Finding Best Compiler Options for Critical Software Using Parallel Algorithms

Gabriel Luque(✉) and Enrique Alba

Andalucía Tech, University of Málaga, Málaga, Spain
{gabriel,eatl}@lcc.uma.es

Abstract. The efficiency of a software piece is a key factor for many systems. Real-time programs, critical software, device drivers, kernel OS functions and many other software pieces which are executed thousands or even millions of times per day require a very efficient execution. How this software is built can significantly affect the run time for these programs, since the context is that of compile-once/run-many. In this sense, the optimization flags used during the compilation time are a crucial element for this goal and they could make a big difference in the final execution time. In this paper, we use parallel metaheuristic techniques to automatically decide which optimization flags should be activated during the compilation on a set of benchmarking programs. The using the appropriate flag configuration is a complex combinatorial problem, but our approach is able to adapt the flag tuning to the characteristics of the software, improving the final run times with respect to other spread practices.

1 Introduction

When a software package is developed, there are a lot of aspects which should be considered. The ISO/IEC 25010 standard [1] defines eight characteristics (with many sub-characteristics), such as security, usability, compatibility, efficiency... which must be taken into account when evaluating a software.

In this work, we focus on software in which the performance is one of the key features requested by the stakeholders. Many systems fall into this category. Critical and real-time systems [2] clearly need to response to the external environment changes as quick as possible. But, there are also non-critical software pieces in which performance is very relevant like program which are constantly executed and their runtime affects to the complete software ecosystem, like drives, kernel OS functions, or communication protocols.

Developers focus on the software design and on the source code to get an efficient program. These aspects are indeed crucial, but they often pay little attention how their software is built. Some world spread compiler options, like O3 or hardware platform specific ones in C/C++, are blindly applied without any additional study. The main contributions of this paper focuses on how the

J. Del Ser et al. (Eds.): IDC 2018, SCI 798, pp. 71–81, 2018.
https://doi.org/10.1007/978-3-319-99626-4_7

optimization flags used during the compilation process affect to the final software performance and how they can be optimally selected to build a more efficient program. We analyze the influence of 181 optimization flags provided by GCC compiler [3]. We model it as a complex combinatorial problem, and provide a parallel automatic tool to decide which options should be activated to get a high-performance software on a set of benchmarking problems.

The rest of this article is organized as follows. The next section introduces the problem solved, modelling it as a combinatorial problem. Section 3 explains the algorithmic approaches proposed in this paper. Section 4 shows the experimental design and Sect. 5 analyses the results from different points of view. Finally, the last section concludes and gives some open research lines.

2 Problem Description

Modern compilers have a rich set of optimization flags and a manual selection of the most adequate configuration is a hard task. Compilers support a number of basic optimization level to make easy this process. However, the question is if these basic levels are adequate for any software and any scenario. Traditional compiler optimizations define a well-studied area, and some issues are been analysed in the literature in the past. But, these works follow a different approach to the proposed in this paper: they study the compiler phase order [4], compare different compiler [5], optimize specific software [6,7], or evaluate specific flags for some hardware platforms [8].

In this paper, we have modelled the problem of selecting the optimal set of compiler flags as a combinatorial problem. Given a program P and a set of flags $F = \{f_1, f_2, \ldots, f_N\}$, this problem consists on finding a binary vector $X = (x_1, x_2, \ldots, x_N)$ (where $x_i \in \{0, 1\}$ means if the flags f_i is activated or not) which minimizes the execution time of P when it is compiled with set of flags indicated by X $\{f_i | x_i == 1 \, \forall i \in \{1, N\}\}$:

$$min_X \; Fitness(X) = executionTime(compile(P, F, X)) \tag{1}$$

where $compile(P, F, X)$ is a process which generates an executable program compiling the program P selecting the flags from F indicated by X and $executionTime(P)$ calculates the execution time of the executable program.

We focus on the optimization flags of GCC compiler, a popular C language compiler. This language is the most prominent one for the developing this kind of critical software and GCC is the most used compiler for this language. We will consider all the 181 flags activated by level O3, the most aggressive and commonly option used to get efficient executable codes.

3 Our Proposed Approach

This problem is very complex in manifold ways. On the one hand, as we showed in Eq. 1, the evaluation of a candidate solution requires the compilation and

execution of a program. Even in the smallest software pieces, these activities requires between 0.5 and 1 second. On the other hand, the search space is quite large: we have two options for each flags (it is activated or not), therefore, there are $2^{181} \approx 3.1 \times 10^{54}$ candidate solutions.

The combination of these two factors makes hard the utilization of methods which require a large number of evaluations. Therefore, we have chosen a trajectory-based metaheuristic with a fast convergence to get some accurate solutions in a reasonable time. We have used Variable Neighborhood Search (VNS) [9]. This technique solves optimization problems by doing systematic changes of neighbourhood within a local search. VNS is a descendent method which does not follow a single trajectory since it explores different predefined neighbourhoods of the current solution using a local search (LS). The current solution is changed by a new one if and only if an improvement has been made. The basic idea is to change the neighbourhood structure when the local search is trapped on a local optimum. There exist several parallel models for VNS [10,11]. In this work, we have used two parallel variants:

- **Parallel moves model:** The parallel moves model is a kind of farmer/worker model allowing to speed up the exploration of the possible moves. At the beginning of each iteration of the algorithm, the farmer sends the current solution to a pool of workers. Each worker explores some neighbouring candidates, and returns back the results to the farmer.
- **Parallel multi-start cooperative model:** The model consists in launching in parallel several cooperative homogeneous VNS. Each VNS is initialized with a different solution. VNSs of the parallel multi-start model periodically interchange the current solution during execution.

Now, we show will how this generic template has been instantiated for our concrete problem. To do this, we have to describe the solution representation, the fitness function and the definition of the different neighbourhoods.

Solution Encoding. In the previous section, we define the solution of this problem as a binary vector. With this representation, a solution is a vector X where $x_i = 1$ indicates if the flags f_i is activated while $x_i = 0$ means that the flag f_i does not appear in the compilation command. But our preliminary experiments showed that this approach was not the most appropriate representation. According to the documentation, the O3 option activates all the flags considered in this work and no more, but the results of our preliminary experiments (see Table 1) showed a significant difference between using the O3 option and the activation of all the flags (without using the O3 option).

Based on these results, we decide to change the meaning of each vector component. Now, the compilation is always performed with the O3 option but our algorithm is able to deactivate some flags from O3 option. Therefore, $x_i = 1$ indicates if the flag f_i is deactivated[1] while $x_i = 0$ means that we allow to O3 to

[1] In GCC, you can activate flag with `-f`*flag* but you can also deactivate it with `-fno-`*flag* if another option (O3 in our case) has previously activate it.

Table 1. CLBG corpus of programs and execution time (in μsec) of the compiled programs with different options

Benchmark	Description	O3	All the flags
`binary-trees`	Allocate/traverse/deallocate binary trees	75579	84924
`chameneos-redux`	Symmetrical thread rendezvous requests	390695	436925
`fannkuch-redux`	Indexed access to tiny integer sequence	78313	128914
`fasta`	Generate and write random DNA sequences	18425	59158
`k-nucleotide`	Hashtable update and k-nucleotide strings	39271	50916
`mandelbrot`	Generate Mandelbrot set bitmap file	13926	23865
`meteor-contest`	Search for solutions to shape packing puzzle	45314	72318
`n-body`	Double precision N-body simulation	46661	184670
`pidigits`	Streaming arbitrary precision arithmetic	21742	39185
`regex-redux`	Match DNA 8mers and substitute magic patterns	25712	32673
`rev-complement`	Read DNA sequences, write their rev-complement	34159	53811
`spectral-norm`	Eigenvalue using the power method	61149	69154
`thread-ring`	Switch from thread to thread passing one token	714616	736524

use the flag f_i. Preliminary experiments show that the order in which the flags are deactivated does not change the overall performance.

Fitness Function. We use the (Eq. 1) as fitness function. The main complication found in our preliminary experiments is that although the hardware platform is dedicated to this work and the tested programs are deterministic, in some cases the time consumed by the same program (compiled with the same flags) is significant different. This is due to the operating system, that is composed by a quite large number of internal processes which are executed in a non-controllable way (by the user). This behaviour could provoke misleading conclusions. To deal with this difficulty, we execute each program five times and get the lowest value. We use this value (instead of the mean, median, or other statistical rate) since it is the most accurate representation of the execution time of the program without any interruption of any external software. However, this approach makes even harder the search process since the evaluation of a candidate solution is now more expensive.

Neighbourhood. Since we encode the solutions as a bit string, we can apply traditional variation operators. In concrete, we use a variant of bit-flip mutation. We define that X' is in the k-neighbourhood of X if the hamming distance (number of different elements between the bit string of X and the one of X') is exactly k. Then, our VNS starts changing only one bit in the solution ($k = 1$) to get a neighbour. When the convergence is detected (we explore 5 neighbours and all these candidate solutions are worse than the current one, in our experiments), it changes the neighbourhood and varies at most two bits ($k = 2$). This process continues until the $N/2$-neighbourhood is reached and in that moment it backs

to the first neighbourhood. VNS also backs to the first neighbourhood when a new best solution is found.

4 Experimental Design

This section is mainly devoted to describe the benchmark problems used to test our algorithm but before describing them, some comments about the setting of our approaches and the testing platform.

Our sequential proposed algorithm has only a single parameter: the stop condition. Since the fitness function is a very time-consuming task, we use a maximum number of evaluation (4000 in our experiments) as stopping criterion. It is a quite low value but most of the runs have already converged before reaching this number of fitness evaluations. In parallel techniques, we use 8 processes which cooperate (in multi-start approach) every 25 evaluations using an unidirectional ring topology. Finally, we have performed 30 independent runs of each experiment to gather statistical information and then apply Kruskal-Wallis test to validate if the results are statistically different. We use these values to decide in the results' tables when the values are similar or not (the actual values are not shown due to the lack of space).

In this study, we propose three VNS variants: the sequential version, $\mathbf{VNS}_{Seq}$, and two parallel ones; one following the farmer/worker scheme, $\mathbf{VNS}_{F-W}$, and a cooperative multi-start approach, $\mathbf{VNS}_{CMS}$. We compare them against four static and commonly used compiler configurations:

- **None**: No flags are activated.
- **All**: All the considered flags in this study are activated.
- **O2**: It uses the O2 option to compile the program.
- **O3**: It uses the O3 option to compile the program.

In tables reporting results (next section), we will present how the execution time is reduced (percentage) with respect to the most basic configuration (**None** one). This value is calculated using the next equation:

$$reduction(Algorithm) = \frac{t_{None} - t_{Algorithm}}{t_{None}} \times 100 \quad (2)$$

A larger value indicates a higher reduction in the execution time and it is better. The hardware platform is composed by 8 machines with Intel i-core7 processor at 2.6 GHz with 8 GB of RAM memory. The operating system is Ubuntu/Linux 14.04.5 LTS and the GCC version is 4.8.4.

In order to obtain a comparable, representative and extensive set of programs we have explored The Computer Language Benchmarks Game (CLBG) [12] which has been used in previous studies in the literature such as [13]. The CLBG has gathered solutions for 13 benchmark problems (Table 1). An interesting feature of this set of programs is most of them (with the exception of `meteor-contest`) can be easily tunable with a parameter or file in order to generate faster or slower executions. Also, this benchmark includes programs with

different characteristics; in concrete, five of the programs are also using some kind of parallelism (`k-nucleotide`, `spectral-norm`, `fasta`, `rev-complement`, and `thread-ring`).

5 Experimental Analysis

In this section, we will show the results obtained in our experiments. We have performed two main experiments and analyses: in our first analysis (Sect. 5.1), we study the numerical performance of the techniques, while in our second study (Sect. 5.2), we analyse the solutions obtained by our proposed tool, and discuss about the optimization flags activated or deactivated according to the problem tested.

5.1 Quality Analysis

Our main goal in this section is to study the quality of the solutions provided by our tool. In the Table 2, we show the reduction obtained by each of the configurations calculated according to the Eq. 2 with respect to the **None** version. Several conclusion can be obtained from that table.

Table 2. Average time reduction (percentage) with respect to the **None** configuration. Boldfaced values represent the best values with statistical significance.

Benchmark	All	O2	O3	VNS_{Seq}	VNS_{F-W}	VNS_{CMS}
`binary-trees`	6.48	12.41	16.77	20.66	19.13	**23.78**
`chameneos-redux`	25.58	27.94	33.45	45.43	50.31	**53.41**
`fannkuch-redux`	25.42	50.21	54.69	**73.51**	**72.97**	**73.38**
`fasta`	3.86	35.85	70.06	74.65	**76.32**	**77.2**
`k-nucleotide`	28.69	40.37	45.00	55.99	50.12	**58.39**
`mandelbrot`	11.09	37.41	48.12	50.86	**54.18**	**54.78**
`meteor-contest`	13.76	25.61	45.96	50.58	50.84	**52.34**
`n-body`	29.95	74.20	80.41	**83.30**	**82.57**	**82.40**
`pidigits`	26.57	51.47	59.25	66.58	71.53	**77.31**
`regex-redux`	30.72	43.83	45.48	52.42	**55.31**	**56.85**
`rev-complement`	12.20	18.42	44.26	50.80	53.51	**67.37**
`spectral-norm`	3.12	10.83	14.34	42.56	**50.74**	**51.34**
`thread-ring`	3.64	4.87	6.51	**12.54**	9.34	10.75
Average	12.77	26.93	36.39	47.25	47.42	50.99

First, the utilization of the optimization flags allows to reduce the final execution time in all the benchmark problems tested in this work. In fact, this

reduction is very important in most of the cases and we can observe a reduction around 50% (or even larger) for 11 out 13 cases. Only in a specific benchmark problem, `thread-ring`, the benefit of the optimization flags is minor (between 3.6% and 12.5%). This is due to this application creates many threads and switches constantly among them. Then, the influence of the code generated by the compiler is smaller since most of the time is spent changing among thread without executing real calculations. This confirms the intuitive idea that the efficacy of the compiler depends on the amount of non-calculation operations in the code (switching context or IO operations).

Second, as we stated in Sect. 3, the O3 configuration obtains different results than the configuration which activate all the flags and this difference is quite important in most of the scenarios. These figures means that the O3 adds some additional flags which are not mentioned in the documentation.

Third, our approaches outperform in all the instances to the classical O3 configuration. The difference varies from 4–5% (for `binary-trees` or `n-body`) to 41.5% (for `fannkuch-redux`). The average reduction provided by VNS with respect to the O3 option is 18%, a significant reduction. This gives us some hints that the blindly utilization of O3 is not the most adequate configuration in general and some additional studies are need to select the most beneficial flags for each program. Even in the cases in which the reduction is small, these gains can be very important for critical systems.

Analyzing our proposals, we observe that the $\mathbf{VNS}_{CMS}$ version outperforms (or it is equal to) the other two version in 12 out of 13 test problems. This is due to the parallel versions maintains a better diversity, and while the serial version is stuck in a local optimum, the parallel versions is able to get out of it and to continue the search for better solutions. However, this slows that the convergence (see time with one processor in Table 3) provoking that parallel versions consume their computational badge without improving the sequential solution or even providing a worse one when the problem requires a high number of evaluations to get the local optimum (like in `thread-ring`).

Finally, in Table 3, we show the average execution time required by our approaches. We show the execution time in a single processor (with 8 threads) aganist a real parallel configuration (with 8 machines) in order to calculate the *speedup*. Comparing the versions on a single processor, we notice that the parallel versions spend more time that serial one. As we said, parallel version have slower convergence but it allows to get better solutions. When the parallel versions are run in an actual parallel platform, the execution time is reduced in a very significant way, with a very good speedup. The $\mathbf{VNS}_{F-W}$ usually is slower than $\mathbf{VNS}_{CMS}$ since they require more synchronization (it waits to the finalization of all the workers before moving to the next iteration).

5.2 Analysis of the Selected Flags

Now, we study the flags selected by our proposals. The Fig. 1 shows how many times is selected each flags in the best found solution of each independent run (as percentage) in some representative problems. Since the number of flags is

Table 3. Average execution time (in sec) of our approaches and the speedup

Benchmark	VNS_{Seq}	VNS_{F-W}			VNS_{CMS}		
		1 proc.	8 proc.	Speedup	1 proc.	8 proc.	Speedup
`binary-trees`	2735.20	3397.12	463.45	7.33	3042.64	435.28	6.99
`chameneos-redux`	2142.90	2800.34	386.25	7.25	2494.12	346.41	7.2
`fannkuch-redux`	3175.00	4011.93	583.98	6.87	3597.28	510.25	7.05
`fasta`	1995.40	2650.69	372.29	7.12	2404.66	328.95	7.31
`k-nucleotide`	4293.60	5147.17	698.39	7.37	4643.53	639.6	7.26
`mandelbrot`	2269.30	2892	429.72	6.73	2711.36	381.34	7.11
`meteor-contest`	2974.00	3854.3	559.4	6.89	3583.97	506.21	7.08
`n-body`	3006.90	3734.57	528.23	7.07	3592.64	526.78	6.82
`pidigits`	1189.60	1605.96	224.92	7.14	1507.1	207.3	7.27
`regex-redux`	2323.80	3086.94	410.5	7.52	2872.22	402.27	7.14
`rev-complement`	2153.00	2883.3	402.7	7.16	2727.64	380.96	7.16
`spectral-norm`	2018.50	2768.57	394.38	7.02	2494.87	351.89	7.09
`thread-ring`	4550.40	5061.86	729.37	6.94	4733.78	660.22	7.17
`Average`	2533.18	3240.36	456.05	7.11	2983.49	418.69	7.13

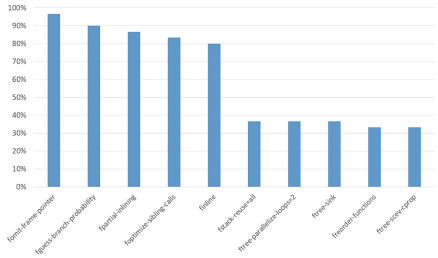
too high to be shown in the figures, only the extremes are presented (the first five and the last five considering they are sorted by percentage).

First, we can notice that depending on the problem, a different set of flags is selected and these flags are chosen according to the characteristic of the problem. For example, we can observe that in `fannkuch-redux` (Fig. 1b), the flag `-finline` is always active. This is logical selection since the source code of this program has five inline functions and they are used constantly.

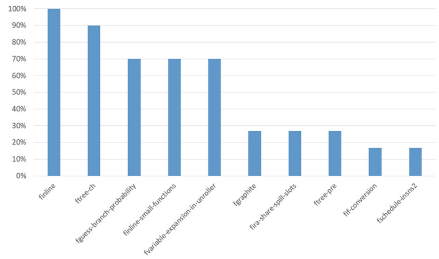
Second, we can notice that in the programs in which we obtained large runtime reductions, `fannkuch-redux`, `fasta`, and `n-body` (see Table 2), there are a clear distinction between the appropriate flags (with percentages close to 100%) and the non-beneficial flags (around 20% or even lower). This can also help to explain why in some test cases the **All** configuration offers a very low performance. This poor performance is due to it activates these non-beneficial flags (which can even harm the final runtime in specific programs).

Also, we can observe in some applications (like `thread-ring`, Fig. 1f) that the distinction of the appropriate flags is not so clear (percentages lower than 80%). This is an expected result since, as we said before, for this problem the optimization flags has only a minor effect in the final runtime.

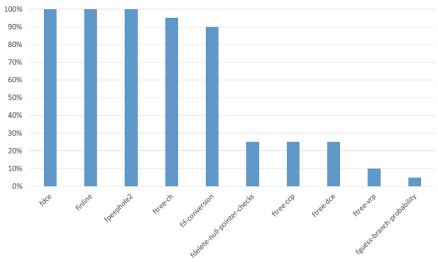
Finally, in the last figure, we aggregate the values for all the functions. We can notice that there is not a clear set of flags which should be chosen always (the difference between the most selected one and the last one is lower than 30%) since, as we said above, it depends on the features of the problem. However, we can extract some information about our benchmark analysing this figure.



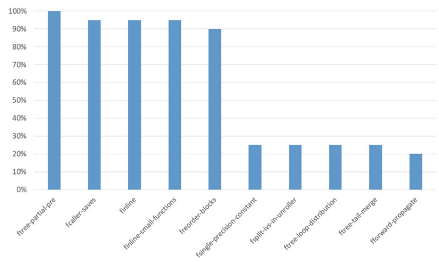
(a) Flags for `binary-tree`



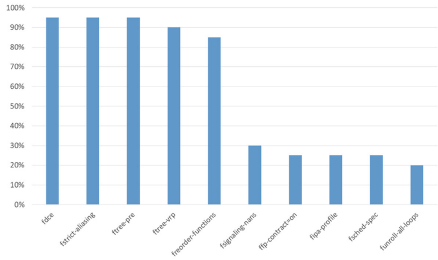
(b) Flags for `fannkuch-redux`



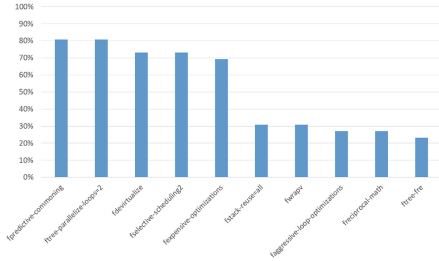
(c) Flags for `fasta`



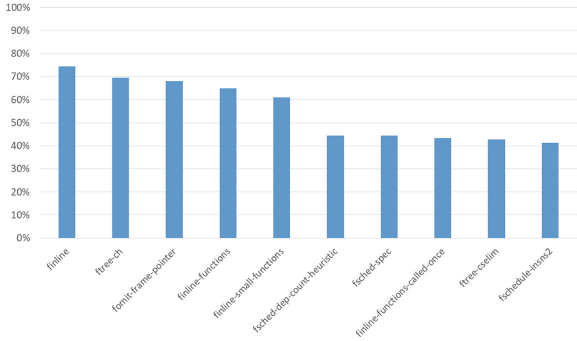
(d) Flags for `meteor-contest`



(e) Flags for `n-body`



(f) Flags for `thread-ring`



(g) Flags considering all the benchmark problems

Fig. 1. Flags selected by our approach for different problems.

For example, since the `-finline` is the most selected flag, we can deduce that our programs use this kind of inline functions frequently. Examining the source codes, we can observe than 9 out 13 programs include some inline functions confirming our deduction from the figure.

6 Conclusions

In this work, we have analysed how the optimization flags can affect to the final performance of a software system. Here, we have modelled the compilation process as a combinatorial problem and we have proposed three techniques to automatically decide which optimizations flags should be activated.

Our results show that world spread levels (like O3 or O2) produce very accurate results but they are not the optimal ones, and the flags chosen can be refined to get better performance. Our proposed techniques are able to outperform the results of O3 in all the tested programs (a wide set of 13 different software pieces). The parallelism (specially, using the multi-start approach) does not only allow to reduce the execution time but it also improves the quality of the solutions. We have also observed that the selection of the flags can be performed with the programs executed in small scenarios and later, you can use that configuration to other scenarios (with larger inputs).

There are several open research lines in this domain. First, besides to flags activated by O3, modern compilers offer a wider set of flags and parametric options. Second, other compilers like LLVM or Microsoft Visual Compiler are also very popular nowadays, and we are interesting in performing a similar analysis for them. Finally, we also plan to analyse other metaheuristics in order to perform a more efficient search.

Acknowledgement. This research has been partially funded by the Spanish MINECO and FEDER projects (TIN2014-57341-R (http://moveon.lcc.uma.es), TIN2016-81766-REDT (http://cirti.es), and TIN2017-88213-R (http://6city.lcc.uma.es).

References

1. Software engineering software product quality requirements and evaluation (SQuaRE) Software product quality and system quality in use models. Standard, International Organization for Standardization, Geneva, CH (2011)
2. Hassan, M.M., Afzal, W., Lindström, B., Shah, S.M.A., Andler, S.F., Blom, M.: Testability and software performance: a systematic mapping study. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 1566–1569. ACM (2016)
3. Stallman, R.M.: GCC DeveloperCommunity: Using the GNU Compiler Collection: A GNU Manual for GCC Version 4.3. 3. CreateSpace, Paramount (2009)
4. Nobre, R., Reis, L., Cardoso, J.: Compiler phase ordering as an orthogonal approach for reducing energy consumption. In: Proceedings of the 19th Workshop on Compilers for Parallel Computing (CPC16) (2016)

5. Machado, R.S., Almeida, R.B., Jardim, A.D., Pernas, A.M., Yamin, A.C., Cavalheiro, G.G.H.: Comparing erformance of C compilers optimizations on different multicore architectures. In: Computer Architecture and High Performance Computing Workshops (SBAC-PADW), pp. 25–30. IEEE (2017)
6. Hoste, K., Eeckhout, L.: Cole: Compiler optimization level exploration. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2008, pp. 165–174. ACM, New York (2008)
7. Zhong, S., Shen, Y., Hao, F.: Tuning compiler optimization options via simulated annealing. In: Second International Conference on Future Information Technology and Management Engineering, FITME 2009, pp. 305–308. IEEE (2009)
8. Kumar, T.S., Sakthivel, S., Kumar, S.: Optimizing code by selecting compiler flags using parallel GA on multicore CPUs. Int. J. Eng. Technol. **6**, 544–551 (2014)
9. Mladenovic, N., Hansen, P.: Variable neighborhood search. Comput. Oper. Res. **24**(11), 1097–1100 (1997)
10. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. Wiley, New York (2005)
11. Crainic, T.G., Toulouse, M.: Parallel meta-heuristics. In: Handbook of Metaheuristics, pp. 497–541. Springer, Heidelberg (2010)
12. Fulgham, B., Gouy, I.: The computer language benchmarks game. http://shootout.alioth.debian.org (2012)
13. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: Energy efficiency across programming languages: how do energy, time, and memory relate? In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, pp. 256–267. ACM, New York (2017)