Using Machine Learning in Compiler Optimizations

Michelle Thalakottur
Dept. of Computer Engineering
MKSSS's Cummins College of Engineering
Pune, Maharashtra, India
michellethalakottur@cumminscollege.in

Chhaya Gosavi
Dept. of Computer Engineering
MKSSS's Cummins College of Engineering
Pune, Maharashtra, India
chhaya.gosavi@cumminscollege.in

***Abstract*—In recent years, Machine Learning in Compiler Optimization has moved from a niche research area to one that is more established. This paper discusses how programs can be characterized for Machine Learning models. Then the role of Machine Learning in a standard compiler structure has been explored by looking various models that can be used to automate compiler optimization, with a focus on the advantages and disadvantages of various supervised, unsupervised models. The paper concludes with a discussion on the significance of this research field and further research directions that can be explored.**

***Index Terms*—compilers, optimization, machine learning**

I. Introduction

One of the best examples of a deterministic system in Computer Science is a compiler, because when given an input, ie, a source program, it will always give the same output, ie, an object file. Machine Learning, however, is probablistic and predicts outputs given a certain input, by learning a relationship between the output and input variables. Thus, the use of Machine Learning in a deterministic system can seem illogical since there is no discernible relationship that has to be learnt in a compiler. But there is a place for Machine Learning even in compiler systems, particularly at the Code Optimization phase.

A typical compiler system goes through several phases (Lexical Analysis, Syntactical and Semantic Analysis, Intermediate Code Generation, Code Optimization, Code Generation) before finally giving an object file as the output. There has been extensive research in the 1960s to automate the Lexical Analysis and Parsing phases of a compiler which has resulted in tools such as lex [1] and yacc [2] being heavily used in compilers today. Nowadays, most research is done in the Code Optimization phase of compilers because of two reasons. Firstly, there has been an exponential increase in the capacity of modern hardware, particularly when it comes to parallelism. However, this increased capacity goes underutilized because a majority of software and software writers are unable to take full advantage of it. Compilers are used for the automatic parallelization of sequential code segments and to optimize code so that it makes better use of the underlying hardware through Instruction Level Parallelism (ILP). Additionally, computer architecture evolves very rapidly and compilers have to adapt to each new generation. Instead of compiler writers writing new heuristics to optimize code, we can instead have a machine learn how to optimize code, based on a certain performance metric like speedup or energy consumption, and use these machine-learnt heuristics in the code optimization phase of compilers.

Machine Learning can be used in several ways [3] in the Code Optimization phase - by learning a heuristic function to evaluate certain optimizations so as to decide whether to apply that optimization to the current program, or by learning a model to directly predict the best optimization. The former is related to intelligent searching of an optimization search space while the latter deals with various types of Machine Learning models, including supervised, unsupervised and online Learning.

This paper attempts to explore the field of Machine Learning based Compilation as follows. Section II shows how Machine Learning models can be trained and depoloyed in compilers. Section III explains how program features are learnt from programs and the importance of representative programs in a particular domain. Section IV discusses various models that are used in the optimization phase of a compiler while Section V looks at Machine Learning models in particular. Section VI concludes the paper and discusses future work.

II. Training and Deploying a model

For a Machine Learning model to be able to learn anything useful about a program, it needs to be able to characterize it first. Several program features like the number of instructions, performance counter values, cache miss rate, etc can be learnt from the compiler intermediate representation of the program, static data structures in the source program and through profiling at runtime. However, when chosing which program features to use to represent your program, special attention has to be given to the particular application domain since optimizations on programs are application specific. For example, some domains want to improve speedup of the program, while others might focus on reducing compilation time. The success of the Machine Learning model depends upon the benchmark programs chosen to be a part of the models training data since the effect of optimization can only be seen when relevant optimizations are applied to a new program.

The learnt program features are aggregated into a feature vector which is then compiled in a dataset into the training data for the model. Various machine learning algorithms can be

used to train the model. The model finds a correlation between the program features and useful optimizations by comparing the outcomes obtained after applying different optimizations, using a certain metric like program length, and maximizing over the chosen metric. Thus optimizations that give shorter programs will be weighted more compared with those whose optimizations yield longer programs.

When deploying the learnt model, the model will recieve a new program for input, whose relevant program features are learnt and aggregated into a feature vector. Through past experience of optimal optimizations on the training data, the model makes a prediction about the optimal optimization that should be applied to the new program to result in an optimized program. The output of the model can be a compiled optimized program that can now be run to give more optimized results.

III. Characterizing Programs using Program Features

A. Feature Representation

We have already seen the importance of proper feature representation while training our machine learning model, since the quality of features chosen to represent the model is highly correlated to the optimal optimization prediction made for a new program. The various program features that can be represented fall into several categories including static features, dynamic features and features generated using Machine Learning methods.

1) Static Code Features: As the name suggests, static code features can be learnt when the program is not running and thus is usually derived from intermediate compiler representations. Raw static code features can also be aggregated into a new feature if there exists some grouping between them. For example, instruction count and type of instructions can also be transformed into a new feature, memory load ratio which calculates the number of load instructions over the total number of instructions in the program. Static code features are helpful since they are readily available and easily extracted. However, they are not as useful when compared to other program features as they do not characterize the application behaviour in a very meaningful way. They also contain information about code segments that get executed rarely which can be left out of the model entirely.

2) Dynamic Features: Dynamic features are extracted from the programs when they are running. They can help encode several program characteristics like frequently executing sections of code, cache miss rate, etc. An advantage of dynamic features is that they capture the program behaviour down to the hardware level. However, this comes with its own downside that the learnt model might be too specific to a particular architecture and wont generalize quickly to a new architecture if all the program features are hardware specific. Additionally, these features are prone to noise because of the myriad of application programs running and competing for resources and collecting this dynamic information introduces additional overhead to the compilation time which might not be desirable.

3) Reaction Based Features: The program can be run with several optimization options while being profiled to characterize program behaviour for these options. These reactions to optimization options are then used to predict program speedup and apply several optimization options to the input program through a process called iterative compilation [4] in which the program is compiled using multiple passes during which several optimization options are applied and the resulting code is actually executed to determine performance enhancement. The GCC compiler has five optimization flags (-O0, -O1, -O2, -O3, -Os) which give different speedups to the input program.

4) Automatic Feature Generation: When characterizing programs, Machine Learning models can also be used to generate program features. These models are able to find relevant data access patterns that might not be apparent and can reduce the time spent extracting static and dynamic features. For generating features, the relevant literature [1] uses neural networks to extract features directly from source code, which aren't directly abstractable. Additionally, as with all Machine Learning models, neural nets trained on one optimization problem can transfer learning to another, thus improving accuracy over time. While automatic feature generation is very useful, it cannot replace the feature characterization seen before with dynamic profiling, reaction based features and static code features and in a well characterized program, both are used.

B. Feature Selection

Selecting the right and relevant features to characterize programs is as important as generating program features. One might think that including every generated feature in our feature vector would lead to better results, however, as the length of the feature vector increases, we are faced with another problem, the curse of dimensionality. This refers to the fact that as the number of features or dimensions of our dataset increases, the number of samples required to provide a good accuracy score also increases. Thus, we need to ensure that our dataset is at a low enough dimension so as to not require more samples but has enough good features to characterize the program well. This can be done in various ways by analysing the correlation between features or by projecting the features into a low dimensional space using Dimensionality Reduction methods.

1) Pearson's Coefficient: Karl Pearsons Correlation coefficient can be used to analyse the correlation between features. The correlation of each feature can be found with the prediction vector, ie, the feature representing the metric that signifies optimal optimization state. The correlation found can be of three types: positive correlation, negative correlation and zero. Positive and negative correlation show how the prediction changes with an increase or decrease in the feature value and thus holds valuable information regarding the behaviour of the program and cannot be removed. However, features that show a correlation very close to zero and zero can be discarded since this shows that the prediction value does not change with any change in the feature value. In addition to correlation

coefficient, mutual information and regression analysis for feature ranking can also be used for feature selection.

2) Dimensionality Reduction: Dimensionality Reduction is the process of transforming a high dimension dataset to a low dimension one by projecting the features to the lower dimension. Algorithms like Principle Component Analysis (PCA) [6] achieve this by projecting the data to a lower-dimensional subspace within the high-dimensional space that the data currently resides in while preserving maximum variance of the data. It is a method of orthogonal linear transformation and can also be used to visualize the data. Methods like Linear Discriminant Analysis (LDA) [6] improve on PCA by preserving class information while projecting to the low-dimensional subspace, can also be used to reduce the dimensionality of the dataset.

IV. Models used in the Optimization Phase

If we were able to understand the effect of a certain optimization or code transformation on the eventual performance, then the problem of choosing an optimization simply becomes that of choosing the right transformation. The naïve approach would be to profile each transformation for each relevant metric and choose the best performing transformation in regards to a metric. However, since applications vastly vary in their optimal optimizations and some optimization problems are undecidable, searching these optimation spaces can take a lot of time and can incur a large overhead if the entire search space is represented. A cost-effective solution is using cost functions to evaluate an optimization decision. Instead of using cost functions to search the optimization search space, machine learning models can also be built to predict the best optimization option directly.

A. Cost Functions

A cost function can be used to determine the quality of an optimization option with regards to a chosen quality metric like code size or execution time. Hand crafted cost functions have been used before in compilers like by Leupers et al. [7] who use a branch and bound algorithm as a hand-crafted cost function to evaluate the usefulness of inling a function by adding up relevant program features and comparing this sum with a threshold value. This one-size-fits-all method can improve performance only upto a certain limit because the interplay of optimization options and their effects might go unnoticed by the compiler writers. Thus, the weights of metrics in a cost function and the threshold value that they are compared against are better off learnt by a machine.

1) Performance Based Metrics: A cost function can be given a performance based metric like runtime of a program or the speedup of the target program. Eg. Curve fitting algorithms like regression analysis can be used to predict the run time of a program for a given input program. The Qilin compiler [8] achieved 25% reduction in execution time by estimating if the target programs performance would be better on a GPU or a CPU while choosing which processor to map the program to, in the problem of additive mapping.

2) Energy Consumption Metric: A regression model can also be used in a cost function that works to minimize the energy consumption of the target program to predict the effect of a certain optimization on energy consumption. Artificial Neural Networks (ANNs) can also be used to automatically construct a power model like by Curtis-Maury et al. [9] who use ANNs to identify energy efficient concurrency levels in the context of concurrency throttling.

B. Predict an optimization directly

Cost functions are useful for evaluating an optimization option directly without having to compile and profile a running program. However, since they are used to search a large optimization space, the use of cost functions will inevitably incur a large overhead that can increase your compilation time. In this case, directly predicting an optimization option can be beneficial. Using such a model, you can predict the loop unroll factor, predict compiler flags that can be set during compilation for optimal optimization, code transformation options, etc. Generally predicting the optimization options for sequential programs is much easier than predicting them for parallel programs where factors such as concurrency and contention come into play. The various machine learning models under supervised, unsupervised and online learning frameworks has been discussed in the following section in detail.

V. Machine Learning Models

Depending on the training data provided to a Machine Learning model, a model can be classified as Supervised or Unsupervised learning models. In Supervised learning, the model takes in labelled training data as input and learns a correlation between the predictive feature and the rest of the features. Examples of supervised ML models are regression and classification algorithms like Regression, Decision trees, Support Vector Machines, etc. Unsupervised learning however does not have a labelled training data and the model has to learn how to group the data in order to make a relevant prediction on a new sample. Unsupervised learning algorithms are used for clustering tasks and for Dimensionality Reduction methods for visualization and reducing training data size. Both these types of learning have been used by various researchers in Machine Learning aided Compiler Optimization.

A. Supervised Learning

1) Regression: Regression [10] is akin to curve fitting and is useful for continuous value input models. It is useful for representing linear relationships and does so by using algorithms like Linear Regression which learns a linear relationship between the output and feature values using weights to give more importance to highly correlated features. To represent non-linear relationships we can use algorithms like Support Vector Machine (SVM) and Artificial Neural Networks (ANNs) which works on the principle that non-linearly separable data can be projected to a higher dimension to make it linearly separable using kernel functions. SVMs can seem to be more useful than Linear Regression models, however,

they need more training samples as a cost of this projection. Linear regression samples require less training data (samples) and its feature vectors can have lesser dimensions but the data has to be linearly correlated. SVMs and ANNs can be used on data that exhibit linear and non-linear relationships but require more training samples and can be prone to overfitting without regularization of the data.

2) Classification: Instead of predicting a real-valued output, classification algorithms [12] predict which class or set of classes a feature vector belongs to. Algorithms like Logistic Regression can be used for classification but it is similar to Linear regression in that it assumes a linear relationship among features since it uses a sigmoid function. K Nearest Neighbour (KNN) algorithm can be used to predict optimal optimizations by looking at the optimizations of the K-nearest neighbours. However it is susceptible to noise and can be slow since it calculates the pairwise distances between the target programs feature vector and each training sample. However, this algorithm is prone to overfitting since it assumes that there exists a hyperplane that can divide the features into categories. This drawback is overcome by the Random Forest algorithm.

B. Unsupervised Learning

1) Clustering: Clustering [13] is an Unsupervised learning method which has unlabelled training data as its input so the model uses the geometric properties of the samples to group together samples into clusters. The structure of the data in a k-dimensional space is made use of when clustering samples using a similarity function to code the distance between two samples. Clustering is useful in characterizing program behaviour since similar programs will be grouped together and can be used when putting together the benchmark datasets discussed in Chapter 2. K-means algorithm is a useful clustering algorithm that is regularly used.

2) Evolutionary Search: Evolutionary Algorithms [15] are perfectly suited for finding optimizations in a large search space, such as the problem of setting compiler flags to control optimization options shown by Garciarena et al [16]. Random flags are generally chosen first and the option is evaluated using a fitness function. Then, similar to genetic crossover, the flags are changed to favour options with a higher fitness score. Mutation also occurs where the optimization is randomly changed, slightly, and this new optimization option is evaluated and the process repeated until no further improvement in the fitness score is observed. Evolutionary algorithms succeed where convex optimization methods like Gradient Descent fail, ie, they are more resistant to converging in a locally optimum point when compared to the latter because of the element of randomness introduced via the mutation process.

VI. Conclusion and Future Work

This paper has discussed various Machine Learning models and methods that can be used in Compiler Optimization. Machine Learning based compilation may seem like a niche research topic but in reality is more widespread and is used in compilers like Milepost gcc [17] as well as the compilers which are a part of the cTuning Compiler Collection [18]. Milepost gcc reportedly reaches upto 2 times speedup over the highest gcc optimization level on several benchmark datasets hence, it is not surprising that the use of Machine Learning in Compiler Optimization is a quickly evolving research topic.

Of course, as is true with any Machine Learning application, the model obtained is only as good as the data it is fed. In that sense, there is still a lot of work to be done in ensuring that the models used in compilers get good quality programs and program features as their input and that the program feature subspace is as accurately represented as possible. Rather than replacing compiler writers, Machine Learning only opens up more challenging and creative research directions that can be explored in the field of Compiler Optimization.

References

[1] M. E. Lesk and E. Schmidt, "Lex: A lexical analyzer generator," 1975.

[2] S. C. Johnson *et al.*, *Yacc: Yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[3] Z. Wang and M. OBoyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[4] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009, pp. 75–84.

[5] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 219–232.

[6] I. K. Fodor, "A survey of dimension reduction techniques," Lawrence Livermore National Lab., CA (US), Tech. Rep., 2002.

[7] R. Leupers and P. Marwedel, "Function inlining under code size constraints for embedded processors," in *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No. 99CH37051)*. IEEE, 1999, pp. 253–256.

[8] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2009, pp. 45–55.

[9] M. Curtis-Maury, K. Singh, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. De Supinski, and M. Schulz, "Identifying energy-efficient concurrency levels using machine learning," in *2007 IEEE International Conference on Cluster Computing*. IEEE, 2007, pp. 488–495.

[10] T. P. Ryan, *Modern regression methods*. John Wiley & Sons, 2008, vol. 655.

[11] M. Kiang, "A comparative assessment of classification methods," *Decision Support Systems*, vol. 35, pp. 441–454, 07 2003.

[12] M. Aly, "Survey on multiclass classification methods," 2005.

[13] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping multidimensional data*. Springer, 2006, pp. 25–71.

[14] S. Shalev-Shwartz *et al.*, "Online learning and online convex optimization," *Foundations and trends in Machine Learning*, vol. 4, no. 2, pp. 107–194, 2011.

[15] C. A. C. Coello, "A comprehensive survey of evolutionary-based multiobjective optimization techniques," *Knowledge and Information systems*, vol. 1, no. 3, pp. 269–308, 1999.

[16] U. Garciarena and R. Santana, "Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, 2016, pp. 1159–1166.

[17] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[18] "ctuning compiler collection," http://ctuning.org/wiki/index.php/CTools:CTuningCC, viewed 5 March 2020.