| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | Impact Factor: 7.488 |

||Volume 8, Issue 7, July 2020||

Machine Learning Based Autotuning in Compiler Optimization

K.AZARUDEEN[1], S.AKASH NARAYANAN[2], S.P.KRISHNA[3], K.RITHIK[4], R.S.VISHWESH[5]

Assistant Professor, Dept. of CSE, Velammal College of Engineering and Technology, Madurai, Tamilnadu, India[1]

UG Student, Dept. of CSE, Velammal College of Engineering and Technology, Madurai, Tamilnadu, India[2,3,4,5]

ABSTRACT: Optimization is the field where most compiler research is prepared today. The tasks of the Front-end (scanning, parsing, semantic analysis) are well implicit and optimized. Code generation is moderately straightforward. High-quality optimization is more of an art than a science. Optimization is the process of converting a piece of code to make it more efficient (either in terms of time or space) without changing its output or side-effects. Compiler optimization leads to the enhancement of machine code and/or intermediate code produced by other phases of the compiler. It results in reduced run time and/or space for the object program. Today's compilers have a plethora of optimizations to choose from, and the correct choice of optimizations can have a significant impact on the performance of the code being optimized. Furthermore, choosing the correct optimizations has been a long standing problem in compilation research. In Initial stages of compilation research, We have poor quality models that help to check whether a code is optimized or not. Also, efficiency of existing systems was minimal. Taking all initial systems and models into account, We proposed a system that helps to choose the best possible optimization technique with better efficiency using modernized supervised or unsupervised machine learning approaches. Here we are going to use the Naive Bayesian Multinomial Model Supervised Machine Learning approach based on the static features extracted from the code.

KEYWORDS: Code Optimization, Code Autotuning, Control Flow Graph, Machine Learning, Intermediate Representation, Translation, Data Flow Graph, Intermediate Representation.

I.INTRODUCTION

Compilers have two jobs – translation and optimization. They must first translate programs into binary correctly. Secondly they have to find the most efficient translation possible. Compiler optimization consists of a set of techniques meant to minimize some attributes of a program such as the execution time, memory or energy consumption, etc.Machine learning predicts an outcome for a new data point based on prior data. This ability to predict based on prior information can be used to find the data point with the best outcome and is closely tied to the area of optimisation. There are primarily three different ways of improving the performance of compiler optimizations subjected to machine learning algorithms such as Optimization Phase Ordering, Optimization Tuning, Optimization Selection. In this chapter, We facilitate Optimization Selection using Naive Bayesian supervised Machine Learning approach taking static features of code into account.

II.METHODOLOGY

Our proposed system is primarily based on the Supervised Machine Learning Naive Bayesian approach. The Naive Bayesian Multinomial Classifier works based on the presence of a particular feature in a class unrelated to the presence of any other feature in the same class.Such principle helps to predict whether a code needs to be optimized or not. It involves three stages. First wetrain the model using static features dataset created with help of a pylint tool. Secondly we separate test and training data using dataframes. Finally, with the help of the score method we can predict the optimized code.

When the code is not optimized it starts suggesting the optimization techniques like dead code elimination, loop unrolling, strength reduction, strength reduction, compile time evaluation. It involves three stages. First we need to extract the essential static features from the dataset we used for optimized code prediction. Secondly start training the model by separating the dataset into test and training data using dataframes. Finally, with the help of the score method we are able to suggest the optimization techniques like dead code elimination, loop unrolling, strength reduction, strength reduction, compile time evaluation etc., needs to be used. Efficiency of the proposed system is 90 percent.

III. SYSTEM ARCHITECTURE

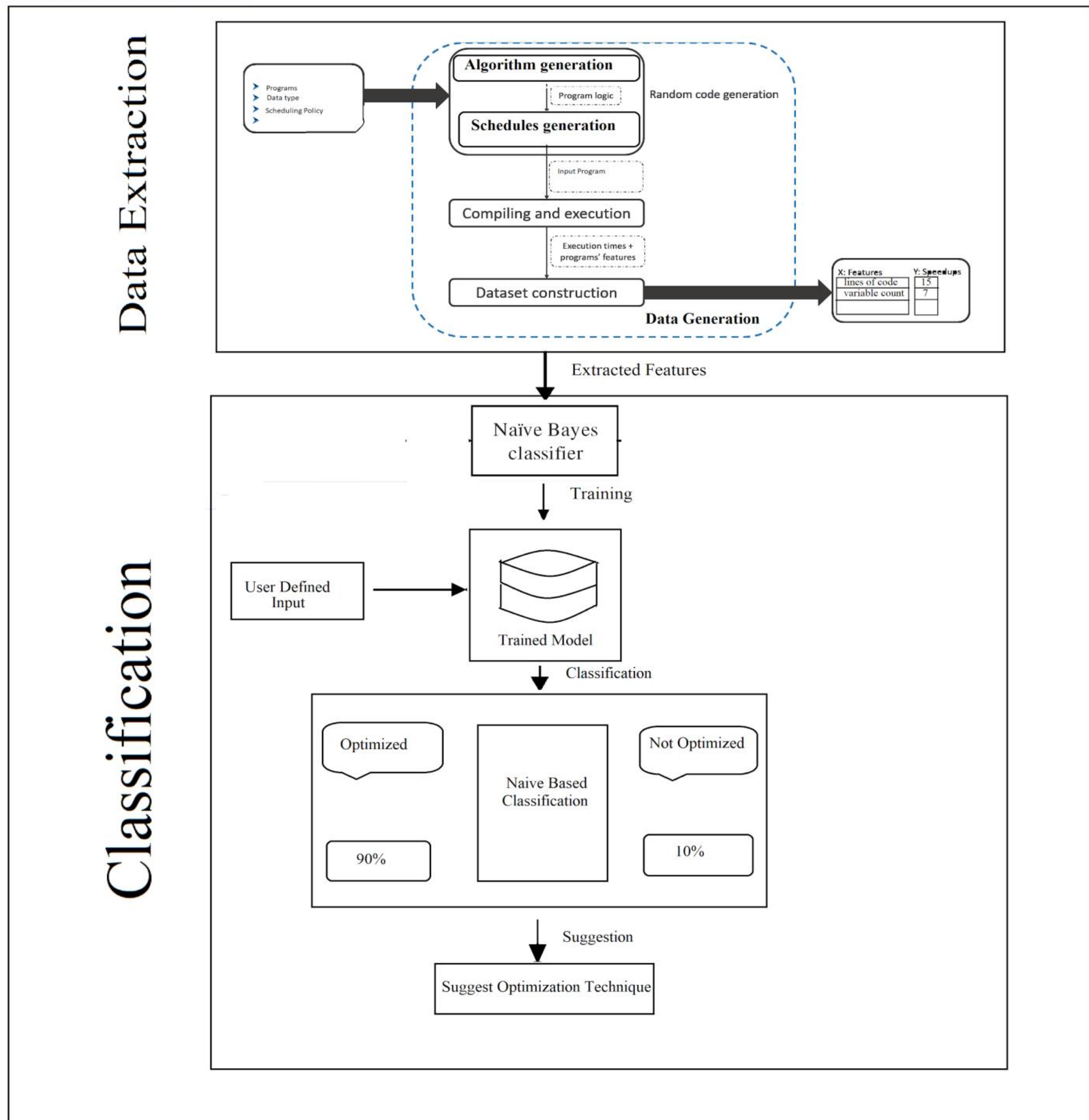

Fig 3.1 Architecture Diagram

A system architecture consists of system components and the sub-systems developed, that will work together to implement the overall system. Our proposed system consists of two sub-systems. The data extraction sub-system is solely responsible for collecting the static features of a code. It can be done with the help of **Pylint. Pylint** is a source-code, bug and quality checker for the python programming language. Components of Data Extraction System are Random Code Generation, Compilation and Execution, Data Set Construction.

The Classification sub-system consists of two main roles training the model and testing against the trained model. Also it helps us to conclude with results optimized or not optimized with help of the trained model. In case the input data is not optimized its suggests with optimization techniques like dead code elimination, loop unrolling,inlining, code reduction etc., . Components of Classification System are **Naive Bayes Classifier, Trained Model, Classifications and Suggestion.**

IV.RESULT

Our proposed system helps to find the effective optimization technique taking the essential static features of the code into account. Results are shown below for the injected static features of the user defined code. Initially our system begins with checking whether the code is optimized or not. Furthermore if it's not optimized it brings an optimization technique for refining a code.

127.0.0.1:5000/predict

Compiler optimization

Using ML

>

To check whether compiler is Optimized or not

line count of code:	250	cyclomatic complexity:	49
essential complexity:	34	design complexity:	16
operator + operand count:	1469	volume:	9673
program length:	0	difficulty:	97
intelligence:	99	effort:	938311
runtime:	3	time estimator:	52128
line count:	139	No. of comments:	92
blank line count:	17	lOCodeAndComment:	0
uniq opertaor:	32	unique operand:	64
total operator:	1081	total operand:	388
branch count:	97		

Predict | have to optimize!!

Fig 4.1 Optimized or Not Prediction

To predict the type of optimization

line of code:	10	No. of comments:	5
operator+operand count:	15	unique operator count:	0
regex count:	0	Basic block count:	3
No. of branches:	3	No. of loops:	0
Statement inside loop :	0	No. of methods:	2
Unused variable count:	0	Variable count:	5

Predict | Inlining

Fig 4.2 Optimization Technique Prediction

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | Impact Factor: 7.488 |

||Volume 8, Issue 7, July 2020||

V.CONCLUSION AND FUTURE SCOPE

Machine learning is used to automate code optimization in various ways. Our proposed system that uses Naive Bayes Supervised Learning for compiler auto tuning. It involves static data taken from a data extraction tool like pylint. With help of static features it can classify optimized or not. In case if it is not optimized, it proceeds with the second phase suggesting the best optimization technique with 90% efficiency. For future work, we would like to implement phase-ordering techniques in a static compiler. In addition, we would also like to incorporate profile information into the feature set, which allows us to improve our prediction.

REFERENCES

1. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society
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. IEEE, 2014, pp. 90–97.
3. D. Binkley, "Source code analysis: A road map," Future of Software Engineering FOSE, vol. 7, pp. 104–119, 2007.
4. J. Cavazos and M. F. P. O'Boyle. Method-specific Dynamic Compilation Using Logistic Regression. In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 229–240, New York, NY, USA, 2006. ACM Press.
5. J. Chipps, M. Koschmann, S. Orgel, A. Perlis, and J. Smith, "A mathematical language compiler," in Proceedings of the 1956 11th ACM national meeting. ACM, 1956, pp. 114–117.
6. Mr. Chirag H. Bhatt, Dr. Harshad B. Bhadka , "Peephole Optimization Technique for analysis and review of Compile Design and Construction", IOSR Journal of Computer Engineering (IOSR-JCE), Volume 9, Issue 4 (Mar. - Apr. 2013).
7. K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems, ser. LCTES '99, 1999, pp. 1–9.
8. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. The Journal of Supercomputing, 23(1):7–22, 2001.
9. G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle. Milepost gcc: machine learning based research compiler. In Proceedings of the GCC Developers' Summit, June 2008.
10. M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff, "Automatic Selection of Compiler Options Using Non-parametric Inferential Statistics," in 14th International Conference on Parallel Architecture and Compilation Techniques (PACT), 2005, pp. 123–132.
11. 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, ser. CGO '08, 2008, pp. 165–174.