International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 06 Issue: 07 | July 2019 www.irjet.net p-ISSN: 2395-0072

Machine Learning Techniques for Code Optimization

Tejas Upadhya, Shivam Raj and Suyash Pathak

Department of CSE, Ramaiah Institute of Technology, Bangalore, India

---

Abstract — Researchers have been trying to use machine-learning-based approaches since the mid-1990s to solve a number of various compiler optimization issues. These techniques primarily Machine Learning Techniques For Code Optimization enhance the quality of the obtained results and, more importantly, make it possible to address two main compiler optimization problems: optimization selection (choosing which optimizations to apply) and phase ordering (choosing the order to apply optimizations). Because of advancing applications, increasing number of compiler optimizations, and new target architectures, the compiler optimization space continues to grow. Generic optimization in compilers cannot fully leverage newly introduced optimizations and therefore cannot keep up with the pace of increasing options. The survey highlights the approaches taken so far, the obtained results, the fine-grain classification among different approaches, and finally, the influential papers of the field.

Keywords — Code Optimization, Autotuning, supervised learning, unsupervised learning, Linear SVM, Random Forest, Decision Tree

I. INTRODUCTION

Compilers have been the backbone of computer science for quite some time now [1,2].These compilers generate a binary executable code which can be easily be understood by the machine which uses an LLVM (Low Level Virtual Machine) nowadays which uses some clever trickery to compile the code. LLVM provides optimizaztion features and these optimizations are implemented as passes by the developers, they do this so as to transform the code into an optimized version of the app, but the LLVM acts only on the intermediate representation layer which is one of the three layers in the compilation process. The three layers are, front-end, intermediate representation, and the backend. Optimising code manually is a very tedious task. Here we talk about many automatic methods for optimising code. Optimisation of the intermediate layer with the help of an LLVM contributes to a vital role in performance.

Translation and optimization are the two major roles of compilers. The code received from high level languages is translated into an intermediate package which is further translated by the compiler into binary. Their second role is to optimise the code i.e find the best translation of the code possible. There are many code translation whose code translations are syntactically and logically correct but their performance would all vary drastically from each other. The majority of the studies or research involved in compiler optimisation is based on the aspect of performance. This aspect has been misnamed optimization because in the maximum number of cases, till lately finding the most effective translation became dismissed as too hard to discover and an unrealistic undertaking. The target was to develop compiler heuristic rules to change the code to improve performance.

Optimization as an area has been studied since the 1800's. There are two fundamental reasons why the two fields have taken so long to converge. First is the advent of moore's law wherein the capacity of transistors doubled every year giving us twice the computational power every year, however the software wasn't able to bridge the gap. Secondly the computer architecture evolved so quick that every new generation has some new features for which then the developers and compiler developers have to take time to develop code and rather clever heuristics to utilize the hardware. Rather than relying on experts to do this repeatedly after a new architecture comes out. So instead of this we can train machine learning models that will learn how to optimise a compiler to make the code to run more efficiently. ML in this scenario is best tailored for code optimisation where the effect of performance is platform dependent.

Machine Learning is about building systems that can learn from data. Learning means getting better at some task, given some performance measure. The ability of machine learning systems to predict based on previous information fed to it can be used to find the data point with the best outcome that is the closest to the point of optimization. It's in these kind of scenarios we find machine learning applicable to our problem. .

II. SUPERVISED LEARNING

Supervised Learning is a type of machine learning in which a model is generated over a labelled training data [5, 6]. The machine is given labelled objects as training data from which it learns and predicts labels for unlabelled data..

A. Linear Models and SVMs

Support vector machine is a kind of supervised learning algorithm which finds a hyperplane with highest margin in a a n-dimensional space where n is the number of features.The hyperplane classifies the data point. [4] proposes to use SVM to train models which atutotunes the JIT compiler of IBM Testarossa and then builds a compilation plan. They have used scalar features to construct feature vectors and using SVM learning to experimentally test the quality of their model using single-iteration and 10-iteration scenarios on SPECjvm 98 benchmark suite

B. Decision Trees and Random Forests

Decision Tree is machine learning algorithm which could be used for both classification as well as regression problems. It graphically represents a tree which has all the possible solutions.A decision can be made based on conditions using decision trees. Random Forest is a ensemble bagging algorithm to get higher accuracy in predictions. It chooses decision trees randomly and gives the label which has most votes. This kind of learning model reduces variance among individual trees.

[7] proposes to use supervised learning to compress the code. It uses IR structure of the compiler code and deduce a decision tree which separates IR code into a stream that compresses more efficiently.

III. UNSUPERVISED LEARNING

Unsupervised learning is used to group data in categories without any labels or classes explicitly being defined. This leads to test sample being unclassified and thus there is no incorrect or correct method to evaluate a solution[8,6]. Unsupervised learning is highly based on density estimation in statistics [9], but also consists of many methods focused on understanding and explaining the essential features such as evolutionary algorithm. Unsupervised learning has no goal or target but provides an area for model evaluation. An environment can check a given model based on parameters like the input value passed to the model function.

A. Clustering Methods

In unsupervised learning the most commonly used technique is clustering. This method helps in the optimization of the compiler by making clusters that are homogenous to each other and reduce sample size. Therefore scalars or nested loops should work with the same sequence.

Another equally important perspective on clustering is to reduce the size of the compiler, which can be reduced to hundreds of orders.

[10] the paper discusses a method to drastically reduce the time taken for training of the machine learning model based on auto-tuning. They use highly specific clustering techniques [11], after reducing the dimensions. They score the approach utilized based on an EEMBCv2 bench mark suite and showed a highly reduced training time by a factor of seven by the proposed method the paper.

Martins et al.[12, 13] addressed the issue of phase-ordering through a clustering-based method of selecting similar functions. The paper proposes to utilize programming elements which ate encoded by the DNA sequence and uses a the distance matrix which is calculated and according to this information constructs a tree for the optimization.

Finally, the optimized compiler is added to the area where exploration speedup versus good and popularly known algorithms like Genetic algorithms is made and evaluated.

B. Evolutionary Algorithms

The premise of an evolutionary algorithm (to be further known as an **EA**) is quite simple and revolves around the process of natural selection and mutation. Candidate solutions in the optimization space are considered as individuals of a

population. Quality of solutions is evaluated based on some fitness functions. Population evolution occurs after repeated fitness function [6] has been applied. Some of the most notable techniques are briefly mentioned here..

The genetic algorithm is utilized in understanding and grasping the different compiler optimization problems that are based on Darwin's theory of natural selection which talks about the processes that are the driving forces for evolution. One of the Genetic Algorithm heuristic is NSGA-II (Non-dominated Sorting Genetic Algorithm II) [14], a popularly known method for many multi-objective optimization problems and have several applications mostly in the computer architecture domain [15, 16]. The computational complexity of classic GA algorithms is increased by NSGA-II.

Neuroevolution of Augmenting Topologies (NEAT)[17] is another interesting evolutionary model. The paper claims it to be a stronger model for learning complex problems because network topology and the weight of the parameters can be changed to generate a best-balanced fitness function.

Cooper et al.[18, 19] used genetic algorithms to address the code size issue of the generated binaries. The results were compared with an iterative algorithm which generated a sequence of fixed optimization at random frequency as well. Based on the comparison, the paper concluded that a new fixed optimization sequences using their GAs that reduces the binary code size.

Agakov et al.[20] adapted several models to accelerate the exploration of an iterative piliation space. He utilized many different approaches like using identically distributed independent distribution and many more such techniques were exploited. iterative compilation, helped to gain significant speed when tested on unseen applications using these two models. A nearest-neighbour classifier is used to predict the best optimizations given an unseen application. Firstly, the tested model which had the lowest score by distance in the vector space feature is identified. Therefore, through the different models it understands and grasps the concept and probability distribution which needs to be utilized for providing the best optimization for neighbouring applications. The learned probability distribution is the used for the new application as the calculated best possible distribution. The methods utilized are exceedingly well optimized for calculating a uniform distribution of probabilities using RIC methodology.

Kulkarni et al. suggests two different approaches to achieve both a better optimization technique selection[21] and ordering of different phases[22]. The correct compiler parameters are selected using NEAT and other tuning features which are static in nature to use the Hotspot compiler with specified benchmark suites.The paper proposes to utilize the NEAT for training the model and the called decision tree. This paper advices to use an intermediate speed-up prediction method that used static features of the current state of the application being studied to query the model and induce the current best optimization to be used when addressing the phase-ordering issue. Thus, iteratively, an application-based sequence of compilers is formed.

Classification		**Reference**
Supervised Learning	**Linear Model/SVMs**	**[23,24,25,26,27,29,50,51]**
	Decision Trees/Random Forests	**[30,31,7,32,33,34,21,35,36,37,38,39]**
Unsupervised Learning	**Clustering**	**[24,12,13,40,10,52]**
	Evolutionary Algorithms	**[20,41,42,43,44,45,18,19,46,47,48,49,21,22,35,13,40]**

CONCLUSION

In the coming years with the advent of machine learning, AI and intelligent systems, these technologies are being increasingly applied to areas that involved a sizeable amount of human intervention especially in the field of high performance computing, in compiler design as described in the paper it has been increasingly used in code optimization problems as described above, other areas include auto-parallelization, security, and energy efficiency. Architecture that supports large-scale parallelism can be exploited using compiler optimizations. Machine learning in recent times is has become more powerful due to the advent of deep learning techniques to construct different heuristics. The machine

learning models allow these intelligent systems perform with minimal input from the developer. In this survey we have taken a comprehensive look at the research done in this area of optimising compilers by showing/comparing the different types of ml algorithms and their performance in code optimisation in the table above.We hope this survey will be resourceful to researchers,developers and help them create something new, creative and open up new avenues of research..

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. Compilers, Principles, Techniques. Addison Wesley.

[2] M. Hall, D. Padua, and K. Pingali. 2009. Compiler research: The next 50 years. Commun. ACM (2009). Retrieved from http://dl.acm.org/citation.cfm?id=1461946.

[3] Bas Aarts, Michel Barreteau, François Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John Gurd, Jan Hoogerbrugge, Ping Hu et al. 1997. OCEANS: Optimizing compilers for embedded applications. In Proceedings of the European Conference on Parallel Processing (Euro-Par'97). 1351–1356.

[4] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. 2011. Using machinestolearnmethod-specificcompilationstrategies.InProceedingsofthe9thAnnualIEEE/ACMInternationalSymposium on Code Generation and Optimization. 257–266. Retrieved from http://dl.acm.org/citation.cfm?id=2190072.

[5]ThomasG.Dietterich.2000.Ensemblemethodsinmachinelearning.InInternationalWorkshoponMultipleClassifier Systems. Springer, 1–15.

[6] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2012. Foundations of Machine Learning. MIT Press.

[7] Christopher W. Fraser. 1999. Automatic inference of models for statistical code compression. ACM SIGPLAN Notices 34, 5 (May 1999), 242–246. DOI:http://dx.doi.org/10.1145/301631.301672

[8] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. Unsupervised learning. In The Elements of Statistical Learning. Springer, 485–585.

[9] Bernard W. Silverman. 1986. Density estimation for statistics and data analysis. Vol. 26. CRC press.

[10]J.Thomson,M.O'Boyle,G.Fursin,and B.Franke.2009.Reducingtrainingtimeinaone-shotmachinelearning-based compiler. Proceedings of the International Workshop on Languages and Compilers for Parallel Computing. 399–407. Retrieved from http://link.springer.com/10.1007.

[11]R.Babuka,P.J.VanderVeen,andU.Kaymak.2002.ImprovedcovarianceestimationforGustafson-Kesselclustering. In Proceedings of the 2002 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE'02), Vol. 2. IEEE, 1081–1085.

[12] L. G. A. Martins and R. Nobre. 2016. Clustering-based selection for the exploration of compiler optimization sequences. ACM Trans. Architect. Code Optim. 13, 1 (2016), 28. Retrieved from http://dl.acm.org/citation.cfm?id= 2883614.

[13] Luiz G. A. Martins, Ricardo Nobre, Alexandre C. B. Delbem, Eduardo Marques, and João M. P. Cardoso. 2014. Exploration of compiler optimization sequences using clustering-based selection. InACMSIGPLANNotices, Vol. 49. ACM, 63–72

[14] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. 6, 2 (2002), 182–197.

[15] Giovanni Mariani, Aleksandar Brankovic, Gianluca Palermo, Jovana Jovic, Vittorio Zaccaria, and Cristina Silvano. 2010. A correlation-based design space exploration methodology for multi-processor systems-on-chip. In Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC'10). IEEE, 120–125.

[16] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio,RobertoZafalon,PrabhatAvasare,GeertVanmeerbeeckетal.2011.Multicube:Multi-objectivedesignspace exploration of multi-core architectures. In Proceedings of the VLSI 2010 Annual Symposium. Springer, 47–63.

[17] Kenneth O. Stanley. 2002. Efficient reinforcement learning through evolving neural network topologies. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02). Citeseer.

[18] K. D. Cooper, P. J. Schielke, and D. Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. ACM SIGPLAN Notices. Retrieved from http://dl.acm.org/citation.cfm?id=314414.

[19] K. D. Cooper, D. Subramanian, and L. Torczon. 2002. Adaptive optimizing compilers for the 21st Century. J. Supercomput. Retrieved from http://link.springer.com/article/10.1023/A:1015729001611.

[20] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K. I. Williams. 2006. Using machine learning to focus iterative optimization. In Proceedings of the International Symposium on Code Generation and Optimization. IEEE, 295–305.

[21] S. Kulkarni and J. Cavazos. 2013. Automatic construction of inlining heuristics using machine learning. Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13). 1–12. Retrieved from http://ieeexplore.ieee.org/xpls/abs.

[22] S.KulkarniandJ.Cavazos.2012.Mitigatingthecompileroptimizationphase-orderingproblemusingmachinelearning. ACM SIGPLAN Notices (2012). Retrieved from http://dl.acm.org/citation.cfm?id=2384628.

[23] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. 2016. Predictive modeling methodology for compiler phase-ordering. In Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms (PARMA-DITAM'16). ACM, New York, NY, 7–12. DOI:http://dx.doi.org/10.1145/2872421.2872424

[24] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017.MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning.ACM Trans. Archit. Code Optim. 14, 3, Article 29 (Sept. 2017). DOI:http://dx.doi.org/10.1145/3124452

[25] Biagio Cosenza, Juan J. Durillo, Stefano Ermon, and Ben Juurlink. 2017. Stencil autotuning with ordinal regression:Extended abstract. In Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES'17). ACM, New York, NY, 72–75. DOI:http://dx.doi.org/10.1145/3078659.3078664

[26] E. Park, J. Cavazos, and L. N. Pouchet. 2013. Predictive modeling in a polyhedral optimization space. International J. Parallel Program. 41, 5 (2013), 704–750. Retrieved from http://link.springer.com/article/10.1007/s10766-013-0241-1.

[27] E. Park, S. Kulkarni, and J. Cavazos. 2011. An evaluation of different modeling techniques for iterative compilation.In Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'11). ACM, 65–74. Retrieved from http://dl.acm.org/citation.cfm?id=2038711.

[28] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. 2011. Using machines to learn method-specific compilation strategies. In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. 257–266. Retrieved from http://dl.acm.org/citation.cfm?id=2190072.

[29] M. Stephenson and S. Amarasinghe. 2005. Predicting unroll factors using supervised classification. In Proceedings of the International Symposium on Code Generation and Optimization. DOI:http://dx.doi.org/10.1109/CGO.2005.29

[30] Bruno Bodin, Luigi Nardi, M. Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Lujan et al. 2016. Integrating algorithmic parameters into benchmarking and space exploration in 3D scene understanding. In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation. ACM, 57–69.

[31] Y. Ding, J. Ansel, and K. Veeramachaneni. 2015. Autotuning algorithmic choice for input sensitivity. ACM SIGPLAN Notices 50, 6 (2015), 379–390. Retrieved from http://dl.acm.org/citation.cfm?id=2737969.

[32] G. Fursin and A. Cohen. 2007. Building a practical iterative interactive compiler. Workshop Proceedings. Retrieved from https://www.researchgate.net/profile/Chuck.

[33] G. Fursin, Y. Kashnikov, and A. W. Memon. 2011. Milepost GCC: Machine learning enabled self-tuning compiler. Int.J. Parallel Program. 39, 3 (2011), 296–327. Retrieved from http://link.springer.com/article/10.1007/s10766-010-0161-2.

[34] G. Fursin, C. Miranda, and O. Temam. 2008. MILEPOST GCC: Machine learning based research compiler. Proceedings of the GCC Summit. Retrieved from https://hal.inria.fr/inria-00294704/.

[35] P. Lokuciejewski and F. Gedikli. 2009. Automatic WCET reduction by machine learning based heuristics for function inlining. Proceedings of the 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART'09). 1–15. Retrieved from https://www.researchgate.net/profile/Peter.

[36] L. Luo, Y. Chen, C. Wu, S. Long, and G. Fursin. 2014. Finding representative sets of optimizations for adaptive multiversioning applications. arXiv preprint arXiv:1407.4075 (2014). Retrieved from http://arxiv.org/abs/1407.4075.

[37] A. Monsifrot, F. Bodin, and R. Quiniou. 2002. A machine learning approach to automatic production of compiler heuristics. Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, and Applications (2002), 41–50. Retrieved from http://link.springer.com/chapter/10.1007/3-540-46148-5.

[38] Eunjung Park, Christos Kartsaklis, and John Cavazos. 2014. HERCULES: Strong patterns towards more intelligen predictive modeling. Proceedings of the 43rd International Conference on Parallel Processing. 172–181. DOI:http://dx.doi.org/10.1109/ICPP.2014.26

[39] K. Vaswani. 2007. Microarchitecture sensitive empirical models for compiler optimizations. International Symposium on Code Generation and Optimization (CGO'07) (2007), 131–143. Retrieved from http://ieeexplore.ieee.org/xpls/abs.

[40] S. Purini and L. Jain. 2013. Finding good optimization sequences covering program space. ACM Trans. Architect.Code Optim. 9, 4 (2013), 56. Retrieved from http://dl.acm.org/citation.cfm?id=2400715.

[41] L. Almagor and K. D. Cooper. 2004. Finding effective compilation sequences. ACM SIGPLAN Notices 39, 7 (2004),231–239. Retrieved from http://www.anc.ed.ac.uk/machine-learning/colo/repository/LCTES04.pdf.

[42] J. Cavazos, C. Dubach, and F. Agakov. 2006. Automatic performance model construction for the fast software exploration of new hardware designs. In Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems. 24–34. Retrieved from http://dl.acm.org/citation.cfm?id=1176765.

[43] J. Cavazos and M. F. P. O'Boyle. 2005. Automatic tuning of inlining heuristics. Proceedings of the ACM/IEEE SC 2005 Conference on Supercomputing. 14–14. Retrieved from http://ieeexplore.ieee.org/xpls/abs.

[44]KatherineE.Coons,BehnamRobatmili,MatthewE.Taylor,BertrandA.Maher,DougBurger,andKathrynS.McKinley.2008.Featureselectionandpolicyoptimizationfordistributedinstructionplacementusingreinforcementlearning. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. ACM, 32–42.

[45] K. D. Cooper, A. Grosul, and T. J. Harvey. 2005. ACME: Adaptive compilation made efficient. In ACM SIGPLAN Notices 40, 7 (2005), 69–77. Retrieved from http://dl.acm.org/citation.cfm?id=1065921.

[46] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. 2017. End-to-end deep learning of optimization heuristics. In Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT'17). 219–232. DOI:http://dx.doi.org/10.1109/PACT.2017.24

[47] C. Dubach, J. Cavazos, and B. Franke. 2007. Fast compiler optimisation evaluation using code-feature based performance prediction. In Proceedings of the 4th International Conference on Computing Frontiers. 131–142. Retrieved from http://dl.acm.org/citation.cfm?id=1242553.

[48] Thomas L. Falch and Anne C. Elster. 2015. Machine learning based auto-tuning for enhanced opencl performance portability. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW'15). IEEE, 1231–1240.

[49] Unai Garciarena and Roberto Santana. 2016. Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion (GECCO'16). ACM, New York, NY, 1159–1166. DOI:http://dx.doi.org/10.1145/2908961.2931696

[50] Liu, H., Zhao, R., Wang, Q. and Li, Y., 2018. ALIC: A low overhead compiler optimization prediction model. *Wireless Personal Communications*, *103*(1), pp.809-829.

[51] Wang, Z. and O'Boyle, M., 2018. Machine learning in compiler optimization. *Proceedings of the IEEE*, (99), pp.1-23.

[52] Zhang, Z., Liu, L., Shen, F., Shen, H.T. and Shao, L., 2018. Binary multi-view clustering. *IEEE transactions on pattern analysis and machine intelligence*.