International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792

Optimizing Compilation using Machine Learning Models

Neha Sharma[1], Nirvana Dogra[2], Mohammed Annan[3], A. Parkavi[4]
[1,2,3]*Student, Dept. of Computer Science and Engg., M. S. Ramaiah Institute of Technology, Bengaluru, India*
[4]*Assistant Professor, Dept. of Computer Science and Engg., M. S. Ramaiah Inst. of Tech., Bengaluru, India*

***Abstract*: Machine learning has been used to improve compiler performance for quite some time now. This optimization can be of various kinds – reducing the code size, optimizing the code space, automatically generating program features and optimizing memory access patterns. In this article, we shed light upon the relationship between compiler performance (optimization) and machine learning, and the types of optimizations that could be made.**

***Keywords*: Genetic Algorithm (GA), Features, optimization, Machine Learning (ML).**

1. Introduction

A compiler does two things – translate the source code of a program from a programming language into executable code, and then optimize this translation. Our goal is to maximize performance, which is also termed as optimization. Machine learning is used to predict an outcome for a sample based on past data. It can only learn from the data that we provide, and the ability to predict making use of past information is related to optimization. These optimizations are of several kinds, and it is upon the user to decide what performance criteria he wants to optimize for his program.

In this paper, we discuss various types of optimizations that can be made, and present a way to blend them for better results. The rest of the paper is laid out as follows: Section II talks about the types of machine learning used in compilers, Section III shows the various types of optimizations that can be applied to compilers using machine learning, and Section IV concludes the paper.

2. Types of machine learning used in compilers

A. *Supervised learning*

Supervised learning consists of three stages – feature generation, building a model, and then using that model for prediction. There are several features that can be used, such as – dataset size, the number of instructions in a program, the data structures used in the program, etc. The entire process can be summarized as in Fig 1.

B. *Unsupervised Learning*

In unsupervised learning, we draw references from unlabelled input data to determine hidden patterns in it. The most widely used algorithm is the k-means clustering algorithm, which groups the input data into k clusters based on similarities between the objects in those clusters. Similar objects are grouped together achieving high intra-cluster similarity and low inter-cluster similarity. Each data point here characterizes program behavior.

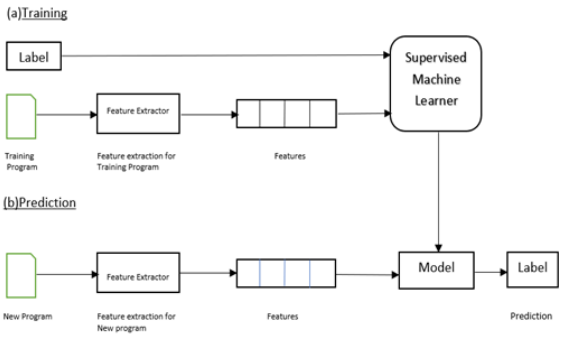

Fig. 1. The figure depicts a general supervised technique in machine learning. A model is trained on the training sample which is then used to predict on the unknown testing sample.

3. Types of optimizations

Machine learning can be adopted to pick the most suitable compiler optimizations based on program features. The nature of these features is therefore, of utmost importance. Various optimizations based on program features are discussed below.

A. *Optimization based on distribution of instructions*

Abid M. Malik in [1] focuses on producing features by exploiting the spatial information of a program. By spatial it refers to the distribution of the instructions in the program. These features are utilized by the predictive models which are constructed using machine learning approach to select prime compiler options for optimization. This spatial information is mainly found inside the compiler in the form of a Data Flow Graph (DFG). The nodes of such a graph store instruction and their edges represent the relationship between the instructions. The author tests this scheme against IBM Milepost-GCC framework. Experimental results display the significance of the spatial information in tuning the compiler optimization by outperforming the Milepost-GCC framework.

B. Optimization based on code space

Keith D. Cooper, Philip J. Schielke and Devika Subramanian in [2] discuss about the problem of selecting well established set of optimizations which is lone capable of producing optimal results for a piece of code. Traditional approaches either standardize some set of optimizations to be capable of optimizing all programs or put the burden on the user by providing them with large number of flags for achieving the purpose. But these approaches had their shortcomings due to which the author incorporated a new technique called Genetic Algorithm to come up with an optimized sequence that would result in smaller code size. Genetic algorithm is a search technique which relies upon processes such as selection, mutation and crossover to come up with solutions rather than random selection. This paper has portrayed GA to be a good fit to solve the problem of finding optimization sequences in compilers due to its capability of handling large space of population (in this case, sequences). Secondly, GA technique makes use of a function (also called fitness function) that assures good quality solutions. They are also quite flexible when it comes to the time spent. Experiments showed that GA outperformed the traditional hand-designed sequence method on a given benchmark set by the authors.

C. Optimization based on automatic feature generation

Hugh Leather, Edwin Bonilla, Michael O'Boyle in [3] try to bring out a novel method of optimization of compilers. The authors use machine learning (ML) to improve the performance of the compiler and make a comparative analysis to the compilers that are handcrafted. The major focus is on finding the features to be used for prediction using an ML model. This is seen to be a necessity as only a good feature selection leads to a good model.

For any ML model, feature selection is a major task and essential for the creation of a good model. For a given program there can be infinite number of features like number of instructions or loop nest level, but selection of any arbitrary features might not lead to a good model. While the selection of features it must be by hand the following problems might be encountered.

- *Irrelevant features:* Like discussed initially, there can be infinite number of features but not all of them might hold importance for the optimization of the compiler.
- Classification clashes: Sometimes, the features selected can be same for different programs but this does not guarantee the best feature selection.
- *Classifier peculiarities:* Similar programs may have same features but this does not guarantee that the feature producer will produce similar optimization results. It is possible that a set of features might work well for a program but might not work for another.
- *Beyond simple features:* There are high chances on missing out some of the essential features after the selection of the common ones. Creating new features becomes harder with each selection.

Essentially three steps have been taken to provide the optimizations climbed in this paper.

- Data generation
- Feature search
- Machine learning
- *Data generation:* Data generation requires gathering data from the input. The data structures describing the program are also extracted for collection of data.
- *Feature search:* A population of features is maintained by the feature search which are derived automatically for compiler (intermediate representation) IR.
- *Machine learning:* It gives us a feedback about how good a feature is. A predictive model is constructed depending on the best features whose quality is decided based on the speedup or slowdown in comparison to the original values.

Finally, the author concludes by comparing the normal GCC compiler implementation and the state-of-the-art ML techniques against his approach. It was found the there was a 76% performance improvement in comparison to GCC compiler and a 48% improvement for the ML technique.

4. Conclusion

In this paper, we have discussed various novel optimizations of compiler that have been brought by either exploiting the spatial distribution of instructions or by automating the process of feature generation and selection. It has been observed that application of ML model can help in making the current compilers highly optimized and are usually more efficient than the hand written compilers. A future scope of study could be inclusion of ML for good feature selection, generation and exploitation of spatial property as a composite.

References

[1] A. M. Malik, "Spatial Based Feature Generation for Machine Learning Based Optimization Compilation," *2010 Ninth International Conference on Machine Learning and Applications*, Washington, DC, 2010, pp. 925-930.

[2] Keith D. Cooper, Philip J Schielke and Devika Subramanian, "Optimization for Reduction of code spaces using genetic algorithms," 1999.

[3] H. Leather, E. Bonilla and M. O'Boyle, "Automatic Feature Generation for Machine Learning based Optimizing Compilation," *2009 International Symposium on Code Generation and Optimization*, Seattle, WA, 2009, pp. 81-91.

[4] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra and J. Méhaut, "A machine learning-based approach for thread mapping on transactional memory applications," *2011 18th International Conference on High Performance Computing*, Bangalore, 2011, pp. 1-10.

[5] A. Matsunaga and J. A. B. Fortes, "On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications," *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, Melbourne, VIC, 2010, pp. 495-504.

[6] M. Curtis-Maury *et al*., "Identifying energy-efficient concurrency levels using machine learning," *2007 IEEE International Conference on Cluster Computing*, Austin, TX, 2007, pp. 488-495.

[7] X. Chen and S. Long, "Adaptive Multi-versioning for OpenMP Parallelization via Machine Learning," *2009 15th International*

Conference on Parallel and Distributed Systems, Shenzhen, 2009, pp. 907-912.

[8] Andre Xian Chang, Aliasgre Zaidy, Eugenio Culurciello, "Efficient compiler code generation for deep learning Snowflake co-processor," 1st Workshop on Energy Efficient Machine Learning, 2018.

[9] Z. Wang and M. O'Boyle, "Machine Learning in Compiler Optimization," in *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879-1901, Nov. 2018.

[10] Antoine Monsifrot, Francois Bodin and Rene Quiniou, "A Machine learning approach to automatic production of compiler Heuristics," International Conference on Artificial Intelligence: Methodology, Systems, and Applications, pp. 41-50, 2002.

[11] John Thomson, Michael O'Boyle, Grigori Fursi n, Bjorn Franke, "Reducing training time in a one-shot machine learning based compiler," International Workshop on Languages and Compilers for Parallel Computing, pp. 399-407, 2009.

[12] Mark Stephenson and Saman Amara Singh, Martin and Una-May O'Reilly, "Meta optimization: Improving compiler Heuristics with Machine learning," PLDI '03 Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pp. 77-90, 2003.

[13] S. Kulkarni, J. Cavazos, C. Wimmer and D. Simon, "Automatic construction of inlining heuristics using machine learning," *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Shenzhen, 2013, pp. 1-12.

[14] L. G. A. Martins, R. Nobre, A. C. B. Delbem, E. Marques and J. M. P. Cardoso, "A clustering-based approach for exploring sequences of compiler optimizations," *2014 IEEE Congress on Evolutionary Computation (CEC)*, Beijing, 2014, pp. 2436-2443.

[15] Vincent. P, Larochelle H., Bengio Y., "Extracting and composing robust features with denoising autoencoders," ICML '08 Proceedings of the 25th international conference on Machine learning, pp. 1096-1103, 2008.