Predictive Modeling Methodology for Compiler Phase-Ordering

Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo and Cristina Silvano
{name.family-name}@polimi.it
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, Italy

ABSTRACT

Today's compilers offer a huge number of transformation options to choose among and this choice can significantly impact on the performance of the code being optimized. Not only the selection of compiler options represents a hard problem to be solved, but also the ordering of the phases is adding further complexity, making it a long standing problem in compilation research. This paper presents an innovative approach for tackling the compiler phase-ordering problem by using predictive modeling. The proposed methodology enables *i)* to efficiently explore compiler exploration space including optimization permutations and repetitions and *ii)* to extract the application dynamic features to predict the next-best optimization to be applied to maximize the performance given the current status. Experimental results are done by assessing the proposed methodology with utilizing two different search heuristics on the compiler optimization space and it demonstrates the effectiveness of the methodology on the selected set of applications. Using the proposed methodology on average we observed up to 4% execution speedup with respect to LLVM standard baseline.

Keywords

Compilers, Autotuning, Phase-ordering, Machine Learning

1. INTRODUCTION

Selecting the best ordering of compiler optimizations for an application has been an open problem in the field for many decades and the problem is known to be NP-complete. The unrealistic exhaustive search is the only solution that seems appealing to achieve the optimal solution. Compiler researchers rely on their insights on the compiler *backend* to come up with some predefined sequences and ordering. This process is usually done tentatively and the selected pass is constructed with little insight on the interaction between the selected compiler options. However, to come up with an optimal solution, researchers might have to spend several years to run different code variants and this is simply unfeasible, given the growing design space composed of different architectures and software models that rely on modern compiler frameworks. As an example, GCC compiler has more than 200 compiler passes and LLVM-OPT has more than 100, and these optimizations are working on different layers of application e.g. *analysis passes, loop-nest passes*, etc. Most of the passes are usually turned off by default and compiler developers rely on software developers to know which optimization can be beneficial for their code. The so-called *average case* has been defined as to get certain *standard optimization levels*, e.g. O1, O2, Os, etc. to introduce a fix sequence of compiler options, that on average can bring good results for most applications. Given the peculiarity of the problem, this certainly is not enough.

Exploiting compiler optimizations in application-specific *embedded domains*, where applications are compiled once and then deployed on the market on millions of devices is troublesome. The reason why is firstly because embedded systems are usually designed with tight extra-functional properties constraints. Secondly, the large variety of embedded platforms can not be faced with the average case provided by standard optimization levels, thus custom compiler optimization sequences might lead to substantial benefits in reference to several performance metrics (e.g. execution time, power consumption, memory footprint).

In the *High Performance Computing* (HPC) domain, parallel computer systems are increasingly more complex. Currently, HPC systems offering peak performance of several Petaflops have hundreds of thousands of cores to be managed efficiently. Those machines have deep software stack, which has to be exploited by the programmer to tune the program. Moreover, to reduce the power consumption of those systems, advanced hardware and software techniques are applied, such as the usage of GPUs that are highly specialized for regular data parallel computations via simple processing cores and high bandwidth to the graphics memory. Numerous scientific and engineering compute-intensive applications spend most of their execution time in loop nests that are suitable for high-level optimizations. Typical examples include dense linear algebra codes and stencil-based iterative methods [28]. *Polyhedral compilation* is a recent attempt to bring mathematical representation focusing on the loop-nest of the *polyhedral model* including many different tools [4, 5, 16, 20].

In this paper, we tackle the phase-ordering problem by using predictive modeling. Our predictive model is able to introduce the next-best compiler optimization to be applied given the current status of the application to maximize the performance. The status of the application is defined by a *vector of representative features* that has been collected dynamically and it is independent from the architecture the code is running on. The proposed predictive model has been trained off-line with different permutations of the compiler flags (allowing repetitions and dynamic sequence length). Therefore, the proposed method receives as input the *program features* and it generates the next-best compiler option to maximize the performance of the application. We selected a set of benchmark applications to assess the benefits of the proposed approach and to prove its feasibility.

In this paper, we propose a predictive modeling methodology to mitigate the phase-ordering problem, In particular,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PARMA-DITAM '16, January 18 2016, Prague, Czech Republic

© 2016 ACM. ISBN 978-1-4503-4052-6/16/01…$15.00

DOI: http://dx.doi.org/10.1145/2872421.2872424

the main contributions are:

- Predictive modeling methodology capable of capturing the correlation between the program features and the compiler optimization at each state.
- The integration of the predictive modeling within a compiler framework. The generated model is trained by means of Machine Learning to focus on the next-immediate best compiler optimization to be applied given the current status of the application for any new previously unobserved program.
- Tackling the phase-ordering problem on utilizing different relative positioning of the sequences of compiler options previously acquired as good sequences from LLVM standard optimization level and explore the design space by using a larger set of compiler flags rather than the individual options.
- Predictive modeling capable of iteratively predicting the next-best compiler option using two search heuristics namely to be applied given the current status of the application being optimized.

We apply prediction modeling techniques originally proposed in [9]. However, the original work was mostly performing predictions on fixed optimization vectors length, while our proposed model is able to iteratively call the function and generate the next-best optimization to be applied, given the current status of the application. This feature is certainly vital for the phase-ordering problem because of: **i)** it opens up to complete the new states towards exploring more regions of interest in the design space and **ii)** it enables us to apply *repetitions* on the application being optimized. Moreover, the original work was tackling the problem of *selection of best compiler optimization*, while the current work is targeting the substantially harder problem of *phase-ordering*.

The rest of the paper has been organized as follows. Section 2 provides a brief discussion on the related work. In Section 3, we introduce the predictive modeling approach to tackle *phase-ordering*. Section 4 presents experimental evaluation of the proposed methodology on an Unix-based Intel platform. Finally, Section 5 summarizes the outcome of the work and some future paths.

2. RELATED WORK

Literature on the *phase-ordering* problem is closely tightened with the *selection of the best compiler options* problem. Therefore, study on the literature can be classified in two main classes: i) *autotuning and iterative compilation approaches* and ii) *applying machine learning to compilation*. Nevertheless, these two approaches have been amalgamated in many ways by exploiting different techniques and methodologies.

2.1 Iterative Compilation and Autotuning

Autotuning addresses automatic code generation and optimization by using different scenarios and architectures. It involves building techniques for automatic optimization of different parameters in order to maximize or minimize the fulfillment of an objective function. One strategy in autotuning consists of coupling the approach with random generation of code variances at each run. It is well known that *Random Iterative Compilation* (RIC) can generally improve application performance in reference to static handcrafted compiler optimization sequences [1]. Given the complexity of the *iterative compilation* problem, it has been proved that drawing compiler optimization sequences at random is as good as applying other optimization algorithms such as genetic algorithms or simulated annealing [1, 7, 8]. Other authors [3] have exploited the Design Space Exploration (DSE) techniques jointly with architectural DSE for VLIW architectures.

2.2 Exploiting Machine Learning

Applying machine learning has been investigated by many researchers in the recent literature on compiler optimization. The papers in [27, 22] were among the very first research works introducing the use of machine learning. Other authors [2, 26, 25, 24] have tackled the problem of selection of compiler options with Bayesian Networks and independent application characterization technique, predictive modeling with dynamic characterization, predictive modeling using *Intermediate Representation* (IR) and Control Flow Graphs (CFG).

There are quite a few studies that have tackled the *phase-ordering* problem. Authors in [19] have applied *Neuro- Evolution for Augmenting Topologies* (NEAT) on Jikes dynamic compiler and come up with sets of good ordering of phases. Other works have approached the problem by exploiting compiler backend optimizations and using statistical tests to reduce code-size [12]. Authors in [18] exhaustively exploring the ordering space at functions' granularity level and evaluate their methodology with search tree algorithms and in [23] the authors have exploited iterative compilation with the information relying on relative passes in previously generated compiler options in the sequence in function level.

Our approach is rather different with respect to the literature, given that we propose a predictive modeling methodology utilizing an independent micro-architecture characterization features for all different *permutations with repetitions* of the compiler options and come up with the prediction of the *next-best compiler option to be applied* on the application given the current status. The proposed work is able to come up with good set of compiler options even with *dynamic lengths* and it is not just limited to fixed vector length. We use previously acquired relative positioning of the promising sequences utilized on LLVM standard optimization levels and treat each of those acquired sequences as one whole. In this case, we could apply phase ordering feasibility on a larger set of compiler options, while generating less design space in the problem.

3. THE PROPOSED METHODOLOGY

Main goal of the proposed methodology is to identify the feasibility of tackling the phase ordering problem using a predictive modeling methodology. Each application optimized with a unique compiler options sequence is passed through a characterization phase, that generates parametric representation of its dynamic features. A model based on predictive modeling correlates these features to the compiler optimizations applied such as to predict the application speedup by using the next-best compiler optimization at each level.

The optimization flow is represented in Figure 1. It consists of three main phases: i) *Data collection* where different instances of the application are executed and the application characterizations are fetched with the speedup achieved by utilizing the specific option, ii) *Training phase* where the predictive modeling is learned on the base of a set of training applications and iii) *Exploitation phase*, where new applications are optimized by exploiting the knowledge stored in the trained predictive model. The model is able to predict, given the current program characterization, the immediate speedups associated to each of the compiler optimization under analysis.

During the second and the third phases, an optimization process is necessary to identify the best compiler optimizations to be enabled to get the best performance. This is done for learning purposes during the *training phase* and for optimization purposes during the *exploitation phase*. To implement the optimization process, a Design Space Exploration (DSE) engine has been used. This DSE engine compiles, executes and measures application performance by enabling and disabling different permutations with repetitions of compiler optimizations.

In our approach the DoE is obtained by *exhaustive explo-*

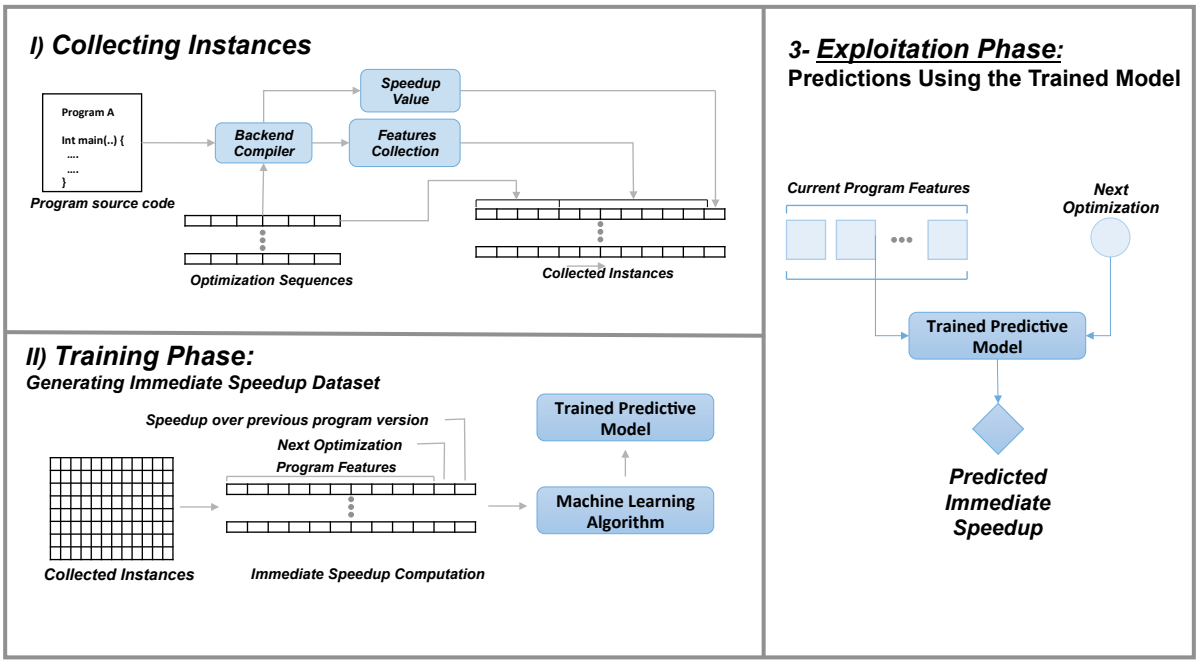

Figure 1: Proposed Predictive Modeling Methodology

ration including all permutations with repetitions of compiler configurations (during the training phase as shown in Figure 1) either by means of the *whole sequence* at once or *the current compiler optimization* that has been applied to the previous state. On the other side, *exploitation phase*, they are generated by means of predicting the *whole sequence* at once or as *the next-best configuration* to be applied given the current status. These two different techniques will be elaborated more in Sections 3.3.1 and 3.3.2 respectively.

3.1 Compiler Phase-ordering Problem

To formulate the phase-ordering problem, first we come up with the *selection of the best compiler sequence* problem. Let us define a Boolean vector $\boldsymbol{o}$ whose elements o_i are the different compiler optimizations. Each optimization o_i can be either enabled $o_i = 1$ or disabled $o_i = 0$. A compiler optimization sequence to be *selected* is represented by the vector $\boldsymbol{o}$ belongs to the n dimensional Boolean space of:

$$\mathcal{O}selection = \{0,1\}^n \tag{1}$$

For the application a_i being optimized and n represents the number of compiler optimizations under study. Therefore, the mentioned research problem consists of an exponential space as its upper-bound. Having $n = 10$, drive us to a total space (2^n) up to $|\mathcal{O}selection| = 1024$ options to select among per interested target application a_i to be optimized and this number itself would be multiplied by different applications $A = a_0...a_N$ under study.

Coming back to the *phase-ordering* problem, let us define a Boolean vector $\boldsymbol{o}$ whose elements o_i are the different compiler optimizations. A *Phase-ordering* compiler optimization sequence represented by the vector $\boldsymbol{o}$ belongs to the n dimensional factorial space $|\mathcal{O}phases| = n!$, where n represents the number of compiler optimizations under study. However, the mentioned bound is for a simplified *phase-ordering* problem given fixed vector length without repetitions. Enabling repetitions and dynamic length will expand the design space size to:

$$|\mathcal{O}phases\_repetition| = \sum_{i=0}^{m} n^i \tag{2}$$

Where n is the number of interesting optimizations under study and m is the maximum desired length for the optimization sequence length. In this case, assuming the same n and m equal to 10, $|\mathcal{O}phases\_repetition|$ will drive up to more than 11 *Billion* different configurations to select per each application. [1]

The o_i in this work consists more than one single compiler optimizations. These set of optimizations are derived from the LLVM standard optimization level. Reader can refer to the specific o_i in Section 4 Table 2. We treat each of the whole sequence (which being referred to as *genes*) as a discrete variable, so that each optimization o_i can be either enabled $o_i = 1$ or disabled $o_i = 0$ and enabling the o_i will enable all its contained sub-optimizations respectively.

3.2 Application Characterization

In this work, we use PIN [21] based *dynamic profiling* framework to analyze the behavior of the different applications at execution time. In particular, the selected profiling framework provides a high level *Micro-architectural Independent Characterization of Applications* (MICA) [14] suitable for characterizing applications in a cross-platform manner. Furthermore, there is no static syntactic analysis, but the framework is based solely on MICA profiling.

In our experimental setup, an application is compiled and profiled on an x86 host processor, while the target architecture where the application executes (i.e. the architecture for which the application shall be optimized) could be a different platform thanks to the high level abstraction of the application characterization carried out with MICA, so as we can easily change the target architecture without the need of changing the profiling infrastructure.

The MICA framework reports data about *instruction type*, *memory and register access pattern*, potential *instruction level parallelism* and a dynamic control flow analysis in terms of *branch predictability*. Overall, the MICA framework characterizes an application in reference to 99 different metrics (or features). However, many of these 99 features are strongly correlated (e.g. the number of memory reads with stride smaller than 1K is bounded by the number of reads with stride smaller than 2K). Furthermore, generating a *pre-*

[1]The problem of phase-ordering does not have a definite upper-bound in the case of having repetitions with unbounded *max_length*($\mathcal{O}$).

dictive model is a process whose time complexity grows up with the number of parameters in use. It is too expensive to include all the 99 features in the model. Given the goal of speeding up the construction, we applied *Principal Component Analysis* (PCA) to reduce the number of parameters to characterize the application. PCA is a technique to transform a set of correlated parameters (application features) into a set of orthogonal (i.e. uncorrelated) principal components. The PCA transformation aims at sorting the principal components by descending order based on their variance [15]. For instance, the first components include the most of the input data variability, i.e. they represent the most of the information contained in the input data. To reduce the number of input features, while keeping most of the information contained in the input data, it is simply needed to use the first k principal components as suggested in [14]. In particular, we set $k = 10$ to trade off the information stored in the application characterization and the time required to train the *predictive modeling*.

3.3 Speedup Prediction Modeling

The general formulation of the optimization problem is to construct a function that takes as input the features of the *current status* of a program being optimized to generate as output the *the next-best optimization* to be applied that maximize the *immediate predicted speedup*. We used the prediction model originally proposed in [9]. However, the original work was mostly performing predictions on fixed optimization vectors length, while our proposed model is able to iteratively call the function and generate the next-best optimization to be applied, given the current status of the application. This feature is certainly vital for the phase-ordering problem because of: **i)** it opens up to complete the new states towards exploring more regions of interest in the design space and **ii)** it enables us to apply *repetitions* on the application being optimized.

An application is parametrically represented by the vector $\boldsymbol{\alpha}$, whose elements α_i are the first k principal components of its dynamic profiling features. Elements α_i in the vector $\boldsymbol{\alpha}$ generally belong to the continuous domain. The optimal compiler optimization sequence $\bar{\boldsymbol{o}} \in \mathcal{O}$ that maximizes the performance of an application is generally unknown. However it is known that the effects of a compiler optimization o_i might depend on whether or not another optimization o_j is applied.

Our models predict optimizations to apply to unseen programs that were not used in training the model. To this purpose, we need to feed as input a characterization of the unseen program. The model is able to predict the speedup of each possible optimization set $\mathcal{O}$ in our predictive optimization space, given the characteristics of the unseen program. We order the predicted speedups to determine which optimization set is predicted best, and we apply the predicted best optimization set(s) to the unseen program. In the experimental Section 4, we use a leave-one-benchmark-out cross-validation procedure for evaluating the models. The proposed predictive modeling is going to introduce two different heuristics on predictive modeling.

3.3.1 DFS Search Heuristic

Depth-First Search (DFS) and its optimized version Depth-First Iterative Deepening [17] are well-known tree traversing algorithms. DFS starts at the root and explores as far as possible along each branch before backtracking. Adapting the heuristic on the current problem, we propose to start from an empty optimization sequence $\boldsymbol{o}_o$, Considering sequence $\boldsymbol{o}_i$, for each of the possible optimizations we predict the immediate speedup δ_i derived from applying o_i after $\boldsymbol{o}_i$ in the compilation process. The *immediate speedup* is computed as:

$$e_i = Exec\_time(\boldsymbol{o}_i)/Exec\_time(\boldsymbol{o}_j) \quad (3)$$

where e_i is the ratio between the execution time of the program compiled using $\boldsymbol{o}_i$ and the execution time of the version of the program generated using $\boldsymbol{o}_j$. We define $\boldsymbol{o}_j$ as $\boldsymbol{o}_i$ followed by o_i.

Then, we order the possible optimizations by the value of the predicted immediate speedup δ. If none of the optimizations o_i to be explored has an associated immediate speedup δ_i greater than 1, we choose $\boldsymbol{o}_i$ as the next sequence to test, and we use it to compile the program and measure corresponding execution time. Otherwise, we repeat the same exploration process starting from sequence $\boldsymbol{o}_j$, that is $\boldsymbol{o}_i$ followed by the still unexplored o_i maximizing predicted immediate speedup δ_i.

Once all the possible optimizations o have been explored, and the original sequence $\boldsymbol{o}_i$ tested, the algorithm backtracks to the previous considered sequence $\boldsymbol{o}_k$, that is, $\boldsymbol{o}_i$ without its last optimization. If a sequence $\boldsymbol{o}_i$ has reached the maximum optimization sequence length N to be considered, we stop applying further optimizations after it. We then evaluate $\boldsymbol{o}_i$ it and backtrack to the previous node. The algorithm explores the complete optimization space using this policy and terminates when reaching the backtracking point for the initial empty sequence $\boldsymbol{o}_o$.

3.3.2 Exhaustive Search Heuristic

The second iterative approach is tackling the exploration with exhaustive search. A model trained using machine learning techniques produces speedup predictions for all the configurations in the complete considered sequence space. Ordered by decreasing predicted speedup values, the sequences are then applied to the program, and their actual speedup is measured. This approach has been successfully used in selection of the best compiler sequences problem, but lacks of applicability in the phase-ordering problem, given the complexity increase of the configuration space.

In our specific case, we modify this methodology to adapt it to our application scenario. In particular, as described previously at Section 3.3.1 Equation 4, the model we trained is able to predict only immediate speedups δ (i.e. the speedup of $\boldsymbol{o}_j$ over $\boldsymbol{o}_i$, where $\boldsymbol{o}_j$ is the optimization sequence obtained by applying o_i after $\boldsymbol{o}_i$. We are able to predict the actual speedup of optimization sequence $\boldsymbol{o}$ by *multiplying* the immediate speedups δ_i predicted at each optimization $o_i \in \boldsymbol{o}$:

$$\boldsymbol{o}_o : \prod_{o_i \in o} \delta(o_i) \quad (4)$$

where o_i is each individual optimization options to be explored. The proposed exhaustive exploration computes the immediate speedups starting from the initial empty sequence $\boldsymbol{o}_o$ to the complete optimization space. In this way, we are able to predict the speedup of every optimization sequence $\boldsymbol{o} \in \mathcal{O}$ and map our system to the classic exhaustive predictions methodology we described.

4. EXPERIMENTAL EVALUATION

In this section, we assess the proposed methodology of the *immediate next-best predictive modeling* on quad-core Intel-Xeon E1607 running at 3.00 GHZ. We have used LLVM compilation tool v3.8 within our framework. A subset of cBench benchmark suite [10] consisting of six different applications has been integrated within the framework. The list of selected applications has been reported in Table 1. Table 2 presents the utilized sets of LLVM optimizations categories in 4 different genes. The utilized passes are part of the LLVM standard optimization levels and we exploited the phase-ordering scenario having them relatively fixed internally, while altering the whole sequence externally at each phase. In this mode, we speculated that we could explore more interesting regions of the design space and reaching higher potential speedups accordingly. The utilized 4 different genes, consists of 30 compiler optimizations in total

Table 1: Applications used in this work

Applications	Description
automotive_bitcount	Bit counter
automotive_qsort1	Quick sort
automotive_susan_c	Smallest Univalue Segment Assimilating Nucleus Corner
automotive_susan_e	Smallest Univalue Segment Assimilating Nucleus Corner
network_dijkstra	Dijkstra's algorithm
network_patricia	Patricia Trie data structure

Table 2: Compiler optimizations under analysis: Derived from LLVM-Opt

Gene	Abbreviation	Relative Positioning of the Optimizations
A	*domtreeRULE*	-domtree -memdep -dse -adce -instcombine -simplifycfg -domtree -loops -loop-simplify -lcssa -branch-prob
B	*simplifycfgRULE*	-simplifycfg -reassociate -domtree -loops -loop-simplify
C	*memdepRULE*	-memdep -domtree -memdep -gvn -memdep -memcpyopt -sccp
D	*loopsRule*	-loops -loop-simplify -lcssa -branch-prob -block-freq -scalar-evolution -loop-vectorize
Total Number of Optimizations Under Analysis		**30**
Unique Number of Optimizations Under Analysis		**13**

and 13 unique optimizations. One of the features of the proposed methodology is that it supports repetitions in the compiler design space. Table 3 represents the maximum achievable speedup enabling repetition feature on every individual applications. We observe that excluding two applications that coincidentally had their gene having the best achievable speedup without repetitions (thus enabling repetition was converging to the very same result), the other four are gaining benefits from enabling this features. We used harmonic mean to better report the average of the speedups rather than arithmetic mean here [13].

The application execution time is estimated by using the *Linux-Perf* tool. Execution time required to process a given data set by a compiled application binary is estimated by averaging four different executions. In order to implement dimension reduction technique, we used PCA having set PCs to 10 in this work. The PCA components have been computed by using the MICA features collected from each application run, normalized by standard deviation across all data sets. Our exploration experiments have been generated by using the data previously collected offline.

WEKA Machine Learning tool [11] has been integrated in our framework to exploit predictive modeling algorithms. More in detail, in this work, we assessed the proposed methodology with *Linear Regression* (LR) Machine Learning algorithm activating the *M5* attribute selection method with default ridge parameter. Experimental results has been carried out by means of *leave-one-out cross-validation*. Given a new unseen application in the training set, the current program feature already obtained offline will impose a bias on the trained model and the predictive modeling will be able to predict the *next-best optimization* to be applied to maximize the immediate speedup in a greedy manner. As mentioned in Equation 2, given the complexity of the problem, we assessed the feasibility of our proposed approach with four different sequences of compiler options with a total of 30 compiler options (13 unique optimizations) to be used in the design space. Generating different permutations with repetitions and enabling dynamic sequence length led to 341 different variations and 2046 for all considered applications.

Table 3: Maximum Achievable Performance Enabling Repetition

Application	Best Opt Found W.Rep	Best Opt Found WOut.Rep	Speedup %
automotive-susan-c	CDD	CD	9.34
network-patricia	CDCD	AD	60.37
automotive-qsort1	CBA	CBA	-
automotive-bitcount	CCBB	CDB	2.41
network-dijkstra	ACAB	AD	36.59
automotive-susan-e	CD	CD	-
Harmonic Mean	-	-	7.68

Table 4: Average speedup of the one-shot prediction for both approaches w.r.t LLVM baseline

Application	Greedy DFS	Exhaustive Predictions
automotive-susan-c	0.9808	0.9658
network-patricia	1.0069	1.002
automotive-qsort1	1.1255	0.9670
automotive-bitcount	1.0848	1.1506
network-dijkstra	0.9988	0.9988
automotive-susan-e	1.0617	1.1015
Average	**1.0431**	**1.0242**

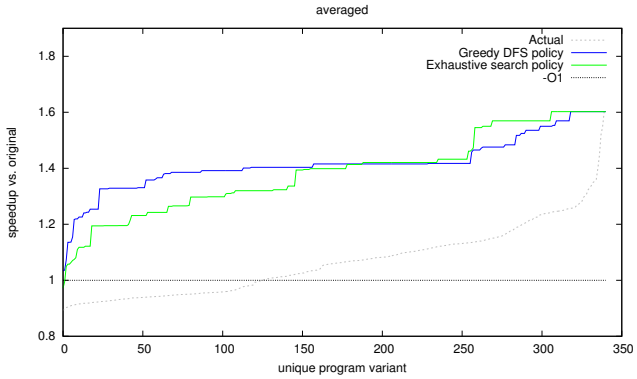

Figure 2: Average Speedup of the proposed methodology among all the applications

The results obtained by exploring the phase-ordering space with the two proposed heuristics is reported in Figure 2. It shows that the revised DFS search approach, namely an *Iterative Deepening First Search* policy (based on a greedy Depth First Search (DFS) heuristic) is doing slightly better with respect to the *exhaustive search heuristic* presented in Section 3.3. We define the *actual speedup* line as as the speedup observed from running the application using the compiler optimizatioj prediction of the machine learning model. Table 4 is presenting the quantitative values of utilizing the two predictive modeling algorithms within our framework. We evaluated our *iterative greedy approaches* with the classic 1-shot predictive speedup approach mentioned in [6, 9, 25] and the results reported in Table 4 are the average output of the one-shot approach per application. One-shot approach is extracting the prediction by means of one-extraction only and observe its speedup gain. In average, these two search algorithms demonstrated respectively 4% and 2% performance speedup over LLVM default performance.

The graph in Figure 2 demonstrates that the performance of the greedy exploration policy in early generation of the prediction is better than the exhaustive search methodology for the selected benchmarks. The horizontal axis represents different variations of the applications. This is rather interesting because in the phase-ordering problem the cardinality of the optimization sequence space $\mathcal{O}$ is too huge for an exhaustive search policy to be applied. On average, by traversing 15% of the compiler design space, we can reach up to 80% of the best found options in the design space.

5. CONCLUSIONS AND FUTURE WORK

This paper presents a method based on predictive modeling to select the next-best immediate compiler option to be applied to maximize the application performance. Experimental results exploiting two different search heuristics on the selected set of benchmarks demonstrated respectively 4% and 2% performance speedup with respect to the default LLVM compiler framework. Future work will focus on building more-accurate predictive models capable of capturing the intra/inter-correlation effects of different compiler options.

6. ACKNOWLEDGMENTS

This work is partially funded by the European Union's Horizon 2020 research and innovation program under grant agreement ANTAREX-671623.

7. REFERENCES

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305. IEEE Computer Society, 2006.

[2] 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.

[3] A. H. Ashouri, V. Zaccaria, S. Xydis, G. Palermo, and C. Silvano. A framework for compiler level statistical analysis over customized vliw architecture. In *Very Large Scale Integration (VLSI-SoC), 2013 IFIP/IEEE 21st International Conference on*, pages 124–129. IEEE, 2013.

[4] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[6] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 24–34. ACM, 2006.

[7] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):21, 2012.

[9] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th international conference on Computing frontiers*, pages 131–142. ACM, 2007.

[10] G. Fursin. Collective benchmark (cbench), a collection of open-source programs with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization, 2010.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[12] M. Haneda, P. M. Knijnenburg, and H. A. Wijshoff. Code size reduction by compiler tuning. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 186–195. Springer, 2006.

[13] T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 73. ACM, 2015.

[14] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[15] K. J. Johnson and R. E. Synovec. Pattern recognition of jet fuels: comprehensive gc× gc with anova-based feature selection and principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 60(1):225–237, 2002.

[16] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

[17] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[18] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1):1, 2009.

[19] S. Kulkarni and J. Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices*, 47(10):147–162, 2012.

[20] V. Loechner. Polylib: A library for manipulating parameterized polyhedra, 1999.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[22] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50. Springer, 2002.

[23] R. Nobre, L. G. Martins, and J. M. Cardoso. Use of previously acquired positioning of optimizations for phase ordering exploration. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, pages 58–67. ACM, 2015.

[24] E. Park, J. Cavazos, and M. A. Alvarez. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 196–206. ACM, 2012.

[25] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International Journal of Parallel Programming*, 41(5):704–750, 2013.

[26] E. Park, S. Kulkarni, and J. Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 65–74. ACM, 2011.

[27] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices*, volume 38, pages 77–90. ACM, 2003.

[28] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.