Machine Learning and Compilers: A Survey of ML Techniques for Enhancing Optimizing Compilers

A Technical Report
presented to the faculty of the
School of Engineering and Applied Science
University of Virginia

by

Ryan Steele

December 1, 2022

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Ryan Steele

Technical advisor: Briana Morrison, Department of Computer Science

Machine Learning and Compilers: A Survey of ML Techniques for Enhancing Optimizing Compilers

CS4991 Capstone Report, 2022
Ryan Steele
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
rws7bf@virginia.edu

ABSTRACT
Optimizing compilers are critical to modern software solutions, as they allow program code written at a comparatively high level of abstraction to be automatically transformed into reasonably performant native machine code. Such optimization is inherently uncertain, as the methods used and application order can greatly influence the final level of optimization in ways that are not obvious and difficult to predict. Given the size of the search space presented by the often large number of available optimization passes and possible orderings, machine learning (ML) techniques may have potential applications for increasing efficiency and practical efficacy by providing a method for estimating which optimizations are most likely to be beneficial, given the structure of the input program. This brief yet reasonably comprehensive overview of the state of ML in optimizing compilers synopsizes major papers in the field and provides an informed assessment of the overall viability of these methods and the potential for future research efforts.

1. INTRODUCTION

Why is software written in languages like C or C++ typically so much faster than its logical equivalent in languages like Python or JavaScript? Although implementations may vary, the most obvious answer to this question is that the first set of languages is usually compiled, while the second is traditionally interpreted. In most cases, this performance difference is caused by more than just the overhead required by interpreters to translate instructions on-the-fly, as compilers will try to optimize the output binary rather than simply providing a naïve translation. This often results in dramatic changes to the code's structure, which can provide very significant performance improvements.

Creating compilers that are both efficient and effective at code optimization has many practical benefits for creating more efficient software. Although modern compilers have become quite good at this task, the problem itself is quite complex, and can be quite computationally expensive. This is due in part to the non-linear nature of code optimizations, where applying one optimization may affect the effectiveness of future optimizations in unpredictable ways, making the choice of optimizations and their order of application critical to the amount of code improvement achieved overall.

This difficulty is compounded by the vast number of potential code optimizations that compilers may choose to employ, making a brute force analysis of all possible combinations of optimizations and phase-orderings intractable in essentially all real-world applications. Thus, the size of the search space for potential solutions is quite large, making ML methods a reasonable approach to finding good solutions. Applying

ML methods has the added benefit of being able to generalize between common program structures, with the ability to quickly predict what kinds of optimizations are likely to be most beneficial. This has the potential to make optimizing compilers more efficient, as well as enabling further levels of optimization that, with current methods, may take an unreasonable amount of time.

Due to the significant amount of work published in this area and the rate of current research, the topic may seem unapproachable to a newcomer. My proposal provides a summary of the notable advancements in ML techniques as they apply to optimizing compilers, and an analysis of the future potential of such methods and possible avenues of future research.

2. RELATED WORKS

Several surveys of the field have been conducted by various authors in recent years. Ashouri et al. [2018] provides a detailed look at ML approaches to optimization selection and ordering. Wang and O'Boyle [2018] and Allamanis et al. [2018] provide a more in-depth summary of the field and present applications and possible new directions. More recent studies include Leather and Cummins [2020], which provides an analysis of existing ML compilers while also discussing future potential. These sources served as a starting point for my proposal and were used to identify important papers, as well as helping to guide my perception of the field.

3. PROJECT SUMMARY

Work on ML in compilers has evolved dramatically from its inception to current-day approaches and research directions. Most notable is the shift in complexity and scope, as earlier works focus mainly on simpler or more fundamental tasks, while more recent work has expanded include more complex optimization, such as seen in programs running in a multi-threaded environment. Much of this has been driven by advances in ML techniques, which has allowed for better and more efficient training on large datasets.

3.1 Review of Early Works

Perhaps the earliest paper to explore the use of ML in compiler design investigated the use of neural networks and decision trees for providing better branch prediction [3]. In this paper, the authors use a corpus of programs with known branching behavior to statically predict the behavior of new programs using a method referred to by the authors as evidence-based static prediction (ESP). Using this method, the authors were able to reduce mispredicted branches by 20% over their set of test programs.

Further work was performed by Monsifrot et al. [2002], in which the authors explore the possibility of using ML to automatically generate heuristics for loop unrolling, a compiler optimization that tries to increase performance by expanding multiple iterations of a loop into a single loop to reduce the control logic (such as counter increments and condition checks) required. Applying this transformation may not always be beneficial and can decrease performance in many cases, hence the need for heuristics. In traditional compilers, such heuristics need to be created manually for every target architecture, requiring a significant amount of time and skill. In their study, the authors demonstrated that in many cases it was possible to achieve better performance with their decision tree learning model when compared to a reference compiler. Stephenson and Amarasinghe [2005] later built upon this work, and were able to achieve an improvement of about 5% on the SPEC 2000 benchmark suite by using nearest neighbor and support vector machine ML algorithms.

3.2 Directions of Current Research

Recent work on ML in compilers has focused heavily on optimizing for parallelized programs, such as programs running on multi-core processors or GPUs, as well as making use of advancements in ML, particularly with reinforcement learning.

For example, Ganapathi et al. [2009] noted that the complexity of such parallelized systems leads to a search space that is very expensive to search exhaustively, and that the non-obvious relation between chosen optimization parameters and overall performance makes it difficult even for experts to find a good solution. In their work, the authors were able to use ML methods to find compilation configurations that performed comparable or better than those selected by experts at about 2000 times the speed. This was accomplished using nearest neighbor based stencils. Many recent studies have built upon this work, exploring ways in which auto-tuning compilation parameters can lead to performance improvements [7, 10].

Rapid advancement in ML techniques has been one of the major drivers for new research incorporating ML in compiler design. Advancements in reinforcement learning, a technique that involves training a model by trying to maximize some reward function in an unsupervised environment, shows significant promise in increasing the effectiveness of ML in compilers. Reinforcement learning based methods have been employed with great success in several compiler pipelines, such as learning optimization orderings for the LLVM compiler [5], in which the authors observed 16% improved performance over traditional compilation.

4. ANTICIPATED BENEFITS

I anticipate that applying ML methods in optimizing compilers to will have a significant impact in computing within the next few decades, both for developers and for users of the compiled software. The main benefit that I expect to see from mainstream integration of ML in compilers is faster compile times, as well-trained ML models will likely be able to determine the best way to compile a program without requiring an intensive parameter search or other such expensive processing. Furthermore, ML methods could likely free up compiler developer time, as they would no longer need to manually tune their optimization methods to each target architecture, allowing them to focus more on other parts of the compiler. Finally, I expect users to eventually see improved performance when running software, as well as an increase in the complexity of software able to perform well on comparatively weaker hardware, possibly with lower energy requirements. Enhancing optimizing compilers with ML is likely to have significant implications on program efficiency in the long run, and as such I believe that further research in the space is clearly warranted.

5. CONCLUSION

I learned a lot about ML and its potential uses in compiler design while creating this report. Integrating ML techniques into the compiler optimization process offers many potential benefits, the most significant of which are likely overall program efficiency and time savings, particularly the possibility of reducing developer time spent in hand-crafting heuristics for individual target architectures. A summary of the field such as the one provided by this project is an invaluable resource for researchers unfamiliar with the field seeking an introductory source, which also serves to provide further resources for additional reading. Although this review is brief, it provides a solid overview of the state of ML in optimizing compilers, which

provides an overall understanding of the field and its development.

6. FUTURE WORK

The most effective next step for expanding upon this project would be to increase the amount of research reviewed from what is provided in this report. A broader survey would provide further insight into the development of the use of ML in the design of optimizing compilers and current practices. It may also include significant works that I overlooked in my initial review of the field.

Other directions that may be taken to build upon what is provided in this report include performing a survey of a more specific use case of ML in compiler design, such as the use of ML for optimizing GPU shader compilation, the potential of reinforcement and deep learning methods to improve integration of ML and compiler optimization, and applications in cloud and distributed computing.

REFERENCES

[1] Allamanis, M., Barr, E., Devanbu, P. and Sutton, C., 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, 51(4), pp.1-37.

[2] Ashouri, A., Killian, W., Cavazos, J., Palermo, G. and Silvano, C., 2018. A Survey on Compiler Autotuning using Machine Learning. *ACM Computing Surveys*, 51(5), pp.1-42.

[3] Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M. Zorn, Z., 1997. Evidence-Based Static Branch Prediction Using Machine Learning. *ACM Transactions on Programming Languages and Systems*, 19(1), pp 188-222.

[4] Ganapathi, A., Datta, K., Fox, A., Patterson, D. A., 2009. A case for machine learning to optimize multicore performance. *Proceedings of the First USENIX conference on Hot topics in parallelism*.

[5] Huang, Q., Haj-Ali, A., Moses, W.S., Xiang, J., Stoica, I., Asanović, K., & Wawrzynek, J., 2019. AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning. *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 308.

[6] Leather, H. and Cummins, C., 2020. Machine Learning in Compilers: Past, Present and Future. *2020 Forum for Specification and Design Languages (FDL)*.

[7] Menon, H., Bhatele, A., & Gamblin, T., 2020. Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization. *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 831-840.

[8] Monsifrot, A., Bodin, F., and Quiniou, R., 2002. A Machine Learning Approach to Automatic Production of Compiler Heuristics. *Artificial Intelligence: Methodology, Systems, and Applications*, pp. 41-50.

[9] Stephenson, M., and Amarasinghe, S., 2005. Predicting Unroll Factors Using Supervised Classification. *International Symposium on Code Generation and Optimization*, pp. 123–134.

[10] Thiagarajan, J.J., Jain, N., Anirudh, R., Giménez, A., Sridhar, R., Marathe, A., Wang, T., Emani, M.K., Bhatele, A., and Gamblin, T, 2018. Bootstrapping Parameter Space Exploration for Fast Tuning. *Proceedings of the 2018 International Conference on Supercomputing*.

[11] Wang, Z. and O'Boyle, M.F., 2018. Machine Learning in Compilers. *Proceedings of the IEEE*, 106(11), pp. 1879-1901