IJMTST is UGC Approved Journal with Journal Id: 43137

International Journal for Modern Trends in Science and Technology, 8(S08): 145-153, 2022

ISSN: 2455-3778 online
DOI: https://doi.org/10.46501/IJMTST08S0826
Available online at: https://www.ijmtst.com/vol8si08.html

Compiler Optimization using Machine Learning Techniques

Satya Mohan Chowdary G[1] | Ganga Bhavani T[2]

[1]Department of IT, Pragati Engineering College, Surampalem, AP, India.

To Cite this Article
Satya Mohan Chowdary G and Ganga Bhavani T. Compiler Optimization using Machine Learning Techniques. International Journal for Modern Trends in Science and Technology 2022, 8(07), pp. 145-153. https://doi.org/10.46501/IJMTST08S0826

Article Info
Received: 26 May 2022; Accepted: 24 June 2022; Published: 28 June 2022.

ABSTRACT

In the last decade, machine learning based activities has moved from an obscure research niche to a mainstream activity. Here in this article, we describe compiler optimization using machine learning techniques. We then provide a comprehensive survey and provide overview for the wide variety of different research areas involved in this. One of the key challenges for compilation is to select the right code transformation, or sequence of transformations for a given program. This requires effectively evaluating the quality of a possible compilation option e.g. how will a code transformation affect eventual performance.

KEYWORDS: Compiler, Auto-Tuning, Machine Learning, Code Optimization

1. INTRODUCTION

Compilers translate programming languages written in human understandable language into machine understandable language where correctness is critical. Machine- learning on the other hand is sub area of Artificial Intelligence aimed at detecting and predicting patterns.

A. Optimization-Important phase in Compiler

Compiler mainly do two things – translation and optimization. There are many different correct translations whose performance varies significantly, traditionally misnamed optimization.

Machine learning based on prior data predicts an outcome for a new data point. 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 optimization. It is at this overlap of looking at code improvement as an optimization problem and machine learning as a predictor of the optimization done where we find machine learning compilation.

An interesting question is therefore why the convergence of optimization and machine learning taken so long? There are two fundamental reasons. Firstly, highly increase in the potential performance of hardware, software is increasingly unable to realize it leading to a software-gap. This gap has yawned right open with the advent of multi-cores. Compiler writers are looking for new ways to bridge this gap.

Secondly, computer architecture evolves so quickly, that it is difficult to keep up. By using the desirable property of being automatic Machine learning has. Rather than relying on expert compiler writers to optimize the code, we can let the machine learn how to optimize a compiler to make the machine run faster, an approach sometimes referred to as auto-tuning. Machine learning is, therefore, ideally suited to making *any* code optimization decision where the performance impact depends on the underlying platform.

Machine learning is part of a tradition in computer science and compilation in increasing automation The

50s to 70s were spent trying to automate compiler translation, e.g. lex tool for lexical analysis i.e. for recognizing tokens and yacc for parsing i.e. for parse trees, in the last decade by contrast has focused on trying to automating compiler optimization.

This article is structured as follows. Overview for machine learning in compilers discussed in Section II. Then we describe how machine learning can be used to search for or to directly predict good compiler optimizations in Section III. This is followed by a comprehensive discussion in Section IV for a wide range of machine learning models that have been employed in prior work. We discuss the challenges and limitations for applying machine learning to compilation in Section V before we summarize and conclude in Section VI.

MACHINE LEARNING IN COMPILERS- OVERVIEW

Given a program, compiler writers would like to know which optimization to apply in order to make the code better. Better often means which executes faster, but can also mean reduced power. Machine learning can be used to build a model used within the compiler, that makes such decisions for any given program.

There are two main stages involved: learning and deploy- ment. The first stage learns the model based on training data, while the second uses the model on new unseen programs. Within the learning stage, we need a way of representing programs in a systematic way. This representation is known as the program features

Figure 1 gives a generic view of supervised machine learning in compilers.

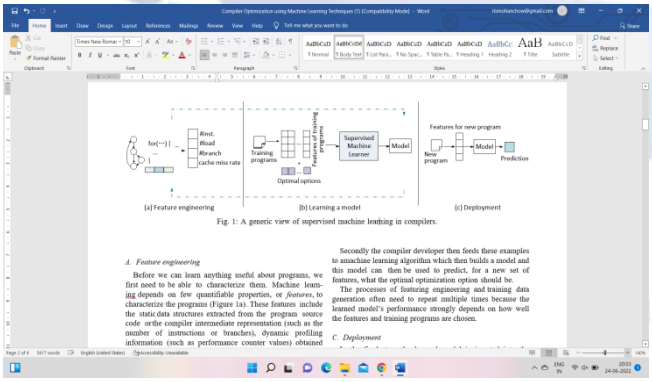

Fig 1: A generic view of supervised machine learning in compilers

A. *Feature engineering*

Before we can learn anything useful about programs, we first need to be able to characterize them. Machine learn- ing depends on few quantifiable properties, or *features*, to characterize the programs (Figure 1a). These features include the static data structures extracted from the program source code or the compiler intermediate representation (such as the number of instructions or branches), dynamic profiling information (such as performance counter values) obtained through runtime profiling, or a combination of the both.

Standard machine learning algorithms typically work on fixed length inputs, so the selected properties will be summarized into a fixed length *feature vector*. Each element of the vector can be an integer, real or Boolean value. The process of feature selection and tuning is referred as *feature engineering*. This process may need to iteratively perform multiple times to find a set of high-quality features to build a accurate machine learning model.

B. *Understanding a model*

The second step is to use training data to derive a model using a learning algorithm. This process is depicted in Figure 1b. The compiler developer will select training programs which are typical of the application domain. For each training program, First we calculate the feature values, compiling the program with different optimization options, and running and timing the compiled binaries to discover the best- performing option. This process produces, for each training program, a training instance that consists of the feature values and the optimal compiler option for the program.

Secondly the compiler developer then feeds these examples to a machine learning algorithm which then builds a model and this model can then be used to predict, for a new set of features, what the optimal optimization option should be.

The processes of featuring engineering and training data generation often need to repeat multiple times because the learned model's performance strongly depends on how well the features and training programs are chosen.

C. *Deployment*

In the final step, the learned model is inserted into the compiler to predict the best optimization decisions for new programs. This is demonstrated in Figure 1c. To make a prediction, the compiler first extracts the features of the input program, and then feeds the extracted feature values to the learned model to make a prediction.

3. METHODOLOGY

One of the key challenges for compilation is to select the right code transformation, or sequence of transformations for a given program. This requires effectively evaluating the quality of a possible compilation option e.g. how will a code transformation affect eventual performance.

A naive approach is to exhaustively apply each legal transformation option and then profile the program to collect the relevant performance metric. Many techniques have been proposed to reduce the cost of searching a large space. However, its main limitation remains: it only finds a good optimization for one program and does not generalize into a compiler heuristic. The two main approaches for solving the problem of selecting compiler options that work across programs. The first strategy attempts to develop a cost function to be used as a proxy to estimate the quality of a potential compiler decision, without relying on extensive profiling. The second strategy is to directly predict the best-performing option.

A. Building a cost function

Many compiler heuristics rely on a cost function to estimate the quality of a compiler option. Depending on the optimization goal, the quality metric can be execution time, the code size, or energy consumption etc. Using a cost function, a compiler can evaluate a range of possible options to choose the best one, without needing to compile and profile the program with each option.

B. *Directly predict the best option*

While a cost function is useful for evaluating the quality of compiler options, the overhead involved in searching for the optimal option may still be prohibitive. For this reason, researchers have investigated ways to directly predict the best compiler decision using machine learning for relatively small compilation problems.

Monsifrot *et al.* pioneered the use of machine learning to predict the optimal compiler decision. This work developed a decision tree based approach to determine whether it is beneficial to unroll a loop based on information such as the number of statements and arithmetic operations of the loop. Their approach makes a binary decision on whether to unroll a loop but not how many times the loop should be unrolled. Later, Stephenson and Amarasinghe advanced by directly predicting the loop unroll factor by considering eight unroll factors, (1, 2, . . . , 8). They formulated the problem as a multi-class classification problem (i.e. each loop unroll factor is a class). They used over 2,500 loops from 72 benchmarks to train two machine learning models (a nearest neighbor and a support vector machines model) to predict the loop unroll factor for unseen loops. Using a richer set of features than, their techniques correctly predict the unroll factor for 65% of the testing loops, leading to on average, a 5% improvement for the SPEC 2000 benchmark suite.

Directly predicting the optimal option for parallel programs is harder than doing it for sequential programs, due to the complex interactions between the parallel programs and the underlying parallel architectures.

These papers show that supervised machine learning can be a powerful tool for modelling problems with a relatively small number of optimization options.

4. MACHINE LEARNING MODELS

In this section, we review the wide range of machine learning models used for compiler optimization. Table I summarizes the set machine learning models discussed in this section.

There are two major subdivisions of machine learning techniques that have previously been used in compiler optimizations: supervised and unsupervised learning. Using supervised machine learning, a predictive model is trained on empirical performance data (labelled outputs) and important quantifiable properties (features) of representative programs. The model learns the correlation between these feature values and the optimization decision that delivers the optimal (or nearly optimal) performance. The learned correlations are used to predict the best optimization decisions for new programs. Depending on the nature of the outputs, the predictive model can be either a *regression* model for continuous outputs or a *classification* model for discrete outputs.

In the other subdivision of machine learning, termed *unsupervised learning*, the input to the learning algorithm is a set of input values merely – there is no labelled output. One form of unsupervised learning is *clustering* which groups the input data items into several subsets. It

does so by first dividing a set of program runtime information into groups (or clusters), such that points within each cluster are similar to each other in terms of program structures (loops, memory usages etc.); it then chooses a few points of each cluster to represent all the simulation points within that group without losing much information.

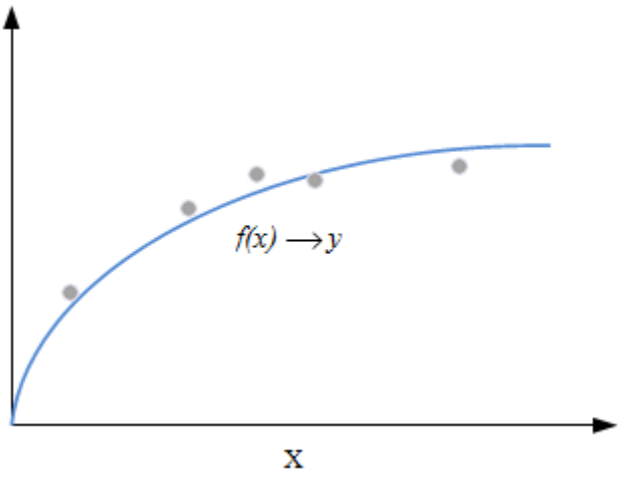

Fig. 2: A simple regression-based curve-fitting example. There are five training examples in this case. A function, f , is trained with the training data, which maps the input x to the output

y. The trained function can predict the output of an unseen x.

A. Supervised learning

Regression: A widely used supervised learning tech- nique is called *regression*. This technique has been used in various tasks, such as predicting the program execution time input or speedup for a given input, or estimating the tail latency for parallel workloads .

Regression is essentially curve-fitting. As an example, con- sider Figure 2 where a regression model is learned from five data points. The model takes in a program input size, X, and predicts the execution time of the program, Y . Adhering to supervised learning nomenclature, the set of five known data points is the training data set and each of the five points that comprise the training data is called a training example. Each training example, (x_i, y_i), is defined by a feature vector (i.e. the input size in our case), x_i, and a desired output (i.e. the program execution time in our case), y_i. Learning in this context is understood as discovering the relation between the inputs (x_i) and the outputs (y_i) so that the predictive model can be used to make predictions for any new, unseen input features in the problem domain. Once the function, f , is in place, one can use it to make a prediction by taking in a new input feature vector, x. The prediction, y, is the value of the curve that the new input feature vector, x, corresponds to.

There are a range of machine learning techniques can be used for regression. These include the simple linear regression model and more advanced models like support vector ma- chines (SVMs) and artificial neural networks (ANNs). Linear regression is effective when the input (i.e. feature vectors) and output (i.e. labels) have a strong linear relation. SVM and ANNs can model both linear and non-linear relations, but typically require more training examples to learn an effective model when compared with simple linear regression models

Table II gives some examples of regression techniques that have been used in prior work for code optimization and the problem to be modelled.

1) Classification: Supervised classification is another technique that has been widely used in prior work of machine learning based code optimization. This technique takes in a feature vector and predicts which of a set of classes the feature vector is associated with. For example, classification can be used to predict which of a set of unroll factors should be used for a given loop, by taking in a feature vector that describes the characteristics of the target loop.

The k-nearest neighbour (KNN) algorithm is a simple yet effective classification technique. It finds the k closet training examples to the input instance (or program) on the feature space. The closeness (or distance) is often evaluated using the Euclidean distance, but other metrics can also be used. This technique has been used to predict the optimal optimization parameters in prior works.

It works by first predicting which of the training programs are closet (i.e. nearest neighbours) to the incoming program on the feature space; it then uses the optimal parameters (which are found during training time) of the nearest neighbours as the prediction output. While it is effective on small problems, KNN also has two main drawbacks. Firstly, it must compute the distance between the input and all training data at each prediction. This can be slow if there is a large number of training programs to be considered. Secondly, the algorithm itself does not learn from the training data;

instead, it simply selects the *k* nearest neighbours. This means that the algorithm is not robust to noisy training data and could choose an ill-suited training program as the prediction.

As an alternative, the decision tree has been used in prior works for a range of optimization problems. These include choosing the parallel strategy for loop parallelization, determining the loop unroll factor, deciding the profitability of using GPU acceleration, and selecting the optimal algorithm implementation. The advantage of a decision tree is that the learned model is interpretable and can be easily visualized. This enables users to understand why a particular decision is made by following the path from the root node to a leaf decision node. For example, Figure 3 depicts the decision tree model developed in for selecting the best- performing device (CPU or GPU) to run an OpenCL program. To make a prediction, we start from the root of the tree; we compare a feature value (e.g. the communication-computation ratio) of the target program against a threshold to determine which branch of the tree to follow; and we repeat this process until we reach a leaf node where a decision will be made. It is to note that the structure and thresholds of the tree are automatically determined by the machine learning algorithm, which may change when we target a different architecture or application domain.

Decision trees make the assumption that the feature space is convex i.e. it can be divided up using hyperplanes into different regions each of which belongs to a different category. This restriction is often appropriate in practice. However, a significant drawback of using a single decision tree is that the model can over-fit due to outliers in the training data. Random forests have therefore been proposed to alleviate the problem of over fitting. Random forests are an ensemble learning method . As illustrated in Figure 4, it works by constructing multiple decision trees at training time. The prediction of each tree depends on the values of a random vector sampled independently on the feature value. In this way, each tree is randomly forced to be insensitive to some feature dimensions. To make a prediction, random forests then aggregate the outcomes of individual trees to form an overall prediction. It has been employed to determine whether to inline a function or not, delivering better performance than a single-model-based approach. We want to highlight that random forests can also be used for regression tasks.

TABLE I: Machine learning methods discussed in Section IV.

Approach	Problem	Application Domains	Models
Supervised learning	Regression	Useful for modelling continuous values, such as estimating execution time, speedup, power consumption, latency etc.	Linear/non-linear regression, artificial neural networks (ANNs), support vector machines (SVMs).
	Classification	Useful for predicting discrete values, such as choosing compiler flags, #threads, loop unroll factors, algorithmic implementations etc.	K-nearest neighbour (KNN), decision trees, random forests, logical regression, SVM, Kernel Canonical Correlation Analysis, Bayesian
Unsupervised learning	Clustering	Data analysis, such as grouping profiling traces into clusters of similar behaviour	K-means, Fast Newman clustering
	Feature engineering	Feature dimension reduction, finding useful feature representations	Principal component analysis (PCA), autoencoders

TABLE II: Regression techniques used in prior works.

Modelling Technique	Application	References
Linear Regression	Exec. Time Estimation	[62], [38], [43]
Linear Regression	Perf. & Power Prediction	[63], [64], [65]
Artificial Neural Networks	Exec. Time Estimation	[62], [46], [39]

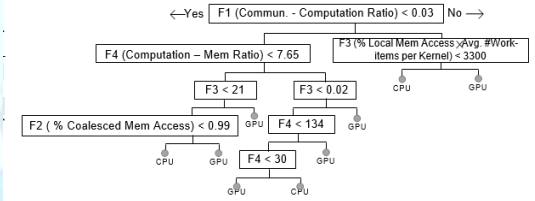

Fig.3:Adecisiontreefordeterminingwhichdevice(CPUor GPU) to use to run an OpenCL program. This diagram isreproducedfrom.

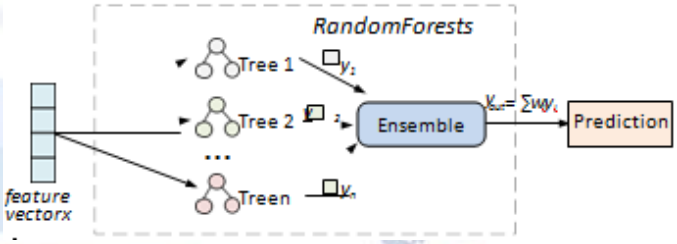

Fig. 4: Random forests are an ensemble learning algorithm. It aggregates the outputs of multiple decision trees to form a final prediction. The idea is to combine the predictions from multiple individual models together to make a more robust, accurate prediction than any individual model.

Logical regression is a variation of linear regression but is often used for classification. It takes in the feature vector and calculates the probability of some outcome. For example, Cavazos and O'Boyle used logical regression to determine the optimization level of Jike

RVM. Like decision trees, logical regression also assumes that the feature values and the prediction has a linear relation.

More advanced models such as SVM classification has been used for various compiler optimization tasks SVMs use kernel functions to compute the similarity of feature vectors. The radial basis function (RBF) is commonly used in prior works because it can model both linear and non-linear problems. It works by mapping the input feature vector to a higher dimensional space where it may be easier to find a linear hyper-plane to well separate the labelled data (or classes).

Deep neural networks: In recent years, deep neural net-works have been shown to be a powerful tool for tackling a range of machine learning tasks like image recognition ,and audio processing . DNNs have recently used to model program source code for various software engineering tasks ,but so far there is little work of applying DNNs to compiler optimization. A recent attempt in this direction is the DeepTune framework , which uses DNNs to extract source code features . The advantage of DNNs is that it can compactly represent. a significantly larger set of functions than a shallow network, where each function is specialised at processing part of the input. This capability allows DNNs to model the complex rela- tionship between the input and the output (i.e. the prediction). As an example, consider Figure 5 that visualizes the internal state of DeepTune when predicting the optimal thread coarsening factor for an OpenCL kernel (see Section II-D). Figure 5 (a) shows the first 80 elements of the input source code tokens as a heatmap in which each cell's colour reflects an integer value assigned to a specific token. Figure 5 (b) shows the neurons of the first DNN for each of the four GPU platforms, using a red-blue heatmap to visualize the intensity of each activation. If we have a close look at the heatmap, we can find that a number of neurons in the layer with different responses across platforms. This indicates that the DNN is partly specialized to the target platform. As information flows through the network (layers c and d in Figure 5), the layers become progressively more specialized to the specific platform.

B. *Unsupervised learning*

Unlike supervised learning models which learn a correlation from the input feature values to the corresponding outputs, unsupervised learning models only take it the input data (e.g. the feature values). This technique is often used to model the underlying structure of distribution of the data.

Clustering is a classical unsupervised learning problem. The k-means clustering algorithm groups the input data into *k* clusters. For example, in Figure 6, a k-means algorithm is used to group data points into three clusters on a 2-dimensional feature space. The algorithm works by grouping data points that are close to each other on the feature space into a cluster. K-means is used to characterize program behaviour It does so by clustering program execution into phase groups, so that we can use a few samples of a group to represent the entire program phases within a group. K-means is also used in the work presented into summarize the code structures of parallel programs that benefit from similar optimization strategies. In addition to k-means, Martins *et al.* employed the Fast Newman clustering algorithm which works on network structures to group functions that may benefit from similar compiler optimizations.

Principal component analysis (PCA) is a statistical method for unsupervised learning. This method has been heavily used in prior work to reduce the feature dimension. Doing so allows us to model a high-dimensional feature space with a smaller number of representative variables which, in combination, describe most of the variability found in the original feature space. PCA is often used to discover the common pattern in the datasets in order to help clustering exercises. It is used to select representative programs from a benchmark suite.

Autoencoders are a recently proposed artificial neural net- work architecture for discovering the efficient codings of input data in an unsupervised fashion. This technique can be used in combination of a natural language model to first extract features from program source code and then find a compact representation of the source code features.

.D. *Discussions*

What model is best, is the $64,000 question. The answer is: it depends. More sophisticated techniques may provide greater accuracy but they require large amounts of labelled training data - a real problem in compiler optimization. Techniques like linear regression

and decision trees require less training data compared to more advanced models like SVMs and ANNs. Simple models typically work well when the prediction problem can be described using a feature vector that has a small number of dimensions, and when the feature vector and the prediction is linearly correlated. More advanced techniques like SVMs and ANNs can model both linear and non-linear problems on a higher dimensional feature space, but they often require more training data to learn an effective model. Furthermore, the performance of a SVM and an ANN also highly depends the hyper-parameters used to train the model. The optimal hyper-parameter values can be chosen by performing cross-validation on the training data. However, how to select parameters to avoid over-fitting while achieving a good prediction accuracy remains an outstanding challenge.

Choosing which modelling technique to use is non-trivial. This is because the choice of model depends on a number of factors: the prediction problem (e.g. regression or classification), the set of features to use, the available training examples, the training and prediction overhead, etc. In prior works, the choice of modelling technique is largely relied on developer experience and empirical results. Many of the studies in the field of machine learning based code optimization do not fully justify the choice of the model, although some do compare the performance of alternate techniques. The OpenTuner frame- work addresses the problem by employing multiple techniques for program tuning. OpenTuner runs multiple search techniques at the same time. Techniques which perform well will be given more candidate tuning options to examine, while poorly performed algorithms will be given fewer choices or disabled entirely. In this way, OpenTuner can discover which algorithm works best for a given problem during search.

One technique that has seen little investigation is the use of Gaussian Processes. Before the recent widespread interest in deep neural networks, these were a highly popular method in many areas of machine learning .They are particular powerful when the amount of training data is sparse and expensive to collect. They also automatically give a confidence interval with any decision. This allows the compiler writer to trade off risk vs reward depending on application scenario.

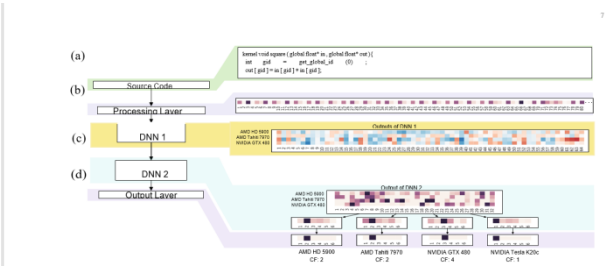

Fig. 5: A simplified view of the internal state for the DeepTune DNN framework when it predicts the optimal OpenCL thread coarsening factor. Here a DNN is learned for each of the four target GPU architectures. The activations in each layer of the four models increasingly diverge (or specialize) towards the lower layers of the model. It is to note that some of the DeepTune layers are omitted to aid presentation.

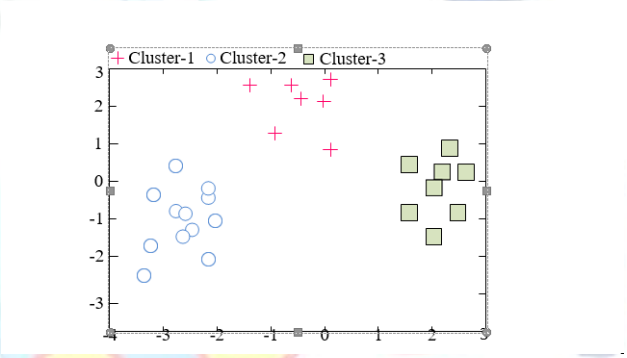

Fig. 6: Using k-means to group data points into three clusters. In this example, we group the data points into three clusters on a 2-d feature space.

Using a single model has a significant drawback in practice. This is because a one-size-fits-all model is unlikely to precisely capture behaviours of diverse applications, and no matter how parameterized the model is, it is highly unlikely that a model developed today will always be suited for tomorrow. To allow the model to adapt to the change of the computing environment and workloads, ensemble learning was exploited in prior works .The idea of ensemble learning is to use multiple learning algorithms, where each algorithm is effective for particular problems, to obtain better predictive performance than could be obtained from any of the constituent learning algorithm alone.

Making a prediction using an ensemble typically requires more computational time than doing that using a single model, so ensembles can be seen as a way to compensate for poor learning algorithms by performing extra computation.

To reduce the overhead, fast algorithms such as decision trees are commonly used in ensemble methods (e.g. Random Forests), although slower algorithms can benefit from ensemble techniques as well.

5. DISCUSSION

New models have to be based on empirical data which can then be verified by independent experimentation. This experiment – hypothesis – test cycle is well known in the physical sciences but is a relatively new addition compiler construction.

As machine learning based techniques require a sampling of the optimization space for training data, we typically know the best optimization for any program in the training set. If we exclude this benchmark from training, we therefore have access to an upper bound on performance or oracle for this program. This immediately lets us know how good existing techniques are. If they are 50% of this optimum or 95% of this optimum immediately tells us whether the problem is worth exploring Furthermore we can construct naive techniques – e.g. a random optimization and see its performance. If this performed a number of times, it will have an expected value of the mean of the optimization speedups. We can then demand that any new heuristic should outperform this – though in our experience there have been cases where state-of the art work was actually less than random.

TABLE III: Summary of features

Feature	Description
Static code features	Features gathered from source code or the compiler intermediate representations, such as instruction counts. See Section V-A1.
Tree and graph based features	Features extracted from the program graph, such as the number of nodes of different types. See Section V-A2.
Dynamic features	Features obtained through dynamic profiling or during runtime execution, such as performance counter values. See Section V-A3.
Reaction-based features	Speedups or execution time obtained by profiling the target program under specific compiler settings. See Section V-B.

TABLE IV: Feature engineering techniques

Problem	Techniques
Feature selection	Pearson correlation coefficient, mutual information, regression analysis. See Section V-D1.
Feature dimensionality reduction	Principal component analysis (PCA), factor analysis, linear discriminant analysis, autoencoder. See Section V-D2.

8

TABLE V: Example code features used in prior works.

Description	Examples
Arithmetic instructions	#floating point instr., #integer instr., #method call instr.
Memory operations	#load instr, #store instr.
Branch instructions	#conditional branch instr, #unconditional branch instr
loop information	#loops, loop depth
parallel information	#work threads, work group size

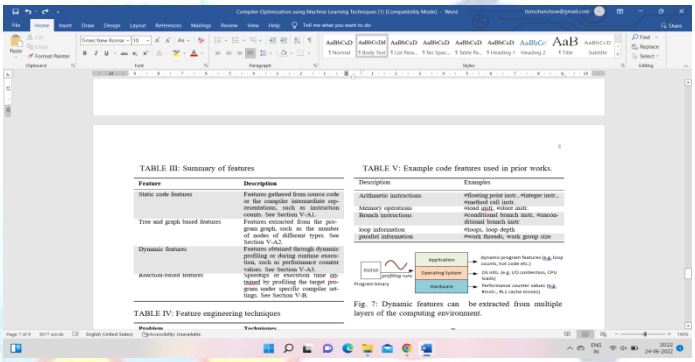

Fig. 7: Dynamic features can be extracted from multiple layers of the computing environment.

6. CONCLUSION

This paper has introduced machine learning based compilation and described its power in determining an evidence based approach to compiler optimization. It is the latest stage in fifty years of compiler automation. Machine learning based compilation is now a mainstream compiler research area and over the last decade or so, has generated a large amount of academic interest and papers. While it is impossible to provide a definitive cataloguer of all research, we have tried to provide a comprehensive and accessible survey of the main research areas and future directions. Machine learning is not a panacea. It can only learn the data we provide. Rather than, as some fear, it dumbs down the role of compiler writers, it opens up the possibility of much greater creativity and new research areas.

Conflict of interest statement

Authors declare that they do not have any conflict of interest.

REFERENCES

[1] Machine Learning in Compiler Optimisation, Zheng Wang and Michael O'Boyle

[2] 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.

[3] P. B. Sheridan, "The arithmetic translator-compiler of the ibm fortran automatic coding system," Communications of the ACM, vol. 2, no. 2, pp. 9–21, 1959.

[4] M. D. McIlroy, "Macro instruction extensions of compiler languages," Communications of the ACM, vol. 3, no. 4, pp. 214–220, 1960.

[5] H. Schoen, D. Gayo-Avello, P. Takis Metaxas, E. Mustafaraj, M. Strohmaier, and P. Gloor, "The power of prediction with social media," Internet Research, vol. 23, no. 5, pp. 528–543, 2013.

[6] Slashdot. (2009) IBM releases open source machine learning compiler. [Online]. Available: https://tech.slashdot.org/story/09/07/03/0143233/ibm-releases-open-source-machine-learning-compiler

[7] H. Massalin, "Superoptimizer: a look at the smallest program," in ACM SIGPLAN Notices, vol. 22, no. 10, 1987, pp. 122–126.

[8] R. J. Adcock, "A problem in least squares," The Analyst, vol. 5, no. 2, pp. 53–54, 1878.

[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker,D. Patterson, J. Shalf, and K. Yelick,"Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in Proceedinngs of the 2008 ACM/IEEE conference on Supercomputing, 2008, p. 4.