2024 IEEE 6th International Conference on Cybernetics, Cognition and Machine Learning Applications (ICCCMLA) | 979-8-3315-0579-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICCCMLA63077.2024.10871813

A Survey of Optimized Compiler Using Advanced Machine learning and Deep Learning Techniques

Lal Bahadur Pandey
Dept. of CSE(AI & ML)
CMR Engineering College
Hydrabad , India
pandey.lalbahadur@gmail.com

Manisha Sharma
Dept. of ETC
Bhilai Institute of Technology
Durg, Chhattisgarh, India
manishasharma1@rediffmail.com

Rajesh Tiwari*
Dept. of CSE
CMR Engineering College
Hydrabad , India
drrajeshtiwari20@gmail.com

Radhe Shyam Panda
Dept. of CSE(AI & ML)
CMR Engineering College
Hydrabad , India
rspanda@cmrec.ac.in

Partha Roy
Dept. of CSE
Bhilai Institute of Technology
Durg, Chhattisgarh, India
drproy77@gmail.com

Abstract- Optimizing compilers is a difficult and time-consuming task, especially when done by hand. As far as we know, the compiler handles both translation and optimization. An efficient compiler system can become more automated and simple, as evidenced by recent studies using deep learning and machine learning approaches. Model training, prediction, optimization, and feature selection are handled by most machine learning and deep learning methods. In this case, choosing the optimal characteristics is necessary in order to use deep learning and machine learning techniques to enhance the optimization quality. This study examines various approaches that might be utilized to enhance and refine the quality of the chosen heuristics as well as the general quality of machine learning and deep learning models in order to boost the compiler's efficiency. The phase-ordering problem, the amount of iterative program evaluations, and the time needed to obtain the best forecast are only a few of the many subjects covered by these approaches.

Keywords— Phase-ordering, machine learning-ML, optimized compiler, deep learning, and survey.

I. INTRODUCTION

These days, deep learning and machine learning approaches are applied to compiler system optimization, as the model does not perform any manual labor in this case. Over the past ten years, there has been a lot of interest in optimizing compilers. A major improvement in software development and performance optimization is an efficient compiler that makes use of deep learning and machine learning. Compilers have historically translated high-level code into machine code with the goal of maximizing memory utilization and execution speed. However, in order to attain optimal performance, more advanced methodologies are needed due to the complexity of modern hardware designs and applications. Performance tuning and code optimization are two common compilation challenges. Code generation, preconference prediction, and autotuning are the functions of machine learning in compiler optimization, while pattern recognition, profile-guidedoptimization, and hardware-specific optimizations are the functions of deep learning models.

For several reputations, same laborious and intricate procedures had to be repeated, and the architecture also needed to be taken into account. Therefore, the best and alternate approach for automated compiler optimization is to use select machine learning and deep learning. I'll be introducing several machine learning and deep learning optimization techniques here. By producing more effective machine code, compiler system optimization can improve software application performance.

This efficiency can improve hardware resource consumption, lower memory usage, and simplify computation times. Advanced language features are supported by an optimized compiler system, enabling programmers to write more effective code without compromising speed.

II. LITERATURE SURVEY

Shewale as well as the others since the effort is too hard, it is not possible to find an optimal, or even good, combination of optimization for an unpredictable programmed on an arbitrary design using standard manual analysis approaches. This results in the development of a special optimization model that applies machine learning and deep learning methods [1].

Two deep learning networks, one based on the CNN model and the other on the LSTM model, were described by Pizzolotto et al. We assessed Shem and so that they may both attain over 98% accuracy in compiler identification and optimization, peaking at over 99% when further data becomes available. We further show that the accuracy of the system is approximately 95% even with relatively small input sequences of only 125 bytes, or about 30 instructions. Therefore, we conclude that both networks handle inputs made up of raw bytes with identical effectiveness [2].

Compiler optimization levels are crucial for binary analysis, according to Chen Y et al., however COTS binaries do not contain them. The first end-to-end system, named HIMALIA, is presented in this study. It is capable of recovering compiler optimization levels from disassembled binary code without requiring any understanding of the semantics of the target instructions set. To do this, we formulate the challenges as a two-layer recurrent neural network training task using deep learning techniques. HIMALIA is further enhanced by two additional techniques: instruction embedding and a novel method of function representation. This dataset was assembled by GCC and includes 378,695 distinct functions from 5828 binaries. As a result, the system's accuracy is roughly 89% [3].

The four characteristics that an approach should have, according to Georgiou et al., are what enable an optimizer to be tuned as part of a compiler's daily development cycle. That embodies mobility, quickness, adaptability, and perceptiveness. Rather than helping the compiler engineer tune a standard optimization level, most traditional auto tuning techniques—like iterative compilation and MLB approaches—are primarily designed to provide better optimization than the standard optimization level for a particular application. These methods usually need to run through thousands of iterations or necessitate a new, costly training phase with every compiler release. Furthermore, they are unable to reveal

optimization trends and their potential causes, with a few notable exceptions being coupled elimination strategies.

As a result, they are unable to demonstrate all four necessary attributes, and as a result, the compiler engineer can only use them to a limited extent [4].

Machine learning-based compilation was presented by Wang Z et al., who also discussed how effective it is in identifying an evidence-based strategy for compiler optimization. After fifty years of compiler automation, this is the most recent phase. In the last ten or so years, machine learning-based compilation has been a popular topic for compiler research and has attracted a lot of scholarly attention and publications. Although a comprehensive catalogue of all study fields and an easily available overview of the primary research fields and their future prospects may be supplied. The use of machine learning is not a panacea. All it can learn is the information we give it. As opposed to what some fear, it allows for considerably more innovation and opens up new study fields, not dumbing down the function of compiler writers [5].

Gong and colleagues report the first quantitative analysis of compiler stability, which explains a significant amount of the performance variation in target code generated by a compiler from semantically comparable but differently structured source code. The stability of the C language compilation processes for loop nests utilizing GCC, ICC, and Clang is investigated in this work. We also computed the performance headroom of the processes. We were able to ascertain the stability and performance headroom of the compiler by analyzing a sizable collection of loop nests extracted from libraries and benchmarks along with their semantically equivalent mutations generated by executing a sequence of semantic-preserving modifications [6].

Michael R. Jantz et al. define phase selection as the process of adjusting the relevant collection of compiler optimization phases for certain functions or programs in order to improve the code generator's performance. Researchers have recently devised heuristic techniques based on feature vectors to carry out phase selection during online JIT compilation. Although program startup times are sped up by these algorithms, steady-state performance did not show any improvement over the fixed single sequence baseline that is used by default. Regretfully, it remains unclear if the current online phase selection heuristic is to blame for the absence of study-state performance increase or if phase selection actually offers little to no speedup in online JIT contexts. This work aims to answer these concerns, while also analyzing the behavior of optimizations linked to phase selection and evaluating and enhancing the efficacy of current heuristic solutions [7].

C-GOOD, a unique DNN framework developed by Duseok Kang et al., produces C code that can execute on any embedded platform. Because the framework is optimization-aware, the user can choose which optimization strategies to apply to a particular network. The user can then explore the design space of network selection and software optimization with the aid of the automatically generated optimized C-code. Along with the systematic application of optimization techniques, a software optimization approach is also offered.

Pipelining, layer-wise Tucker decomposition, layer-wise quantization, combining batch normalization into weights, and input size reduction are all part of this process. The proposed techniques have been evaluated on three different hardware platforms: the Samsung Reconfigurable Processor, the Odroid XU4, and the Jetson TX2. The experimental results show that the baseline C code generated by C-GOOD performs better than the Darknet framework in terms of frame-per-second performance. Additionally, compared to the baseline unoptimized code, the optimization procedure increases speed by 2.2 to 25.83 times for a certain hardware platform [8].

It has been demonstrated by Sameer Kulkarni et al. that method-specific optimization orderings can result in notable speed gains when using the Jikes RVM JIT compiler. Additionally, research has shown that a neural network that provides appreciable performance gains over a well-planned optimization scheme can be automatically generated using the neuro-evolution technique. We demonstrate up to 20% improvements in total execution time. This shows that machine learning models may successfully be applied to determine the method's optimization order within a compiler. The current study is encouraging since it offers a new perspective on phase ordering issues that have been researched for many years. This method's potential for improvement has not yet been fully realized; further work will be needed to enhance the machine learning algorithm and yield even greater benefits [9].

According to Kovac M et al., enormously parallel and heterogeneous compilation is the way of the future, with specialized accelerator devices and instruction sets in both edge and cluster computing. Software development, though, will eventually become the bottleneck. The software would need to find solutions for a variety of issues, including heterogeneous device mapping, capability discovery, parallelization, adaption to new ISAs, and many more, in order to fully realize the promise of hardware marvels. It will be hard for human developers to manually control this systematic complexity. It is necessary to assign these issues to knowledgeable compilers. We highlight the latest findings using various techniques such as reinforcement learning, polyhedral optimization, and deep learning [10].

The sequence in which optimization techniques are executed can greatly influence the level of performance achieved, as noted by Shewale C et al. Consequently, we designed a novel compiler optimization prediction framework. Our model consists of three key operational steps: feature extraction, model exploitation, and model training. The model training phase encompasses both the generation of candidate sample sets and their initialization. Subsequently, the inputs were directed to the feature extraction phase, where enhanced entropy, dynamic, and static features were obtained. In the feature exploitation phase, an improved Hunger Games search algorithm is utilized to identify the most effective characteristics, which are then optimized. Leveraging these selected features, we applied a Convolutional Neural Network in this study to predict compiler optimization. The findings indicate that our innovative compiler optimization model surpasses previous techniques [11].

Erik and everyone else we present the Bayesian Compiler Optimization framework (BaCO), a plug-and-play auto scheduling solution for contemporary scheduling languages that is intended to be used with a variety of hardware backends. Modern auto tuners take 2.7×-10× longer to reach expert-level performance than BaCO. Compiler users can save up their time by assigning BaCO the difficult and time-consuming work of scheduling, as the policy and mechanism are separated from one other. Even if we demonstrate that BaCO can deliver high-performing solutions in less than a minute, software development still finds this time to be unacceptably long. Being able to use an auto tuner while developing is the holy grail of auto scheduling; ideally, this allows the user to execute autotuning each time they compile their code. In this manner, users can regularly monitor both functional and non-functional properties across the many stages of the program

lifecycle. Increasing the auto tuner's efficiency would, in fact, allow for a new level of autotuning-in-development-loop paradigm that is not possible with the autotuning technology available now [12].

Hui Liu and colleagues presented ELOPS, a compiler optimization parameter selection technique that can provide optimization settings for various applications on its own. The model accepts program characteristics as input and outputs the compiler optimization parameters that are close to ideal. During the ELOPS model's training data creation phase, an enhanced particle swarm optimization (PSO) method for examining the compiler optimization parameter space was put forth, enabling more efficient resolution of limited multi-objective optimization problems. On two different platforms, we saw speedups of 1.39× and 1.36× in comparison to GCC's typical optimization level of O3. The improved PSO conducts a methodical investigation of the compiler optimization space in order to acquire more precise sample data during the model construction stage. We suggested FCR, a technique for describing program characteristics that combines static and dynamic elements to depict the programs. We can offer more and better sample parameters for creating the compiler optimization prediction model by using this novel program feature representation technique and identifying the compiler's ideal parameters. We used Stacking ensemble learning to create a statistical model based on this data, which we then integrated into the compiler framework. We then used the functionality to select the compilation process and optimization setting. In order to determine the best compilation optimization parameters, the ELOPS model may now automatically extract features from the new target program and use the information it has accumulated. The ELOPS strategy outperformed the current way with program execution time speedups of 1.29× and 1.26× on two separate platforms, respectively, compared to the compiler's default standard optimization level -O3 [13].

Yang, Y, and everyone else in this work, 142 primary works on DL for SE from 21 publication venues—including conference proceedings, symposiums, and journals—were subjected to a systematic literature review (SLR). We developed four study topics to thoroughly examine different facets related to DL model applicability to SE activities. Our SLR revealed a considerable increase in the amount of research interest in DL for SE. After thorough research and analysis, three DL architectures with 30 distinct DNNs were employed in primary studies; when compared to other DNNs, RNN, CNN, and FNN are the three neural networks that are most frequently used. Additionally, we examined the popularity of three distinct model selection procedures and generalized them. In RQ3, we included a thorough overview of the essential methods for data collecting, data processing, model optimization, and model evaluation so that readers could fully comprehend the DNN training and testing procedure. In RQ4, we categorized main research based on the particular SE tasks they resolved, examined the distribution of DL approaches applied in various SE activities, and provided a succinct overview of each study. We saw that DL approaches were used to cover six SE actions across 23 SE themes. Lastly, we determined a number of present issues that would require further attention in research on the application of DLs in SE [14].

Li J et al. offer the DNN Graph Compiler, a tensor compiler that generates high-performance code for deep neural network graphs through a hybrid approach combining methods from expert-tuned kernels and compiler optimization [15]. Deep learning-specific optimization problems such aggressive graph operation fusion, low-precision computing, memory layout and static tensor shape optimization, constant weight optimization, and memory buffer reuse are all addressed by One DNN Graph Compiler [16]. According to experimental results, for performance-critical DNN computation graphs and end-to-end models on Intel® Xeon® Scalable Processors, there are notable performance improvements over the current tensor compiler and primitive's library [17].

III. Discussions

As I mentioned in the survey above, one of the most frequent issues encountered when utilizing machine learning techniques for compiler optimization is the phase ordering problem. In order to obtain optimal model performance and generalization, the sequence of data pre-processing and transformation processes must be carefully and empirically considered, as demonstrated by the machine learning phase ordering problem. Effectively navigating the complicated environment of transformation orders requires a combination of automated search techniques, subject expertise, and empirical judgment. The goal of several research studies on this issue was to lessen the phase-ordering problem's negative effects on the compiler's effectiveness and performance. The problem-solving approach of using artificial neural networks shows that program profiles can be utilized for code optimization. Algorithms utilizing reinforcement learning perform 16% better than it. A compiler optimization method called loop unrolling seeks to decrease loop overhead and increase program performance.

Traditionally, to expose greater instruction-level parallelism and minimize the amount of loop control instructions executed, this is accomplished by manually expanding the loop body size (unrolling). Combining deep learning (DL) and machine learning (ML) with loop unrolling optimization offers a new way to potentially improve compiler speed even more. Loop pipelining is frequently combined with this issue. Throughout our investigation, we noted that numerous scholars put up proposals for systems and assessed them in comparison to the current loop unrolling technique. They employed a number of machines learning methods, including the decision tree algorithm and boosting. Even though a machine learning algorithm is quick, it is not quick enough to identify the optimal heuristics for compiler optimization. Two popular approaches to solving this issue include using active learning and narrowing the target of the iterative search. While it might seem that the ideal course of action would be to narrow the search, it's important to remember that active learning makes the heuristics much easier to use by reducing both the computation time and the number of training samples required. The disadvantages of random iterative compilation techniques are lessened by auto-tuning systems, which use Bayesian networks to provide application-specific optimization that guarantees three times faster results than random iterative compilation. Solving the phase-ordering problem is one possible future use for this technology.

IV. Conclusion

An analysis of the research work based on optimized compiler techniques has demonstrated that machine learning and deep learning may now be used to solve challenges in this field quickly and effectively. Our survey's results enabled us to draw the conclusion that compiler performance has significantly increased as a result of the application of deep learning and machine learning approaches for compiler optimization. To achieve compiler optimization, a number of cutting-edge methods have been used, including both established and recently developed approaches like artificial neural networks (ANN), supervised and unsupervised learning, and other advanced techniques. Selecting a solution for a

compiler's issues requires first identifying the issues it faces. Most of the time, researchers combined a number of methods to get the best outcomes. This survey study aims to gather the findings from studies on compiler optimization using machine learning and deep learning techniques. Further research into the complex and cutting-edge techniques mentioned, as well as a combination of these, may be part of the future scope of this field of work in order to achieve compiler optimization beyond the state-of-the-art performance of the same. Our intention is to assist researchers in drawing comparisons between their proposed methods and those that have already been examined in our study.

REFERENCES

[1] Shewale, C., Shinde, S. B., Gurav, Y. B., Partil, R. J., & Kadam, S. U. (2023). Compiler optimization prediction with new self-improved optimization model. International Journal of Advanced Computer Science and Applications, 14(2).

[2] Pizzolotto, D., & Inoue, K. (2020, September). Identifying compiler and optimization options from binary code using deep learning approaches. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 232-242). IEEE.

[3] Chen, Y., Shi, Z., Li, H., Zhao, W., Liu, Y., & Qiao, Y. (2019). Himalia: Recovering compiler optimization levels from binaries by deep learning. In Intelligent Systems and Applications: Proceedings ofthe 2018 Intelligent Systems Conference (IntelliSys) Volume 1 (pp. 35-47). Springer International Publishing.

[4] Georgiou, K., Chamski, Z., Amaya Garcia, A., May, D., & Eder, K. (2022). Lost in translation: Exposing hidden compiler optimization opportunities. The Computer Journal, 65(3), 718-735.

[5] Wang, Z., & O'Boyle, M. (2018). Machine learning in compiler optimization. Proceedings of the IEEE, 106(11), 1879-1901.

[6] Gong, Z., Chen, Z., Szaday, J., Wong, D., Sura, Z., Watkinson, N., ... & Torrellas, J. (2018). An empirical study of the effect of source-level loop transformations on compiler stability. Proceedings of the ACM on Programming Languages, 2(OOPSLA), 1-29.

[7] Jantz, M. R., & Kulkarni, P. A. (2013, March). Performance potential of optimization phase selection during dynamic JIT compilation. In Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (pp. 131-142).

[8] Kang, D., Kim, E., Bae, I., Egger, B., & Ha, S. (2018, November). C-GOOD: C-code generation framework for optimized on-device deep learning. In 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (pp. 1-8). IEEE.

[9] Kulkarni, S., & Cavazos, J. (2012, October). Mitigating the compiler optimization phase-ordering problem using machine learning.In Proceedings of the ACM international conference on Object oriented programming systems languages and applications (pp. 147-162).

[10] Kovac, M., Brcic, M., Krajna, A., & Krleza, D. (2022, May). Towards intelligent compiler optimization. In 2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO) (pp. 948-953). IEEE.

[11] Shewale, C., Shinde, S. B., Gurav, Y. B., Partil, R. J., & Kadam, S. U. (2023). Compiler optimization prediction with new self-improved optimization model. International Journal of Advanced Computer Science and Applications, 14(2).

[12] Hellsten, E. O., Souza, A., Lenfers, J., Lacouture, R., Hsu, O., Ejjeh, A., ... & Nardi, L. (2023, March). Baco: A fast and portable Bayesian compiler optimization framework. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (pp. 19-42).

[13] Liu, H., Xu, J., Chen, S., & Guo, T. (2022). Compiler optimization parameter selection method based on ensemble learning. Electronics, 11(15), 2452.

[14] Yang, Y., Xia, X., Lo, D., & Grundy, J. (2022). A survey on deep learning for software engineering. ACM Computing Surveys(CSUR), 54(10s), 1-73.

[15] Li, J., Qin, Z., Mei, Y., Cui, J., Song, Y., Chen, C., ... & Lavery, D. (2024, March). oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation. In 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (pp. 460-470). IEEE.

[16] Vadali Pitchi Raju, Tushar Kumar Pandey, Rajeev Shrivastava, Rajesh Tiwari, S. Anjali Devi, Neerugatti Varipallay vishwanath,(2024) "Computational genetic epidemiology: Leveraging HPC for largescale AI models based on Cyber Security", Journal of Cybersecurity and Information Management (JCIM), Vol. 13, No. 02, pp. 182-190, ISSN (Online) 2690-6775, ISSN (Print) 2769- 7851,Doi : https://doi.org/10.54216/JCIM.130214.

[17] Rajesh Tiwari, Satyanand Singh, G. Shanmugaraj, Suresh Kumar Mandala, Ch. L. N. Deepika, Bhanu Pratap Soni, Jiuliasi V. Uluiburotu, (2024) "Leveraging Advanced Machine Learning Methods to Enhance Multilevel Fusion Score Level Computations", Fusion: Practice and Applications, Vol. 14, No. 2, pp 76-88, ISSN: 2770-0070 , Doi : https://doi.org/10.54216/FPA.140206.