Application of artificial intelligence in compiler design

Swati Rajwal
Emory University
13 PUBLICATIONS **30** CITATIONS

Pinaki Chakraborty
Netaji Subhas Institute of Technology
3 PUBLICATIONS **9** CITATIONS

Application of artificial intelligence in compiler design

Swati Rajwal and Pinaki Chakraborty*
Department of Computer Science and Engineering, Netaji Subhas University of Technology, New Delhi 110078, India
* Corresponding author. E-mail: pinaki_chakraborty_163@yahoo.com

Abstract

Compilers are complex programs and specialized algorithms are available to implement the different phases of compilers. Nevertheless, some researchers have used artificial intelligence techniques to improve the performance of the syntax analysis, code optimization and code generation phases of compilers, and to check the correctness of compilers. This paper reviews the artificial intelligence, machine learning and deep learning techniques that have been successfully used so far for designing and testing compilers.

Keywords: Compiler, artificial intelligence, syntax analysis, code optimization, code generation, compiler testing.

Introduction

A compiler is a software tool that converts programs written in a high-level language into a binary format that can be executed by a computer. Compilers hold a pivotal position in computer science. Extensive research on compilers has been conducted since the 1950s. Advances in compiler design allowed development of various programming languages which are now used to develop software used by millions of people around the world.

Artificial intelligence (AI) is the ability of computer programs to perform computations that have not been explicitly specified. Machine learning (ML) is a technique of achieving AI where computational models are trained on sample data to make accurate predictions. Deep learning (DL) is a form of ML where computational models with unbounded number of layers are trained on large volumes of data. Humenik and Pinkham used simple AI techniques to improve the performance of compilers in 1990. Several researchers have used ML, DL and other types of AI techniques to enhance compiler design since then (Table 1). AI techniques have been used for analyzing the syntax of programs, selecting and ordering optimization techniques to improve the code, generating codes for parallel computers and testing compilers.

Application of AI in different phases of compilers

Syntax analysis

Compilers analyze the programs to identify the different statements in them and ensure that they have been written as per the specification of the programming language. The process is known as syntax analysis. The syntax of a programming language is specified using formal rules. Compilers apply these rules one by one to analyze programs. Although several traditional algorithms for syntax analysis have been known since the 1960s, some later researchers have used AI techniques to make the syntax analysis process faster (Fig. 1). Humenik and Pinkham (1990) and Chakraborty (2008, 2009) assigned priorities to the rules with the priorities being calculated according to simple heuristics. The rules with higher priority are given precedence (Chakraborty, 2013).

____

Cite this paper as: Rajwal, S., & Chakraborty, P. (2023). Application of artificial intelligence in compiler design. In *Proceedings of Forty-first National Conference on Recent Trends in 5G Technology & Artificial Intelligence* (pp. 234–239).

Table 1. Summary of AI techniques used in developing and testing compilers

AI technique	Brief description
Artificial neural network (ANN)	An ANN is a computational model consisting of a large number of nodes, known as artificial neurons, which are arranged in multiple layers. Each artificial neuron receives inputs from a number of artificial neurons in the previous layer, applies an activation function that may be linear or nonlinear on a weighted sum of the inputs, and transmits the result to artificial neurons in the next layer.
Deep reinforcement learning	In deep reinforcement learning, an ANN with a large number of layers is used to solve a numerical task by trial and error.
Farthest-first traversal	The farthest-first traversal can be used to select a solution of a numerical problem from a set of several candidate solutions. In each iteration, the algorithm checks a candidate solution whose properties differ most widely from the one checked in the last iteration.
Genetic algorithm (GA)	A GA starts with a set of random candidate solutions for a numerical problem and then iteratively applies biology-inspired operations like selection, mutation and crossover till an optimal solution is obtained.
Genetic programming (GP)	GP is a technique in which computer programs are evolved by iteratively applying biology-inspired operations like mutation and crossover.
Heuristic	A heuristic is an approach that can be used readily to find an approximate solution of a complex numerical problem.
K* algorithm	A K* algorithm uses heuristic to classify a new data item based on its similarities with previously known data items.
k-nearest neighbours (k-NN) algorithm	The k-NN algorithm classifies a new data item based on its similarities with its k nearest items in the dataset.
Linear regression	Linear regression is a type of regression analysis that estimates the relationship between a dependent variable and a linear combination of parameters.
Long short-term memory (LSTM)	An LSTM is an ANN with feedback connections and memory in the form of gates that can process sequence of data.
Multilayer perceptron	A multilayer perceptron is an ANN without any feedback connection and in which the artificial neurons use threshold activation functions.
Regression analysis	Regression analysis is the estimation of the relationship between a dependent variable and one or more independent variables. Different techniques are available for regression analysis.

Code optimization

Compilers typically use a wide range of optimization techniques to make the binary code being generated shorter and faster. Conventionally, the programmers developing the compilers need to select the optimization techniques to be used in them. However, some researchers have used AI to decide which optimization techniques are to be included in compilers. The AI techniques that have been used so far to select optimization techniques are genetic programming (GP) (Stephenson *et al*., 2003), genetic algorithm (GA) (Agakov *et al*., 2006), probabilistic ML (Fursin *et al*., 2011), artificial neural network (ANN) (Dubach *et al*., 2007) and long short-term memory (LSTM) (Cummins *et al*., 2017). Programmers developing compilers can manually analyze only a few optimization techniques. Alternatively, AI can be used to study the performance of various available optimization techniques on a large number of sample programs, and only the best optimization techniques can be then included in the compilers. Fursin *et al*. (2011) and Cummins *et al*. (2017) followed this approach and could achieve up to 11% and 14% speedup, respectively.

Compilers typically include a number of optimization techniques to improve the code being generated. These optimization techniques are applied on the programs one after another. The outcome of an optimization technique influences the outcome of the subsequent optimization techniques and the overall result. The task of finding an order in which optimization techniques should be applied on the programs to obtain plausible results is known as the phase ordering

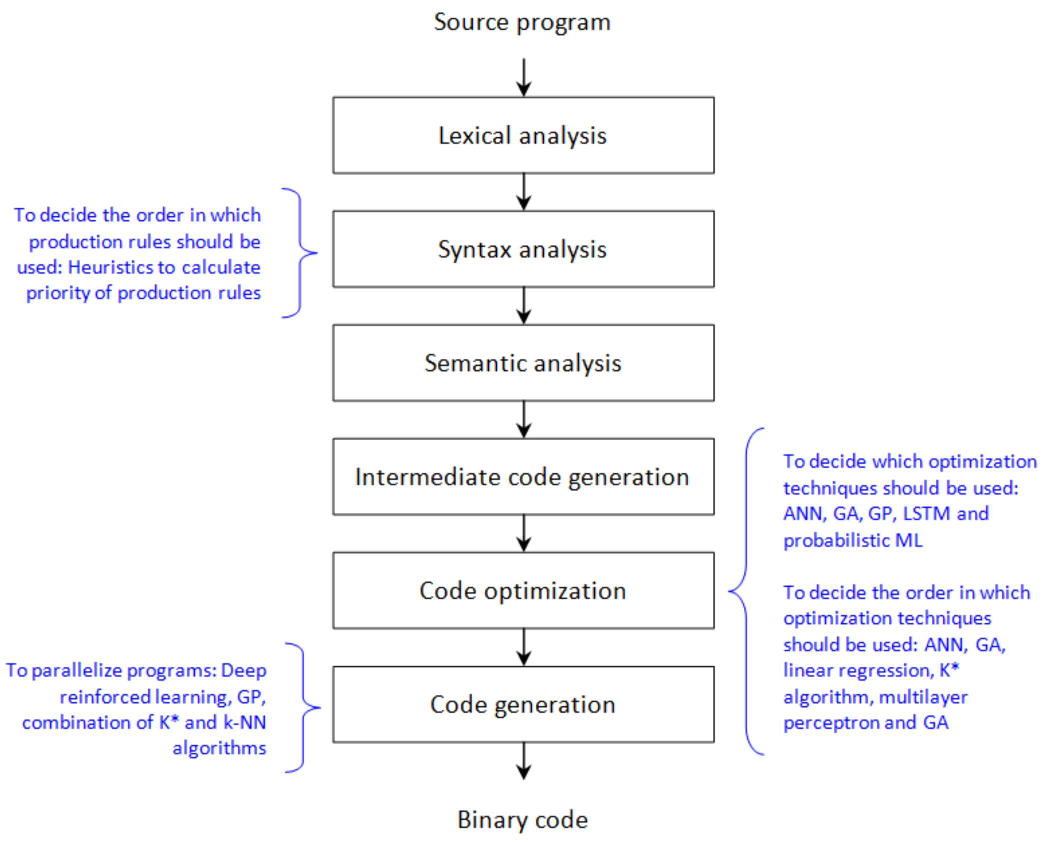
Source program
Lexical analysis
Syntax analysis
Semantic analysis
Intermediate code generation
Code optimization
Code generation
Binary code
To decide the order in which production rules should be used: Heuristics to calculate priority of production rules
To decide which optimization techniques should be used: ANN, GA, GP, LSTM and probabilistic ML
To decide the order in which optimization techniques should be used: ANN, GA, linear regression, K* algorithm, multilayer perceptron and GA
To parallelize programs: Deep reinforced learning, GP, combination of K* and k-NN algorithms

Fig. 1. The phases of a compiler and the AI techniques that have been used in them.

problem. Conventionally, the programmers developing the compilers are supposed to solve the phase ordering problem, *i.e.* decide the order in which the different optimization techniques will be applied on the programs. However, in the recent years, some researchers have used AI to solve the phase ordering problem. The AI techniques used for the purpose include ANN (Kulkarni and Cavazos, 2012), linear regression, multilayer perceptron, K* algorithm (Ashouri *et al.*, 2017) and GA (Almohammed *et al.*, 2019). These AI techniques can explore various permutations of optimization techniques and predict their performance. Ashouri *et al*. (2017) found that the AI techniques needed to explore only 0.001% of the possible permutations of the optimization techniques before being able to find a plausible solution to the phase ordering problem. Kulkarni and Cavazos (2012) reported that the use of ANN to solve the phase ordering problem resulted in 8% faster binary code.

Code generation

After an analysis of the programs and optimizing them, compilers generate binary code equivalent to them for ready execution. Most computers have a single central processing unit and it is easier to generate binary code for them. However, cluster computers and supercomputers have multiple processing units which can execute different parts of the same program in parallel. AI techniques can be used to determine which parts of a program can be executed in parallel so as to maximize the speed of execution. Stock *et al*. (2012) used a combination of K* and k-NN algorithms to determine the parts of the programs that should be executed in parallel. Deep reinforced learning (Haj-Ali *et al.*, 2020) and GP (Leather *et al.*, 2014) are two other AI techniques that have been used for the same purpose.

Application of AI in testing compilers

All software products are tested before deployment. Compilers are important software tools and hence tested thoroughly. Compilers should be able to generate binary code for correct programs and detect errors in incorrect programs. Compilers are typically tested using an approach

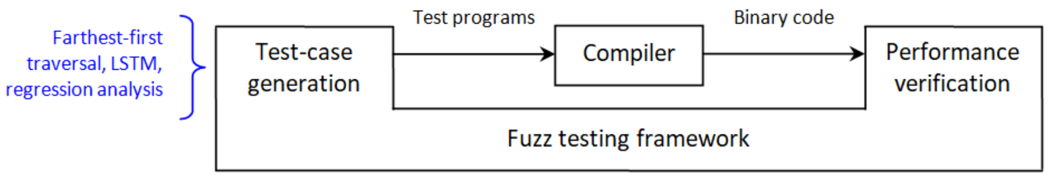

Fig. 2. Fuzz testing of compilers and AI techniques that have been used in test-case generation.

known as fuzz testing (Fig. 2). In this approach, a specialized software tool known as the test-case generator is used to automatically generate a large number of random test programs and the performance of the compiler on those test programs is recorded. However, this approach is time consuming and may not be able to detect all errors in the compiler. Consequently, AI techniques are used in the test-case generators to generate a small but better representational set of test programs. AI techniques like farthest-first traversal (Chen *et al*., 2013), regression analysis (Chen *et al*., 2017) and LSTM (Cummins *et al*., 2018; Xu *et al*., 2020) have been used to generate programs for testing compilers. The use of AI can help in reducing testing time by half (Chen *et al*., 2017).

Conclusion

Compilers are being developed and used since the 1950s. Several researchers have used AI techniques in compilers in the last three decades and got encouraging results. Given the importance of compilers in computer science, we call for further research on the scope of using AI in complier design and testing.

References

Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., & Williams, C. K. I. (2006). Using machine learning to focus iterative optimization. In *Proceeding of the International Symposium on Code Generation and Optimization* (pp. 295–305).

Almohammed, M. H., Alwan, E. H., & Fanfakh, A. B. M. (2019). Programs features clustering to find optimization sequence using genetic algorithm. In *Proceedings of the International Conference on Information, Communication and Computing Technology* (pp. 40–50).

Ashouri, A. H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., & Cavazos, J. (2017). Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization*, *14*(3), article 29.

Chakraborty, P. (2008). Use of heuristics in shift-reduce parsers. In *Proceedings of the International Conference on Data Management* (pp. 103–109).

Chakraborty, P. (2009). Design and implementation of a cross compiler. *Journal of Multidisciplinary Engineering Technologies*, *3*(2), 6–15.

Chakraborty, P. (2013). Modeling a parser as an expert system. In: Segura, J. M., & Reiter, A. C. (Eds.) *Expert System Software: Engineering, Advantages and Applications*, Nova Science Publishers, pp. 91–102.

Chen, J., Bai, Y., Hao, D., Xiong, Y., Zhang, H., & Xie, B. (2017). Learning to prioritize test programs for compiler testing. In *Proceedings of Thirty-ninth IEEE/ACM International Conference on Software Engineering* (pp. 700–711).

Chen, Y., Groce, A., Zhang, C., Wong, W. K., Fern, X., Eide, E., & Regehr, J. (2013). Taming compiler fuzzers. *ACM SIGPLAN Notices*, *48*(6), 197–208.

Cummins, C., Petoumenos, P., Murray, A., & Leather, H. (2018). Compiler fuzzing through deep learning. In *Proceedings of the Twenty-seventh ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 95–105).

Cummins, C., Petoumenos, P., Wang, Z., & Leather, H. (2017). End-to-end deep learning of optimization heuristics. In *Proceeding of the Twenty-sixth International Conference on Parallel Architectures and Compilation Techniques* (pp. 219–232).

Dubach, C., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F., & Temam, O. (2007). Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the Fourth International Conference on Computing Frontiers* (pp. 131–142).

Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., & Bodin, F. (2011). Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, *39*(3), 296–327.

Haj-Ali, A., Ahmed, N. K., Willke, T., Shao, Y. S., Asanovic, K., & Stoica, I. (2020). NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the Eighteenth ACM/IEEE International Symposium on Code Generation and Optimization* (pp. 242–255).

Humenik, K., & Pinkham, R. S. (1990). Production probability estimators for context-free grammars. *Journal of Systems and Software*, *12*(1), 43–53.

Kulkarni, S., & Cavazos, J. (2012). Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices*, *47*(10), 147–162.

Leather, H., Bonilla, E., & O'Boyle, M. (2014). Automatic feature generation for machine learning-based optimising compilation. *ACM Transactions on Architecture and Code Optimization*, *11*(1), article 14.

Stephenson, M., Amarasinghe, S., Martin, M., & O'Reilly, U. M. (2003). Meta optimization: Improving compiler heuristics with machine learning. *ACM SIGPLAN Notices*, *38*(5), 77–90.

Stock, K., Pouchet, L. N., & Sadayappan, P. (2012). Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization*, *8*(4), article 50.

Xu, H., Wang, Y., Fan, S., Xie, P., & Liu, A. (2020). DSmith: Compiler fuzzing through generative deep learning model with attention. In *Proceeding of the International Joint Conference on Neural Networks* (pp. 1–9).