DECO: Diversity-based Exploration of Compiler Optimizations

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*—In general, most of compiler optimizations focus on improving the execution time and performance of compiled code. However, this 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. So, a fine-grained understanding of resource consumption and analysis of compilers behavior regarding optimizations becomes necessary. As well, compilers may have a huge number of optimization combinations and it very hard and time-consuming to find the optimal sequence of optimization options that satisfies user key objective. In this paper, we propose DECO (Diversity-based Exploration of Compiler Optimizations), an automatic generator of compiler optimization sequences based on sequences diversity. In this approach, we apply the Novelty Search (NS) technique to determine the optimal sequence that could be used to produce a code which consumes less system resources. To do so, we explore the search space of possible optimization combinations by considering diversity of sequences as the unique objective function to be optimized. In fact, instead of having a fitness-based selection, we select optimization sequences based on a novelty score showing how different they are compared to all other solutions evaluated so far. We conduct experiments by providing a quantitative study of the impact of compiler optimizations explored by NS on non-functional properties like CPU and memory consumption. We run experiments on a commonly used set of benchmarks and we identified the optimal set of optimizations regarding performance. The results show that...**

Keywords. *iterative compiler optimization, non-functional testing, novelty search.*

I. Introduction

Compilers tend to improve source code in a safe and profitable way. Improvement of source program can refer to several different characteristics of the produced code such size, energy consumption, execution time, among others. 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 environments like embedded systems.

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. By the past, researchers have shown that the choice of optimization sequences can have an effect on software performance. However, these universal and predefined sequences may not produce always good performance results and may be highly dependent on the benchmark and the source code they have been tested on [1] [2].

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.

As well, compilers may have a huge number of optimization combinations and it very hard and time-consuming to find the optimal sequence of optimization options that satisfies user key objective. For example, GCC 4.8.4 has around 76 option flags that can be used in various optimization levels (-O1, -O2, -O3, -Ofast), and most of them are turned off by default. This results in a huge space with 2^{76} possible optimization combinations.

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. So, a fine-grained understanding of resource consumption and analysis of compilers behavior regarding optimizations becomes necessary.

In this paper we want to explore the relationship between runtime execution of optimized code and resource consumption. Thus, we propose DECO, Diversity-based Exploration of Compiler Optimization options, an automatic generator of compiler optimization sequences based on sequences diversity. In this approach, we apply the Novelty Search (NS) algorithm to automate this process and determine the optimal sequence

that could be used to produce a code which consumes less system resources. To do so, we explore the search space of possible optimization combinations by considering diversity of sequences as the unique objective function to be optimized. In fact, instead of having a fitness-based selection, we select optimization sequences based on a novelty score showing how different they are compared to all other solutions evaluated so far. We conduct experiments by providing a quantitative study of the impact of compiler optimizations explored by NS on non-functional properties like CPU and memory consumption. We run experiments on a commonly used set of benchmarks and we identified the optimal set of optimizations regarding performance. The results show that...

The primary contribution of this paper can be summarized as follows: (1) The paper proposes a diversity-based technique for exploring the space of possible compiler optimization sequences called novelty search and, to the best of our knowledge, this is the first paper in the literature to introduce diversity in iterative compiler optimization; (2) The paper proposes also a docker-based infrastructure to perform a fine-grained analysis of resource consumption of optimized code; (3) We report results of comparison between generated sequences by NS and standard optimization sequences in GCC across a set of valuable benchmarks.

The paper is organized as follows: section 2 describes the related work. The approach overview and the NS adaptation for compiler optimization are presented in Section 3. We describe the experimental setup and methodology in section 4. The results of our experiments are discussed in Section 5. Finally, concluding remarks and future work are provided in Section 6.

II. Previous work

A. Compilers' Optimization Techniques

We already mentioned in the introduction that many studies have investigated the effect of different compiler optimizations on applications performance. Generally, authors tried to explore the search space of optimization sequences based on a specific key objective such as execution time, compilation time, energy consumption,etc.

B. Iterative Optimization

Finding effective compilation sequences [1]
Cole: compiler optimization level exploration [2]
Compiler Optimization: A Genetic Algorithm Approach [3]
Performance potential of optimization phase selection during dynamic JIT compilation [4]
Fast and effective orchestration of compiler optimizations for automatic performance tuning [5]
Tuning compiler optimization options via simulated annealing [6]
Identifying compiler options to minimize energy consumption for embedded platforms [7]
Deconstructing iterative optimization [8]
Automatic selection of compiler options using genetic techniques for embedded software design [9]
A Genetic Algorithm approach towards compiler flag selection based on compilation and execution duration [10]
Exploration of compiler optimization sequences using clustering-based selection [11]
Evaluating iterative optimization across 1000 datasets [12]
MILEPOST GCC: machine learning based research compiler [13]
Compiler optimizations for low power systems [14]
Is Compiling for PerformanceCompiling for Power? [15]
Analyzing the effects of compiler optimizations on application reliability [16]
Automatic selection of GCC optimization options using a gene weighted genetic algorithm [17]
Evaluation of GCC Optimization Parameters [18]
Compiler-based optimizations impact on embedded software power consumption [19]
Post-compiler software optimization for reducing energy [20]
A REST Service Framework for Fine-Grained Resource Management in Container-Based Cloud [21]

C. Novelty search

The idea of NS, introduced by Lehman and Stanley in 2008 [22]. In this approach, individuals in an evolving population are selected based on how different they are compared to other solutions evaluated so far. They also argue that objective fitness functions can be deceptive, leading the evolutionary search to local maxima rather than towards the goal. In contrast, NS ignores the objective and simply searches for novel behavior, and therefore cannot be deceived. So mainly, NS acts like GAs. However, NS needs extra changes. First, NS add a measure of individuals behavior to the GA. This depends on the context of the search and the way we represent individuals. Then, a new novelty metric is used to reward individuals with different behavior from past-discovered solutions. Finally, an archive must be added to the algorithm which is a kind of a data base that remembers individuals that were highly novel when they were discovered in past generations. Novelty search has been generally evaluated in deceptive tasks. In fact, It was often applied to evolutionary robotics (in the context of neuroevolution) [23] [24].

III. Novelty Search Compiler Optimization Exploration

The goal of the novelty search approach for compiler optimization 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. 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. [8]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. Meta-heuristic algorithms strive to escape from the local optima in multi-modal search space by using different techniques. 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.

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.

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

```
initialize_archive(archive, L)
population ← random_sequences(S)
repeat
    for sequence ∈ population do
        for program ∈ benchmark do
            perfomance ← execute(sequence, program)
        end for
        novelty_metric(sequence) ←
        distFromKnearest(archive, population, K)
        if novelty_metric > T then
            archive ← archive ∪ sequence
        end if
    end for
    new_population ←
    generate_new_population(population)
    generation ← generation + 1
until generation = N
return best_sequence
```

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. 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.

B. Compiler optimization options

The compiler optimization level design space is very huge which is the motivation for proposing a novelty search algorithm for exploring the search space. GCC exposes its various optimizations via a number of flags that can be turned on or off through command-line compiler switches. In our experimental setup, we use the GNU Compiler Collection (GCC) 4.8.4 compiler. This compiler provides a wide range of command-line optimizations that we can enable or disable (it includes more than 150 options for optimization). We choose to evolve only optimization options that are appearing in the default standard optimization levels O0-O3 of GCC. There are 78 options. 10 options are enabled by default and the remaining options are defined within O1, O2, O3 and Ofast levels. 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, the goal is to investigate different combinations of optimization sequences using novelty search in order to explore as much as possible the search space. To do so, we run the generated sequences on different selected benchmarks described above and catch most effective sequences.

//explain how levels are nested

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		

IV. EXPERIMENTAL SETUP

A. Benchmarks

To explore the impact of compiler optimizations a set of test input programs are needed. We collected a set of programs from different benchmarks. We run experiments on commonly used benchmarks in previous works. These programs are a mix of different sources. They have been chosen based on different characteristics. First, programs should be a greedy in term of resources so that we can inspect and study the effect of optimizations on produced executables. Then, we should consider also programs that target multi-core platforms. The media based benchmarks are suitable for these settings since they usually rely on a large library and large processing files. So, we choose n programs form MediaBanch benchmark with high memory and cpu consumption. It is composed of complete applications originating from the image/video processing, communications and DSP fields. As well, for the multi-core platforms we used n programs from ParMiBench. We choose other classic programs usually used in previous works from MiBench benchmark like (give programs names). //provide a table containing the characteristics of input programs

B. Novelty Parameters

Our experiments use the classical Novelty Search algorithm, where we evolve a set of optimization sequences through generations. Novelty search is implemented as described in Section X.

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.

TABLE II
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	1000
Individual length	76	Elitism	10
Scaling archive prob.	0.05	New in archive z	3

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. The value that yielded the highest performance scores was chosen. The resulting parameter values are listed in Table 2.

C. Docker as a deployment environment

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 goal 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.

/* Docker uses 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.

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

D. Monitoring environment

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 our experiments, we focus on the CPU and memory consumption of each container so these metrics 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.

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 extraction of performance metrics using cgroups file systems at runtime. Thus, cAdvisor collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters, historical resource usage, and histograms of complete historical resource usage. We note that resource usage information is collected in raw data.
//give references to papers that used cadvisor

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.

For that purpose, we use InfluxDB, a time series data base as a 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.

V. Evaluation

In this section we evaluate the implementation of our approach with several experiments. To do so, 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 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 induce to low consumption of memory and CPU resources.

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 input Benchmark programs. Then, we evaluate the memory footprint and CPU consumption of programs and we compare the results. The goal of this initial experiment is to provide a fine-grained 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 novelty search approach for automatic optimization sequences generation by comparing the results founded in first experiments 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.

A. Impact of standard GCC optimization levels on resource consumption

In this experiment we assess the impact of standard optimization levels on the resource consumption. Before running our programs using optimization options generated by novelty search, we compile first the different benchmarks introduced in section X using standard optimization options. We place the input programs in shared volume between the host machine and the containers. In Docker environment, we call this repository the Data Volume. It is a specially-designated directory within one or more containers that share data with the host machine. So once a container is created it will execute the needed binary from this volume repository.

Monitoring information should also be provided to inform about resource utilization required/needed and to automate the resource management

B. Evaluation of the diversity-based exploration of compiler optimizations

VI. Experimental Results

A. Execution Time

B. Memory Usage

C. CPU Consumption

VII. Conclusion, limitations and Future work

REFERENCES

[1] 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.

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

[4] M. R. Jantz and P. A. Kulkarni, "Performance potential of optimization phase selection during dynamic jit compilation," *ACM SIGPLAN Notices*, vol. 48, no. 7, pp. 131–142, 2013.

[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] 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.

[7] 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.

[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. 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.

[10] 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.

[11] 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.

[12] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu, "Evaluating iterative optimization across 1000 datasets," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 448–459.

[13] 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.

[14] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler optimizations for low power systems," in *Power aware computing*. Springer, 2002, pp. 191–210.

[15] M. Valluri and L. K. John, "Is compiling for performancecompiling for power?" in *Interaction between Compilers and Computer Architectures*. Springer, 2001, pp. 101–115.

[16] 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.

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

[18] R. D. Escobar, A. R. Angula, and M. Corsi, "Evaluation of gcc optimization parameters," *Revista Ingenierias USBmed*, vol. 3, no. 2, pp. 31–39, 2015.

[19] M. E. Ibrahim, M. Rupp, and S.-D. Habib, "Compiler-based optimizations impact on embedded software power consumption," in *Circuits and Systems and TAISA Conference, 2009. NEWCAS-TAISA'09. Joint IEEE North-East Workshop on*. IEEE, 2009, pp. 1–4.

[20] 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.

[21] L. Li, T. Tang, and W. Chou, "A rest service framework for fine-grained resource management in container-based cloud," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 645–652.

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

[23] 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.

[24] 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.