Combining Machine Learning with a Genetic Algorithm to Find Good Complier Optimizations Sequences

Nilton Luiz Queiroz Junior, Luis Gustavo Araujo Rodriguez and Anderson Faustino da Silva
Department of Informatics, State University of Maringá, Maringá, Brazil

Keywords: Optimization Selection Problem, Machine Learning, Genetic Algorithms.

Abstract: Artificial Intelligence is a strategy applied in several problems in computer science. One of them is to find good compilers optimizations sequences for programs. Currently, strategies such as Genetic Algorithms and Machine Learning have been used to solve it. This article propose an approach that combines both, Machine Learning and Genetic Algorithms, to solve this problem. The obtained results indicate that the proposed approach achieves performance up to 3.472% over Genetic Algorithms and 4.94% over Machine Learning.

1 INTRODUCTION

Artificial intelligence is applied in several problems, like scheduling problems (Ansari and Bakar, 2014), natural language processing (Tambouratzis, 2016), estimate good compilers optimizations sequences (Martins et al., 2016).

Compilers are programs capable of transforming source code to a target code, in a process that is divided into several stages. A critical step in this process is to apply a compiler optimization sequence (transformations in the code) to improve the quality of the target code (Aho et al., 2006).

Modern compilers such as (`GCC`, `ICC` and `LLVM`) provide standard compiler optimization levels (`O1`, `O2`, `O3`), which can be used to optimize the source code. However, specific levels are only appropriate for particular programs. This is because of the selection process for the optimizations that will be applied to a specific program, becoming into a program-dependent problem.

Artificial intelligence techniques are the most common in chose compilers optimizations for specific programs (Park et al., 2012; Zhou and Lin, 2012; Jantz and Kulkarni, 2013b; Jantz and Kulkarni, 2013a; Lima et al., 2013; Junior and da Silva, 2015). The Artificial intelligence approaches to select compiler optimizations can be divided in two: Iterative Compilation and Machine Leaning.

Iterative Compilation (`IC`) evaluates the quality of the target code generated by different sequences, and returns the best target code. These approaches can use search methods, like Genetic Algorithms, and take much time to converge to good solutions.

On the other hand, Machine Learning (`ML`) approaches attempt, from previously-successful compilations, to predict sequences that will have a good performance in new programs. These methods, in general, are faster than iterative compilation, but, they usually have worse results (Jantz and Kulkarni, 2013b).

It is worth mentioning that finding the best compiler optimization sequence for a particular program is an undecidable problem, due to the size of the search space (quantity of optimizations provided by the compiler and possible combinations).

This article describes a hybrid approach, that combines the best of `IC` and `ML`, to mitigate the optimization selection problem (`OSP`). The objective is to describe an approach that initially uses `ML` to select potential optimization sequences, considering the characteristics of the test program, and then applies `IC` to adapt the potential optimization sequences to the test program. Thus, it is expected that adapting a solution will improve the performance instead of only using potential sequences.

The results indicate that the hybrid approach outperforms both `IC` and `ML` in terms of balancing performance (speedup x number of evaluated sequences). Furthermore, the average speedup achieved by the hybrid approach is superior when comparing it to the best compiler optimization level of `LLVM`.

Junior, N., Rodriguez, L. and Silva, A.
Combining Machine Learning with a Genetic Algorithm to Find Good Complier Optimizations Sequences.
DOI: 10.5220/0006270403970404
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 1*, pages 397-404
ISBN: 978-989-758-247-9
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

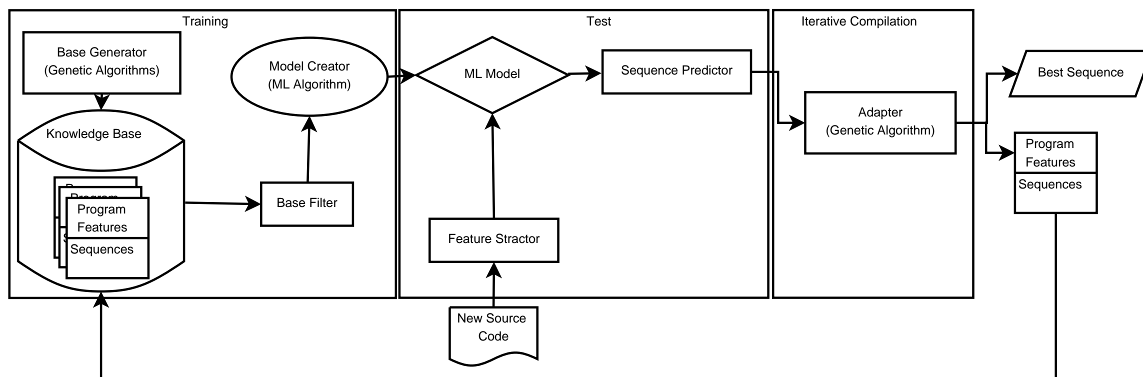

Figure 1: Mixed Approach Architecture.

2 RELATED WORKS

Zhou and Lin (Zhou and Lin, 2012) employed IC, utilizing a genetic algorithm called NSGA-II, to investigate multi-objective compilations. Jantz and Kulkarni (Jantz and Kulkarni, 2013a) proposed a strategy to reduce the search space in order to explore dependencies between the optimizations. These works applied only IC and, thus, the starting point of the solution did not consider the program characteristics. Furthermore, Jantz and Kulkarni only explored the relationship between the optimizations.

Malik (Malik, 2010) utilized a histogram, constructed from the data-flow graph, to establish the similarity between programs and, thus, select a good optimization sequence from a database. Malik made calculations on the histogram and created ML models, utilizing decision-tree algorithms and Support Vector Machines (SVM), to predict a good optimization sequence. Park *et al.* (Park et al., 2012) introduced a program-characterization model, using a control-flow graph, and added information about the instructions to each node. Junior and da Silva (Junior and da Silva, 2015) evaluated the performance of different configurations for case-based reasoning (CBR). These works applied ML, without taking into consideration an adaptive solution for each program.

Martins *et al.* (Martins et al., 2016) implemented an approach that combines *clustering* with IC, resulting in a very similar work to the one proposed in this article. However, both have different strategies to select the initial optimization sequences. While Martins *et al.* utilized a clustering algorithm, this article describes an approach that applies SVM.

3 THE HYBRID APPROACH

IC is an appealing option because it achieves better results than ML. On the other hand, ML is interesting because it applies supervised strategies that are capable of accelerating the convergence to a good solution. This article describes a hybrid approach to solve the OSP, which aims to combine the best of both previously-discussed approaches, and can be synthesized as follows.

Suppose there exists a training set (database), containing S good optimization sequences for P programs. First, the approach creates a model based on the training set, which is used to predict good optimization sequences for a particular test program. In the second step, utilizing the created model, the approach selects N potential optimization sequences for the test program. Afterwards, the N sequences will feed a solution adapter, which utilizes a strategy based on IC to adapt (improve) the solution found in the ML phase. Finally, the best target code found by the adapter is returned to the user and the database is updated with the new knowledge. The Figure 1 illustrates the process.

3.1 The Machine Learning Phase

The ML phase follows a traditional model, which consists of two stages: training and testing. The training stage aims to create a database containing good optimization sequences for different programs. In addition, it provides the model that will be utilized, by the test phase, to predict potential optimization sequences for the test program.

Training consists in a database generator that employs an IC strategy. In addition, it creates a database containing, for each test program, a tuple with the following information: program features

and good optimization sequences (sequences that possess a better performance than the best compiler optimization level). This information will be filtered by the optimization goal (for example execution time, code size) and utilized to create a model that will be applied during the test stage. This model connects program features with good optimization sequences, and therefore, it is possible to predict potential sequences given a determined set of characteristics.

Test predicts potential optimization sequences for the test program. First, this step extracts the characteristics of the test program which are initially collected to predict potential sequences. Afterwards, these characteristics feed the model, which in turn provide `N` potential optimization sequences. The assumption utilized in `ML` is that *similar* programs are those that react similarly when compiled with the same optimization sequences. Therefore, if a model is capable of identifying similarities between programs, it is possible to use sequences, found for training programs, to compile test programs.

3.2 Iterative Compilation Phase

After `N` potential optimization sequences are selected in the initial phase, the next step is to adapt these sequences to the test program. This process is performed through an `IC` strategy.

In various works, `IC` usually begins with the process of discovering good sequences randomly or employing heuristics to selectively investigate the search space. In addition, these works do not utilize program features to assist during the search process.

The advantage of a preliminary `ML` phase is that it provides assistance to guide, based by program features, the process of adapting a solution. Thus, it aims to guide the process of exploring the search space, offering capabilities to discover potential search points and not random points. Consequently, better results are expected than that obtained by other strategies that investigate the search space using unguided processes.

In summary, this stage performs two tasks. First, based on `N` potential sequences, an `IC` strategy adapts the optimization sequences, provided by the previous phase, to the test-program features and returns the best sequence. Second, it updates the database in order to maintain the acquired knowledge and reuse it.

3.3 Strategies to Feed the Database

Although the hybrid approach can operate without feeding the database, utilizing a scheme for such process generates knowledge in medium and long terms. Thus, two approaches to feed the database are proposed:

1. Constant feeding: for every test program compiled, the hybrid approach stores new information in the database (characteristics and sequences) and recreates the model.
2. Batch feeding: for every test program compiled, the hybrid approach stores new information in the database, and after `K` compiled programs, it recreates the model.

The constant feeding strategy has the potential of providing recently-discovered knowledge, but has a cost to recreate the model. On the other hand, batch feeding reduces that cost, but does not provide knowledge when it is recently discovered.

3.4 Strategies for Program Representation

In this article, a program is represented utilizing the characteristics proposed by Namolaru *et al.* (Namolaru et al., 2010), which are systematically extracted from relationships between the program entities and defined by the specificities of the programming language. The appeal of using these characteristics is that Namolaru *et al.* proves their influence in applying optimizations.

In fact, this article uses two representation approaches:

1. Based on hot functions: this indicates that the program will be represented using only its hottest function, because it is a portion of the code that the compiler will have the highest benefit in optimizing.
2. Based on the entire program structure: this indicates that the extracted characteristics describe the entities of the full program.

4 INSTANTIATION OF THE HYBRID APPROACH

The hybrid approach, described in the previous section, can be instantiated using different strategies. Thus, this section aims to describe how it was implemented.

4.1 Implementation

The proposed strategy was implemented as a tool of LLVM 3.7.0 (LLVM3.7, 2016), which was chosen based on the fact that it allows full control over the optimizations. This means that it is possible to enable a list of optimizations through the command line, in which the position of each optimization indicates its order of applying.

In addition, two plugins were implemented for LLVM: (1) libWuLars: used for extracting the program's hottest function, as proposed by Wu and Larus (Wu and Larus, 1994); and (2) libFeaturesExtractor: used for extracting the characteristics proposed by Namolaru *et al.* (Namolaru et al., 2010).

4.1.1 The Machine Learning Phase

Search Space. The search space is formed by the optimizations that exist on the three LLVM compiler optimization levels: O1, O2 and O3.

Test Programs. Microbenchmarks, taken from the test suite of LLVM, were utilized to create the database. Table 1 presents the microbenchmarks.

Table 1: Microbenchmarks.

ackermann	fp-convert	partialsums	whetstone
flops-6	nsieve-bits	spectral-norm	flops-3
methcall	reedsolomon	fib2	mandel-2
quicksort	dt	intmm	puzzle-stanford
ary3	hash	perlin	flops-4
flops-7	richards_bench	strcat	mandel
misr	fannkuch	fldry	queens
random	heapsort	lists	flops-5
bubblesort	oourafft	perm	matrix
flops-8	salsa20	towers	queens-mcgill
n-body	fbench	flops-1	fasta
realmm	himenobmtxpa	pi	fasta-redux
chomp	oscar	treesort	mandelbrot
flops	sieve	flops-2	binary-trees
recursive	ffbench	lpbench	regex-dna
dry	huffbench	puzzle	pidigits

Database Creation. The hybrid approach uses a genetic algorithm (GA) to reduce the search space and find a good compiler optimization sequence for each training program. The GA consists in randomly generating an initial population, which will evolve through an iterative process. This procedure involves choosing the parents; applying genetic operators; evaluating new individuals; and deciding which individuals will compose the new generation.

This iterative process is performed until a stopping criterion is reached. The first generation is composed of individuals that are generated by a uniform sampling of the optimization space. The process for a population to evolve requires two genetic operators:

1. crossover; and
2. mutation.

Crossover has a probability of 60% to create a new individual. In this case, a tournament strategy ($Tour = 5$) selects the parents. Mutation has a probability of 40% to transform an individual. In addition, each individual has an arbitrary initial length, which ranges from 1 to $|Number\ of\ Optimizations\ in\ Space|$. Thus, the crossover operator can be applied to individuals of different lengths. In this case, the length value for the new individual is the average of its parents. Four types of mutation operations were used:

1. Insert a new optimization into a random point;
2. Remove an optimization from a random point;
3. Exchange two optimizations from random points; and
4. Alter one optimization in a random point.

Both operators have the same probability of occurrence. In addition, only one mutation is applied over the individual selected to be transformed. This iterative process uses elitism, which maintains the best individual in the next generation. Furthermore, it executes twice: over 100 generations and 50 individuals; and over 20 generations and 10 individuals. The stop criterion consists in whether the standard deviation of the current fitness score is less than 0.01, or the best fitness score does not change in three consecutive generations. Finally, they are merged and used as one reduced search space. The strategy utilized to reduce the search space is similar to the strategy proposed by Martins *et al.* (Martins et al., 2016) and Purini and Jain (Purini and Jain, 2013).

Model Creation. The ML algorithm is responsible for creating the model, and applies the SVM classifier to identify similar programs. The instantiation process uses the SVM version from the *scikit-learn* library (ScikitLearn, 2016), with a linear kernel and the decision function one-versus-rest. This ML algorithm was chosen based on results obtained comparing SVM and other ML approaches presented in literature (Park et al., 2011; Malik, 2010).

4.1.2 Iterative Compilation Phase

Initial Sequences. These are extracted from training programs that are considered similar to the test program. Thus, each training program P will contribute with a quantity of sequences. This value is given by:

$$N_p = \left\lceil Population_size \times \frac{Decision_val_P}{\sum\limits_{x \in Base} Decision_val_x} \right\rceil \quad (1)$$

The sequence-extraction process from different programs is done in order to generate diversity. Therefore, the solution adapter uses a guided procedure to explore potential search points.

It is worth noting that `Population_size` is the number of potential sequences that the `ML` model will provide. Thus, when this value is reached, the selection process is interrupted and the chosen sequences are sent to the adaptation phase.

The Solution Adapter. The solution adapter is the `GA` utilized to create the database, with an initial population of 10 individuals, and a maximum of 20 generations.

4.2 Utilizing the Approach

The hybrid approach can be described in the following steps:

1. Creation of the database by applying a `GA`;
2. Exclusion of the optimization sequences that under-perform the best `LLVM` compiler optimization levels, for each program of the database;
3. Creation of the prediction model applying `SVM`;
4. Extraction of the characteristics of the test program;
5. Calculation of the decision function, according to the model created in Step 3, for the characteristics of the test program;
6. Calculation of the number of optimization sequences that each test program will provide in the 10 potential optimization sequences, according to the Equation (1);
7. Selection of the 10 safe[1] compiler optimization sequence and validation of the potential optimization sequences that will be provided to the solution adapter of the (`GA`);
8. Attribution of the valid optimization sequences, as in the first generation of the `GA`;
9. Execution of the `GA` on selected sequences;
10. Re-feeding of the database with all sequences created by the `GA`.

Steps 1 to 7 summarize the `ML` phase. Step 8 is a transition phase from `ML` to `IC`. Step 9 consists in `IC`. Lastly, Step 10 retains the knowledge created by `IC`. Step 1 is executed once because it is part of the training phase of `ML`, which is not dependent on the parametrization of the model.

[1] A safe sequence means that it does not cause error on the compiler infrastructure during the target code generation.

5 EXPERIMENTS

This section analyzes the performance of the approach proposed in this article. First, the performance will be analyzed and compared to `IC` and `ML`. Afterwards, the performance, utilizing different data sets and hardware platforms, will be analyzed as well.

5.1 Methodology

Experimental Architecture. The experiments were conducted on a hardware with an Intel Core i7-3770 processor with a frequency of 3.40GHz, 8 MB cache, 8 GB of RAM and the Ubuntu 15.10 operating system with kernel 4.2.0-35-generic.

Test Programs. The Collective Benchmark (cBENCH) was utilized for testing purposes (excluding *stringsearch* and *ispell*), with data set 1; and the Polyhedral Benchmark (POLYBENCH) (excluding JACOBI-1D), with data set extralarge. Each benchmark suite consists in a batch, for the batch feeding strategy.

Metrics. The evaluation uses four metrics to analyze the results: (1) `Speedup`; (2) `NPS`: number of programs achieving speedup over the best compiler optimization level, also called covering; (3) `NoS`: number of sequences evaluated; and (4) `ReT`: response time. The speedup is calculated as follows:

$$Speedup = Runtime_Level_O0/Runtime$$

Strategies. Table 2 summarizes the evaluated approaches.

Table 2: Evaluated Approaches.

Approach	Program Representation	Feeding Strategy	Compilation Order	Maximum NoS
The Proposed Hybrid Approaches				
HHBP	Hot	Batch	Poly-cBench	200
HHBC	Hot	Batch	cBench-Poly	200
HHC	Hot	Constant	Alphabetical	200
HFBP	Full	Batch	Poly-cBench	200
HFBC	Full	Batch	cBench-Poly	200
HFC	Full	Constant	Alphabetical	200
Machine Learning Approaches				
MLH	Hot	-	-	10
MLF	Full	-	-	10
Iterative Compilation Approaches				
IC50	-	-	-	5000
IC10	-	-	-	200
B10	-	-	-	10

The IC50 and IC10 approaches are applications of the GA used in the creation of the database. IC50 is a strategy with 50 individuals in the population, on the other hand, IC10 has 10 individuals. The B10 approach consists in applying the 10 sequences found by Purini and Jain, and returning the best target code (Purini and Jain, 2013).

5.2 Performance

The obtained results are presented in Table 3.

Table 3: Summary of Experiments (GMS: geometric mean speedup, AVG: Average).

Strategy	Speedup			NPS	NoS		
	Best	GMS	Worst		Max	AVG	Min
HHBP	4.466x	1.997x	1.055x	46	110	51.085	10
HHBC	4.500x	1.988x	1.067x	45	117	54.845	10
HHC	4.479x	1.972x	1,056x	46	180	55.458	10
HFBP	6.000x	1.990x	1.062x	45	140	50.135	10
HFBC	4.442x	1.958x	1.068x	44	99	51.983	10
HFC	4.459x	1.954x	1.054x	43	119	52.796	10
MLH	4.491x	1.910x	1.050x	33	10	10	10
MLF	4.060x	1.903x	1.056x	35	10	10	10
IC50	4.353x	2.083x	1.075x	56	650	286.169	100
IC10	6.714x	1.930x	1.035x	46	120	53.678	10
B10	3.751x	1.801x	1.046x	24	10	10	10
O1	4.7x	1.692x	0.992x	-	1	1	1
O2	4.368x	1.843x	1.033x	-	1	1	1
O3	4.361x	1.844x	1.052x	-	1	1	1

Speedup. It can be seen that the hybrid approach, in all its variations, outperforms all the evaluated approaches, except IC50. Among the different variations made in the hybrid approach, a better performance is reached when applying the batch feeding strategy. In addition, the speedups achieved, using only the characteristics of the hottest function, are superior to those reached by the full program representation. This also occurs in the pure ML approaches. Thus, it demonstrates that optimizing the hot functions is a smart strategy to mitigate the OSP.

An interesting result is that constant feeding underperforms when compared to batch feeding. This indicates that inserting knowledge and utilizing it immediately does not always provide benefits.

Comparing the results obtained by the hybrid approaches to the best compiler optimization level (O3) shows that the worst result obtained by the hybrid approach (HFC) had a gain of 5.965%, and the best (HHBP) had a gain of 8.297%. On the other hand, comparing the hybrid approach with pure ML, the performance gain ranges from 4.555% to 4.940%, which proves that combining IC and ML brings benefits in terms of GMS. Furthermore, the gain achieved by the hybrid approach compared to IC10 ranges from 1.244% to 3.472%. However, the performance gain of the IC50 approach ranges from 4.306% to 6.602% over the hybrid approach, and it is 7.927% over the gain obtained by the IC10 approach.

Covered Programs. The NPS indicates the superiority of the strategy IC50, which covers 95% of the programs. The hybrid approach and IC10 cover 78% of the programs, while B10 and ML cover 41% and 51%, respectively.

Number of Evaluated Sequences. A factor that increases the response time is the quantity of sequences that will be evaluated. It is important to note that to *evaluate a sequence* means to compile and execute the program, to measure its runtime. Therefore, reducing the NoS is crucial to reduce the response time of the system.

The results show that it is possible to outperform the compiler optimization levels (concerning the GMS) evaluating a few sequences; however, reducing the NoS. This is the case for MLH, MLF, and B10.

Increasing the NoS increases the speedup and the NPS, consequently increasing the response time. This is the case for IC50 that reaches a better speedup than the hybrid approach; however evaluating 5 times more sequences. This indicates that IC can reach good speedups at the cost of a high response time. Therefore, there exists a *trade-off* between performance (speedup) and response time, which the hybrid approach balances.

Response Time. The evaluated response time is the total time spent by the system, excluding the creation of the database since this step is executed only once offline. The lowest Ret comes from the ML and B10 approaches that consumed 42 and 43 minutes in average per program. On the other hand, the IC10 and IC50 approaches consumed an average of 2 hours and 18 minutes, and 14 hours and 30 minutes respectively. Lastly, the hybrid approach, consumed an average of 2 hours and 58 minutes per program.

The hybrid approach outperforms IC10 up to 3.47% (in regards to the speedup), adding up 29% of the time. In addition, IC50 outperforms the IC10 approach up to 7.927%; however, with a increase in time of 530%. This results indicate that the approaches that are more time-consuming reach the best speedups, besides proving how efficient the hybrid approaches are in cost-benefit.

Observing all approaches, it is worth highlighting that when the number of evaluated sequences grows, the performance increases as well; however, not in

the same proportion. Thus, it is important to analyze the objective of the system to decide which strategy should be utilized. Some strategies achieve lower speedups with a lower response time, while others have a high response time but achieve higher speedups.

5.3 Performance using Different Data sets and Hardware

The experiments for different data sets were performed only with constant feeding, and CBENCH due to the availability of several data sets. Table 4 displays the speedups based on different data sets.

Table 4: Evaluated Input.

Input	Hot Function			Full Program		
	Best	GMS	Worst	Best	GMS	Worst
1	4.116x	1.987x	1.064x	3.352x	1.966x	1,061x
10	3.178x	1.961x	1.072x	3,197x	1,947x	1,080x
20	3.184x	1.895x	1.076x	3,089x	1,890x	1,060x

It can be seen that the alteration of the data sets influences the performance, as reported in the literature (Chen et al., 2010). The performance loss was only up to 4.855%, independent from the characterization of programs utilized.

It is important to note that the extraction of good sequences, from the database, considers only static characteristics. This has the disadvantage of not considering the program behavior by exchanging data sets. On the other hand, it has the advantage of not requiring the execution of the program for feature extraction, which will impact on the system response time. Therefore, a loss of up to 4.855% in performance is appealing against the cost of executing programs.

The experiments with different hardware architectures were also executed only with the approach of constant feeding, but using CBENCH and POLYBENCH. Table 5 exhibits the GMS obtained in the architecture described in Section 5.1 (Core-i7) and the following infrastructure: Intel Xeon E5504 processor with a frequency of 2.00GHz, 4 MB of cache and 24GB of RAM (Xeon). The experiment with the second architecture considers that there is already a database created previously. Thus, the presented results utilize the same database created in the first architecture.

Table 5: Evaluated Architectures.

	Hot	Full	O1	O2	O3
Xeon	2.243x	2.233x	1.935x	2.086x	2.098x
Core i7	1.972x	1.954x	1.692x	1.843x	1.844x

It can be observed that the architecture with the Xeon processor had better results when compared to the Core-i7 processor. Furthermore, the gain in performance of the hybrid approach over the the best compiler optimization level (O3) is up to 6.491% in the Core-i7 architecture, and up to 6.465% in the Xeon architecture. The gains on the O2 and O1 levels are very similar, where the variations reach at most a 1%, from one architecture to another. The results indicate that for both architectures, the gains in the hybrid approach are approximately the same proportion.

The results indicate that:

- it is possible to achieve good speedups using a database created in a different hardware platform;
- representing programs based on their hot functions is an efficient strategy to reduce the loss in performance when the data set is changed, as well as the hardware platform; and
- using a hybrid approach is a smart strategy to mitigate the OSP, regardless of the data set or the hardware platform.

6 CONCLUSIONS AND FUTURE WORKS

Finding a good compiler optimization sequence is a program-dependent problem. An efficient approach performs two steps. The first inspects the program features, and based on these features, it predicts potential previously-successful compilations. The second step adapts the potential sequences to the program. In addition, considering that a substantial amount of runtime is spent in a small portion of the code, such an approach is guided by the features extracted from hot functions. Thus, this paper proposed a hybrid approach to mitigate the optimization selection problem, which combines iterative compilation and machine learning.

The results indicate that the proposed hybrid approach outperforms both iterative compilation approaches and machine learning approaches. Therefore, the hybrid approach is an efficient approach to mitigate the optimization selection problem, because, even when compared to the most aggressive iterative compilation approach, it had a better cost-benefit.

It is intended, as a future work, to evaluate the hybrid approach with other instantiations, altering characteristics for program representations, machine learning algorithms and solution adapters.

REFERENCES

Aho, A. V., S., L. M., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques and tools*. Prentice Hall.

Ansari, A. and Bakar, A. A. (2014). A comparative study of three artificial intelligence techniques: Genetic algorithm, neural network, and fuzzy logic, on scheduling problem. In *2014 4th International Conference on Artificial Intelligence with Applications in Engineering and Technology*, pages 31–36.

Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., and Wu, C. (2010). Evaluating Iterative Optimization Across 1000 Datasets. *SIGPLAN Notices*, 45(6):448–459.

Jantz, M. R. and Kulkarni, P. A. (2013a). Exploiting Phase Inter-dependencies for Faster Iterative Compiler Optimization Phase Order Searches. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 1–10.

Jantz, M. R. and Kulkarni, P. A. (2013b). Performance potential of optimization phase selection during dynamic jit compilation. *SIGPLAN Notices*, 48(7):131–142.

Junior, N. L. Q. and da Silva, A. F. (2015). Finding Good Compiler Optimization Sets - A Case-based Reasoning Approach. In *Proceedings of the International Conference on Enterprise Information Systems*, pages 504–515.

Lima, E. D., De Souza Xavier, T., Faustino da Silva, A., and Beatryz Ruiz, L. (2013). Compiling for Performance and Power Efficiency. In *International Workshop on Power and Timing Modeling, Optimization and Simulation*, pages 142–149.

LLVM3.7 (2016). The LLVM Compiler Infrastructure Project. http://llvm.org.

Malik, A. M. (2010). Spatial Based Feature Generation for Machine Learning Based Optimization Compilation. In *Ninth International Conference on Machine Learning and Applications*, pages 925–930.

Martins, L. G. A., Nobre, R., Cardoso, J. M. P., Delbem, A. C. B., and Marques, E. (2016). Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. *ACM Transactions on Architecture and Code Optimization*, 13(1):8:1–8:28.

Namolaru, M., Cohen, A., Fursin, G., Zaks, A., and Freund, A. (2010). Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 197–206. ACM.

Park, E., Cavazos, J., and Alvarez, M. A. (2012). Using Graph-based Program Characterization for Predictive Modeling. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 196–206, New York, NY, USA. ACM.

Park, E., Kulkarni, S., and Cavazos, J. (2011). An evaluation of different modeling techniques for iterative compilation. In *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 65–74.

Purini, S. and Jain, L. (2013). Finding Good Optimization Sequences Covering Program Space. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–23.

ScikitLearn (2016). Scikit-Learn: Machine Learning In Python. http://scikit-learn.org.

Tambouratzis, G. (2016). Applying pso to natural language processing tasks: Optimizing the identification of syntactic phrases. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 1831–1838.

Wu, Y. and Larus, J. R. (1994). Static Branch Frequency and Program Profile Analysis. In *Proceedings of the Annual International Symposium on Microarchitecture*, pages 1–11, New York, NY, USA. ACM.

Zhou, Y.-Q. and Lin, N.-W. (2012). A Study on Optimizing Execution Time and Code Size in Iterative Compilation. In *International Conference on Innovations in Bio-Inspired Computing and Applications*, pages 104–109.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS