STM JOURNALS

Journal of
Computer Technology & Applications

ISSN: 2229-6964 (Online)
ISSN: 2347-7229 (Print)
Volume 16, Issue 1, 2025
January–April
DOI (Journal): 10.37591/JoCTA

https://journals.stmjournals.com/jocta

Review JoCTA

Educating Compilers to Learn: Utilizing Machine Learning for More Brilliant Code Optimization

Manish Kumar Jha[1,*], Shambhu Kumar Mishra[2]

Abstract

This study explores the use of machine learning (ML) approaches to compiler optimization. The now-traditional static compilation techniques are transformed into adaptive, dynamic systems capable of making context-specific advancements. Traditional compilers rely mostly on heuristic or rule-based optimization techniques. While these techniques work well in general cases, they consistently fail to adapt well within the limits of code structures that modern machines display. This limitation is especially acute in today's computational landscape, where software applications are increasingly complex and require ever-growing resources. On the other hand, machine learning models like neural networks and reinforcement learning (RL) offer a paradigm shift because compilers can learn from past performance data and adjust optimizations dynamically to the unique features of individual codebases. This study further evaluates the application of supervised, unsupervised, and reinforcement learning models to the prediction of optimal compiler configurations. It aims at improving critical performance metrics like execution time, memory usage, and energy efficiency. The experimental results show enhancements across various benchmarks, including reduced execution times, less memory usage, and superior energy efficiency compared to state-of-the-art compiler optimization methods. Reinforcement learning models were highly adaptable, yielding great performance in dynamic and variable computational environments, whereas supervised and unsupervised models proved more effective in predictable and clustered codebases. These results emphasize the potential of ML-based compiler optimization in the creation of smarter and more flexible compiler systems. In doing so, ML might empower compilers to dynamically respond to a diverse set of requirements imposed by applications and computational environments, thus creating the next generation of self-optimizing compilers with unprecedented performance scalability. This work validates the position of ML as an essential ingredient in modern compiler designs in order to cope with increasing demands of advanced computational workloads.

Keywords: Machine learning, compiler optimization, fortification learning, neural systems, versatile compilation

***Author for Correspondence**
Manish Kumar Jha
E-mail: manishdirect@gmail.com

[1]Research Scholar, Department of Mathematics, Patliputra University, Patna, Bihar, India
[2]Professor, Department of Mathematics, Patliputra University, Patna, Bihar, India

Received Date: November 14, 2024
Accepted Date: November 25, 2024
Published Date: December 18, 2024

Citation: Manish Kumar Jha, Shambhu Kumar Mishra. Educating Compilers to Learn: Utilizing Machine Learning for More Brilliant Code Optimization. Journal of Computer Technology & Applications. 2025; 16(1): 50–54p.

INTRODUCTION

In the area of compiler design, compiler optimization is a component of effective computer program execution, which empowers the change of high-level code into machine code that works successfully over various shifted models [1]. The procedures of optimization move forward not as it were the execution speed but to minimize memory and vitality utilization as these are basic in high-performance computing and resource-constrained situations [2]. Various conventional compiler optimization strategies depend on heuristic-based rules, which are created from generalized programming standards. Whereas valid in certain

settings, these inactive strategies struggle to adjust to the increasing complexity and found in computer program applications, which require a more refined approach [3, 4].

Machine learning techniques are very flexible in data driven. It can be used to develop various context-aware compiler optimizations. By using the procedures of machine learning, we can educate compilers to learn from their past executions, different designs, and make choices that optimize code structures based on their personal characteristics. This machine learning based approach changes compilers from inactive means into a better framework that can move forward and adjust to changed workloads [5]. In this study, we investigate three machine learning models: (1) supervised learning, (2) unsupervised learning, and (3) fortification learning for compiler optimization. We compare the adequacy of these three machine learning strategies to get speedier execution, diminished memory impression, and lower control utilization, subsequently showing the potential of machine learning in upgrading compiler [6, 7].

RESEARCH METHODOLOGY

In the research methodology, we inquire about the use of three machine learning approaches, that is supervised learning, unsupervised learning, and fortification learning, to analyze and optimize compiler optimizations. Each machine learning approach test was prepared and approved on a mixed dataset of open-source projects, which are different in its complexity and space.

Directed Learning

Supervised learning models expect ideal compiler leadings based on titled datasets of code tests. Our dataset contains various code sections which are combined with compiler optimizations that provide favorable execution measurements, such as decreased execution time or minimum memory use [8]. A visual arrange show was at that point drafted to take various features such as ‘code complexity’, ‘circle settling profundity’, and ‘memory’ required as inputs, forecasting the best compiler optimization for each code segment.

Feature Understanding

To show exactness, a “significance investigation” was conducted to recognize the most logical and impressive highlights for compiler optimization expectation. As shown in Table 1, highlights like “memory utilization” and “loop (circle) structure” displayed its importance, which fixed with earlier evaluations, showing these parameters in successful code optimization of compiler design [9].

Unsupervised Learning

The unsupervised learning approach included “clustering strategies”, particularly “K-means clustering”, which go ahead to collect “code structures” based on the need of its comparative optimization. This grouping allows the application focused on different optimization techniques to each batch. Computational based applications are gathered and optimized for “memory utilization”, whereas less computational based applications can be optimized for its “execution speed” [10].

Batch Study

To check batch correctness, we used “outline coefficient scores”. The batches of code sections into bunches with related basic qualities adapted to a normal outline score. It indicates that machine learning shows bunches requiring particular compiler optimizations.

Table 1. Include significance and expectation exactness for directed learning model.

Feature	Importance score	Optimization hail expectation exactness (%)
Code complexity	0.85	92
Memory usage	0.78	87
Loop structure	0.76	90

Support Learning

The support learning approach uses a "benefit-based structure" which allows the compiler to learn/make changes in the processes of optimization. The use of Q-learning calculation was tested by our support learning. It allows the compiler to get benefits based on advancements in 'speed', "memory utilization", and/or "to control the productivity" [11].

Support Learning at One Structure

The support learning system gives component values based on the decrease in its "execution time" and "memory uses" and thus, it computes setups that adapt effectiveness. This benefit-based structure is given to the support learning, establish with adaptability, to change to different code inputs, upgrading its viability over different programming paradigms.

Comparative Analysis of Machine Learning Models

To evaluate the effectiveness and practicality of each model of machine learning in compiler optimization, a comparative analysis was conducted. This analysis focused on "performance metrics", 'adaptability', and "application suitability". As a result, each model of machine learning presents unique advantages, thus making it good for different types of compiler optimization tasks, which are highlighted below.

Supervised Learning vs. Unsupervised Learning

The machine learning model called "supervised learning" is designed to predict specific compiler configurations, which are based on "labeled data", "code structures" and "repetitive patterns". The supervised learning model showed high accuracy in predicting optimal performance settings for code samples with standard features like "structure of loops" and "pattern of memory allocation" [9]. But as a limitation, supervised learning models depend on labeled datasets which can limit adaptability, especially when it is applied to any new code structure or domains which is outside its training data.

The machine learning model called "unsupervised learning", specifically "K-means clustering", offered a greater flexibility by categorizing code segments based on inherent similarities without requiring labeled data. By grouping similar types of codes, the compiler could apply customized optimization techniques, and thus improving efficiency for each unique requirement of the batch [10]. Although the unsupervised model's accuracy in predicting optimizations was lower than supervised learning, it was more adaptable to diverse datasets with different code complexity, and thus making it ideal for large-scale, and heterogeneous software environments [9, 10].

RL's Adaptability

RL outperformed both supervised and unsupervised models in adaptability and real-time optimization and can handle varying performance requirements in different code structures. Through continuous feedback, the RL model learned optimal configurations dynamically and refined them with each iteration [11]. The RL model's self-improvement made it very effective in reducing execution time and memory usage across different code samples. Although training the RL model takes a lot of computational resources, its real-time adaptability makes it very useful for compiler optimization in complex and evolving environments [12].

Model Suitability

- *Supervised learning* is great for predictable code and lots of labeled data, it gives good performance prediction for code with common patterns [8, 9].
- *Unsupervised learning* is good for big and diverse datasets, it groups code segments and allows for flexible optimizations per cluster [10].
- *Reinforcement learning* is best for dynamic and complex tasks where adaptability and real-time feedback is important, it is great for continuous optimization [11, 12].

This shows that a hybrid approach can give you everything. Use supervised learning's accuracy, unsupervised learning's clustering and reinforcement learning's adaptability to get high performance configurations for different software environments [13].

RESULTS AND DISCUSSION

The ML models were assessed on the SPEC CPU2006 and NAS Parallel Benchmarks, recognized for testing code execution proficiency in compilers as shown in Table 2. Execution measurements, counting execution time, memory proficiency, and control utilization, were utilized to evaluate each model's effectiveness.

Execution Speed and Adaptability

Reinforcement learning illustrated the most elevated flexibility and execution time changes, with a normal 22% increment in speed over changed code structures. This advantage stems from the RL model's energetic alteration capabilities, permitting it to reconfigure settings based on input differences [12]. Directed learning models moreover accomplished striking speed advancements, especially with tedious and data-heavy applications that profited from predictable designs in optimization.

Memory Proficiency and Clustered Optimizations

The unsupervised learning demonstrates accomplished critical memory reserve funds, especially in clustering computationally seriously code sections. By fitting optimizations to particular clusters, the unsupervised approach has given a normal memory decrease of 15%, making it well-suited for large-scale information preparing applications that require effective memory administration [13].

Power Utilization and Asset Efficiency

Both RL and directed models contributed to outstanding decreases in control utilization. Control investment funds found the middle value of 18% in the RL demonstrate, which powerfully balanced setups to adjust execution with vitality utilization. This result highlights RL's versatility for resource-constrained situations, such as versatile or inserted frameworks where control productivity is a need [14]. Table 3 outlines the structure of the support learning model's preparing cycle, displaying the input circle component utilized to refine compiler configurations.

Table 2. Execution advancements for different ML models.

ML approach	Average execution time decrease (%)	Memory utilization diminishment (%)	Power productivity enhancement (%)
Supervised learning	15	18	12
Unsupervised learning	10	15	9
Reinforcement learning	22	25	18

Table 3. Reinforcement learning feedback loop for compiler optimization.

Stage	Description
(1) Model initialization	The RL model initializes with a set of baseline compiler configurations, chosen from either default settings or random configurations to allow broad exploration of optimization choices. This setup provides a foundation for identifying performance improvements in subsequent iterations.
(2) Execution code and monitoring	The RL-guided compiler executes the given code segment using the initial configuration. During execution, key performance metrics, such as runtime, memory consumption, and power usage, are monitored closely. These metrics are logged as part of the feedback process.
(3) Reward function evaluation	A reward is assigned based on observed performance metrics. Higher rewards are given to configurations that achieve lower execution times, reduced memory usage, or enhanced power efficiency. The reward function is designed to prioritize improvements that align with the primary optimization goals.
(4) Policy adjustment	The RL model uses reward information to update its internal policy, learning which configurations lead to better performance. This update allows the model to incrementally refine its decision-making to prioritize configurations with higher expected rewards in future iterations.
(5) Iterative refinement	The RL process repeats from Code Execution and Monitoring to Policy Adjustment, with each cycle further refining the compiler's optimization strategy. Over multiple iterations, the RL model converges toward configurations that provide optimal performance for the specific codebase.

CONCLUSION

This work shows the power of machine learning in compiler optimization and how ML models can be used to dynamically adjust and optimize code for different metrics. Compilers can become more adaptable, faster, and memory-efficient through the use of supervised, unsupervised, and reinforcement learning techniques. Each ML model has its strengths: supervised learning is for data driven configuration prediction; unsupervised learning is for clustering-based optimization of specific code structures and reinforcement learning is for real time adaptability, so compilers can evolve and improve over time.

Future work can explore hybrid ML models that combine supervised and reinforcement learning to create more robust and adaptive optimization frameworks. A hybrid approach can allow compilers to use supervised learning for standard and predictable code and reinforcement learning for complex and evolving tasks so that it can be more flexible across code types. Also, more work on clustering algorithms and feature engineering can improve the performance of unsupervised learning models and memory management and resource allocation for better efficiency.

Moreover, exploring deep reinforcement learning (DRL) can allow compilers to handle more high dimensional optimization tasks especially in environments with lots of code variability. Using ML in compiler design not only improves performance but also enables compilers to self-improve over time which is what is needed for advanced, high performance and embedded systems.

REFERENCES

1. Siemieniuk A, Chelini L, Khan AA, Castrillon J, Drebes A, Corporaal H, Grosser T, Kong M. OCC: An automated end-to-end machine learning optimizing compiler for computing-in-memory. IEEE Trans Comput-Aided Des Integr Circuits Syst. 2021 Aug 2; 41(6): 1674–86.
2. Chen C, Zhang P, Zhang H, Dai J, Yi Y, Zhang H, Zhang Y. Deep learning on computational-resource-limited platforms: A survey. Mob Inf Syst. 2020; 2020(1): 8454327.
3. Zhang H, Xing M, Wu Y, Zhao C. Compiler Technologies in Deep Learning Co-Design: A Survey. Intelligent Computing. 2023 Jun 19; 2: 0040.
4. Patel H, Ramanan BA, Khan MA, Williams T, Friedman B, Drabeck L. Automating Code Adaptation for MLOps--A Benchmarking Study on LLMs. arXiv preprint arXiv:2405.06835. 2024 May 10.
5. Sutton RS, Barto AG. Reinforcement Learning: An Introduction. 2nd ed. Cambridge (MA): MIT Press; 2018.
6. Li M, Liu Y, Liu X, Sun Q, You X, Yang H, Luan Z, Gan L, Yang G, Qian D. The deep learning compiler: A comprehensive survey. IEEE Trans Parallel Distrib Syst. 2020 Oct 13; 32(3): 708–27.
7. Babu RG, Nedumaran A, Manikandan G, Selvameena R. Tensorflow: Machine learning using heterogeneous edge on distributed systems. In Deep Learning in Visual Computing and Signal Processing; Apple Academic Press. 2022 Oct 20; 71–90.
8. Werner M, Servadei L, Wille R, Ecker W. Automatic compiler optimization on embedded software through k-means clustering. In Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD. 2020 Nov 16; 157–162.
9. Lee B, Lu L, Wang T, Kim T, Lee W. From zygote to morula: Fortifying weakened aslr on android. In 2014 IEEE Symposium on Security and Privacy. 2014 May 18; 424–439.
10. Tripathy HK, Mishra S, Rout M, Balamurugan S, Mishra S, editors. Optimized Computational Intelligence Driven Decision-making: Theory, Application and Challenges. Hoboken (NJ): John Wiley & Sons; 2024.
11. Rahmani TA, Belalem G, Mahmoudi SA, Merad-Boudia OR. Equalizer: Energy-efficient machine learning-based heterogeneous cluster load balancer. Concurr Comput: Pract Exp. 2024 Oct 25; 36(23): e8230.
12. Raschka S, Patterson J, Nolet C. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. Information. 2020 Apr 4; 11(4): 193.
13. Kumar D, Bezdek JC, Palaniswami M, Rajasegarar S, Leckie C, Havens TC. A hybrid approach to clustering in big data. IEEE Trans Cybern. 2015 Sep 29; 46(10): 2372–85.
14. Haneda M, Knijnenburg PM, Wijshoff HA. Optimizing general purpose compiler optimization. In Proceedings of the 2nd conference on Computing frontiers. 2005 May 4; 180–188.

