NOTICE: A Framework for Non-functional Testing of Compilers

Mohamed Boussaa, Olivier Barais, and Benoit Baudry
Diverse team INRIA, Rennes, France
Email: {mohamed.boussaa, olivier.barais, benoit.baudry}@inria.fr

Gerson Sunyé
AtlanMod team INRIA, Nantes, France
Email: gerson.sunye@inria.fr

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

This paper proposes NOTICE, a component-based framework for non-functional testing of compilers through the monitoring of generated code in a controlled sand-boxing environment. We evaluate the effectiveness of our approach by verifying the optimizations performed by GCC compiler. Our experimental results show that our approach is able to auto-tune compilers according to user requirements and construct optimizations that yield to better performance results than standard optimization levels. We also demonstrate that NOTICE can be used to automatically construct optimization levels that represent optimal trade-offs between multiple non-functional properties such as execution time and resource usage requirements.

Keywords. *software quality, non-functional properties, compilers, testing.*

I. Introduction

Compiler users tend to improve software programs in a safe and profitable way. Modern compilers provide a broad collection of optimizations that can be applied during the code generation process. For functional testing of compilers, software testers generally use to re-execute a set of test suites on different optimized software versions and compare the functional outcome that can be either pass (correct behaviour) or fail (incorrect behaviour, crashes or bugs) [1]–[3].

For non-functional testing, improvement of source code programs in term of performance can refer to several different non-functional properties of the produced code such as code size, resource or energy consumption, execution time, among others [4], [5]. Testing non-functional properties is more challenging because compilers may have a huge number of potential optimization combinations, making it hard and time-consuming for software developers to find/construct the sequence of optimizations that satisfies user specific key objectives and criteria. It also requires a comprehensive understanding of the underlying system architecture, the target application and the available optimizations of the compiler.

In some cases, these optimizations may negatively decrease the quality of the software and deteriorate application performance over time [6]. As a consequence, compiler creators usually define fixed and program-independent sequence optimizations, which are based on their experiences and heuristics. For example, in GCC, we can distinguish optimization levels from O1 to O3. Each optimization level involves a fixed list of compiler optimization options and provide different trade-offs in term of non-functional properties. Nevertheless, there is no guarantee that these optimization levels will perform well on untested architectures or for unseen applications. Thus, it is necessary to detect possible issues caused by source code changes such as performance regressions and help users to validate optimizations that induce to performance improvement.

We note also that when trying to optimize software performance, many non-functional properties and design constraints must be involved and satisfied simultaneously to better optimize code. Most of previous works try to optimize a single criterion (usually the execution time) [7]–[9] and ignore other important non-functional properties, more precisely resource consumption properties such as memory or CPU usage, that must be taken into consideration and can be equally important in relation to the performance. Sometimes, improving program execution time can be so expensive at the expense of resource usage which may decrease system performance. For example, embedded systems for which code is generated often have limited resources. Thus, optimization techniques must be applied whenever possible to generate efficient code and improve performance (in term of execution time) with respect to available resources (in term of CPU or memory usage) [10]. Therefore, it is important to construct optimization levels that represent multiple trade-offs between non-functional properties, enabling the software designer to choose among different optimal solutions which best suit the system specifications.

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

- The paper introduces a novel formulation of compiler optimizations exploration problem using Novelty Search [11]. We evaluate the effectiveness of our approach by verifying the optimizations performed by GCC compiler. Our experimental results show that NOTICE is able to auto-tune compilers according to user choices (heuristics, objectives, programs, etc.) and construct optimizations that yield to better performance results than standard optimization levels.
- We propose a micro-service infrastructure to ensure the deployment and monitoring of different variants of optimized code. In this paper, we focus more on the relationship between runtime execution of optimized code and resource consumption profiles (CPU and memory usage) by providing a fine-grained understanding and analysis of compilers behavior regarding optimizations.
- We also demonstrate that NOTICE can be used to automatically construct optimization levels that represent optimal trade-offs between multiple non-functional properties, such as execution time, memory usage, CPU consumption, etc.

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

II. Motivation

A. Compilers Optimizations

In the past, researchers have shown that the choice of optimization sequences may impact software performance [4], [8]. 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], [4]. Indeed, each one of these optimizations interacts with the code and in turn with all other optimizations in complicated ways. Similarly, code transformations can either create or eliminate opportunities for other transformations and it is quite difficult for users to predict the effectiveness of optimizations on their source code program. As a result, most software engineering programmers that are not familiar with compiler optimizations find difficulties to select effective optimization sequences.

To explore the large optimization space, users have to evaluate the effect of optimizations according to a specific performance objective (see Figure 1). Performance can depend on different properties such as execution time, compilation time, resource consumption, code size, etc. Thus, finding the optimal optimizations combination for an input source code is a challenging, very hard, and time-consuming problem. Many approaches [2], [12]–[14] have attempted to solve this optimization selection problem using techniques such as genetic algorithms (GAs), machine learning, 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 [15]. Thus, a fine-grained understanding of resource consumption and analysis of compilers behavior regarding optimizations becomes necessary to ensure the efficiency of generated code.

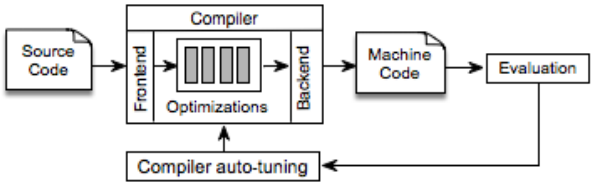

Fig. 1. Process of compiler optimizations exploration

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.

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. The diversity of available optimization options makes the design space for optimization level very huge, increasing the need for heuristics to explore the search space of feasible optimization sequences. For instance, we count 76 optimization flags that are enabled by 4 default optimization levels (O1, O2, O3, Ofast). For instance, O1 reduces code size and execution time without performing any optimizations that take a great deal of compilation time. It turns on 32 flags. O2 increases both compilation time and the performance of generated code. It turns on all optimization flags specified by O1 plus 35 other options. O3 is more aggressive level which enables all O2 options plus 8 more optimizations. Finally, Ofast is the most aggressive level which enables optimizations that are not valid for all standard-compliant programs. It turns on all O3 optimizations plus one more aggressive optimization. This results in a huge space with 2^{76} possible optimization combinations. The list of optimizations is available here [16]. 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.

III. Evolutionary Exploration of Compiler Optimizations

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

A. Novelty Search Adaptation

In this work, we aim at providing a new alternative for choosing effective compiler optimization options compared to the state of the art approaches. In fact, since the search space of possible combinations is too large, we aim at using a new search-based technique called Novelty Search [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 define. For instance, in a previous work [8], Chen et al. showed that handful optimizations may lead to higher performance than other techniques of iterative optimization. In fact, the fitness-based search may be trapped into some local optima that 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 [17], [18]. 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.

Generally, NS acts like GAs (Example of GA use in [19]). 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 (line 1 & 2). 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 optimization sequence yields to the lowest memory consumption so that, we can consider it as the best solution. Finally, genetic operators (mutation and crossover) are applied afterwards to fulfill the next population. This process is iterated until reaching the maximum number of evaluations.

Algorithm 1: Novelty search algorithm for compiler optimizations exploration

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

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 (see example in figure 2). As well, we keep the same order of invoking compiler flags since that does not affect the optimization process and it is handled internally by GCC.

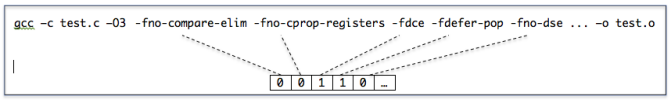

Fig. 2. Solution representation

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 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.

B. Novelty Search For Multi-objective Optimization

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

For instance, NS can be easily adapted to multi-objective problems. In this adaptation, the SBSE formulation remains the same as described above (Algorithm 1). However, in order to evaluate the new discovered solutions, we have to consider two main objectives and add the non-dominated solutions to the Pareto non-dominated set. We apply the Pareto dominance relation to find solutions that are not Pareto dominated by any other solution discovered so far, like in NSGA-II [20], [33]. Then, this Pareto non-dominated set is returned as the result. The metric used to evaluate one solution compared to previously discovered ones is called Hypervolume (HV) [24]. There is typically more than one optimal solution at the end of NS. The size of the final Pareto set yielded is bounded by the size of the initial population chosen.

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

In general, there are many non-functional properties that can be influenced by compiler optimizations, e. g., performance (execution time), code quality, robustness, etc. In this paper, we focus on the efficiency of optimized code in term of resource consumption (memory and CPU). Therefore, we need to deploy the test harness, i.e., the produced binaries, on an elastic infrastructure that provides to compiler users facilities to ensure the deployment and monitoring of different variants of optimized code. For this purpose, we propose NOTICE, a non-functional testing infrastructure based on System Container techniques such as Docker[1] environment.

Consequently, we rely on this technology and benefit from all its advantages to:

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

Before starting to monitor and test generated code, we have to describe the deployment environment of NOTICE.

A. System Containers as Deployment Environment

NOTICE represents a component-based infrastructure based on Docker Linux containers to monitor the execution of produced binaries by 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. Using Docker, we can define preconfigured applications and servers to host as virtual images. We can also define the way the service should be deployed in the host machine. A simple way to build images automatically is to use Dockerfiles which represents configuration files. We use Docker Hub[2] for building, saving, and managing all our Docker images. It represents a cloud-based registry service for building and shipping application or service containers. Once Docker images are defined, we can instantiate different containers.

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

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

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 resource constraints. So once execution is done, resources reserved for the container are automatically released to enable spawning next containers. Therefore, the host machine will not suffer too much from the performance trade-offs.

We resume, in the following, the main advantages of this approach:

1) The use of containers induces less performance overhead and resource isolation compared to using a full stack virtualization solution. Instrumentation tools for memory profiling like Valgrind [21] can induce too much overhead.
2) Thanks to the use of Dockerfile, NOTICE can be easily configured by compiler users to define the target compiler to test (e. g., GCC compiler version), the container OS, the input program under test and the optimization options, etc. Thus, NOTICE uses the same configured Docker image to execute different instances of generated code. For hardware architecture, containers share the same platform architecture as the host machine (e.g., x86, x64, ARM, etc.).
3) Docker uses Linux control groups (cgroups) to group processes running in the container. This allows us to manage the resources of a group of processes, which is very valuable. This approach increases the flexibility when we want to manage resources, since we can manage every group individually.
4) Although containers run in isolation, they can share data with the host machine and other running containers. Thus, non-functional data relative to resource consumption can be easily gathered and managed by other containers (i.e., for storage purpose, visualization)

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 related to resource usage.

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. Thus, 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 NOTICE run only one optimized version of code within each container.

To ease the monitoring process, NOTICE integrates 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 host machine. It will then have access to 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 used 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 resource usage monitoring, it becomes necessary to log it into a database at runtime. Thus, we link our monitoring component to a back-end database.

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 are continuously sent to this component. When a container is killed, NOTICE is able to access to its relative resource usage metrics through the database. We choose a time series database because we are collecting time series data that corresponds to the resource utilization profiles of programs 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 maximum memory usage of container "$generated\_code\_v1$" since its creation:

```
select max (memory_usage) from stats
```

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

where container_name='generated_code_v1'

To give an idea about data stored in InfluxDB, Table 1 describes the different stored metrics:

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

TABLE I
RESOURCE USAGE METRICS RECORDED IN INFLUXDB

Apart from that, NOTICE provides also information about the size of generated binaries and the compilation time needed to produced code. For instance, resource usage statistics are collected and stored using NOTICE. It would be nice to view resource consumption graphs within a complete dashboard. It is relevant to show performance profiles of memory and CPU consumption of our running programs overtime. To do so, we present a front-end visualization component for performance profiling.

3) Front-end Visualization Component: NOTICE provides a dashboard to run queries and view different profiles of resource consumption of running components through web UI. Thanks to this component, we can compare visually the profiles of resource consumption of running containers.

To do so, we choose Grafana[7], one of the best time-series visualization tools available for Docker. It is considered as a web application running within a container. We run Grafana 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, in Figure 3, an overall overview of the different components involved within NOTICE.

Our testing infrastructure will run different jobs within Docker containers. First, in the top level layer, we use NOTICE to generate different versions of code using our target compiler (e.g., GCC compiler). Then, we wrap generated code within multiple instances of our preconfigured Docker image. Each container will execute a specific job. For our case, a job represents a program compiled with a new optimization sequence (e.g., using 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

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

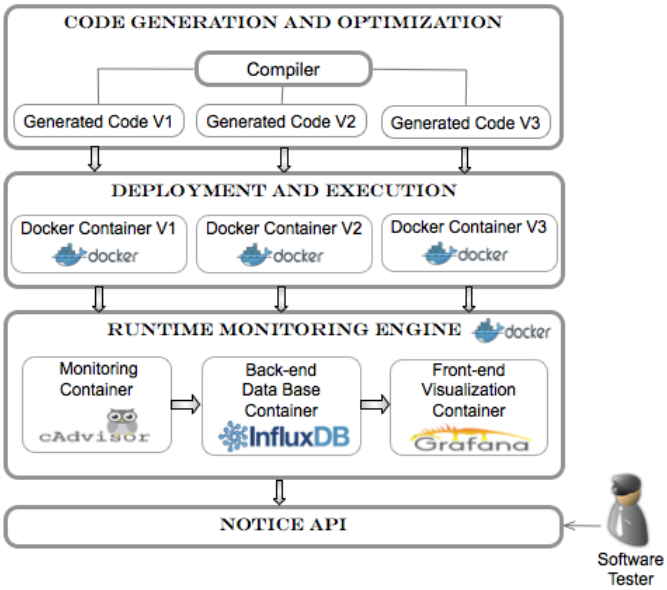

Fig. 3. NOTICE architecture overview

and draw up charts. The use of the front-end visualization component is optional and NOTICE can directly access to information stored in the database through REST API calls.

Remark. *We would notice that this testing infrastructure can be generalized and adapted to other case studies other than compilers (e.g., testing model-based code generators). Using Docker technology, any software application/generated code can be easily deployed within containers (i.e., by configuring the Docker image). It will be later executed and monitored using NOTICE monitoring components.*

V. EVALUATION

So far, we have presented a sound procedure and automated component-based framework for extracting non-functional properties of generated code. In this section, we evaluate the implementation of our approach by explaining the design of our empirical study; the research questions we set out to answer and different methods we used to answer these questions. The experimental material is available for replication purposes at [16].

A. Research questions

Our experiments aim at answering the following research questions:

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

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

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

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

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

B. Experimental Setup

1) Programs Used in the Empirical Study: To explore the impact of compiler optimizations a set of input programs are needed. To do so, we use a random C program generator called Csmith [15]. Csmith is a tool that can generate random C programs that statically and dynamically conform to the C99 standard. It has been widely used to perform functional testing of compilers [1], [3], [22]; not the case for checking non-functional requirements. Csmith can generate C programs that utilize a much wider range of C features including complex control flow and data structures such as pointers, arrays, and structs. Csmith programs come with their test suites that explore the structure of generated programs. Authors argue that Csmith is an effective bug-finding tool because it generates tests that explore atypical combinations of C language features. They also argue that larger programs are more effective for functional testing. Thus, we run Csmith for 24 hours and gathered the largest generated programs. We depicted 30 C programs with average source lines 12K.

Moreover, we run experiments on commonly used benchmarks in iterative compilation named Collective Benchmark (Cbench) [23]. 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. Selected Csmith and Cbench programs are described in more details at [16].

2) Parameters Tuning: An important aspect for meta-heuristic search algorithms lies in the parameters tuning and selection, which is necessary to ensure not only fair comparison, but also for potential replication. NOTICE implements 3 mono-objective search algorithms (NS, RS and GA [19]) and 3 multi-objective optimizations (NS, RS and NSGA-II [20]). Each initial population/solution of different algorithms is completely random. The stopping criterion is when the maximum number of fitness evaluations is reached. The resulting parameter values are listed in Table 2. The same parameter settings are applied to all algorithms under comparison.

NS, which is our main concern in this work, is implemented as described in Section 3. During the evolutionary process, each solution is evaluated using the novelty metric. Novelty is calculated for each solution by taking the mean of its 15 nearest optimization sequences in term of similarity (considering all sequences in the current population and in the archive). Initially, the archive is empty. Novelty distance is normalized in the range [0-100]. Then, to create next populations, an elite of the 10 most novel organisms is copied unchanged, after which the rest of the new population is created by tournament selection according to novelty (tournament size = 2). Standard genetic programming crossover and mutation operators are applied to these novel sequences in order to produce offspring individuals and fulfill the next population (crossover = 0.5, mutation = 0.1). In the meanwhile, individuals that get a score higher than 30 (threshold T), they are automatically added to the archive as well. In fact, this threshold is dynamic. Every 200 evaluations, we check how many individuals have been copied into the archive. If this number is below 3, the threshold is increased by multiplying it by 0.95, whereas if solutions added to archive are above 3, the threshold is decreased by multiplying it by 1.05. Moreover, as the size of the archive grows, the nearest-neighbor calculation that determines the novelty scores for individuals becomes more computationally demanding. So, to avoid having low accuracy of novelty, we choose to bound the size of the archive (archive size is 500). Hence, it follows a first-in first-out data structure which means that when a new solution gets added, the oldest solution in the novelty archive will be discarded. Thus, we ensure individuals diversity by removing old sequences that may no longer be reachable from the current population.

TABLE II
ALGORITHM PARAMETERS

Parameter	Value	Parameter	Value
Novelty nearest-k	15	Tournament size	2
Novelty threshold	30	Mutation prob.	0.1
Max archive size	500	Crossover	0.5
Population size	50	Nb generations	100
Individual length	76	Elitism	10
Scaling archive prob.	0.05	Solutions added to archive	3

Algorithm parameters were tuned individually in preliminary experiments. For each parameter, a set of values was tested. The parameter values chosen are the mostly used in the literature [11]. The value that yielded the highest performance score was chosen.

3) Evaluation Metrics Used: For mono-objective algorithms, we use to evaluate solutions using the following metrics:

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

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

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

When comparing two mono-objective algorithms, it is usual to compare their best solutions found so far during the optimization process. However, this is not applicable when

comparing two multi-objective evolutionary algorithms since each of them gives as output a set of non-dominated (Pareto equivalent) solutions. For this reason, we use performance indicator to compare multi-objective algorithms. Thus, for multi-objective algorithms we use to evaluate solutions using the following metric:

-*Hypervolume (HV)*: corresponds to the proportion of objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. The HV reference point is the point obtained by taking the maximum value observed. Thus, the HV metric can be computed as the area between the Pareto frontier and the HV reference point. Larger values for this metric mean better performance. The most interesting features of this indicator are its Pareto dominance compliance and its ability to capture both convergence and diversity [24].

4) Setting up infrastructure: To answer previous research questions, we configure NOTICE to run different experiments. Figure 4 shows a big picture of the testing and monitoring infrastructure considered in these experiments. First, a meta-heuristic (mono or multi-objective) is applied to generate specific optimization sequences for GCC compiler (step 1). During all experiments, we use GCC 4.8.4, as it is introduced in the motivation section, although it is possible to choose another compiler version using NOTICE since the process of optimizations extraction is done automatically. Then, we apply the generated optimization sequences to the input program under test and deploy the output binary within a new instance of our preconfigured docker image (step 2). While executing the optimized code inside the container, we collect at runtime performance data (step 4) and record it in a new time-series database using our InfluxDB back-end container (step 5). Next, NOTICE accesses remotely to stored data in InfluxDB using REST API calls and assigns new performance values to the current solution (step 6). The choice of performance metrics depends on experiment objectives (Memory improvement, speedup, etc.).

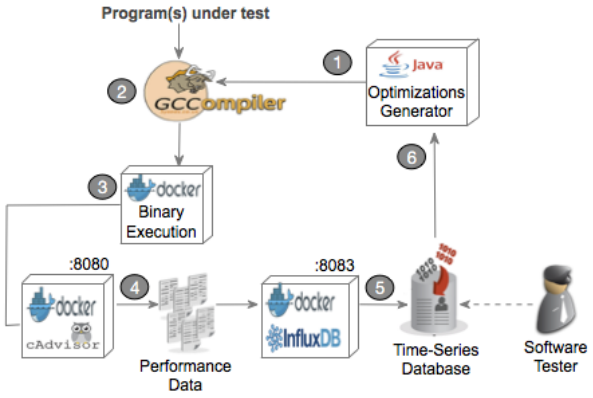

Fig. 4. NOTICE experimental infrastructure

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

C. Experimental Methodology and Results

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

1) RQ1. Mono-objective SBSE Validation:

a) Method: To answer the first research question RQ1, we implement three mono-objective search algorithms for compiler optimizations exploration (RS, GA and NS) in order to evaluate the non-functional properties of optimized code. We compare the performance results of different heuristics to standard GCC optimization levels (O1, O2, O3 and Ofast). In this experiment, we are trying to optimize for execution time (S), memory usage (MR) and CPU consumption (CR). Each non-functional property is improved separately and independently of other metrics.

As it is shown on the left-hand side of figure 5, given a set of input programs *"the training set"*, a list of optimizations and a performance objective, we use NOTICE to search for best optimization sequence. Thus, we generate a set of random Csmith programs *"the training set"* (10 programs) and apply generated sequences to these programs. In this setting, the code quality metric is equal to the average performance improvement (S, MR or CR) and that, for all programs under test.

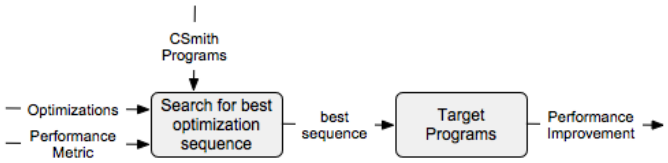

Fig. 5. fig

b) Results: results

-NS better than 3 algos

-conflicting results for standard levels

Key findings for RQ1.
blabla

2) RQ2. Sensitivity:

a) Method: Another interesting experiment is to test the sensitivity of *"training set"* programs to compiler optimizations and evaluate the general applicability of best optimal optimization sets previously discovered in RQ1. To do so, we apply best discovered optimizations to new unseen Csmith (20 random programs) and Cbench programs (20 programs) and we compare then, the performance improvements (see right-hand side of figure 5). The idea of this experiment is to test whether only Csmith programs are more sensitive to compiler optimizations or not. If so, this will be useful for compiler users and researchers to use NOTICE in order to build general optimization sequences from their representative training set programs.

b) Results: results

Key findings for RQ2.
blabla

3) RQ3. Impact of optimizations on resource consumption:

a) Method: To answer RQ3, we study the impact of standard optimization levels and best generated optimizations on memory and CPU consumption using NOTICE. Following again a mono-objective approach, we try in this experiment to maximize the speedup *S* per-benchmark and study, at the same time, the impact of speedup *S* on resource consumption namely memory footprint and CPU usage. In this experiment, we apply standard optimizations and different mono-objective heuristics individually to 5 Cbench programs and use NOTICE to profile applications in term of resource usage. The goal of this experiment is to: (1) use NOTICE infrastructure to provide an understanding of optimizations behavior, in term of resource consumption, when trying to optimize for execution time; (2) prove the usefulness of resource consumption reduction as a key objective for performance improvement.

b) Results: results

Key findings for RQ3.
blabla

4) RQ4. Trade-offs between non-functional properties:

a) Method: Finally, to answer RQ4, we use NOTICE to find trade-offs between non-functional properties. In this experiment, we choose to focus on the trade-off *<ExecutionTime–MemoryUsage>*. We report the comparison results of our NS adaptation for optimizations generation to the current state-of-the-art multi-objective approaches namely NSGA-II and RS.

Sequences evaluation is done across 5 Cbench programs as it is conducted in RQ1. We evaluate the quality of the obtained Pareto optimal optimization levels, both qualitatively by visual inspection of the Pareto frontiers, and quantitatively by using the HV metric.

b) Results: RESULTS

Key findings for RQ4.
blabla

D. Discussions

For RQ1, experiments take about 21 days to run all algorithms. These optimization times might seem long. However, it should be noted that this search can be conducted only once, since in RQ2, we show that best optimizations can be used with unseen programs of the same category as the training set, used to generate optimizations. This has to be proved with other case studies. As an alternative, it would be great to test model-based code generators. Code generators apply to same rules to generate new software programs. Thus, we can use NOTICE to define general-purpose optimizations from a set of generated code artifacts. Multi-objective search as conducted in RQ4, takes about 7 days, which we believe is acceptable for practical use. Nevertheless, speeding up the search speed may be an interesting feature for future research.

E. Threats to Validity

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

External validity refers to the generalizability of our findings. In this study, we perform experiments on different widely used benchmark programs in iterative compilation. We also use to generate 30 Csmith programs (10 as training set and 20 for evaluation). However, we cannot assert that the best discovered set of optimizations can be generalized to industrial applications since optimizations are highly dependent on input programs and target architecture. In fact, experiments conducted on RQ1 and RQ2 have to be replicated to other case studies to confirm our findings; and build general optimization sequences from other representative training set programs chosen by compilers users. Moreover, we build NOTICE to handle only optimizations performed by GCC versions, we did not investigate optimizations performed by other commonly used compilers, such as Clang or LLVM. In future work, we plan to provide more advanced version of NOTICE with multi-compiler evaluation.

Internal validity is concerned with the causal relationship between the treatment and the outcome. Meta-heuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. Are we providing a statistically sound method? or it is just a random result. Due to time constraints, we run all experiments only once. Following the state-of-art approaches in iterative compilation, previous works [2], [25] did not provide statistical tests to prove the effectiveness of their approaches. This is because experiments take too long time. However, we can deal with these internal threats to validity by performing at least 5 independent simulation runs for each problem instance.

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 using search-based techniques (SBSE) to guide the search toward relevant optimizations regrading performance, energy consumption, code size, compilation time, etc. Experimental results have been usually compared to standard compiler optimization levels. The vast majority of the work on iterative compilation focuses on increasing the speedup of new optimized code compared to standard compiler optimization levels. [2], [2], [4], [5], [8], [12]–[14], [25]–[29]. It has been proven that optimizations are highly dependent on target platform and input program. Compared to our proposal, none of previous works has studied the impact of compiler optimizations on resource usage. In this work, we rather focus on compiler optimizations related to resource consumption, while bearing in mind performance improvement.

Novelty Search has never been applied in the filed of iterative compilation. Our work presents the first attempt to introduce diversity in the compiler optimization problem. The idea of NS has been introduced by Lehman et al. [11]. It has been often evaluated in deceptive tasks and especially applied to evolutionary robotics [30], [31] (in the context of

neuroevolution). NS can be easily adapted to different research fields. In the field of software testing, NS, as we introduced, has been applied to only one work [32]. Authors in this work has adapted the general idea of NS to the test data generation problem where novelty score was calculated as the Manhattan distance between the different vectors representing test data. The evaluation metric of generated test suites is the structural coverage of code. In our NS adaptation, the evaluation metric represents the non-functional measurements and we are measuring the novelty score using the systematic difference between optimization sequences of GCC.

For multi-objective optimizations, we are not the first to address this problem. New approaches have emerged recently to find trade-offs between non-functional properties [2], [25], [33]. Hoste et al. [2], which the most related work to our proposal, propose COLE, an automated tool for optimizations generation using a multi-objective approach namely SPEA2. In their work, they try to find Pareto optimal optimization levels that present a trade-off between execution and compilation time of generated code. Their experimental results show that the obtained optimization sequences perform better than GCC standard optimization levels. NOTICE provides also a fully automated approach to extract non-functional properties. However, NOTICE differs from COLE because first, our proposed container-based infrastructure is more generic and can be adapted to other case studies (i.e., compilers, code generators, etc.). Second, we provide facilities to compiler users to extract resource usage metrics using our monitoring components. Finally, our empirical study investigates different trade-offs compared to previous works in iterative compilation.

VII. Conclusion and Future Work

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

In this paper, we have introduced first, a novel formulation of the compiler optimization problem based on Novelty Search. The idea of this approach is to drive the search for best optimizations toward novelty. This work presents the first attempt to introduce diversity in iterative compilation. Experiments have shown that Novelty Search can be easily applied to mono and multi-objective search problems. In addition, we have reported the results of an empirical study of our approach compared to different state-of-the-art approaches, and the results obtained have provided evidence to support the claim that Novelty Search is able to generate effective optimizations.

Second, we have presented an automated tool for automatic extraction of non-functional properties of optimized code, called NOTICE. NOTICE applies different heuristics (including Novelty Search) and performs non-functional testing of compilers through the monitoring of generated code in a controlled sand-boxing environment. In fact, NOTICE uses a set of micro-services to provide a fine-grained understanding of optimization effects on resource consumption. We evaluated the effectiveness of our approach by verifying the optimizations performed by GCC compiler. Results showed that our approach is able to automatically extract information about memory and CPU consumption. We were also able to find better optimization sequences than standard GCC optimization levels.

As a future work, we plan to explore more trade-offs among resource usage metrics e. g., the correlation between CPU consumption and platform architectures. We also intend to provide more facilities to NOTICE users in order to test optimizations performed by modern compilers such as Clang, LLVM, etc. Finally, NOTICE can be easily adapted and integrated to new case studies. As an example, we would inspect the behavior of model-based code generators since different optimizations can be performed to generate code from models [34]. Thus, we aim to use the same approach to find non-functional issues 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] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[2] 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.

[3] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 216–226.

[4] 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.

[5] 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.

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

[7] P. A. Ballal, H. Sarojadevi, and P. Harsha, "Compiler optimization: A genetic algorithm approach," *International Journal of Computer Applications*, vol. 112, no. 10, 2015.

[8] 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.

[9] M. Demertzi, M. Annavaram, and M. Hall, "Analyzing the effects of compiler optimizations on application reliability," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 184–193.

[10] 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.

[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] 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.

[13] 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.

[14] 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.

[15] 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.

[16] "Notice settings," https://noticegcc.wordpress.com/.

[17] 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.

[18] 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.

[19] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 7–22, 2002.

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, 2002.

[21] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[22] E. Nagai, A. Hashimoto, and N. Ishiura, "Scaling up size and number of expressions in random testing of arithmetic optimization of c compilers," in *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, 2013, pp. 88–93.

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

[24] K. Deb, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001, vol. 16.

[25] A. Martínez-Álvarez, J. Calvo-Zaragoza, S. Cuenca-Asensi, A. Ortiz, and A. Jimeno-Morenilla, "Multi-objective adaptive evolutionary strategy for tuning compilations," *Neurocomputing*, vol. 123, pp. 381–389, 2014.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] P. Lokuciejewski, S. Plazar, H. Falk, P. Marwedel, and L. Thiele, "Multi-objective exploration of compiler optimizations for real-time systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*. IEEE, 2010, pp. 115–122.

[34] 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.