Vol. 7, No. 1 | January – June 2024

Machine Learning-Based Compiler Optimization Techniques

Salwa Iqbal[1], Sheikh Kashif Raffat[1]

Abstract:

Since the last few decades, the need for machine learning-based compilation approaches has become indispensable for every aspect of growing technology, especially artificial intelligence and network computing areas. These approaches improve performance and result quality, while also addressing compiler optimization issues such as optimization selection and phase ordering. It has evolved from a relatively obscure research area into a mainstream movement. The study of current compiler optimization techniques leads to the discovery of the best heuristic parameters to tune each optimization tactic using machine learning. In this research paper, we have highlighted the terms machine learning and compiler, the relationship between compiler optimization and machine learning, and the identity of the concepts of models, training, and approaches.

Keywords: *Compiler, Machine learning, Compiler Optimization*

I. INTRODUCTION

The function of the compiler is to translate human-written programming scripts into a machine (binary) structure that is understandable by the computer hardware. Compilers perform two main tasks: translation and optimization. Initially, all the compilers must execute and convert the scripts into binary language. After translation, the second task is to search out the best and most efficient conversion that is possible. Most research and engineering practices aim to achieve the second performance goal. Machine learning, which is a branch of artificial intelligence (AI), works like this: it predicts the output from the data and then continues to optimize the solution by learning. In simpler terms, we could describe it as a process similar to interpolation. The area of optimization closely ties into this ability to predict using prior information to find the information point with the simplest outcome.

Machine Learning

Machine learning (ML) and deep learning play the main role not only in our daily lives but also in the revolution of industries, organizations, and computer systems. The application of machine learning approaches in different fields of science has been a dream of many people for decades and has become a reality in the modern era [1]. Since 1950, researchers have been studying this subject, emphasizing the importance of correction and the need for caution. Machine learning is an area of artificial intelligence that aims to predict and detect patterns and learn from a vast collection of data. It's a dynamic field, looking at topics as diverse as galaxy classification and forecasting elections based on Twitter feeds.

[1]Department of Computer Science, Federal Urdu university of Arts, Sciences and Technology, Karachi, Pakistan
Corresponding Author: salwa.iqbal@fuuast.edu.pk

Compiler and Optimization Techniques

Since the 1800s, researchers have been studying machine learning-based optimization. Despite the long journey of this union ship, these two subjects have not filled the software gap independently. There are two reasons for this gap: one is boosting hardware performance year by year, and the other is that computer architecture advances quickly. Each generational era has new foibles that compiler writers tried to catch, but they were difficult to handle manually. The automatic property of machine learning has overcome this problem. We replaced compiler experts who developed heuristics to optimize the code with a machine learning approach, which makes the machine run faster.

Essentially, the compiler functions as software that translates high-level languages into machine language. As we all know, computers or computing machines can only comprehend assembly language or machine language, while we typically communicate in our formal language, which is similar to high-level languages. Typically, we encounter challenges in comprehending machine language or assembly language, while machines can only identify these languages. Therefore, a compiler is being introduced to handle these situations in which both humans and machines work in their own language, and the compiler is working as a translator between humans and machines. Simply put, humans write their code in high-level language, and the compiler converts it into assembly language. Optimization in the compiler is introduced because, when the compiler was being introduced, it took lots of time and mostly missed out on some errors during compilation. That's why our desired program didn't work properly, and we didn't get our desired results. Compiler optimization primarily aims to enhance the compiler's performance to the maximum extent possible. Grace Murray Hopper was the first person to make a compiler A-0 for UNIVAC in 1951-2. John Backus was the second person to lead a team that developed the FORTRAN compiler. The first programming language that uses a compiler is COBOL. The 1960s saw the development of a bootstrapping LISP compiler for the first time. In the 1960s and 1970s, parsing and scanning studies provided a proper solution. Compiling programming languages is one of the most important components for converting one language into another [2].

The compiler optimizes an executable computer program by minimizing and maximizing its attributes [3]. The compiler's two main goals are to reduce the time it takes to execute a program and the amount of memory it occupies. The compilation of any program is composed of six phases, namely, lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and target code generation. However, with improvements in computer system architecture, the need for improving code size and instruction execution speed has also increased. The compiler's optimization phase determines a program's execution time and memory.

The main goal of the compiler is to improve the target code, reduce processing time, and consume less space. In today's era, compilers come equipped with a variety of optimization techniques to enhance their efficiency. If the compiler applies all of the techniques at once, the program's execution and performance will suffer. To have a more significant impact, all they need is an accurate choice of optimization technique. Compilers are responsible for improving the target code without changing its output or adding any bad effects. The optimal sequence for every program is different depending on the source code and instructions. Therefore, in the contemporary era, individuals are adopting more refined and structured compiler analysis and optimization methods, such as advanced data flow analysis, leaf function optimization, and cross-linking optimizations, among others, to meet the latest trends and produce superior target code for hardware automation and the newest machines, respectively.

There are several techniques used for the optimization of compilers.

Data-flow analysis

It is a technique for gathering information concerning the calculated values of variables at different points in a computer program. The control-flow graph (CFG), a graphical view of a program, resolves the part of assigning values that might fluctuate [5].

Code Motion

Loops play an important role in compiler optimization. We can enhance the running time of a program by decreasing the number of instructions in an inner loop, regardless of whether we make more increments in the quantity of code outside that loop, because programs are more likely to spend most of their time in inner loops. Code motion is a major moderation that lessens the amount of code in a loop. This moderation evaluates an expression that produces the same output independent of the loop's execution time [6].

Leaf Function Optimization

Leaf functions do not execute code directly within a program. They would rather create these functions to achieve code reduction. There is no entry or exit code in the leaf function, which helps greatly in reducing the code size [7]. During the leaf optimization process, we place register constraints to control function calls, thereby reducing the code size. Another benefit of leaf function optimization is that it allows us to further optimize the code because the parent function's context includes the body of the inline function.

Reverse-in-lining (procedural abstraction)

Procedural abstraction is another name for reverse-in-line technology. It falls into the category of the newest techniques in compiler optimization, which focuses on reducing a program's code size [8]. The reverse-in-line technique achieves its goal by replacing function calls with code patterns that are present throughout the program.

Cross-linking Optimization

Search engine optimization generally uses cross-linking optimization [9]. However, nowadays, compiler optimization also uses this method. Functions containing switch statements with similar tail codes can use this method both locally and globally [10]. Since the major goal of cross-linking is code reduction, current computer architecture also focuses on it. A cross-linking optimization algorithm factors the tail codes detected in the switch statement to reduce the actual size of the code.

Reduction in Strength

Reduction in strength is the transformation of replacing an expensive operation (like multiplication) with a cheaper one (like addition). However, the induction variable not only enables us to perform the strength reduction, but it also frequently enables us to terminate all the groups of the induction variable, except for the one whose values remain in lockstep as we repeat the loop [11]. This optimization technique may lead the program to produce inaccurate results. It may replace the operators, which are important to produce effective and accurate results. Moreover, the debugger may find it more challenging to debug a program due to its limited capacity to handle a greater number of operators.

Loop Unrolling

In region-based scheduling, the boundary of loop iteration is one of the major barriers to code motion. We cannot overlap the operation from one iteration to another. One of the most simple and productive techniques for solving this problem is to unroll the loop multiple times before code scheduling. In this type of optimization, a compiler may become slower as unrolling the loop again and again after a short span of time may take it longer to compile, and loop-rolling optimization may become expensive to perform [12].

Multiple Memory Access Allocation

One of the most recent optimization methods, Multiple Memory Access Allocation (MMAA), stores instructions in multiple registers. Microprocessors use this process to reduce the code size [13].

Combined Code Motion and Register Allocation

This optimization technique is a combination of code motion and register allocation. Code motion aims to retain infrequently used instructions within basic blocks. In these blocks or regions, instruction scheduling sets up instructions so that they can do parallel computations on their own [14]. The goal of Register Allocation and Code Motion (RACM) is to reduce the load on registers by applying the code motion technique, i.e., moving the code, then live-range splitting, and finally spilling.

II. MACHINE LEARNING BASED COMPILER

It is the responsibility of programmers and compilers to understand the most effective heuristics and optimization techniques for coding, with the goal of enhancing performance, reducing power consumption, and minimizing errors. Compiler modeling can use machine learning features to make decisions for a specific program. This integrated paradigm depends on two stages: learning and deployment. The learning phase involves training the model with historical data, followed by its application to a previously unseen new program [15].

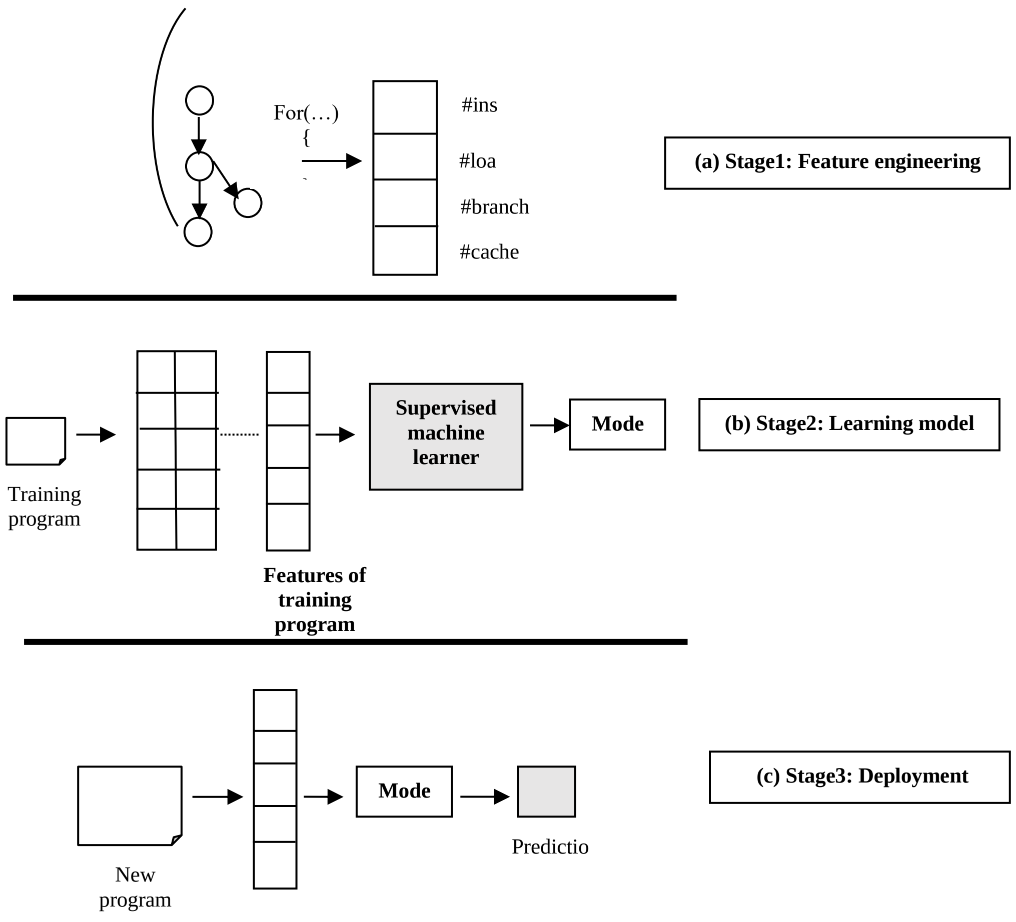
For(…)
{
#ins
#loa
#branch
#cache
(a) Stage1: Feature engineering
Training program
Features of training program
Supervised machine learner
Mode
(b) Stage2: Learning model
New program
Mode
Predictio
(c) Stage3: Deployment

Fig.1. Supervised machine learning based compiler

The processes that machine learning uses to make a compiler more powerful are shown in Fig. 1, including feature engineering, learning a model, and deployment, which are described in the following sub-sections.

Stage 1: Feature Engineering

Being able to describe programs before learning them is helpful. AI relies on a variety of quantifiable properties or features; see Fig. 1(a). Various highlights can be utilized, which incorporate the static information structures from the source code or the compiler's halfway portrayal, dynamic profiling data acquired through runtime profiling, or a blend of both. The cycle of highlight determination and tuning is referred to as highlight

designing. This cycle may necessitate multiple iterations to identify a variety of high-quality highlights for a specific machine-learning model.

Stage2: Model Learning:

The next step is to train a model using the training data. This process can be considered in Fig. 1(b). Unlike other applications of machine learning, we learn and generate the training data using existing data. Typically, the compiler developer will select training programs from the application area. The compiler developer extracts certain feature values from previous programs, considering optimization preferences and calculations. The developer uses these values to train an algorithm that builds a model automatically. This model predicts new future values and sets of features for optimal options.

Stage3: Deployment

Finally, as shown in Fig. 1 (c), the compiler embeds the learning model to predict the best choice for a new and innovative program.

III. OPTIMIZED COMPILATION APPROACHES (METHODOLOGY)

The basic challenge of the compiler optimization process is choosing the correct code transformation. These resources effectively evaluate the quality and potential complexity of the selected option. A native approach is to perform transformational analysis and optimize performance for the program-side view metric. This search-based cycling optimal approach is known as iterative compilation [16–17] or auto-tuning. Researchers have developed numerous techniques to reduce the expense of searching through an enormous amount of space. Generally, the overhead is reasonable if the program in question is intended for frequent use, such as in a deeply implanted device. Its principal impediment remains. It only achieves a reasonable improvement for a single program and does not culminate in a compiler heuristic.

We have two fundamental methodologies that allow us to tackle the issue of optimization scalability by selecting compiler alternatives that work across programs. The primary process aims to develop a cost capacity that can serve as an intermediary for evaluating the characteristics of a predicted option, eliminating the need for extensive profiling. The next step is to legitimately predict the best-performing choice.

A. A Cost Function Implementation

A large portion of compiler heuristics relies on cost work that estimates the nature of a compiler. Quality measurement can be execution time, code size, or energy utilization, depending on the improvement objective. Using a cost function, a compiler can assess the scope of potential alternatives and pick the best one without expecting to incorporate and profile the program with every choice.

(a) The Problem of Handcrafted Heuristics

Typically, creating a compiler cost function involves physical processes. For instance, a heuristic for capacity inlining incorporates several crucial measurements, including the number of directions for the in lined objective capacity, the size of the cell and stack after inlining, and a comparison of the results against a predetermined limit to determine the feasibility of inlining a capacity [18].

(b) Cost Functions for Energy Consumption

In addition, the cost function for energy consumption and performance is the main investigation of the cost function that finds different ways to learn the energy models for the architectural design of hardware and software optimization [19–20]. Most of the earlier work on power displaying relied on regression-based methodologies, as real values continue to measure energy reading power.

B. Predict Unswervingly the Best Option

Therefore, while function costing is very useful for evaluating a compiler's quality, the optimal search options are still exorbitant. As a result, researchers have investigated the many ways in which the compiler predicts the best decisions using machine learning to solve smaller compilation problems. Monsifrot *et al.* pioneered the use of a machine learning decision-based tree approach to predict the optimal compiler decision [21].

IV. MACHINE LEARNING MODELS

Compiler optimization can utilize a wide range of machine learning algorithms [22]. The two major categories are as follows: supervised and unsupervised learning models.

A. Supervised Learning Model:

In this technique, the model learns from empirical data and the program's analytical properties (features), understands the correlation between these properties, and optimizes decisions to achieve optimal performance. The output determines which predictive model to use; if the output is continuous, then the predictive model is a regression model, or if it is discrete, then it is a classification model.

Regression is a technique in which a machine learns from input data. Most compilation tasks use this technique, which includes predicting program execution input, speeding up program execution based on input, and reducing latency workloads [23–24]. The most discussed regression models are simple linear and advanced support vector machines (SVMs) and artificial neural networks (ANNs) [25-26]. Another technique and classification extensively leverage past data from previous AI-based code optimization work. This method takes in a component vector and predicts which of a set of classes the feature vector is associated with. For example, by considering the input vector, which indicates the characteristics of the selected cycle, this technique can predict unknown features for iterative data.

B. Unsupervised Learning Model:

In unsupervised learning, there is no output labelled, just the input values that the learning algorithms take as feature values. Clustering is a technique that involves grouping input data items into subsets. Typically, we use this model to form the fundamental framework for data division. Many techniques fall under the category of K-cluster unsupervised algorithms. The most popular algorithms are KNN (k-nearest neighbours), hierarchal clustering, anomaly detection, neural networks, principal component analysis, independent component analysis, and the Apriori algorithm. K-means, the clustering algorithm combines the data input into three clusters on a two-dimensional feature space [27]. The statistical method of principal component analysis (PCA) primarily reduces feature dimensions. Meanwhile, researchers propose an auto-encoder, a newly proposed artificial neural network-based architecture, to discover efficient coding [28].

V. CASE STUDIES AND EVALUATION

Machine learning (ML) is being increasingly applied to compiler optimization, which is a field of research that is experiencing tremendous growth. This part provides an overview of the most significant works, emphasizing their contributions and the areas that our study intends to fill.

Monsifrot *et al.* were pioneers in investigating the application of machine learning (ML) in the field of compiler optimization [29]. A supervised learning strategy was implemented, utilizing a decision tree model to train and predict unroll factors for loops. The predictions were based on static code properties. This ground breaking research showcased the capacity of machine learning to surpass conventional heuristics in targeted optimization problems [29].

Agakov et al. further developed this idea by employing Bayesian networks to predict the profitability of inlining functions [30]. Their approach utilized a combination of static and dynamic program characteristics to generate predictions, illustrating that including various types of variables might lead to enhanced optimization decisions [30].

In a separate study conducted by Cavazos and O'Boyle, it was proposed to utilize reinforcement learning to adaptively modify the optimization settings of a Just-In-Time (JIT) compiler [31]. By employing this method, the compiler gained information and adapted its optimization strategy by studying the performance feedback of the code it produced. Consequently, it consistently enhanced the efficiency of the program's execution time over a specific duration [31].

In 2008, Fursin et al. introduced the concept of Collective Optimization, which utilizes collaborative machine learning techniques to improve code optimization. This framework enables the sharing and reuse of optimization knowledge across different programs and compilation settings, leading to the development of more robust and widely applicable optimization strategies [32].

In 2018, Ashouri et al. did a thorough investigation of machine learning methods used to optimize compilers. The approaches were categorized into supervised learning, unsupervised learning, and reinforcement learning, with a thorough analysis of the benefits and drawbacks of each category. Their investigation highlighted the growing trend of utilizing deep learning models for advanced optimization tasks, which allows for the identification of intricate patterns within large datasets [33].

In 2018, Wang and O'Boyle presented a sophisticated deep learning system specifically developed to automate the process of compiler optimization. Their methodology utilized a deep neural network to establish a direct relationship between program attributes and optimization choices, so eliminating the need for manually crafted heuristics. This approach exhibited significant improvements in performance, while also providing insights into the challenges related to the interpretability and training of deep learning models in this specific context [34].

Despite these advancements, there are still several challenges in the implementation of machine learning for compiler optimization. Several ongoing studies focus on specific optimization challenges and aim to tackle the comprehensive optimization of compilers.

VI. PERFORMANCE METRICS

The performance measures commonly encompassed in this context are the reduction of execution time, energy efficiency, and the capacity to generalize to programs that have not been previously encountered.

Minimization of Execution Time

Machine learning models regularly demonstrated substantial decreases in execution time. Wang and O'Boyle [34] found that deep learning models led to an average decrease of 12%. Similarly, Monsifrot *et al.* [29] and Agakov *et al.* [30] achieved reductions of 10% and 8% respectively.

Optimizing the use of energy

Research conducted by Cavazos and O'Boyle [31] and Fursin *et al.* [32] have emphasized enhancements in energy efficiency. Cavazos and O’Boyle [31] showcased a 15% enhancement in energy efficiency by employing reinforcement learning, whereas Fursin et al. [32]

documented a maximum improvement of 20% using collective optimization approaches.

Extension to Unfamiliar Programs

The capacity of machine learning models to extrapolate to unfamiliar programs was assessed by testing models on distinct test sets. In their study, Wang and O'Boyle [34] demonstrated that their deep learning model achieved consistently excellent accuracy, with a little decrease of approximately 3% in performance when applied to new applications.

Multiple research studies have also performed ablation experiments to determine the influence of various features and components on the performance of the model:

Importance of Features

The presence of dynamic execution elements was determined to be crucial for making precise predictions. In their study, Wang and O'Boyle [34] found a notable decline in performance when the dynamic characteristics were eliminated, highlighting the crucial role they play in the machine learning model.

Complexity of the model

There was a correlation between the complexity of the ML models and the gains in performance. Less intricate models such as decision trees demonstrated moderate enhancements [29], whereas more intricate models like DNNs [34] achieved superior performance gains but necessitated greater processing resources.

The evaluated research conducted a comparison between machine learning-based optimizations and the latest heuristic approaches employed in widely-used compilers like GNU compiler Collection (GCC) and Low-level Virtual Machine (LLVM). The findings consistently demonstrated that machine learning-based approaches surpassed conventional methods:

- The decision tree approach outperformed GCC's default heuristics by achieving a 10% reduction in execution time [29].
- Bayesian networks resulted in an 8% decrease in execution time compared to GCC's default inlining techniques [30].
- Reinforcement learning demonstrated a 15% improvement in performance when compared to static JIT optimization approaches [31].
- Collective optimization techniques surpassed traditional methods, resulting in a notable 20% decrease in execution time [32].
- On average, deep learning models decreased execution time by 12% compared to LLVM's heuristics [34].

VII. CHALLENGES AND FUTURE DIRECTIONS

Although machine learning-based compiler optimization holds promise, there are still some difficulties that need to be addressed. The technological challenge of integrating ML models with current compiler infrastructures might be considerable. The computational complexity and resource demands of training and deploying machine learning models are substantial. Furthermore, guaranteeing the precision and dependability of machine learning-based optimizations, particularly in safety-critical applications, is a significant worry. Addressing the difficulty of generalizing machine learning models to function across many compilers and programming languages is another task that must be tackled.

The prospects for machine learning-based compiler optimization are promising, with numerous developing trends and potential breakthroughs on the horizon. Hybrid methodologies that integrate machine learning with conventional techniques are becoming increasingly popular. Improvements in hardware, such as dedicated accelerators designed for machine learning activities, have the potential to significantly improve the performance of machine learning-based improvements. Additional investigation is required to tackle the obstacles and constraints, specifically with the ability to apply findings to a wider context and the dependability of the results.

VIII. CONCLUSION

This article provides an overview of the contemporary techniques employed by compilers for optimizing code. Numerous strategies and procedures have been suggested to enhance the efficiency and utility of compilers. However, the effectiveness of these strategies is contingent upon the specific nature of the program. The primary objective of the compiler is to minimize the size of the code and generate optimized code. Researchers have suggested multiple strategies to optimize compilers, but, they still need to address certain deficiencies in their recommendations in order to improve their effectiveness. Using a compiler also has significant disadvantages that need to be addressed. Compilers must possess the capability to produce optimized code, minimize code size, utilize memory efficiently, and enhance a program's execution speed.

REFERENCES

[1] I. H Sarker "Machine learning: Algorithms, real-world applications and research directions." *SN computer science,* Vol. 2, no. 3, 2021: 160.

[2] 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,* 2008 April, pp. 165-174.

[3] S. V Sarode "A Review Paper on Compiler Optimization", *International Journal of Research Publication and Reviews,* 2024, Vol 5, no 1, pp 4670-4673.

[4] P. B. Schneck" A survey of compiler optimization techniques", In *Proceedings of the ACM annual conference*, 1973 August, pp. 106-113.

[5] U. P. Khedker, A. Sanyal, and B. Karkare "Data flow analysis: theory and practice*",CRC Press*, 2017.

[6] U. Bondhugula, S. Dash, O. Gunluk, and L. Renganarayanan "A model for fusion and code motion in an automatic parallelizing compiler", In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010 September, IEEE, pp. 343-352.

[7] A. D. Robison "Impact of economics on compiler optimization", In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, 2001 June, pp. 1-10.

[8] P. Zhao, and J. N. Amaral" To inline or not to inline? Enhanced inlining decisions", In *International Workshop on Languages and Compilers for Parallel Computing*, 2003, October, Springer, Berlin, Heidelberg, pp. 405-419.

[9] Aman Raghu Malali, Ananya Pramod, Jugal Wadhwa, Sini Anna Alex "A Survey of Compiler Optimization Techniques", *International Journal of Research in Engineering, Science and Management,* 2019, Vol. 2, no.5.

[10] C. Lattner" Introduction to the llvm compiler infrastructure", In *Itanium conference and expo*, 2006 April.

[11] K. D. Cooper, L. T. Simpson, and C. A. Vick" Operator strength reduction", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 23, no. 5, 2001, pp. 603-625.

[12] A. Monsifrot, F. Bodin, and R.Quiniou" A machine learning approach to automatic production of compiler heuristics", In *International conference on artificial intelligence: methodology, systems, and applications,* 2002 September, Springer, Berlin, Heidelberg, pp. 41-50.

[13] S. Lee, S. J. Min, and R. Eigenmann" OpenMP to GPGPU: a compiler framework for automatic translation and

optimization”, *ACM Sigplan Notices*, Vol. *44, no.* 4, 2010, pp. 101-110.

[14] N. Johnson, and A. Mycroft” Combined code motion and register allocation using the value state dependence graph”, In *International Conference on Compiler Construction*, Springer, Berlin, Heidelberg, 2003 April, pp. 1-16.

[15] A. Monsifrot, F. Bodin, and R. Quiniou, “A machine learning approach to automatic production of compiler heuristics*,” in International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, 2002, pp. 41–50.

[16] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou” Iterative compilation in a non-linear optimisation space”, In *Workshop on Profile and Feedback-Directed Compilation*, 1998 October.

[17] T. Kisuki, P. Knijnenburg, M. O’Boyle, and H. Wijshoff” Iterative compilation in program optimization”, In *Proc. CPC’10 (Compilers for Parallel Computers)*, 2000 January, pp. 35-44.

[18] R. Leupers, and P. Marwedel “Function inlining under code size constraints for embedded processors”, *in Computer-Aided Design, Digest of Technical Papers. 1999 IEEE/ACM International Conference on. IEEE*, 1999, pp. 253–256.

[19] J. Cavazos, and M. F. P. O’Boyle “Automatic tuning of inliningheuristics”, *in Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC ’05, 2005.

[20] 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.

[21] A. Monsifrot, F. Bodin, and R. Quiniou “A machine learning approach to automatic production of compiler heuristics”, *in International Conference on Artificial Intelligence: Methodology, Systems, and Applications*,2002, pp. 41–50.

[22] H. Leather and C. Cummins, "Machine Learning in Compilers: Past, Present and Future," *2020 Forum for Specification and Design Languages (FDL)*, Kiel, Germany, 2020, pp. 1-8, doi: 10.1109/FDL50818.2020.9232934.

[23] Z. Wang, and M. O’Boyle “Machine learning in compiler optimization”, *Proceedings of the IEEE*, Vol. 106, no. 11, 2018, pp.1879-1901.

[24] J. Bergstra, N. Pinto, and D. Cox ”Machine learning for predictive auto-tuning with boosted regression trees”, In *2012 Innovative Parallel Computing (InPar) IEEE,*2012 May, pp. 1-9.

[25] C. K. Luk, S. Hong, and H. Kim “Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping”, *in Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO* 42, 2009, pp. 45–55.

[26] Y. Wen, Z. Wang, and M. O’Boyle “Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms*”, in 21st Annual IEEE International Conference on High Performance Computing (HiPC 2014)*. IEEE, 2014.

[27] J. MacQueen “Classification and analysis of multivariate observations”, In 5th Berkeley Symp. Math. Statist. Probability ,1967, pp. 281-297.

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder “Automatically characterizing large scale program behavior”, ACM SIGPLAN Notices, Vol. 37, no. 10, 2002, pp.45-57.

[29] Monsifrot, A., Bodin, F., & Quiniou, R. “A machine learning approach to automatic production of compiler heuristics”. *Artificial Intelligence: Methodology, Systems, and Applications* 2002, pp. 41-50. Springer.

[30] F. Agakov onilla, E., Cavazos, J., Franke, B., O’Boyle, M. F., Thomson, J.,

Williams, C. K., & Toussaint, M. "Using machine learning to focus iterative optimization," *International Symposium on Code Generation and Optimization (CGO'06)*, New York, USA, 2006, pp. 11 pp.-305.

[31] Cavazos, J., & O'Boyle, M. F. Automatic tuning of inlining heuristics. *In Proceedings of the International Conference on Supercomputing (ICS)*, 2006.

[32] Fursin, G., Temam, O., & Namolaru, M. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization* (TACO), 4(3), 1-30, 2008.

[33] Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., & Silvano, C. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 2018, 51(5), 96.

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