Université Libre de Bruxelles

Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle

Automatic configuration of GCC using irace

L. Pérez Cáceres, F. Pagnozzi, A. Franzin, and T. Stützle

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2017-008

June 2017
Last revision: October 2017

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:
IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2017-008

Revision history:

TR/IRIDIA/2017-008.001	June 2017
TR/IRIDIA/2017-008.002	October 2017

The information provided is the sole responsibility of the authors and does not necessarily reflect the opinion of the members of IRIDIA. The authors take full responsibility for any copyright breaches that may result from publication of this paper in the IRIDIA – Technical Report Series. IRIDIA is not responsible for any use that might be made of data appearing in this publication.

Automatic configuration of `GCC` using irace

Leslie Pérez Cáceres, Federico Pagnozzi, Alberto Franzin, and Thomas Stützle

IRIDIA, CoDE
`leslie.perez.caceres@ulb.ac.be`, `federico.pagnozzi@ulb.ac.be`,
`alberto.franzin@ulb.ac.be`, `stuetzle@ulb.ac.be`
Université Libre de Bruxelles,
Brussels, Belgium

Abstract. Automatic algorithm configuration techniques have proved to be successful in finding performance-optimizing parameter settings of many search-based decision and optimization algorithms. A recurrent, important step in software development is the compilation of source code written in some programming language into machine-executable code. The generation of performance-optimized machine code itself is a difficult task that can be parametrized in many different possible ways. While modern compilers usually offer different levels of optimization as possible defaults, they have a larger number of other flags and numerical parameters that impact properties of the generated machine-code. While the generation of performance-optimized machine code has received large attention and is dealt with in the research area of auto-tuning, the usage of standard automatic algorithm configuration software has not been explored, even though, as we show in this article, the performance of the compiled code has significant stochasticity, just as standard optimization algorithms. As a practical case study, we consider the configuration of the well-known GNU compiler collection (`GCC`) for minimizing the run-time of machine code for various heuristic search methods. Our experimental results show that, depending on the specific code to be optimized, improvements of up to 40% of execution time when compared to the `-O2` and `-O3` optimization flags is possible.

Keywords: irace, automatic configuration, parameter tuning, gcc

1 Introduction

The performance of computational procedures such as optimisation algorithms is commonly a major concern for both developers and users. Algorithm performance depends on several elements related to the characteristics of the algorithm itself, the particular problem being solved and the environment in which the execution will be performed. Normally, algorithms expose parameters that allow the user to adjust their behaviour to the problem solved and the execution circumstances. Generally, parameter settings or *configurations* have a strong impact on the performance of algorithms and, consequently, finding high-performing configurations is essential to reach peak algorithm performance. The process of finding good configurations can be challenging, requiring significant expertise, consuming large amounts of time and be computationally expensive given the large number of parameters that some algorithms have. To alleviate this situation, a number of automatic algorithm configuration tools, also called *configurators*, have been proposed in the literature. Examples include ParamILS [15], SMAC [14], GGA [2] and irace [17]. These tools have shown to be able to obtain very good parameter settings when configuring different types of algorithms, in some cases strongly improving the performance obtained by expert-defined default parameter settings [12,24]. Configurators aim at efficiently using the available computational resources to search the parameter space for high-performance parameter settings. The use of such tools can enable developers to test and make available more design features, and provides a simple and general approach to obtain the best performance of an algorithm.

Very often the configuration of optimisation algorithms is limited to their parameter settings, overlooking the code compilation as an element that can affect algorithm performance. On the contrary, in contexts where producing high-performance and portable code is very important, this issue is widely acknowledged. Several projects define application-specific "autotuners" that adjust the program produced to the system on which it will be installed. Examples of such work include ATLAS [25], Spiral [21], FFTW3 [6] and Patus [5]. On the other side, compilers such as GCC [8], provide several optimisation options that aim at improving the quality of the generated executable. Default levels of optimisation that activate different sets of options are defined in GCC using the `-Ox` settings. These optimisation levels are defined in a general fashion and further improvements are possible depending on the operations performed by the compiled code and the architecture in which the algorithm is executed. In this sense, GCC is like any other computational procedure whose performance can be optimised by setting its parameters. Selecting the optimisation options of GCC, or any other compiler, to obtain the best executable performance is itself an algorithm configuration problem, which has been tackled by methods such as OpenTuner [1] and COLE [13], or in more specific approaches [4], [7] and [20]. Consequently, when configuring optimisation algorithms, it is possible to optimise the compilation of the code as part of the algorithm configuration process.

The irace package [17] is an automatic algorithm configurator that provides an implementation of iterated F-race [3] and other approaches to automatic configuration. Irace is freely available as an R package, it provides several options to adjust the configuration process (e.g. parallel execution) and it does not require knowledge of R or about the inner workings of irace itself. Irace has been widely applied in the literature and is a state-of-the-art algorithm configurator. In this work, we exploit irace to configure the optimisation options of GCC for minimising the execution time on six different benchmarks that execute different optimisation algorithms. This work shows that (i) the running time of optimisation algorithms can be further improved by configuring the compilation options, (ii) the best options depend on the benchmark and the machine used, and (iii) irace is a suitable configurator to configure the compilation process. Beyond these contributions, irace can be used to improve the execution time of any other compiled code and we will make our procedure, which is tested here on the most recent stable version of GCC, version 7.1, available for other users.

The remainder of this article is organised as follows. First, section 2 describes the algorithm configuration problem, gives details of the configuration process performed by irace and discusses related work. The details of the compilation configuration benchmarks used are provided in Section 3 and Section 4 gives an analysis of the base performance of the compilation of GCC for the different benchmarks. In Section 5, we provide the experimental results obtained by irace when configuring the compilation options of GCC and we analyse them. Finally, we discuss future work and conclude in Section 6.

2 Automatic algorithm configuration

The configuration of algorithms, is the task of finding a set of algorithm parameter values, also called *algorithm configuration*, that exhibit good empirical performance for a particular class of problem instances. Algorithm configuration can be defined as an optimization problem over a parameter search space, which has the goal of identifying parameter settings that maximize algorithm performance. In general, an algorithm configuration scenario defines: a *parameter search space* consisting of the algorithm parameters defined as variables and their domains, a set of *training* and *test* instance sets, and a total configuration *budget*. Broadly, parameters can be classified in two types: (i) parameters that indicate the selection of algorithmic components (e.g. the crossover operator for an evolutionary algorithm, or the branching strategy for an exact algorithm), and (ii) parameters that control the behavior of algorithmic components (e.g. the length of a tabu list, or the size of a perturbation) and whose domain commonly correspond to numbers of a discrete or continuous domain. The type of parameters are commonly best represented by *categorical* or *ordinal* variables, while the former as *numerical* ones. Additionally, parameters can have *conditional* relations that is, their use depends on the value of other parameters (e.g. the use of the tabu list length parameter is conditional to the selection of tabu search as local search). The *homogeneity* of a scenario indicates the degree of consistency of the relative performance of configurations in the parameter space across the instance set. Highly homogeneous scenarios, have configurations that are consistently good (or bad) for all problem instances, while in heterogeneous scenarios certain configurations perform best in a subset of the instances and poorly in other instances. An algorithm can be configured for optimizing different performance measures e.g. solution quality, running time, SAT count, etc. The evaluation of the performance of a configuration is commonly defined as the aggregation of the selected performance measure over the instance set (commonly the mean or median). Given that optimization algorithms are often stochastic, the real performance of configurations can be only estimated and several repetitions are required in order to have a precise estimation.

The irace package [17] is an algorithm configurator that implements configuration procedures based on iterated racing, e.g. iterated F-race [3]. Irace is a general-purpose configurator and it only requires the definition of a configuration scenario as described above. The supported parameter types are *categorical*, *integer*, *real* and *ordered* (a categorical parameter that defines a precedence of values in its domain). Irace iteratively applies a racing procedure in which several configurations are incrementally evaluated on bigger subsets of the training instance set. Statistical tests (the Friedman test by default) are performed to identify configurations that obtain poor performance. These poor configurations are eliminated from the race and the execution continues with the surviving configurations until the termination criterion of the iteration is met. After each iteration, new configurations are sampled from a probabilistic model that is updated to be centered around the best configurations obtained in the previous iteration (*elites*). This way, irace iteratively converges to high-performing areas of the parameter search space, while increasing the precision of the performance estimation by increasing the number of instances in which the elite configurations are evaluated. For more details about irace we refer to [17].

As already mentioned, the configuration of algorithms is a main concern when developing and applying algorithms. The use of automatic configuration tools not only facilitates the algorithm configuration process but also allows developers and researchers to focus on proposing and improving techniques for a wide scope of scenarios, while delegating the tedious configuration task to specialized tools. Several approaches can be found in the literature to automatically generate code and optimize compilation options. These approaches optimize programs to obtain the best performance in particular architectures, multicore or cluster environments, GPU, etc. Most of these methods are program-specific techniques that apply expert knowledge to implement procedures designed for particular scenarios. Examples of such techniques are ATLAS [25] for linear algebra software, Spiral [21] for digital signal processing algorithms, FFTW3 [6] for discrete Fourier transform computation and Patus [5] for stencil computations. As most computational procedures, compilers have parameters that can be optimised thus, a number of approaches have been developed to configure compiler options. An example of such initiatives is ACOVEA [16], an open-source project that currently is not in development. ACOVEA implemented a genetic algorithm based approach to configure the GCC compiler options to obtain lower execution times. Similarly, Milepost GCC [7] is an open-source project that aims at using machine learning techniques to learn high-performing GCC settings, with the aim of reusing this knowledge to improve the performance of programs in particular architectures. The *Tool for automatic compiler tuning* (TACT) [20] is a genetic-based compiler configurator. TACT supports the configuration for single and multiple objectives by obtaining a pareto-optimal set of configurations.

A general autotuning framework, called OpenTuner, is proposed in [1] and applied to configure the options of GCC. OpenTuner provides a framework where domain-specific configuration procedures can be instantiated, while also offering several general-purpose features to configure computational programs. OpenTuner was used to instantiate a specialized autotuner to configure the optimization options of GCC, the approach obtained considerable speed ups of execution performance in several scenarios. A drawback of OpenTuner, from the perspective of the automatic configuration of optimization algorithms, is that it does not provide an explicit method to handle the stochastic behavior of algorithms or the evaluation of problem instances. The evaluation of the configurations is fully delegated to the user and therefore, the use of problem instances and repetitions in the evaluation must be handled by the user. COLE [13], is a compiler optimizer that implements an evolutionary algorithm based on SPEA2 [27]. COLE is able to optimize the compilation for mutiple objectives (e.g. running time and memory use) by searching pareto-optimal sets of compilation options. In [13], COLE was used to configure the optimization flags of `GCC` (version 4.1.2), compiling the SPEC CPU2000 benchmarks [11]. The results showed that the default optimization levels of `GCC` could be strongly improved. In this work, we evaluate the use of irace to configure the optimization parameters of `GCC` for optimizing the performance obtained by 6 optimization algorithms. We do not apply any `GCC`-dependent processing mechanism to the evaluation of configurations; in this regard, we apply irace like for any other configuration scenario. We argue that, since the evaluation of the compilation with `GCC` is a stochastic procedure, irace is an adequate method to perform the configuration of the `GCC` optimization options. The experiments show that irace can significantly improve the performance obtained by the tested optimization algorithms without requiring any specific knowledge about the compilation process or the targeted algorithms.

3 Configuration scenarios

The experiments presented in this paper configure the optimization options of `GCC` [8] to compile a set of optimization algorithms benchmarks. We use `GCC` version 7.1, the optimization options considered in the configuration were obtained from the documentation of `GCC`[1]. The total number of `GCC` parameters selected are 367 categorical and integer parameters. Enabling the optimization options in `GCC` requires to select an optimization level, thus, the set of `GCC` parameters includes a parameter to select the optimization level (`-O1`, `-O2` or `-O3`). We define two types of `GCC` configuration scenarios by making two sets of parameters:

– $\mathtt{GCC}_{\text{flags}}$: 171 categorical parameters consisting of only options that enable/disable main optimization options.
– $\mathtt{GCC}_{\text{flags+num}}$: 366 parameters, 173 categorical and 193 integer.

The domains in the `GCC` parameter definition do not provide the upper bound of some numerical parameter domains. In these cases we set as upper bound the default value of the parameter multiplied by a constant (4 in this work). Configurations that generate invalid executables or failed in the compilation were penalized returning a large numerical value to irace. Other types of specific error handling were not implemented. The compilation performed by `GCC` is optimized using the $\mathtt{GCC}_{\text{flags}}$ and $\mathtt{GCC}_{\text{flags+num}}$ parameter sets in 6 optimization algorithms benchmarks, resulting in 12 different configuration scenarios. The optimization algorithms were executed with fixed parameter settings and fixed evaluation budgets (e.g. fixed number of iterations or solutions generated) and each benchmark defines a set of training and test instances. The goal of the configuration process is to find `GCC` parameter settings that minimize the execution time of the benchmark algorithms (using the defined fixed settings). The optimization algorithm benchmarks are the following:

ACOTSP: framework of ant colony optimization algorithms [23] for solving the traveling salesman problem (TSP). ACOTSP is implemented in `C`. The

[1] The `GCC` optimization options are available at `https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/Optimize-Options.html` and the parameter definition can be obtained in the `params.def` file in the source code of `GCC`.

compilation is configured on a training set of 20 TSP instances and evaluated on 100 TSP instances of sizes 1000 and 1500. We configure two versions of ACOTSP, one that applies 3-opt local search to each tour built (ACOTSP ls3) and a version without local search (ACOTSP ls0).

ILS: iterated local search implementation for solving TSP. ILS is implemented in `C`. The compilation is configured on a training set of 10 TSP instances and evaluated on 50 TSP instances of size 1500.

LKH: state-of-the-art Lin-Kernighan heuristic implementation by K. Helsgaun [9], [10]. LKH is implemented in `C`. The compilation of this algorithm is configured on a training set of 10 TSP instances and evaluated on 50 TSP instances of size 1000.

TS: tabu search implementation for solving the quadratic assignment problem (QAP). TS is implemented in `C`. The compilation is configured on a training set and test set of 50 QAP instances.

EMILI: Iterated greedy algorithm instantiated with the EMILI framework for solving the permutation flowshop problem (PFSP). EMILI is implemented in `C++`. The compilation was configured on a training set of 30 PFSP instances with 20 machines and 50 to 100 jobs and evaluated on 120 PFSP instances with 5 to 20 machines and 20 to 500 jobs.

The execution of `irace` was given a configuration budget of 10 000 evaluations, the statistical test used was the t-test and the performance of the candidate configurations corresponds to the execution time of the optimization algorithm benchmarks. We have chosen not to provide an initial `GCC` configuration to `irace`, which is commonly done when good parameter settings are known (for `GCC`, the `-O3` or `-O2` options are possible initial settings). We omit the initial configurations to evaluate the ability of `irace` to find high-performing configurations without additional information of the parameter space. The main tests were run under Cluster Rocks 6.2, which is based on CentOS 6.2. The machines used were:

– *m1*: 2 AMD Opteron (2.4 GHz), 2 cores, 1 MB cache and 4 GB RAM.
– *m2*: 2 Intel Xeon (2.33 GHz), 4 cores, 6 MB cache and 8 GB RAM.
– *m4*: 2 AMD Opteron (2.1 GHz), 16 cores, 16 MB cache and 64 GB RAM.
– *m5*: 2 Intel Xeon (2.5 GHz), 12 cores, 16 MB cache and 128 GB RAM.

The *m2* machine was used by default to perform the experiments unless specified otherwise. More details about the configuration scenarios are available in the supplementary material provided with this paper [19].

4 `GCC` configuration scenarios analysis

Our premise is that the optimization options of `GCC` can improve greatly the performance of the described benchmarks. In order to prove this, we perform experiments to compare the performance of the benchmarks when compiled by `GCC` with and without optimization options enabled. Table 1 gives the speed up obtained by using the options `-O3`, `-O2` and `-O1` for `GCC`, respectively. We can observe that the speed up obtained by the optimization options is strongly influenced by the benchmark. Nevertheless, all benchmarks improve their performance by using the optimization options. EMILI is the benchmark that shows

Table 1. Speed up of 10 executions of the benchmarks, by setting `GCC` to use `-O3`, `-O2` and `-O1`, compared to `GCC` with no optimization options.

speed up	ACOTSP ls0	ACOTSP ls3	ILS	LKH	TS	EMILI
`-O3`	1.52	1.66	1.72	1.57	3.19	10.31
`-O2`	1.47	1.67	1.68	1.59	2.98	10.54
`-O1`	1.35	1.54	1.66	1.57	2.88	7.20

the biggest improvement in performance by reducing 10 times its running time compared to the one obtained by the executable generated without optimization. This could be related to the fact that the code of EMILI is written in `C++`, which allows more optimization. All the benchmarks obtain their best performance using `-O3` or `-O2`, showing that that those optimization levels, which are commonly advised for compilation, already lead to significant improvements.

The homogeneity of configuration scenarios regarding the problem instances is an important element for the algorithm configuration procedure. Very homogeneous scenarios allow to estimate the performance of a configuration based on less problem instances, more budget can be then used to explore the parameter search space. In terms of compilation, the homogeneity quantifies how consistent is the relative performance of executables obtained by different settings of optimization options across the different instances. In order to investigate the homogeneity of the benchmarks, we sampled uniformly 1 000 random configurations of `GCC` and we evaluate their performance over the training set. We removed the data of the configurations that produced failed compilations or executions. With this data, we calculate the Kendall concordance coefficient (W) [22] considering problem instances as blocks and configurations as groups. W is a normalization of the Friedman statistic and can be interpreted as a measure of how consistent is the ranking of the performance of the configurations over the instances. Table 2 gives the W coefficients for the different scenarios, the higher the number obtained the more homogeneous is the scenario. In addition, we also report the number of not failing configurations and the number of these configurations that obtain significantly better performance compared with `-O3`. If few configurations were used to calculate W, this indicates that the benchmark has many `GCC` settings that produce invalid executables or failed compilations. The number of configurations that are statistically better than `-O3`, gives an indication of how difficult is to optimize the performance for each benchmark. If many configurations showed better performance than `-O3`, good settings will be easy to find in the parameter space. On the contrary, if only few configurations are better than `-O3`, finding good configurations will be more challenging. The values of W indicate that in general, the $\texttt{GCC}_{\text{flags}}$ scenario is more homogeneous than the $\texttt{GCC}_{\text{flags+num}}$. TS-$\texttt{GCC}_{\text{flags}}$ shows to be a perfectly homogeneous scenario, indicating that the compilation could be optimized evaluating configurations in only one or very few instances and the results could be generalized to the rest of them. On the contrary, LKH has the lowest homogeneity evidencing that different instances benefit of different compilation options. This is surprising giving that the LKH scenario uses only one instance size, and therefore such variability between instances was not expected. The $\texttt{GCC}_{\text{flags+num}}$ scenarios are less homo-

Table 2. Kendall concordance coefficient W of uniformly sampled configurations of the 12 different `GCC` configuration benchmarks. Numbers in parenthesis indicate the number of configurations used (from the 1 000 uniformly generated) to compute W and the number of those configurations that have a significantly better performance compared with the mean performance of 10 executions with `-O3` (comparison performed by paired t-test with significance level 0.05).

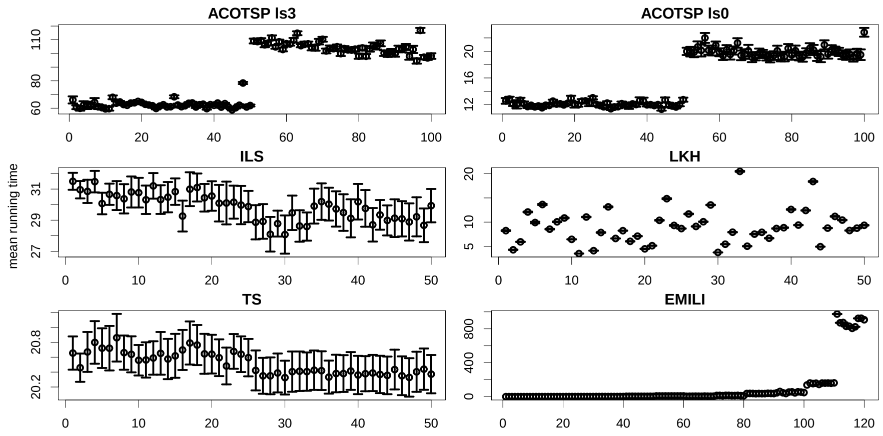
W	ACOTSP ls0	ACOTSP ls3	ILS
$\texttt{GCC}_{\text{flags}}$	0.92 (747|387)	0.97 (747|564)	0.97 (426|418)
$\texttt{GCC}_{\text{flags+num}}$	0.75 (466| 91)	0.58 (466| 32)	0.64 (318|224)
	LKH	TS	EMILI
$\texttt{GCC}_{\text{flags}}$	0.89 (472| 0)	1.00 (534| 8)	1.00 (332| 0)
$\texttt{GCC}_{\text{flags+num}}$	0.79 (352| 0)	0.81 (350| 0)	1.00 (304| 1)

geneous (with the exception of EMILI) indicating that some parameter values might be better for certain instances. This could be related to the instance size or type. These results confirm the differences between the configuration scenarios and therefore the potential benefits configuring their compilation.

Another important aspect of configuration scenarios is the stochastic behavior of algorithms. Algorithms that are strongly affected by stochasticity make the estimation of the performance of configurations more difficult. We explore the stochasticity effects on the evaluation of the performance of the `GCC` compilation using the `-O3` option by calculating a confidence interval based on 20 executions of the benchmarks. Figure 1 gives the confidence intervals of 20 evaluations over the instance test set. The confidence intervals clearly show the different effect of the stochasticity in the measured performance of these two benchmarks. ILS presents high variability of the intra-instance execution times, showing that the stochasticity has a great effect on this scenario. On the contrary, the LKH benchmark has a very low intra-instance variability but shows a high inter-instance variability. This is, again, surprising and indicates that the execution time of some components of this algorithm are affected by other instances features.

We presume that the best settings of the optimization options depend on the characteristics of the operations performed by the code to be compiled, but also of the machine in which the algorithm will be executed. To observe this, we report in Figure 2 the confidence intervals of the speed ups obtained by compiling the benchmarks with `-O3` on different machines, compared with `GCC` without optimization. These results indicate that the improvements in the performance depend on both the machine and the benchmark. For example, while ACOTSP ls3 obtains the biggest improvement in performance on $m2$, ILS obtains the biggest improvement on $m5$. This indicates that some of the optimization options activated by using `-O3` are very favorable for ILS when executed on $m5$, while these same settings seem to be less favorable for ACOTSP ls3 on the same machine. Moreover, different machines exhibit different variability of the results, which can indicate that some machines could be more affected by the stochastic behavior of the algorithms. We can derive from these results the strong impact the system can have on the performance of a compilation process, even if it

Fig. 1. Confidence intervals of the running time obtained by 20 executions of the benchmarks compiled with GCC using -O3 over the instances test set.



has been adjusted to the particular scenario. Good configurations are then not always portable between systems.

5 Experimental results

We configure the benchmarks described in Section 3 using only the optimization flags of GCC (GCC$_{\text{flags}}$) and the flags with the numerical parameters (GCC$_{\text{flags+num}}$). Figure 3 gives the mean speed up of 10 GCC settings obtained by 10 executions of irace compared to the execution times obtained by GCC using -O3.

In general, irace is able to obtain speed ups over -O3 for all the benchmarks. While for some scenarios the improvements are greater than for others, the executions of irace are on average better than -O3. When configuring only the flags (GCC$_{\text{flags}}$), all the benchmarks clearly improve their performance with the exception of ACOTSP ls3, for which two executions of irace obtain slightly worse performance than -O3 (0.98 and 0.99). Regardless of this, the mean speed up obtained by irace in this benchmark is of 1.03. The benchmarks for which the largest speed up is obtained on the GCC$_{\text{flags}}$ scenario are ILS and EMILI.

The configuration of the optimization flags together with the numerical parameters (GCC$_{\text{flags+num}}$) obtains mixed results across the different benchmarks. The speed up obtained by irace on the ACOTSP ls3, ILS and LKH scenarios is increased, while the speed up is reduced (compared to GCC$_{\text{flags}}$) for ACOTSP ls0, TS and EMILI. For EMILI, the speed up obtained for GCC$_{\text{flags+num}}$ is only slightly smaller compared with the one obtained for GCC$_{\text{flags}}$. For TS, the mean speed up is reduced compared to the results obtained for GCC$_{\text{flags}}$. Nevertheless, all the configurations obtained by irace have better mean performance than -O3. The results obtained for GCC$_{\text{flags+num}}$ on ACOTSP ls0, are worse than the results obtained when configuring GCC$_{\text{flags}}$; despite this, the mean speed up obtained over -O3 is 1.03 and of the 10 executions the worst speed up is only 0.99.

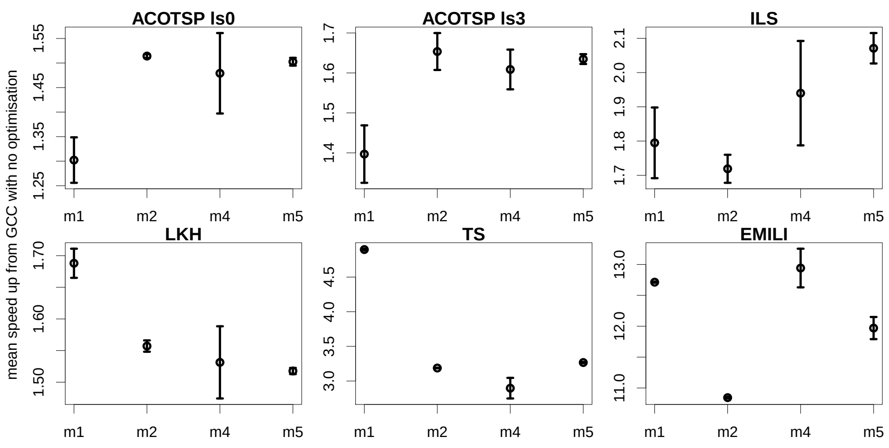


Fig. 2. Confidence intervals of the mean speed up of the running time of 20 executables compiled with `GCC` using `-O3` from `GCC` with no optimization on 4 different machines.

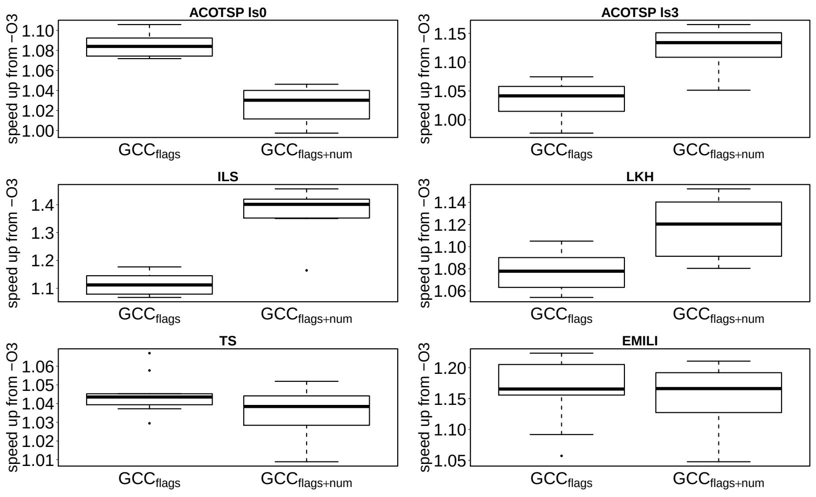


Fig. 3. Mean execution time speed up of 10 `GCC` configurations found by 10 `irace` executions from the mean running time of 10 compilations performed by `GCC` using `-O3`.

The reduced performances when configuring $\texttt{GCC}_{\text{flags+num}}$ for ACOTSP ls0, EMILI and TS can be explained by the size and difficulty of the configuration space. The $\texttt{GCC}_{\text{flags+num}}$ scenarios have considerably more parameters (366) and, as we maintained the same configuration budget as in the $\texttt{GCC}_{\text{flags}}$ scenarios, the search may be too limited. Additionally, the $\texttt{GCC}_{\text{flags+num}}$ scenarios seem to be (see Table 2) less homogeneous, and they also produce more failed executions than the $\texttt{GCC}_{\text{flags}}$ scenarios. This might actually help the configuration process, in some cases, by easily reducing the search space of interest. The possible approaches to tackle the configuration of such scenarios with irace are: (i) increasing the configuration budget (if feasible), (ii) providing `-O3` or `-O2` as initial configurations and/or (iii) increasing the number of instances required to start the elimination of configurations. The last one would allow better estimation of the performance of the configurations, which may account for the higher variability of the intra- or inter-instance execution times as seen in Figure 2 for some of the benchmarks.

The performance data obtained from the executions of irace can be useful in order to analyze the characteristics of the configuration scenarios. Such analysis can provide guidelines to set up the compilation with GCC on scenarios with similar characteristics. We trained random forest models with the performance data obtained by the 10 executions of irace on each benchmark. The procedure and the settings used to train the random forest model are described in [18]. For the random forest implementation we use the `ranger` `R` package [26]. When building the training data set, the configurations that produced failed compilations or executions were removed and thus only valid execution data points are included in the data set. As is common for these performance models, the instances are included as a variable in the training data. Additionally, instance features could be added to the data set to provide more information on the impact of the instances on performance.

Table 3 gives the five most important parameters for each of the benchmarks. Note that since the instances are included in the data set as a variable, for some of the benchmarks the instances are the variable that explains most of the variability. This is the effect of the variability of the execution times required for different instances, which can be observed in Figure 1. The instances variable is therefore more important in benchmarks that have greater inter-instance variability, such as LKH, and less important when the running time for all the instances is similar, as for TS. Even if there are parameters that are important for more than one algorithm, it is not possible to find a parameter that consistently has the same impact on the execution time of all the algorithms. For example, EMILI has a completely different set of important parameters compared to the rest of the algorithms. This difference can be explained by the fact that EMILI is the only program among the benchmarks written in C++ with an Object Oriented philosophy, while the others are written in C. For the benchmarks with algorithms written in C (ACOTSP, ILS, TS and LKH), the most important options are the ones that attempt to optimize memory allocation and the use of the registers (`falign-labels`, `fstrict-aliasing`, `fcaller-saves`, `falign-function`, `fomit-frame-pointer`), the linking process (`flto-partition`) and the optimization of the internal representation of the source code used by the compiler (the `ftree` flags). On the contrary, for EMILI, the optimization seems to be more focused on inlining (`finline`), branching optimization(`fif-conversion`, `fcode-hoisting`) and trying to avoid unnecessary function calls (`fipa-pure-const`), which is consistent with an object oriented code where it is common to have a large number of very small functions. Most of the parameters retain a similar importance when we extend the tuning to the numerical parameters. In fact, the list of parameters does not change for ACOTSP ls3 and for the others, with the exception of ILS, for which the changes are minimal. In the case of ILS, the numerical parameter `max-unswitch-insns`

Table 3. Variable importance % obtained by random forest models trained using data from 10 executions of irace.

$\texttt{GCC}_{\text{flags}}$					
ACOTSP ls0		ILS		TS	
instance	89.9	falign-labels	39.1	falign-labels	45.1
falign-labels	2.5	instance	15.3	fguess-branch-probability	6.2
fstrict-aliasing	1.1	fcaller-saves	4.4	fstrict-aliasing	5.4
ftree-ch	0.8	ftree-pre	3.2	ftree-loop-im	5.2
flto-partition	0.5	falign-functions	2.0	ftree-ter	5.0
ACOTSP ls3		LKH		EMILI	
instance	87.3	instance	95.4	finline	50.3
falign-labels	5.5	falign-labels	2.0	instance	40.0
fcaller-saves	0.6	flto-partition	0.2	fif-conversion	3.3
fstrict-aliasing	0.5	finline-limit	0.2	fcode-hoisting	2.4
falign-functions	0.4	fomit-frame-pointer	0.1	fipa-pure-const	1.3
$\texttt{GCC}_{\text{flags+num}}$					
ACOTSP ls0		ILS		TS	
instance	90.5	max-unswitch-insns	29.4	falign-labels	38.9
falign-labels	1.9	falign-labels	13.6	ftree-ter	7.0
ftree-ch	1.3	instance	10.1	ftree-loop-optimize	3.8
fstrict-aliasing	0.9	ftree-dominator-opts	5.3	ftree-loop-im	2.3
ftree-ter	0.5	ftree-loop-optimize	2.9	fomit-frame-pointer	2.2
ACOTSP ls3		LKH		EMILI	
instance	87.4	instance	93.8	finline	44.1
falign-labels	3.5	falign-labels	2.0	instance	40.2
fcaller-saves	1.0	flto-partition	0.5	fif-conversion	3.0
fstrict-aliasing	0.9	parloops-schedule	0.3	sccvn-max-scc-size	2.4
ftree-ch	0.5	finline-limit	0.3	fipa-pure-const	1.2

is the most important one according to our analysis. This parameter controls the threshold used by the compiler to decide if to unswitch a loop, that is moving a loop invariant condition outside of the loop.

Further optimization on the execution time could be then achieved using this information by restricting the configuration process to the set of variables that show to have great impact in the execution times.

6 Conclusion and future work

Reducing the execution time of optimization algorithms, even in cases in which the main objective is not fast execution, is of great interest for developers and users. We have shown that significant reductions of computation times can be obtained by optimizing compilation options. The optimization of compilation options may either be considered as an extra step to improve execution times after an algorithm configuration process, but also as part of the full configuration process to address interactions between algorithm components and compilation options. In preliminary experiments with GCC, we showed that, for the optimization algorithms used, the execution times are stochastic and depend on the computational platform and the characteristics of the code itself. Here, we configured the GCC compiler options, without including any particular knowledge of the compilation process itself, using a general-purpose algorithm configuration software, irace. The experimental results with irace are encouraging and show that irace can find settings that significantly improve over the default optimization flags available in GCC, the commonly recommended -O2 and -O3 settings.

In future work we will extend these experiments to new benchmarks with algorithms that present different characteristics and also to standard compiler benchmark sets used for autotuning methods. For optimization algorithms, it is also of interest to study the effect of instance types and size and their relationship with the `GCC` options. The information obtained in these experiments could be then further analyzed to extract features from the benchmarks that cause certain optimization flags to have greater effect over performance. Another direction for future work is to specialize the settings of irace to configure compilation options such as a set of initial promising configurations or additional pruning techniques to improve the search. Additionally, we will investigate the use of other automatic configuration tools such as SMAC and compare our approach to other methods specifically designed for the optimization of compiler flags such as OpenTuner and COLE. Finally, we plan to make available the configuration files used in this work to configure the options of `GCC`, so that other researchers can attempt to configure their own algorithms to obtain better performance.

Acknowledgments.

We acknowledge support from the COMEX project (P7/36) within the IAP Programme of the BelSPO. Thomas Stützle acknowledges support from the Belgian F.R.S.-FNRS, of which he is a senior research associate. The authors would like to thank Manuel López-Ibáñez for his many helpful remarks and assistance.

References

1. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.M., Amarasinghe, S.: Opentuner: An extensible framework for program autotuning. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. pp. 303–315. ACM New York, NY, USA (2014)
2. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent, I.P. (ed.) CP 2009, LNCS, vol. 5732, pp. 142–157. Springer (2009)
3. Birattari, M., Yuan, Z., Balaprakash, P., Stützle, T.: F-race and iterated F-race: An overview. In: Bartz-Beielstein, T., et al. (eds.) Experimental Methods for the Analysis of Optimization Algorithms, pp. 311–336. Springer (2010)
4. Blackmore, C., Ray, O., Eder, K.: Automatically tuning the GCC compiler to optimize the performance of applications running on the ARM cortex-M3. Tech. rep., CoRR, `https://arxiv.org/abs/1703.08228` (2017)
5. Christen, M., Schenk, O., Burkhart, H.: Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium. pp. 676–687. IPDPS '11, IEEE Computer Society (2011)
6. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005), special issue on "Program Generation, Optimization, and Platform Adaptation"
7. Fursin, G., Kashnikov, Y., Memon, A.W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C.K.I., O'Boyle, M.: Milepost gcc: Machine learning enabled self-tuning compiler. International Journal of Parallel Programming 39(3), 296–327 (2011)

8. GNU Project, Free Software Foundation: GCC, the GNU compiler collection. `https://www.gcc.gnu.org` (1987)
9. Helsgaun, K.: An effective implementation of the Lin-Kernighan traveling salesman heuristic. European Journal of Operational Research 126, 106–130 (2000)
10. Helsgaun, K.: General k-opt submoves for the Lin-Kernighan TSP heuristic. Mathematical Programming Computation 1(2–3), 119–163 (2009)
11. Henning, J.L.: Spec cpu2000: measuring cpu performance in the new millennium. Computer 33(7), 28–35 (2000)
12. Hoos, H.H.: Automated algorithm configuration and parameter tuning. In: Hamadi, Y., et al. (eds.) Autonomous Search, pp. 37–71. Springer (2012)
13. Hoste, K., Eeckhout, L.: Cole: Compiler optimization level exploration. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 165–174. CGO '08, ACM Press, New York, NY (2008)
14. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello Coello, C.A. (ed.) LION 5, LNCS, vol. 6683, pp. 507–523. Springer (2011)
15. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267–306 (2009)
16. Ladd, S.R.: ACOVEA (Analysis of compiler options via evolutionary algorithm). `https://github.com/Acovea/libacovea` (2000)
17. López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T., Birattari, M.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives 3, 43–58 (2016)
18. Pérez Cáceres, L., Bischl, B., Stützle, T.: Evaluating random forest models for irace. In: GECCO'17 Companion. ACM Press (2017)
19. Pérez Cáceres, L., Pagnozzi, F., Franzin, A., Stützle, T.: Automatic configuration of gcc using irace: Supplementary material. `http://iridia.ulb.ac.be/supp/IridiaSupp2017-009/` (2017)
20. Plotnikov, D., Melnik, D., Vardanyan, M., Buchatskiy, R., Zhuykov, R., Lee, J.H.: Automatic tuning of compiler optimizations and analysis of their impact. In: Alexandrov, V., et al. (eds.) 2013 International Conference on Computational Science. Procedia Computer Science, vol. 18, pp. 1312–1321. Elsevier (2013)
21. Püschel, M., Franchetti, F., Voronenko, Y.: Spiral. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1920–1933. Springer, US (2011)
22. Siegel, S., Castellan, Jr, N.J.: Non Parametric Statistics for the Behavioral Sciences. McGraw Hill, 2 edn. (1988)
23. Stützle, T.: ACOTSP: A software package of various ant colony optimization algorithms applied to the symmetric traveling salesman problem (2002), `http://www.aco-metaheuristic.org/aco-code/`
24. Stützle, T., López-Ibáñez, M.: Automatic (offline) configuration of algorithms. In: Laredo, J.L.J., et al. (eds.) GECCO (Companion), pp. 681–702. ACM Press (2015)
25. Whaley, C.R.: Atlas (automatically tuned linear algebra software). In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 95–101. Springer, US (2011)
26. Wright, M.N., Ziegler, A.: ranger: A fast implementation of random forests for high dimensional data in C++ and R. Arxiv preprint arXiv:1508.04409 [stat.ML] (2015)
27. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength Pareto evolutionary algorithm for multiobjective optimization. In: Giannakoglou, K.C., et al. (eds.) EUROGEN. pp. 95–100. CIMNE, Barcelona, Spain (2002)