and program independent.

We think also 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. In a previous work for example, Chen et al. [4]showed that a handful optimizations may lead to higher speedup 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. However, all these alternatives use a fitness-based selection to guide the search. Therefore, diversity maintenance in the population level is key for avoiding premature convergence. Considering diversity as the unique objective function to be optimized may be a key solution to this problem.

So during the evolutionary process, we use to select optimization sequences that remain in sparse regions of the search space in order to guide the search through novelty. In the meanwhile, we choose to gather non-functional metrics of explored sequences namely memory and CPU consumption. These metrics will provide us a more fine-grained understanding and analysis of compiler's behavior regarding optimizations.

Algorithm 1: Novelty search algorithm for compiler optimizations exploration

Require: Optimization sequences S
Require: Benchmark programs Benchmark
Require: Novelty threshold T
Require: Limit L
Require: Nearest neighbors K
Ensure: Best optimization sequence best_sequence

```
1:  initialize_archive(archive, L)
2:  population ← random_sequences(S)
3:  repeat
4:    for sequence ∈ population do
5:      for program ∈ benchmark do
6:        perfomance ← execute(sequence, program)
7:      end for
8:      novelty_metric(sequence) ←
        distFromKnearest(archive, population, K)
9:      if novelty_metric > T then
10:       archive ← archive ∪ sequence
11:     end if
12:   end for
13:   new_population ←
      generate_new_population(population)
14:   generation ← generation + 1
15: until generation = N
16: return best_sequence
```

1) Optimization sequences 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 which represent compiler options are represented as genes in a chromosome. So, a solution represents the CFLAGS value used by GCC to compile programs. A solution has always the same size which is the total number of involved flags. But, 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). This measure expresses how unique the optimization sequence is. 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. Algorithm 1 shows an overview of the evolutionary process of NS. 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 technologies

To facilitate the testing of code generators, we need to deploy produced binaries on an elastic infrastructure that provides preconfigured virtual server images, storage and network connectivity 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. For this purpose, we propose a testing infrastructure based on docker 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 will provide a platform as a service (PaaS) style of deployment for software programs. Consequently, we will rely on this technology and benefit from all its advantages to:

1) Deploy preconfigured application to test within docker containers
2) Automate optimization sequences generation
3) Monitor service containers
4) Gather performance metrics (CPU, Memory, I/O, etc.)

As a consequence, we are going to integrate a collection of docker technologies to define the adequate infrastructure for testing and monitoring code generators. So which docker technologies must be glued together? and how can we integrate our NS optimizations generator within this architecture?

In the following, we will describe how can we ease the deployment and configuration of generated code within docker.

A. Docker as a deployment environment

Before start monitoring and testing applications, we have to deploy different applications 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 will run virtually on 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. Using docker images we can 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.

Docker uses as well Linux control groups 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 gives a lot of 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 will be able to create too many containers as long as we have new programs to execute and the system does not suffer too much from the performance trade-off.

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.

B. Monitoring components

Docker Containers also rely on control groups (cgroups) to expose a lot of metrics about accumulated CPU cycles, memory, and block I/O usage. These metrics are exposed through pseudo-file systems inside each container. For example, memory and CPU stats are located within $memory.stat$ and $cpuacct.stat$ pseudo-files. In order to achieve docker monitoring of our running applications within docker containers, we aim to use some Docker facilities to ease the extraction of performance metrics.

1) cAdvisor: Monitoring component: So, we use google containers called cAdvisor as Container Advisor. It is a tool developed by Google to monitor their infrastructure. This container will provide us an understanding of the resource usage

and performance characteristics of our running containers. In fact, it automates the mechanism stated above about the[1] extraction of performance metrics using cgroups file systems[2] at runtime. Thus, cAdvisor collects, aggregates, processes, and[3] exports information about running containers. We note that resource usage information is collected in raw data.

cAdvisor does not need any configuration on the host machine. We have just to run cAdvisor container on our docker host. It will then have access to the resource usage and performance characteristics of all running containers. For example, Cadivsor accesses at runtime to CPU consumption metrics available at the cgroup filesystem via stats found in $/sys/fs/cgroup/cpu/docker/(longid)/$ and reports all statistics via web UI ($http : //localhost : 8080$) to view live resource consumption for each container. Stats related to memory consumption can be found as well in $/sys/fs/cgroup/memory/docker/(longid)/$. CAdvisor will automate the process of service discovery and metrics aggregation so that, instead of gathering manually metrics located in cgroups file systems we will use it to ease this task. It has been widely in different projects namely Heapster[1] and Google Cloud Platform[2].

Cadvisor may induce a little overhead, because it does very fine-grained accounting of the memory usage on running container. This is may not affect the gathered performance values since we run only one generated program by GCC within each container.

cAdvisor only monitors and aggregates data over a 60 seconds interval. It collects ephemeral data in real-time for each container. This limitation can in many cases be overseen but we would like to record all data over time. This is useful to execute queries and define metrics from historical data. Thereby, To make gathered data from cAdvisor truly valuable for monitoring resources usage, it becomes necessary to log it in a database at runtime. Fortunately, cAdvisor can easily be plugged together with a database.

2) InfluxDB: Back-end Database component: For that purpose, we use InfluxDB, an open source distributed time series database as backend to record data. It collects Docker container performance metrics through cAdvisor for long-term retention, analytics and visualization. When a new container is launched it will be automatically fetched by cAdvisor and its statistics will be continuously sent into InfluxDB while the container is running. When a container is killed, all statistics will be deleted afterward. cAdvisor must be plugged with the new created times-series database created in influxDB to use it as a datastore back-end. Hence, we add the ip port of the database to cadvisor image. So, container statistics are sent over TCP port (e.g, 8086) exposed by influxdb. InfluxDB allow the user to execute SQL 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 been started:

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

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

```
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 performance metrics stored in InfluxDB: For the moment, we set our back-end for docker monitoring tool (cAdvisor+influxDB). It would be nice to pull all the pieces together visually to view nice charted graphs within a complete dashboard. It is relevant to show performance profiles of memory and CPU consumption of our running applications overtime. To do that, we present Grafana for performance profiling.

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

TABLE II
RESOURCE USAGE METRICS RECORDED IN INFLUXDB

3) Grafana: Front-end Visualization Component: Once we have gathered and stored performance data, the next step is visualizing them. That is the role of Grafana. Grafana can be considered as a web application running within a container. It is one of the best time-series metric visualization tools available. It can easily work with data held in InfluxDB. It provides us a dashboard to run queries against the linked database and chart them accordingly in a very nice layout. Grafana will be the endpoint that we will use to visualize the recorded data. It is easily configurable and will let us choose what to render through Web UI. We run Grafana within a docker container and we link it to InfluxDB by setting the data port 8086 so that it can easily request data from the database.

We recall that influxDB has also an interface to query the database and show graphs. But, Grafana let us store our searches and display the results in much pretty looking graphs. That is why we used Grafana. As well, we can set up the profile of resource consumption for different running containers. Thereby, we can compare the profile of CPU and memory consumption among containers. The only thing we have to care about is the metric definitions (using SQL queries like in InfluxDB). Once we define which data we have to extract from the database for visualization, we can draw up whatever we want using Grafana dashboard. Finally, we can save the created dashboard as well as all its dependencies like the defined metrics, database source... We can also export the data currently being viewed within different panels into static JSON or CSV document. Thereby, we can perform statistical analysis on this data to detect bugs or performance anomalies.

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.

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 the input program. For our case, a job represents a program compiled with new optimization sequence generated by NS. In the meanwhile, we run our monitoring components (Cadvisor, InfluxDB and Grafana). CAdvisor collects usage statistics of all running containers and save them at runtime in the time series database InfluxDB. Grafana comes later to allow the end users to define performance metrics and draw up charts.

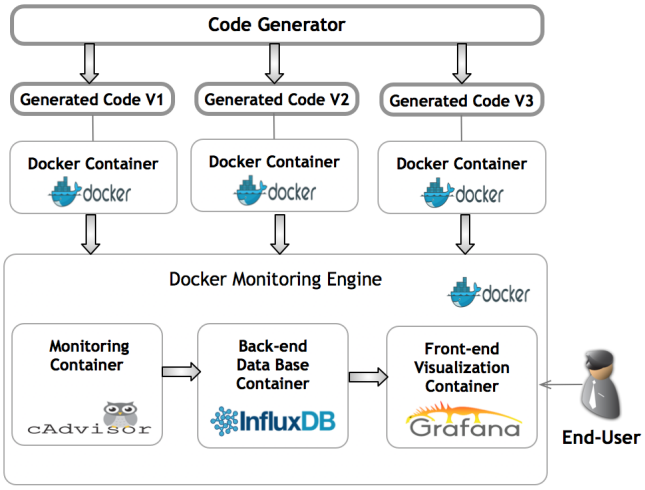

Fig. 1. Overview of the docker-based testing architecture

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:

1. RQ1: How do standard GCC optimization levels influence on the resource consumption of running 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.

2. RQ2: To what extent can the proposed diversity-based exploration of optimization options impact the resource consumption of running programs? In a second experiment, we assess our NS approach for automatic optimization sequences generation by comparing the results founded by applying standard optimization sequences to new results provided by our approach. In general, these experiments show that our novelty-based approach produces optimization sequences with higher performance and less resource consumption than standard optimization levels in GCC. In this experiment, we focused as well on the tradeoff execution time/memory consumption.

A. Case Study 1: FFMPEG

In the first experiment, we set up our infrastructure for testing and monitoring the generated code. In this part, We compiled the FFMPEG library using standard GCC optimizations(O1, O2, O3, Ofast) and we studied the impact of these optimizations on memory consumption and execution time using our docker based infrastructure.

1) FFMPEG: Multimedia Encoding Library: FFmpeg is a complete, cross-platform solution to record, convert and stream audio and video. It is a very fast video and audio converter. It processes a number of input files specified by the $-i$ option, and generate different output files according to the input plugin. FFmpeg allows different types of video conversion and that depends on the input and output format (video, audio, subtitle, attachment, data). Some command options have to be specified for each type of video or audio processing. In fact, FFmpeg defines a set of option flags through command line to configure and generate the output file (e.g. number of threads, size, bitrate, the video or audio codec). Video or audio processing with FFmpeg consumes lot of resources in term of memory usage. So, testing GCC on top of this library is very interesting.

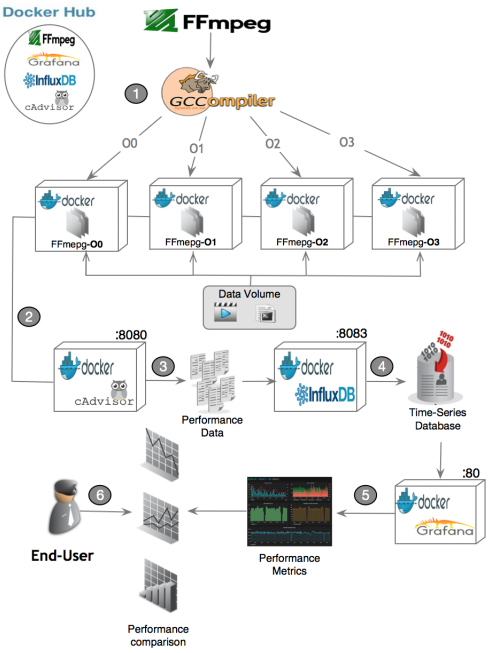

Fig. 2. Overview of the different components involved in testing and monitoring of FFMPEG containers

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 different versions in order to compare the memory usage profiles and execution time of different variants using our testing architecture. An overview of the

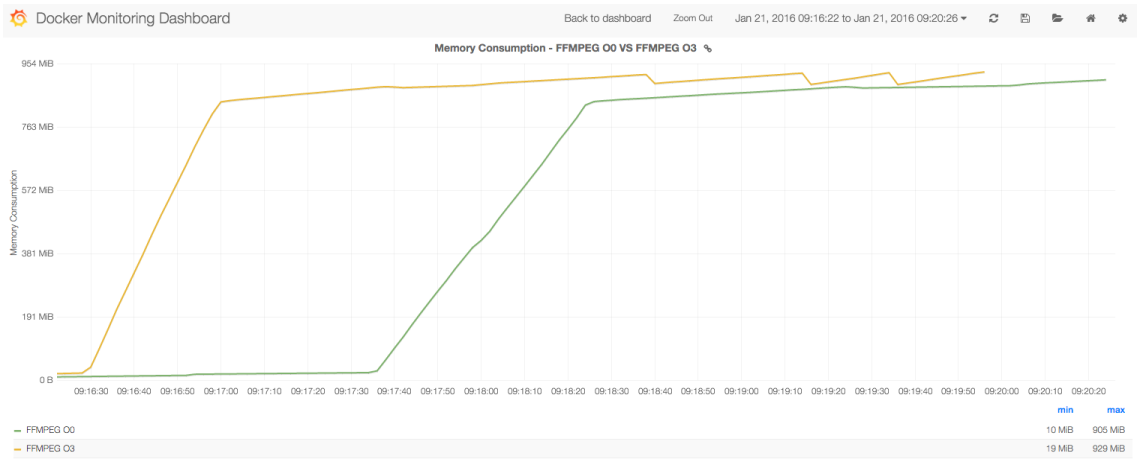

Fig. 3. Runtime memory consumption profiles of workload running in FFMPEG containers compiled with O0 (no optimization) and O3 options

different components involved in testing and monitoring of FFmpeg containers is shwon 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 a Docker container with a Linux image. We configure each container to install FFmpeg with a specific configuration and we uploaded all the FFmpeg images in Docker Hub. Docker Hub is a cloud-based registry service for building and shipping application or service containers. We are using it for building, saving and managing all our docker images. Then, we execute the same ffmpeg testing examples within each ffmpeg instance container. We choose 15 ffmpeg example commands that cover multiple domains[3] like video conversion, sound extraction, encoding file for iPod or PSP, etc. This list of examples is saved in a script file to execute within 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 containers 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 for each container.

Before running the FFmepg workload on different containers, we run monitoring components (cAdvisor, InfluxDB and Grafana) to start gathering usage statistics.

Using Grafana, we are able to query database and define performance metrics. We use for this experiment to gather statistics about memory consumption and execution time of different containers.

To obtain comparable and reproducible results, we used the same hardware across all experiments to run Docker: 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]http://goo.gl/11VLYM

3) Experimental results: The first part of this experiment is to compare the no-optimization container with the high-optimization one in order to study the impact of optimizations on resource consumption. Figure 3 shows runtime statistics for two running ffmpeg containers O0 and O3 with the same input workload. This chart present the memory usage profile of two components (FFMPEG O0 and O3) started in the same time and running in parallel. Visually, we can see that the execution of O3 (yellow) is faster than O0 (green) by around 20s. However, we can see that memory usage remains higher than O0 from the beginning to the end of running the ffmpeg examples.

To better understand resource usage of optimized containers, we run the same experiment for all ffmpeg containers and we collected 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 remark that the memory usage is increasing as soon as we apply more aggressive optimization.

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 on system resources.

B. Case Study 2: Novelty-based exploration of optimization sequences

For the second experiment, we present a set of experiments to evaluate the performance optimized programs across target benchmarks. The goal of this section is to show that our approach for exploring the search space of optimizations is able to generate efficient sequences of optimization options in term of performance and resource consumption. We define an efficient sequence as a set of optimization options that lead to use less resource consumption than GCC default optimizations.

1) Settings: We want to keep the same architecture settings as first case study for this experiment. However, in

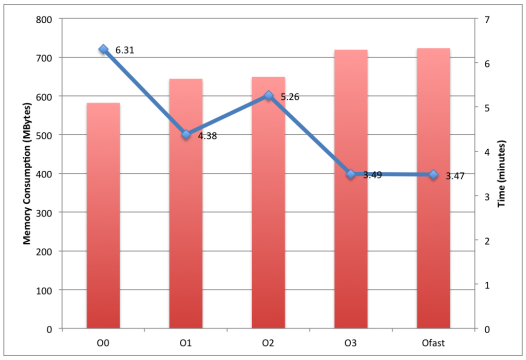

Fig. 4. Comparison of average memory consumption and execution time of running workload in FFMPEG containers compiled with standard GCC optimization options

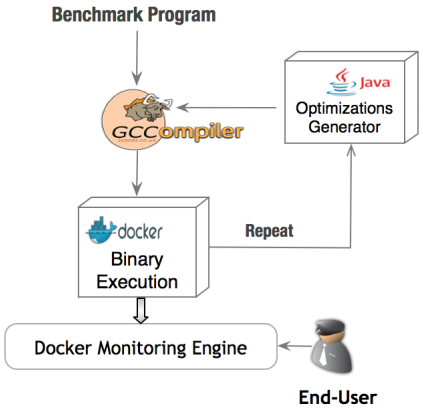

Fig. 5. Overall process of optimization space exploration and monitoring

this example, 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 used, an input program and a specific target compiler, we will be able to execute and monitor optimized code. Figure 5 shows more details about this process. First, our novelty-based test sequences generator will generate a huge amount of diverse optimizations. We compile an input program with these generated sequences using our GCC compiler. Then, program execution is done 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.

2) Benchmark programs: To explore the impact of compiler optimizations a set of test input programs are needed. We run experiments on commonly used benchmarks named Collective Benchmark (cBench)[ref]. 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 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: Our experiments use the classical NS algorithm, where we evolve a set of optimization sequences through generations. 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, by 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 1500 evaluations, it is checked 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 queue data structure (first-in first-out) which means that when a new solution gets added (enqueued), the oldest solution in the novelty archive will be discarded or dequeued. 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[ref]. The value that yielded the highest per-

TABLE IV
PARAMETERS OF THE EVOLUTIONARY 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	No generations	100
Individual length	76	Elitism	10
Scaling archive prob.	0.05	Solutions added to archive	3

formance 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 our set of benchmark programs. It presents the percentage of saved memory after applying standard and novelty optimizations comparing to O0 (no optimization applied). 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 were 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. As well, We remark that the impact of applying standard optimizations on memory consumption for each program differs from one program to another. we can see that using O1 for bzip2e and O2, O3 for tiffmedian can even increase by almost 13 % the memory consumption (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.

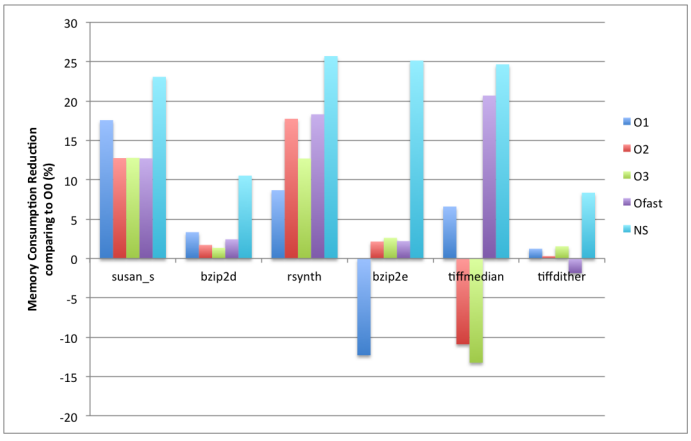

Fig. 6. Evaluating the amount of saved memory after applying standard optimization options comparing to new generated optimizations using NS

To study the correlation between execution time and memory consumption of running programs, we present in Figure 7 an evaluation of the speedup. We compared the speedup (accrording to O0) of the best optimization sequences by NS (gathered in Figure 6) and standard optimization options. The first observation is that optimizations yields high level

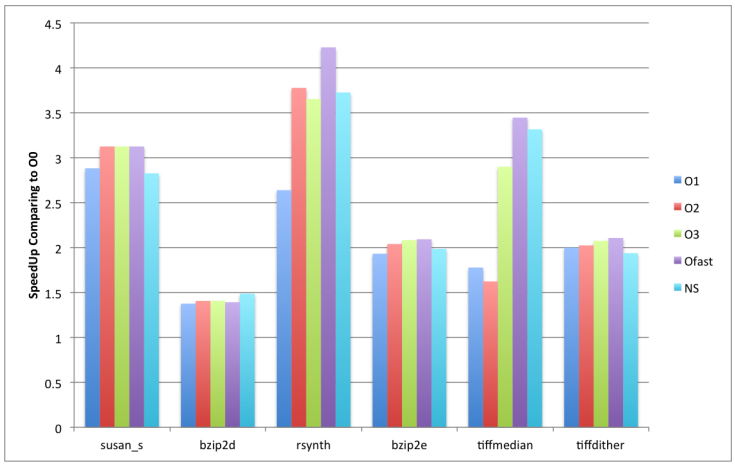

Fig. 7. Evaluating the speedup after applying standard optimization options comparing to new generated optimizations using NS

of speedup for all benchmarks (between 1.5 and 4.3). The second observation we can make is that different optimizations do not differ too much in their efficiency. We can 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 resource usage using NS does not have an impact on the performance of program.

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 softwares performance. Researches have investigated this optimization problem to catch relevant optimizations regrading performance, energy or code size improvements over standard optimization sequences. The vast majority of the work on iterative compilation focuses on increasing the speedup of new optimized code comparing to standard optimizations. It has been proven that optimizations are highly dependent on target platform and input program. In approach, we present a generic infrastructure that provides more understanding about resource consumption.

VII. Conclusion and Future work

In this paper we have described a new approach for testing and monitoring of code generators using a component-based infrastructure. We used a set of docker components in order to provide a fine-grained understanding of resource consumption. We investigated the problem of GCC compiler optimizations through the use of NS as search engine. Then, we studied the impact of optimizations on memory consumption and

execution time across two case studies. Results show that this approach is able to find good optimization sequences across different programs.

As a future work, we are planning to explore more trade-offs among resource usage metrics e.g. the correlation between CPU consumption and platform architectures. We aim as well to compare our findings using NS approach to different multi-objective evolutionary algorithms. Finally, our proposed docker-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 process.

REFERENCES

[1] 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.

[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] 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.

[4] 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.