International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792

Compiler Optimization using Machine Learning Techniques

Amogh S. Inamdar[1], Sindhuja V. Rai[2], Anagha M. Rajeev[3], Sini Anna Alex[4]
[1,2,3]*Student, Department of CSE, Ramaiah Institute of Technology, Bengaluru, India*
[4]*Assistant Professor, Department of CSE, Ramaiah Institute of Technology, Bengaluru, India*

***Abstract*: Compilation optimization is very essential in order to increase the running speed of programs as well as minimize the object file size. Machine learning algorithms are used to select the best compiler options that result in such improvements. Compiler auto-tuning refers to the process of optimizing the performance of the code during the intermediate code-generation phase of compilation. This paper deals with machine learning based compilation optimization on feature processing, compiler auto-tuning and compiler optimization techniques such as loop nest optimizations and automatic generation of optimization heuristics for a target processor by machine learning. It then explores the concept of evolving iterative compilers which attempts a large number of optimization strategies and choosing the best one. It proposes an approach of selecting compiler transformations – namely probabilistic optimization. Using this approach, we can achieve significant performance improvements.**

***Keywords*: Compiler heuristics, loop nests, feature mining, compiler auto-tuning, Intermediate Code Generation (ICG), k-Nearest Neighbors (kNN), Principal Component Analysis (PCA)**

1. Introduction

Optimization occurs without changing the high-level source code itself, only its compiled representation. One of the major problems with manual optimization is that it can be architecture-dependent, thereby reducing the portability of the compiled target and requiring multiple optimizations to be specified, in accordance to the computer architectures used. Problems in compiler optimization are many-fold, and include issues such as selecting the best steps to be taken for optimization, dealing with phase-dependencies of these steps, i.e. the need for correct ordering of the steps. Selecting each optimization step cannot be done in a vacuum- each step chosen has an influence on other steps, and the selection of the correct subset of optimization options available greatly influences performance. Feature processing is of great significance for most machine learning techniques and also provides ways to acquire spatial information within a program and convert it into features that are required by the different techniques.

Loop optimization techniques are used to transform loop nests and improve the performance of the code on a target architecture, including exposing parallelism. Finding and evaluating an optimal, semantic-preserving sequence of transformations is a complex problem. The sequence is guided using heuristics and/or analytical models and there is no way of knowing how close it gets to optimal performance or if there is any headroom for improvement. The automatic generation of optimization heuristics for a target processor by machine learning is addressed and the potential of this method on an always legal and simple transformation loop unrolling is evaluated. Iterative compilation is used to narrow the gap between compiled code and hand-written code by attempting a large number of different optimization strategies, and choosing the best. This approach can easily be transferred to other, or even yet to be invented, processors and extract high levels of performance unachievable by traditional techniques with no additional native compiler effort.

Autotuning involves defining a ‘tuning parameter’ and optimizing the system (in this case, a compiler at the ICG step) to optimize this parameter. Machine learning can be used to select relevant features and sequences of action, to plan and execute the passes made for code optimization. It is used to compare and optimize the various solutions in the parameter space via feature analysis and dimensionality reduction is usually performed to select only the best features. In this paper, we explain the aforementioned methods of compiler optimization and describe these techniques in detail. The rest of the paper is arranged as follows: An overview of the methods, a modest description of different machine learning techniques, a description of related work and references to the researched work.

2. Overview

A. Feature mining and generation

GCC provides four different optimization levels, namely, O0, O1, O2 and O3, for proper selection of the best algorithms. In feature processing, there are different methods used to find the best optimization plan [1]-[3]. Iterative search looks for global optimal solutions but, due to the wide gap between research and practical use, it is not the most commonly used approach. Heuristic algorithms search for near optimal solutions but, just like iterative searches, they are not suitable for practical purposes as they take a tremendous amount of time in finding the solutions [4]. Machine learning algorithms, on the other hand, predicts the near optimal solutions in real-time manner making it an effective approach for compiler

optimization. Feature design is one of the most important aspects of Machine Learning. Program features are generally defined before learning and are divided into static and dynamic features.

Spatial information, which basically shows how instructions are distributed within a program, is also very significant for compilation optimizations. Predictive modeling techniques rely on picking the best features to characterize an optimization space. Machine learning techniques are widely used to build predictive models that select the best compiler options [5]. The performance of the machine learning techniques is highly dependent on the quality of the features selected. The spatial information is stored in a Data Flow Graph (DFG) which is a directed graph with each node representing an instruction and each edge representing the data dependency between the instructions.

B. Autotuning

Compiler auto-tuning is a complex problem that can be modeled as a linear system of many parameters. For better performing target code, there usually have to be trade-offs between different types of optimization such as loop unrolling and resource allocation. Optimization of this auto-tuning is usually done by making multiple passes over the target code, optimizing a certain aspect of that code and trying to achieve better performance little by little. The operations performed during these passes are usually specified as predefined steps by the compiler designer. Systems such as GCC have specified optimization levels that perform certain sequences of transformations that usually lead to better performance on certain benchmarks. However, the word 'optimization' may be a misnomer here, as these transformations do not necessarily lead to better performing target code and can even degrade performance [1]. A 'tuning parameter' is defined and the system is optimized in autotuning (in this case, a compiler at the intermediate code generation step) to optimize the parameter. A parameter space is generated, consisting of multiple solutions, and the feasible ones are compared and the best one returned. Machine learning can be used to select relevant features and sequences of action, to plan and execute the passes made for code optimization and to compare and optimize the various solutions in the parameter space via feature analysis and dimensionality reduction. A common algorithm to perform this is Principal Component Analysis (PCA) [14]

C. Loop nest optimization

Loop optimization techniques are used to transform loop nests and improve the performance of the code on a target architecture, including exposing parallelism. Finding and evaluating an optimal, semantic-preserving sequence of transformations is a complex problem. The sequence is guided using heuristics and/or analytical models and there is no way of knowing how close it gets to optimal performance or if there is any headroom for improvement. In this work, state-of-the art code optimizers are considered and then Machine Learning algorithms are used to make predictions for better, yet clearly achievable performance for the loop nests using these code optimizers. Recognizing the inherent behavior of loop nests using hardware performance counters and Machine Learning algorithms presents an automated mechanism for compiler writers to identify where to focus on making improvements in order to achieve better performance.[8]

D. Iterative Compilation

Iterative compilation is used to narrow the gap between compiled code and hand-written code by attempting a large number of different optimization strategies, and choosing the best. The implication of this work is that built-in compiler heuristics which select optimization strategies are not doing as good a job as is possible. This paper describes a new approach to selecting compiler transformations – probabilistic optimization.[10] It explains how stochastic methods can be used to select the high-level transformations, directed by execution time feedback, where optimization space coverage is traded off against searching in known good regions. This approach can easily be transferred to other, or even yet to be invented, processors and extract high levels of performance unachievable by traditional techniques with no additional native compiler effort.

3. Machine learning techniques

A. Supervised learning

Supervised learning or "learning by example" involves techniques where the learning model is first trained on a dataset of examples, i.e. problems that have already been classified or solved. The aim is to generalize the data model learnt to the entire space of applications that the model will receive for optimization. This method is used for classification, ranking, and regression problems [13]. A popular type of supervised learning uses linear models, such as Linear Regression (LR), Support Vector Machine (SVM) [15], and k-Nearest Neighbors (kNN) that can handle fluctuations in input data. SVM can also perform classification and regression on non-linear data using the kernel trick. Some systems use Bayesian Networks as classifiers, and it has been shown that these can lead to optimization superior to the GCC's –O2 and –O3 in embedded processors [16]. Decision trees and ensemble methods such as random forests have been used for code compression [17] and optimization [18].

B. Unsupervised learning

Unsupervised learning or "learning by observation" is a class of techniques that focuses on identifying common features or patterns in the given data samples. A major type of unsupervised learning is clustering, which seeks to partition the input data space into similar clusters. It has been used to reduce training time of autotuning with machine learning [19]. Dimensionality reduction, as mentioned previously, is a method that is used to reduce the number of characteristics or features

of the data, which can drastically speed up and simplify the optimization process. Evolutionary algorithms, inspired by natural genetic models, are a powerful tool for autotuning. Algorithms such Non-dominating Sorting Genetic Algorithm-II (NSGA-II) [20] and Neuroevolution of Augmenting Topologies (NeAT) [21] have been used to tackle optimization and phase ordering problems.

C. Reinforcement learning

Reinforcement Learning (RL) aims to implement a reward/penalty system that enables learning to obtain the optimum value of either, inspired by psychology. It has been used for building an instruction scheduler that can outperform the commercial Compaq scheduler on evaluated benchmarks from SPEC95 suite [22]. NeAT has also been used for reinforcement learning to find optimal instruction placements where it outperformed all other state-of-the art methods [23].

4. Related work

A. Feature mining- design

Machine learning based optimization has a training part and a prediction part. In the training period, the learning model is trained on large datasets and the parameters are calculated. In the prediction period, the programs are compiled and the features are extracted from the compiled programs and these feature vectors are taken as input to the model thereby predicting a good optimization plan. Due to the fact that it is hard to define static features that have a positive effectiveness, a template based static feature is proposed which generates features having contrasting quantities and qualities.

Most of the feature analysis done so far only gathers the features once, at the start of the optimization, but, on the other hand, a better approach is to make use of the entire feature collection which enhances the quality of learning and also the precision of the prediction. A multi-phase feature extraction prototype is designed that extracts features before 10 passes known as milestone pass. In the training period, the program features are to be extracted before the execution of each milestone pass. The prediction period is similar where the compiler analyzes the code structure and dynamically executes the optimization plan.

K Nearest Neighbors (KNN), Support Vector Machines (SVM), Logistic Regression, etc. are various machine learning models that deal with prediction periods that are similar to traditional machine learning problems.

B. Spatial based- use of machine learning in compilers

The state of the compiler is first examined before choosing an optimization that improves the performance of a program. Important data such as data flow graphs, control flow graphs and abstract syntax trees are collected at the Single Static Assignment (SSA) level [5] [6]. The data is then reassigned to a set of features used by the tools of machine learning whose quality is dependent on the quality of these features. Benchmark applications are used to generate a training set. The applications are compiled many times, each time with different compiler options, discovering the best options for a given application by running the newly compiled program. Each tuple in the training set contains the feature vectors as well as the best compiler options for a given program. The training set is then given to the machine learning tool in order to build a model that predicts the best compiler options for new applications.

C. Spatial information framework

There are four classes of program features that are used to select the best compiler options: i) code size-based features, ii) hot instruction-based features, iii) parallelism-based features, iv) memory access-based features. These features can be considered as indicators of a program's performance. Behavior of the cache is controlled by code-based features as well as memory access patterns, execution time of a program is controlled by hot instruction-based features and optimizations like loop unrolling, instruction scheduling, etc. are controlled by features based on parallelism. All such information can be collected at the SSA level of a compiler. A histogram, which is a bar graph, is used to capture the spatial information, like the weightage of each element in a dataset in a DFG [7]. The IBM Milepost GCC is used to implement this framework.

D. Loop nest optimization

In this work, 4 candidate code optimizers were considered for conducting the experiments which included Polly [11], a Polyhedral Model based optimizer for LLVM. 2 out of those 4 optimizers could perform auto-parallelization of the loop nests. For the experiments, two Intel architectures were used. For the auto-parallelization related experiments, only one thread is mapped per core. Two interesting correlations among hardware performance counters and the characteristic behavior of the loop nests were discovered -the hardware performance counters values from Kaby Lake architecture (after disabling loop transformations and vector code generation) were sufficient to get well trained ML model to make predictions for a similar architecture like the Skylake architecture, and, for predicting the most suited candidate for serial code and for the auto-parallelized code for a loop nest, the same set of hardware performance counters, collected from profiling a serial version, can be used to train the ML model and achieve satisfactory results.

For training and evaluating the Machine Learning model, Orange [12] was used. Random Forest (RF) was used as the classifier for all the experiments. The trained models were evaluated on Accuracy and Area Under Curve (AUC). The predicted optimizer's execution time as compared to that of the most suited optimizer's execution time was the same in case of correct predictions and higher in case of mispredictions. The ML experiments were repeated thrice in order to validate our results, taking into account the unique instances from the three validation datasets for measurements.

E. Automatic production of compiler heuristics using machine learning

Machine learning techniques offer an automatic, flexible and adaptive framework for dealing with the many parameters involved in deciding the effectiveness of program optimizations.[9] Classically a decision rule is learnt from feature vectors describing positive and negative applications of the transformation. To summarize the approach, the steps involved in using a machine learning technique for building heuristics for program transformation are:

1. Finding a loop abstraction that captures the "performance" features involved in an optimization, in order to build the learning set,
2. Choosing an automatic learning process to compute a rule in order to decide whether loop unrolling should be applied,
3. Setting up the result of the learning process as heuristics for the compiler.

F. Application characterization

Application Characterization techniques seek to identify the behavior of the target application in its environment, and characterize it to be able to optimize it accordingly. This is done by selecting relevant features of the application. This analysis can be static or dynamic. Static analysis reads the code as-is, and obtains features from the source code (or compiler representations) [25] or structures such as Control Flow Graphs (CFGs) [26]. Dynamic analysis involved characterizing an application according to its runtime behavior, and may be architecture dependent or architecture independent. The methodology involves collecting Performance Counters (PCs) that indicate program performance and choke points for data, and using these to characterize the application. A hybrid of the two types of characterization can also be used, as developed in the HERCULES system [24]. Dimensionality reduction techniques are usually performed on the extracted features, to reduce optimization complexity.

G. Evolving iterative compilation

This describes a probabilistic search algorithm for finding good source level transformation sequences for typical embedded programs written in C. Two competing search strategies provide a good balance between optimization space exploration and focused search in the neighborhood of already identified good candidates. The work integrates both parameter-less global and parameterized local transformations in a unified optimization framework that can efficiently operate on a huge optimization space spanned by more than 80 transformations. The evaluation of this optimization toolkit, based on three real embedded architectures and kernels and applications from the UTDSP benchmark suite, successfully demonstrates that the approach is able to outperform any other existing approach and gives an average speed up of 1.71 across platforms.

But there is a major drawback to this technique – the substantial amount of compile and evaluation time required to achieve the results. The main reason for this is that the optimization of each program is carried out individually, starting afresh each time. If there was a way to automatically gauge the similarity between programs, this technique should be primed with previously acquired information – to learn from experience – which could dramatically speed up search, and improve the results.

5. Conclusion

This paper presents an overview on compiler optimization using machine learning techniques.

References

[1] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan, "Finding effective optimization phase sequences," in ACM SIGPLAN Notices, vol. 38, no. 7, ACM. New Orleans, LA: ACM, 2003, pp.12–23.

[2] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in Code Generation and Optimization, 2009. CGO 2009. International Symposium on, IEEE. Seattle, WA: IEEE, 2009, pp. 81–91.

[3] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in Code Generation and Optimization, 2005. CGO 2005. International Symposium on, IEEE. San Jose, California: IEEE, 2005, pp. 123–134.

[4] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," ACM SIGPLAN Notices, vol. 34, no. 7, pp. 1–9, 1999.

[5] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle. MILEPOST GCC: machine learning based research compiler. In Proceedings of the GCC Developers' Summit, Ottawa, Canada, June 2008.

[6] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 2006.

[7]] A. Papadopoulos and Y. Manolopoulos. Structure based similarity search with graph histograms. In Proceedings of the 10th International Workshop on Database and Expert Systems Applications 1999.

[8] Aniket Shivam, Neftali Watkinson, Alexandru Nicolau, David Padua, Alexander V. Veidenbaum, Towards an Achievable Performance for the Loop Nests, arXiv:1902.00603v1 [cs.PF] February 2019.

[9] Antoine Monsifrot, Fran¸cois Bodin, and Ren´e Quiniou, Machine Learning Approach to Automatic Production of Compiler Heuristics, IRISA-University of Rennes France.

[10] John D. Thomson, Doctor of Philosophy, Using Machine Learning to Automate Compiler Optimisation, Institute of Computing Systems Architecture, School of Informatics University of Edinburgh, 2008.

[11] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation. Parallel Processing Letters, 22(04), 2012.

[12] J. Dem˘sar and et. al. Orange: data mining toolbox in python. The Journal of Machine Learning Research, 14(1):2349–2353, 2013.

[13] Amir H. Ashouri, William Killian , John Cavazos , Gianluca Palermo , Cristina Silvano, A Survey on Compiler Autotuning using Machine Learning, ACM Computing Surveys (CSUR), v.51 n.5, p.1-42, January 2019

[14] Amir Hossein Ashouri. 2016. Compiler Autotuning Using Machine Learning Techniques. Ph.D. Dissertation. Politecnico di Milano, Italy. http://hdl.handle.net/10589/129561.

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

[16] A.H. Ashouri, G. Mariani, G. Palermo, and C. Silvano. 2014. A Bayesian network approach for compiler auto-tuning for embedded processors. (2014), 90–97.
[17] Christopher W. Fraser. 1999. Automatic inference of models for statistical code compression. ACM SIGPLAN Notices 34, 5 (may 1999), 242–246.
[18] A Monsifrot, F Bodin, and R Quiniou. 2002. A machine learning approach to automatic production of compiler heuristics. International Conference on Arti_cial Intelligence: Methodology, Systems, and Applications (2002), 41–50. http://link.springer.com/chapter/10.1007/3-540-46148-5
[19] J Thomson, M O'Boyle, G Fursin, and B Franke. 2009. Reducing training time in a one-shot machine learning based compiler. International Workshop on Languages and Compilers for Parallel Computing (2009), 399–407. http://link.springer.com/10.1007
[20] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio, Roberto Zafalon, Prabhat Avasare, Geert Vanmeerbeeck, and others. 2011. Multicube: Multi-objective design space exploration of multi-core architectures. In VLSI 2010 Annual Symposium. Springer, 47–63.
[21] S Kulkarni and J Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. ACM SIGPLAN Notices (2012). http://dl.acm.org/citation.cfm?id=2384628.
[22] Amy McGovern, Eliot Moss, and Andrew G Barto. 1999. Scheduling straight-line code using reinforcement learning and rollouts. Tech report No-99-23 (1999).
[23] Katherine E Coons, Behnam Robatmili, Matthew E Taylor, Bertrand A Maher, Doug Burger, and Kathryn S McKinley. 2008. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 32–42.
[24] Eunjung Park, Christos Kartsaklis, and John Cavazos. 2014. HERCULES: Strong Patterns towards More Intelligent Predictive Modeling. 2014 43rd International Conference on Parallel Processing (2014), 172–181. DOI: http://dx.doi.org/10.1109/ICPP.2014.26
[25] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. 2006. Using machine learning to focus iterative optimization. In Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society, 295–305.
[26] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP MFP O'Boyle. 2009. Towards a holistic approach to auto-parallelization: integrating pro_le-driven parallelism detection and machine-learning based mapping. ACM Sigplan Notices (2009), 177–187.