Evolutionary Optimization of Compiler Flag Selection by Learning and Exploiting Flags Interactions

Unai Garciarena
Faculty of Informatics
University of the Basque Country (UPV/EHU)
ugarciarena001@ikasle.ehu.es

Roberto Santana
Intelligent Systems Group
University of the Basque Country (UPV/EHU)
roberto.santana@ehu.es

ABSTRACT

Compiler flag selection can be an effective way to increase the quality of executable code according to different code quality criteria. Evolutionary algorithms have been successfully applied to this optimization problem. However, previous approaches have only partially addressed the question of capturing and exploiting the interactions between compilation options to improve the search. In this paper we deal with this question comparing estimation of distribution algorithms (EDAs) and a traditional genetic algorithm approach. We show that EDAs that learn bivariate interactions can improve the results of GAs for some of the programs considered. We also show that the probabilistic models generated as a result of the search for optimal flag combinations can be used to unveil the (problem-dependent) interactions between the flags, allowing the user a more informed choice of compilation options.

Keywords

compiler flag selection, compiler optimization, probabilistic modeling, EDAs

CCS Concepts

•**Mathematics of computing** → *Probability and statistics;* •**Computing methodologies** → **Genetic algorithms;** *Bayesian network models; Search methodologies;* •**Software and its engineering** → *Automatic programming;*

1. INTRODUCTION

Evolutionary algorithms (EAs) have been recognized as a very effective tool for software improvement. There are a variety of ways in which genetic improvement can help to produce better programs, from the automatic correction of bugs [16], to software transplantation [4], and the creation of Pareto optimal program benchmarks with respect to different criteria [9]. Another scenario where EAs can help to enhance the characteristics of the programs is during the

compilation process. In many situations, executable code can be significantly optimized at the time of compilation. Code optimization can produce significant gains in terms of the time needed to solve practical problems and other resources (e.g. energy, memory, etc) required to solve them.

In this paper we investigate a question usually overlooked when optimizers are applied for compilation optimization. We analyze whether and how the information obtained during the optimization process can produce valuable knowledge about the relationships between the compilation transformations. The straightforward way to use this knowledge is by modifying the search operators in such a way that they incorporate statistical regularities present in the best solutions (e.g. pairwise dependencies between variables). This is the basic strategy used in this paper.

Iterative compilation approaches explore the space of possible optimization sequences until an acceptable solution is found [10]. We constrain our analysis to a particular framework of compilation optimization, the selection of the right combination of compilation flags. Usually, compilers provide a high flexibility for code quality by means of a large set of compilation flags that allow the user to balance the different criteria that describe the quality of the executable code. However, this flexibility has a downside since an exhaustive search over all these compilation options is prohibitive. Therefore, EAs and other heuristic based strategies have been proposed for the compiler flag selection problem [10, 13, 17, 25, 26].

Although EAs can be a more efficient alternative than manually setting the compilation options, they may fail when there are strong or widespread interactions between the variables of the problem (the flags). Traditional genetic operators are usually unable to learn the relevant linkage between the variables. Capturing and exploiting the interactions between the variables of the problems can be essential for optimizing the set of compilation flags. Interactions between compilation flags can be of different types. For instance, the addition of one flag may make irrelevant the use of another flag for the compilation process. Furthermore, these interactions can also depend on the architecture or the code being compiled. In the more general problem of finding the most effective set and ordering of optimization phases for code compilation, it has been already shown that the independence relationships between optimization phases can be used to reduce the search space [11]. In this context, machine learning methods have been proposed to detect and exploit the interactions between the compilation phases [12, 31].

The existence of interactions in the compiler flag selection problem, with the known limitation of classical EAs to deal with interactions indicate that it is a relevant question to investigate to what extent competent genetic algorithms can deal with the flag compilation option. In this paper we propose the use of estimation of distribution algorithms (EDAs) [15, 19] for the compilation flag optimization problem. EDAs are EAs that learn a probabilistic model of the promising solutions in order to capture a description of relevant patterns shared by these solutions. These patterns are represented using a probabilistic graphical model (PGM) where the (not necessarily linear) interactions between the variables are stored as dependency relationships and marginal probabilistic distributions. In addition to investigate the behavior of EDAs as optimizers, we address the question of whether the probabilistic models capture information about the interactions between the flags. Although an extensive list of EA approaches to compilation optimization have been proposed, we found no reference to previous use (and analysis) of probabilistic modeling for this problem.

As a framework to evaluate the performance of the algorithms, we optimize the compilation of code for C++ programs compiled by the GNU Compiler Collection (GCC) on an Intel CPU processor and using Boolean flags. We build on previous work were EAs were applied to the flag selection problem in this context. We focus on the minimization of the execution time of the programs. Previous research [5] has shown that compilation options that generate faster code can produce as a side effect a decrease in the energy consumed by the program. Therefore, at least in some contexts, compiler optimization for speed translates into gains in energy consumption [7, 30].

The paper is organized as follows: In the next section we introduce the problem of compiler flag selection and explain the context of application used in this paper. Section 3 discusses research related to our proposal. Our probabilistic modeling approach is explained in Section 4. Section 5 introduces the experimental framework used to evaluate our proposal and presents the numerical results. The main contributions of the paper are summarized in Section 6, where some lines for future research are also discussed.

2. COMPILER FLAG SELECTION

As previously mentioned, the question of making an intelligent choice of the compilation flags can be posed as an optimization problem and as such it has been previously addressed with EAs. In this section we present the details of the representation and fitness functions used.

We will assume a set of n Boolean flags is available. Let $\mathbf{X} = (X_1, \ldots, X_n)$ denote a vector of discrete random binary variables. We use $\mathbf{x} = (x_1, \ldots, x_n)$ to denote an assignment to the variables.

2.1 GCC compilation problem

GCC includes different levels of optimization, each of which has a number of flags that can be enabled or disabled. Since the number of GCC optimization options is large, it is infeasible to perform an exhaustive evaluation of all combinations. We start from a set of 60 compilation options used by Hoste and Eeckhout in [10]. They used these flags with the GCC 4.1.2 compiler. Here, we use a more advanced compiler (v 4.9.2) for which two of the original 60 flags (-fipa-type-escape and -fsplit-ivn-in-unroller) were already deprecated.

In GCC, different optimization levels group combination of several flags are used to define different trade-offs between the criteria that serve to describe the quality of the code. Below we show the three levels included in the 58 variables specifying the total number of flags in each level for the GCC 4.9 compiler and, between parenthesis, the corresponding number of flags from that level that are included in the set of 58 flags we use.

- -O1: Basic optimization level. 39 (28) variables.
- -O2: Recommended optimization level. Includes all flags existing in -O1. 73 (54) variables.
- -O3: Highest level of optimization. 82 (58) variables.

In our approach, $n = 58$ and each $\mathbf{x} = (x_1, \ldots, x_{58})$ represents the inclusion/exclusion of the corresponding flag from the gcc compilation command.

In the selection of the compilation flags by the EAs, the level groups are not considered, i.e. the level set to which a given flag belongs to is not relevant for its use in the compilation command. However, flags are sequentially ordered in the EA representation according to the level groups.

2.2 Fitness function

The fitness functions computes the time spent by the execution of the function. The evaluation of a solution comprises: 1) The construction of the compilation command from the binary vector. 2) Compiling the code. 3) Executing the program. Details about the implementation of the function are given in Section 5.

3. RELATED WORK

EAs have been previously applied to the problem addressed in this paper [10, 13, 17, 25, 26]. In all these applications, traditional crossover and mutation operators were employed to evolve the solutions. The emphasis was on the behavior of the algorithms and the analysis of the solution space, and not in the role of the recombination operators in the improvements of the algorithm. In [2], a GA was applied to find the optimal combination of flags in terms of the time spent by the executable code. Experiments showing the effectiveness of the approach were conducted using the GCC compiler. Authors also acknowledged that the best combination of flags was application dependent.

In [10], Hoste and Eeckhout proposed the Compiler Optimization Level Exploration (COLE), a multi-objective EA-based approach to compute Pareto optimal optimization levels. They compared the set of compilation flags included in the Pareto front approximations with the individual compiler optimization appearing in the standard -O1,-O2,-O3, and -Os optimization levels. They showed that the solutions in the Pareto front could yield better optimization levels than GCC's manually set options.

One of the few examples where the problem of flags interactions is addressed by modifying the recombination operators is the work described in [17], where a weight is associated to each gene to indicate an estimated fitness of the corresponding optimization option to the input source code. Weights are updated during evolution. This approach can be seen as a naive univariate analysis of the best solutions, similar to one of the simplest EDAs, the univariate marginal distribution algorithm (UMDA) [20]. In the same paper, Li

et al. also proposed to fix the values of a group of positively correlated genes for a number of generations. This is suggested by the authors as a way to alleviate the disruption of interactions inherent in the recombination operators.

Machine learning techniques have been applied in different phases of the compiler design. In [31], regression techniques are applied to predict program performance from the settings of compiler optimization flags, associated heuristics and microarchitectural parameters. Sanchez et al. [24] use support vector machines to discover method-specific compilation strategies in a commercial Just-in-Time compiler. One research particularly related to our work is [1], where Bayesian networks (a PGM class) are applied to find optimal compiler transformations. In this approach, the set of compiler transformations to be applied is represented as a probability distribution (encoded by the Bayesian network) to be sampled. Although this approach also uses probabilistic modeling, the application domain, and also the class of PGMs applied are different to the one used in this paper.

Methods that combine the use of machine learning techniques and evolutionary algorithms have been also proposed for compiler optimization. In [29], neural networks are evolved for selecting which optimizations apply to the compilation process. In two of the benchmarks evaluated, the authors reported performance improvements ranging from 5% to 50%.

In addition to the direct optimization of compilation options, other approaches have applied evolutionary algorithms [14] and other heuristic search algorithms [18] to search the program transformation space in order to find the program transformation sequence that produces the optimal code.

Although we did not find previous applications of EDAs to the problem of optimizing code for compilation, these algorithms have been applied for software testing [23].

4. MODELING DEPENDENCIES FOR COMPILER FLAG SELECTION

We will represent a population as a set of vectors $\mathbf{x}^1, \ldots, \mathbf{x}^N$ where N is the size of the population. Similarly, x_i^l will represent the assignment to the ith variable of the lth solution in the population. p denotes a distribution, $p(x_i)$ the marginal probability $X_i = x_i$. We focus on binary problems.

4.1 Evolutionary algorithms for flag selection

We will use three EAs to address the flag selection problem: 1) A genetic algorithm with one-point crossover and bit-flip mutation. 2) A univariate marginal distribution algorithm (UMDA) [20]. 3) A dependency-tree based EDA [3, 27]. The three algorithms share the same general framework described in Algorithm 1, only the "variators" change. Algorithm parameters will be presented in Section 5.

4.2 EDAs

In this section we explain the main differences between the two probabilistic modeling approaches. We denote a probability distribution by p, the marginal probability for $\mathbf{X}_I = \mathbf{x}_I$ as $p(x_I)$, and the conditional probability distribution of $X_i = x_i$ given $X_j = x_j$ as $p(x_i \mid x_j)$.

A probability distribution $p_{\mathcal{T}}(\mathbf{x})$ that is conformal with a tree is defined as:

$$p_{\mathcal{T}}(\mathbf{x}) = \prod_{i=1}^{l} p(x_i | pa(x_i)), \qquad (1)$$

Algorithm 1: **EA**

---
1 Set $t \Leftarrow 0$. Generate an initial population D_0 of $N \gg 0$ random solutions.
2 **do** {
3   Select from population D_t a set D_t^S of $k \leq N$ points using truncation selection.
4   Generate a new population D_{t+1} from D_t^S applying the variator operator of choice.
5 $t \Leftarrow t + 1$
6 } **until** Termination criteria are met.

---

where $pa(X_i)$ is the parent of X_i in the tree, and $p(x_i|pa(x_i)) = p(x_i)$ when $pa(X_i) = \emptyset$, i.e. X_i is a root of the tree. We allow the existence of more than one root in the PGM (i.e. forests) although for convenience of notation we refer to the model as tree.

The simplest example of tree distributions is the univariate model, where variables are assumed to be independent, i.e. none of the variables has a parent. For this model the probability of a solution is the product of the univariate probabilities for all variables:

$$p_u(\mathbf{x}) = \prod_{i=1}^{l} p(x_i) \qquad (2)$$

Our choice of the dependency tree and univariate models to investigate flag selection with EDAs is motivated by the fact that these models represent different assumptions about the types of interactions that exist between the variables of the problem. UMDA assumes that variables are independent. If the problem has no interactions, or if these interactions are sufficiently weak, then UMDA can produce good results. Tree-EDA learns the strongest pair-wise dependencies between the variables of the problem. Therefore, if pairwise dependencies are essential to solve the problem, Tree-EDA can outperform UMDA and other EAs that do not learn dependencies. A detailed explanation of how UMDA and Tree-EDA learn and sample their models is beyond the scope and space limitations of this paper. Details on the methods used to learn and sample the models can be obtained from [3, 20, 27].

5. EXPERIMENTS

In all the experiments presented in this paper we use truncation selection with parameter $T = 0.5$. The population size was fixed to $N = 100$, and the maximum number of generations, that was set as the termination criterion, was $n_{gen} = 50$. All programs were implemented in C++. The fitness evaluation function involves the compilation of the executable file and the execution of the program. This objective function was implemented by using the command *pclose* of C++ that allows calls to the command line. Experiments were run using an i7 Pentium processor.

To evaluate the algorithms, we use Auto/Industrial instances from the MiBench benchmark[1] [8] which is a freely available set of embedded benchmarks with problems from different domains. Our choice was motivated by the aim of investigating the behavior of EDAs also in embedded architectures. The work presented in this paper is a first step in
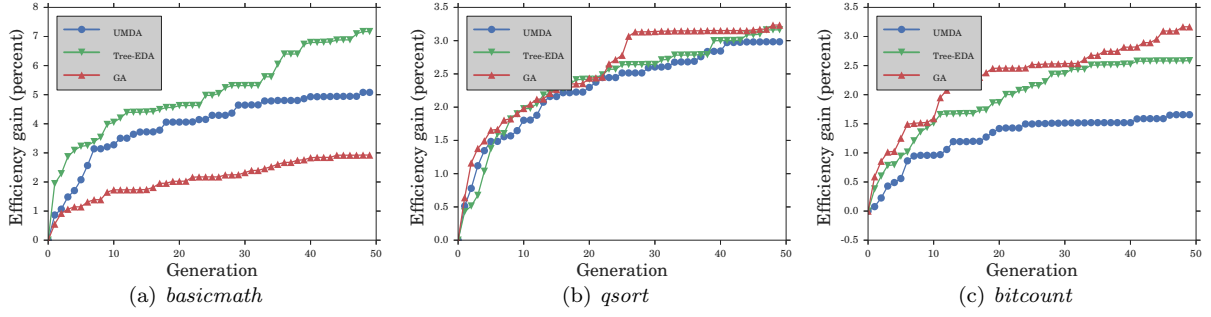
---
[1]http://wwweb.eecs.umich.edu/mibench/

Figure 1: Mean efficiency gain (percent) at each generation for the three EAs investigated (Small input).
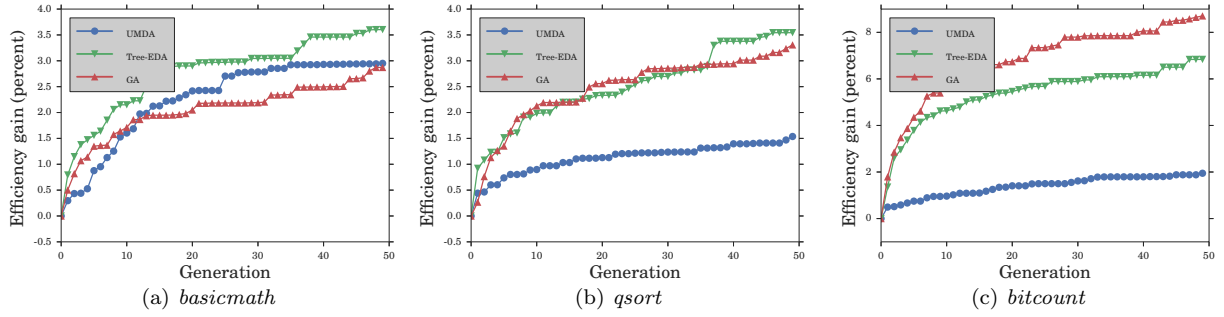

Figure 2: Mean efficiency gain (percent) at each generation for the three EAs investigated (Large inputs).
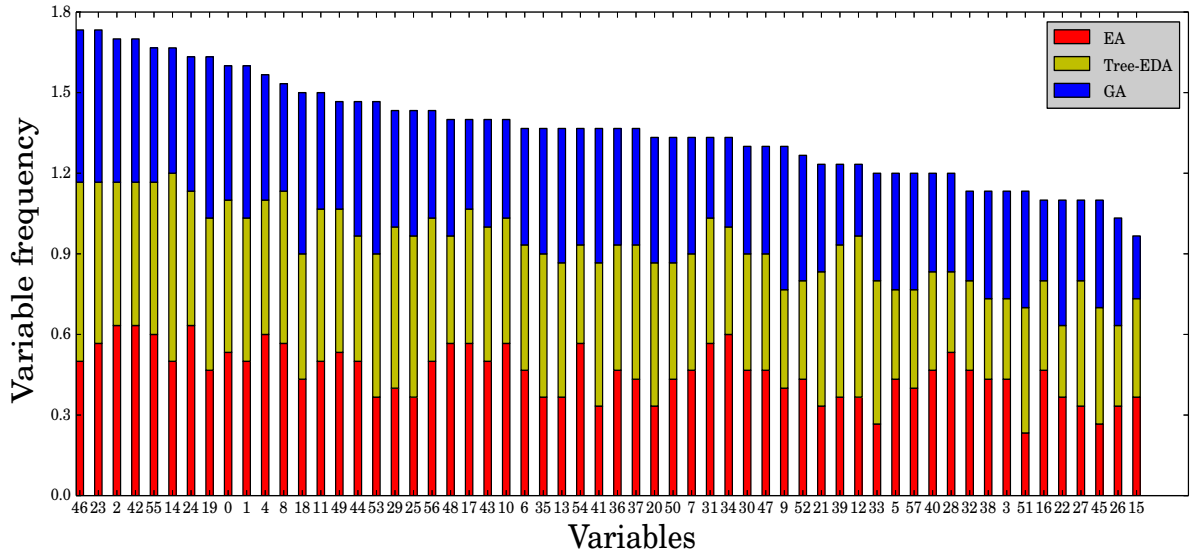

Figure 3: Frequency of the flags found by the three EAs in all the executions.

Auto./Industrial
basicmath
qsort
bitcount
susan (smoothing)
susan (edges)
susan (corners)

Table 1: Mibench instances used for the experimental framework.

tests, illustrating that significant differences exist between the algorithms. In Table 2, statistical differences, considering $p = 0.01$.

small	UMDA vs Tree	UMDA vs GA	Tree vs GA
basicmath	1.555e-4	6.400e-12	1.818e-15
qsort	-	-	-
bitcount	4.610e-09	7.797e-11	1.620e-3

large	UMDA vs Tree	UMDA vs GA	Tree vs GA
basicmath	2.435e-5	2.300e-3	8.295e-8
qsort	3.300e-14	1.743e-13	4.316e-1
bitcount	3.375e-16	1.355e-16	8.945e-08

Table 2: Results of the statistical tests.

that direction. The instances considered in our experiments are shown in Table 1. Some of these applications provide solutions to two levels of complexity of the same problem. One considering useful simple scenarios (small), while the other one is developed for a bigger or more complex problem (large). For all benchmarks we used two different types of inputs of the programs corresponding to small and large problems.

5.1 Comparison between the algorithms

Our first experiment focuses on determining the difference between the performance of GA, UMDA, and Tree-EDA. We ran 30 executions of each algorithm and compared their performance in terms of the average best fitness reached by the algorithms in all the runs.

Figure 1 shows the efficiency gain (percent) in each generation for the three EAs investigated and problems *basicmath* (a), *qsort* (b), and *bitcount* (c) when the input file is small. Similar results are shown in Figure 2 for the same problems but when the input file is large. The efficiency gain is computed as:

$$eg = \frac{f_0 - f_t}{f_0} \tag{3}$$

where f_0 is the fitness (elapsed time) of the best solution in the first generation and f_t is the best fitness in generation t. Since the algorithms use elitism, the best fitness is monotonically nondecreasing function along generations.

An analysis of the charts shown in Figure 1 and Figure 2 allows a number of conclusions: 1) The EAs allow efficiency improvements of up to 9% in some cases. 2) UMDA is never the best choice for none of the programs considered. This fact seems to indicate the need of capturing the dependencies to solve the problem. 3) Tree-EDA is always the first or second best algorithm for all the problems. It is better than the GA for the *basicmath* problem, the two algorithms are tied for problem *qsort*, and the GA is better than Tree-EDA for the *bitcount* problem. The relatively good results of the GA might be explained by the fact that this algorithm uses a one-point crossover that is less disruptive than uniform crossover or UMDA. Furthermore, the one-point crossover might be beneficial when interacting variables are close in the sequential representation.

For each of the six problems, we also tested for significant differences between the algorithms. A multiple comparison statistical test was conducted using the best solutions reached in each of the 30 runs. The Kruskal Wallis test was applied first, and the Dunn test was applied as post-hoc afterwards to look for statistical differences between each pair of algorithms. Table 2 shows the p-values output by the

Another relevant issue is to determine which are the most frequent flags found by the algorithms. Using the best solutions found in each of the 30 executions for problem *bitcount* large we computed the frequency of each of the 58 flags. Figure 3 shows the frequencies for all flags computed for all EAs. There are important differences in the frequency with which the flags are found by the algorithms. The three flags that appeared more frequently in the best solutions were *-fschedule-insns2*, *-ftree-salias*, and *-fkeep-static-consts*. These flags attempt to optimize execution time. Instrutions are reordered in order to avoid executions stalls due to required data bein unavailable. Also, memory location accesses are analyzed. The least frequent flags were *-frerun-loop-opt*, *-ftree-ter*, and *-ftree-ch*.

5.2 Analysis of the problem structure

In the next experiment we investigate whether Tree-EDA can produce efficiency gains for the Susan program. This is an image recognition package used for the analysis of magneto resonance images of the brain. Among the functions included are: recognizing corners, detecting edges, and smoothing the image. In this application different inputs of the same program can require different flag combinations. Therefore it is an interesting benchmark to evaluate the behavior of the algorithm for different tasks. In this case, the small input data is a black and white image of a rectangle while the large input data is a complex picture [8].

Figure 4 and show the efficiency gains achieved by Tree-EDA for the three different tasks, small and large inputs. In this example, Tree-EDA is able to obtain gains well above 8%. These gains differ according to the task to be solved. Higher gains in efficiency by means of flag combination optimization are achieved for the edge-detection task.

We used the Susan program benchmark to investigate and compare the most frequent interactions captured by the Tree-EDA and whether these interactions are common to all programs or depend on the task. To compute the interactions, we inspected all the trees learned by Tree-EDA in the first 10 generations of the algorithm for all the 30 executions. We only used trees corresponding to the first 10 generations since much of the efficiency gains are obtained in the first generations and because in the last generations some spurious interactions can arise between the variables due to genetic drift. For each problem, we calculate how frequent is each edge (i,j) in the 300 trees considered (30 executions × 10 generations).

Figure 5 shows the heat maps with the most frequent flag interactions for the three tasks ran by Susan program and the large input. The main conclusion from the analysis of
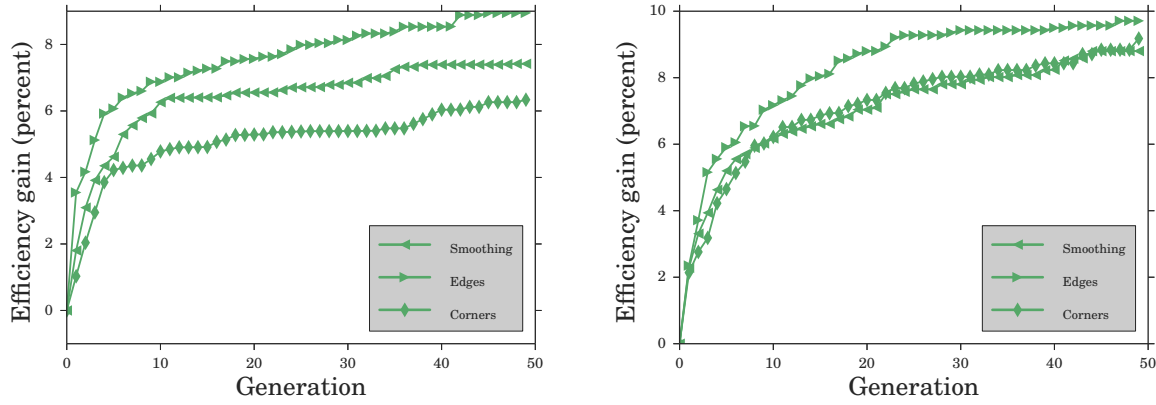
Figure 4: Mean efficiency gain (percent) at each generation for Tree-EDA on three different functionalities of program Susan when a) small images are considered and b) large images are considered.

the figures is that strong interactions between the variables arise but they are relatively few. It is important to notice that the tree models captures a maximum of $n - 1$ bivariate interactions between the variables. Therefore, they may miss some of the interactions of the problem. However, the fact that the interactions found are strong suggest that there exist pairs of flags that have a strong contribution to the optimization problem. These are the pairs of flags that should be taken into account by the user at the time of applying the solutions found by the EAs or trying to improve them.

As an additional step we investigated whether the optimal combinations of flags found for the three tasks were sufficiently different between them so that the type of task being solved could be identified from them. This approach intends to characterize of the programs (or to the tasks being solved) in terms of the flags combinations. We address this problem as a classification problem where we have three classes, each one corresponding to one of the applications of the Susan program: 1) Smoothing, 2) Edges, 3) Corners. For each of them, we have the optimal combination of flags found by Tree-EDA. The classification problem is trying to infer the class from the analysis of the flag combinations. We would expect that if flag combinations are unique to the tasks, then classes can easily be recognized and the classification accuracy will be high.

We solve the classification problem using the random forest classifier [6], one of the most popular classifiers used by the machine learning community. The confusion matrices obtained as the result of applying the classifier are shown in Equation (4) where CM_1 is the confusion matrix obtained for problem Susan, small inputs, and CM_2 is the one obtained when large inputs are used. It can be seen from the analysis of the confusion matrices that the classifier fails to accurately recognize the classes. The accuracies are around the expectation for a random classifier on a 3-class problem. We also computed the feature importance for all features (flags). This information is shown in Figure 6.

The figure shows that some of the flags have an important contribution to the classification but in this particular example this contribution is not translated into a high accuracy. Nevertheless, we suggest that considering classification problem from the analysis of the optimal flag combinations can be another way to characterize the problems.

6. CONCLUSIONS

In this paper we have proposed the use of EAs based on probabilistic modeling to make the right choice of compilation optimization options. Three EAs have been evaluated on different instances of a real-world benchmark: A GA with one-point crossover and bit-flip mutation, UMDA, and Tree-EDA. Our results show that modeling the dependencies is indeed required for an efficient optimization since UMDA, the algorithm that assumes all variables to be independent, produces the worst results. We found that for a number of instances the dependency-tree based EDA strategy outperforms the GA with statistical significant differences.

We also analyzed the most frequent flags found by the algorithms and the most frequent interactions identified by Tree-EDA. In the first case, we found differences in the frequencies of the flags inclusion in optimal combinations. We also found that the number of strong interactions captured by the trees is relatively small. More research is needed to characterize these interactions and to develop a method to exploit them a posteriori. Finally, we proposed the use of a classification approach for characterizing different programs (or procedures) in terms of their associated optimal flag combinations. In the example considered in this paper, the classifier did not find indeed major differences between flag combinations. However, more research is needed to explore the validity of this approach in other scenarios, e.g. comparing different programs and not different tasks to be fulfilled by the same program.

Although we have focused on compiler optimization with minimization of the computation time as the goal, the same approach could be applied to optimize other parameters that describe the behavior of the algorithm, like the size of the program or the energy consumption [21]. Similarly, the evolutionary algorithms could be used to optimize compilation options in different platforms. Finally, EDAs that learn

$$CM_1 = \begin{pmatrix} 11 & 13 & 6 \\ 10 & 10 & 10 \\ 10 & 9 & 11 \end{pmatrix} \quad CM_2 = \begin{pmatrix} 5 & 14 & 11 \\ 11 & 11 & 8 \\ 8 & 9 & 13 \end{pmatrix} \quad (4)$$
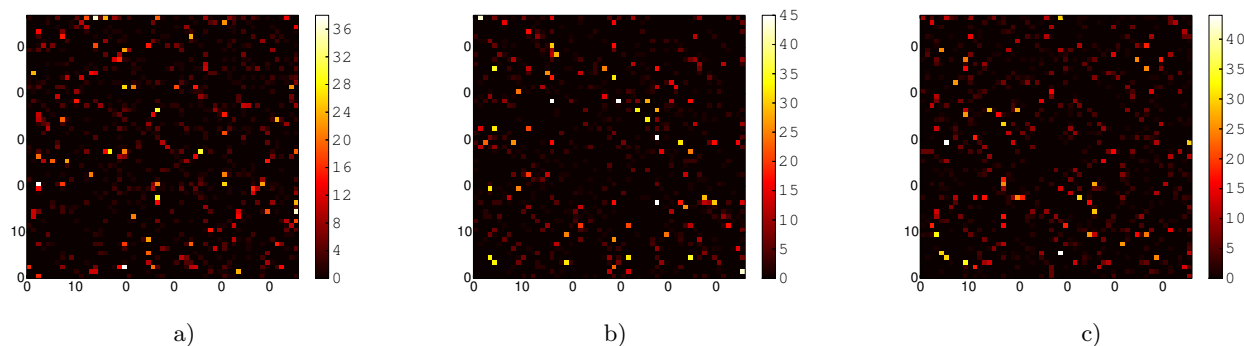
Figure 5: Most frequent interactions captured by Tree-EDA for the three different functionalities of program Susan when large images are considered. a) Smoothing, b) Edges, c) Corners
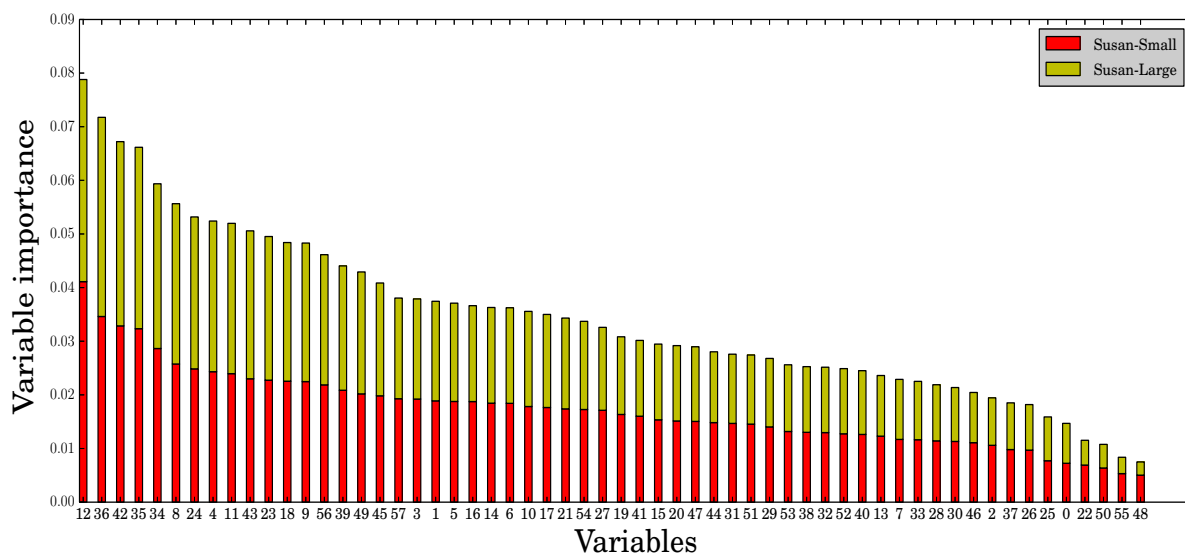

Figure 6: Feature importance found by the random forest classifier when classifying programs from the optimal flag sets.

more complex models [22, 28], able to represent a wider class of patterns of interactions between the compilation flags could be also investigated.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] A. H. Ashouri, G. Mariani, G. Palermo, and C. Silvano. A Bayesian network approach for compiler auto-tuning for embedded processors. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium on*, pages 90–97. IEEE, 2014.

[2] P. A. Ballal, H. Sarojadevi, and P. Harsha. Compiler optimization: A genetic algorithm approach. *International Journal of Computer Applications*, 112(10), 2015.

[3] S. Baluja and S. Davies. Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. In D. H. Fisher, editor, *Proceedings of the 14th International Conference on Machine Learning*, pages 30–38, 1997.

[4] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 257–269. ACM, 2015.

[5] D. Branco and P. R. Henriques. Impact of GCC optimization levels in energy consumption during C/C++ program execution. In *Scientific Conference on Informatics, 2015 IEEE 13th International*, pages 52–56. IEEE, 2015.

[6] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[7] L. N. Chakrapani, P. Korkmaz, V. J. Mooney III, K. V. Palem, K. Puttaswamy, and W.-F. Wong. The

emerging power crisis in embedded processors: what can a poor compiler do? In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 176–180. ACM, 2001.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[9] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–14. ACM, 2012.

[10] 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*, pages 165–174. ACM, 2008.

[11] M. R. Jantz and P. A. Kulkarni. Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pages 1–10. IEEE, 2013.

[12] P. Joseph, M. Jacob, Y. Srikant, and K. Vaswani. Statistical and machine learning techniques in compiler design. *The compiler design handbook, optimization and machine code generation. CRC Press, Boca Raton*, 2008.

[13] J. Kukunas, R. D. Cupper, and G. M. Kapfhammer. A genetic algorithm to improve linux kernel performance on resource-constrained devices. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pages 2095–2096. ACM, 2010.

[14] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *ACM SIGPLAN Notices*, volume 38, pages 12–23. ACM, 2003.

[15] P. Larrañaga and J. A. Lozano, editors. *Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2002.

[16] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[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*, pages 1–8. IEEE, 2008.

[18] S. Long and G. Fursin. A heuristic search algorithm based on unified transformation framework. In *Parallel Processing, 2005. ICPP 2005 Workshops.*

International Conference Workshops on, pages 137–144. IEEE, 2005.

[19] J. A. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea, editors. *Towards a New Evolutionary Computation: Advances on Estimation of Distribution Algorithms*. Springer, 2006.

[20] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. In *Parallel Problem Solving from Nature - PPSN IV*, volume 1141 of *Lectures Notes in Computer Science*, pages 178–187, Berlin, 1996. Springer.

[21] J. Pallister, S. J. Hollis, and J. Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015.

[22] M. Pelikan. *Hierarchical Bayesian Optimization Algorithm. Toward a New Generation of Evolutionary Algorithms*, volume 170 of *Studies in Fuzziness and Soft Computing*. Springer, 2005.

[23] R. Sagarna, A. Mendiburu, I. Inza, and J. A. Lozano. Assisting in search heuristics selection through multidimensional supervised classification: A case study on software testing. *Information Sciences*, 258:122–139, 2014.

[24] R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using machines to learn method-specific compilation strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 257–266. IEEE Computer Society, 2011.

[25] T. Sandran, N. Zakaria, and A. J. Pal. An optimized tuning of genetic algorithm parameters in compiler flag selection based on compilation and execution duration. In *Proceedings of the International Conference on Soft Computing for Problem Solving (SocProS 2011) December 20-22, 2011*, pages 599–610. Springer, 2012.

[26] N. Sankar Chebolu and R. Wankar. A novel scheme for compiler optimization framework. In *Advances in Computing, Communications and Informatics (ICACCI), 2015 International Conference on*, pages 2374–2380. IEEE, 2015.

[27] R. Santana, P. Larrañaga, and J. A. Lozano. Protein folding in simplified models with estimation of distribution algorithms. *IEEE Transactions on Evolutionary Computation*, 12(4):418–438, 2008.

[28] S. Shakya and R. Santana, editors. *Markov Networks in Evolutionary Computation*. Springer, 2012.

[29] G. Sher, K. Martin, and D. Dechev. Preliminary results for neuroevolutionary optimization phase order generation for static compilation. In *Proceedings of the 11th Workshop on Optimizations for DSP and Embedded Systems*, pages 33–40. ACM, 2014.

[30] M. Valluri and L. K. John. Is compiling for performance compiling for power? In *Interaction between Compilers and Computer Architectures*, pages 101–115. Springer, 2001.

[31] K. Vaswani, M. J. Thazhuthaveetil, Y. Srikant, and P. Joseph. Microarchitecture sensitive empirical models for compiler optimizations. In *Code Generation and Optimization, 2007. CGO-07. International Symposium on*, pages 131–143. IEEE, 2007.