Static Performance Prediction of Compiler Optimizations

Michael Dennis

***Abstract*—Tunning the interaction of various compiler optimizations to be optimal for a specific platform is a daunting task with complicated interacting considerations. The problem is exacerbated by the fact that what is optimal on one architecture likely performs terribly on another. Since manually tuning a compiler for each new architecture is infeasible, this project will use machine learning to create a model specific to each architecture that will accurately predict the runtime performance of code based on static features to better inform compiler optimizations. Once the predictive model is created, it can be quickly used to determine near optimal settings for compiler tunning parameters specific to each block of code. To give this research direction, we will begin our work with stencil code, moving on to dynamic programing after preliminary results. These classes of algorithms provides more structure to the problem by focusing on programs whose performance is minimally input dependent allowing our methods to achieve more accurate results.**

I. Introduction

COMPILER optimization is a complex subject. The quality of any individual optimization often depends on fine details of the machine that vary widely from platform to platform. What could be a powerful optimization on one machine could slow down performance on another and it is not uncommon for a compiler to try a very complex series of optimizations only to find that performance has not changed or even decreased. Tuning parameters of compilers for the wide variety of systems by hand is a problem beyond the scale of practicality.

To handle the complexity of choosing the best configuration of optimizations many have used iterative compilation assisted by machine learning techniques to tune parameters to a specific block of code. Running an iterative machine learning algorithm with recompilation vastly slows down the process and ultimately results in less time for other optimizations to be fine tuned. Additionally, an iterative technique often finds local mixima, which offer less than optimal performance gains. More significantly, iterative compilation techniques fail to utilize the structure of the optimization itself, learning how the optimization effects the performance of this program but not programs in general. This project will move the iterative and expensive machine learning to compiler-tuning time by creating a predictive model of the performance of applications on the host architecture to use in later optimizations. The hope of this model is to learn in detail about how the machine performs and to use this model to predict how optimizations will effect the performance of yet unseen programs. Once this model is created for a particular architecture and reused for every subsequent compilation resulting in more efficient compilation and a more informed search for optimizations. This paper will focus on the creation of the predictive model on the host architecture.

To give this work direction we will compare models created from strictly stencil codes to models created with a larger set of codes that allows for more flexible iteration patterns and optional expressions at each level. These two sets will use the same sets of features, and will be compared by the predictive capabilities of their models. Since the second type of code is a superset of stencil code, it will necessarily show at least as much error as the stencil code predictive model. Seeing the extent of the decrease in accuracy as the code complexity increases will demonstrate the challenges of performance prediction from static features as well as the necessity for more quality code features to be used to counteract the decrease in performance.

II. Technique

A. Code Structure

We have limited the scope of our work to two specific code structures for purpose of comparison. The first code structure is simply stencil codes, a very well studied, well structured type of program. The other code structure is a slight generalization of stencil codes, adding the ability for each level to have memory accesses and calculations and allowing loops to iterate in more complex configurations. This code structure allows any loop structure that can be mode from a constant step, a lower bound starting at the outer loop's index variable, and a constant number of iterations. Since this second type of code structure is significantly more complicated, it will be possible to observe how predictive power decreases as complexity increases.

B. Feature Selection

In order to create the predictive model it is important to select quality static features. In our work we have found that it is best to use features that are not calculated, rather observed. Since metrics such as working set size and memory reuse can be calculated from simple, observable static features such as loop factors, memory accesses, and distribution statistics describing the location of memory accesses, giving the model the strictly observable static features gives the feature vector at least as much discriminatory power as if we used to calculated features, but it does not bias the model to use known forms of performance prediction and it limits the chances of multiple features being closely related by their calculations.

For raw features we have chosen the following: Number of Floating Point Adds, Subtracts, Multiplications and divisions, Number of memory accesses, the minimum and maximum and average distance between consecutive memory accesses, Number of iterations of the loop and how much each iteration changes the centroid of higher loop levels. Each of these features are repeated for each level of the loop. For the purposes of these initial experiments we have limited ourselves to 5 nested loops with the understanding that this can later be extended to more loops for a production model. However, though the model in this form is limited to structures with a finite number of loops, this limitation will not decrease the applicability of the model as any optimization whose feature changes are within this constant number of loops will be fully described.

C. Feature Validation

From feature selection we proceeded to determine the descriptive power of our features. Since we are grouping programs by a finite set of features, many programs are going to share the same feature set, and thus one would naturally expect there to be more variance in measurements of different programs with the same features than there would be in different measurements of the same program. To calculate this difference we selected 100 vectors at random and generated 30 stencils for each vector. These stencils where tested each 30 times shuffled with the tests for other stencils with the same features to amortize performance inconsistencies on the host platform. We use this data to calculate a mean percent error as well as a variance of sample means of percent errors for each vector. By the central limit theorem we can then treat each of these measurements for each vectors as a normal distribution and compute the probability density function of percent error as a mixture of these normal distributions. Again sampling from this set we can obtain a mean percent error across all vectors and a confidence interval for this percent error. Following this same methodology we can compute the average error in measurement for a specific code structure and feature set.

We followed the above process on both generated stencil codes and generated codes following the previously mentioned broader structure. The results suggest that, with this feature vector, stencil codes could be predicted with an average error of 5.529277% (sd 0.247) and the more general structure could be with an average error of 12.09777% (sd 0.460). These are both compared to an average measurement error of 2.78825 % (sd 0.416). The fact that stencil codes are predicted better than the more broad category speaks to greater variance in running times of the broader class do to increased complexity. While this feature vector seems adequate to predict the performance of stencil codes it is not well suited for the more complex category. To counteract this decrease in accuracy it is necessary to increase the predictive power of the feature vector in order to add features that can discriminate between the best and worst performers within a specified vector.

D. Model Training

Using the identified features of the code and knowing they give a good basis for performance prediction we, collected data uniformly randomly inside the space of codes with this structure (subject to time and space limitations). A code generator was created that was able to generate code with specific features and test it on the host system. Each such test will represent a data point, giving a correlation between this list of features and performance. After collecting this data for some time a predictive model was generated using Multivariate Adaptive Regression Splines(MARS) [2]. Choosing MARS as our regression algorithm is important as it allows for a piecewise predictive model, which is representative of computing performance where sudden discontinuities can occur when predictive parameters hit certain values such as cache size or bandwidth of some hardware component. Additionally, MARS produces a regression model as a mathematical function that is continuous and differentiable which will be important for later mathematical analysis. The predictive model that has been generated is ready to be integrated into the compiler as a description of performance for the host architecture and will not change in future compilations.

E. Model Utilization

During the compilation process, the compiler will consider the optimizations that it can make, and generate equations representing the space of possible optimizations and how they effect the aspects of the code used by our static predictive model. An equation will be given to the predictive model for each variable that it relies upon. These equations will contain variables that are meaningful tuning parameters to the compiler, but to the model they are simply values that are used to calculate important features of the code. For example, if a compiler does standard blocking of matrix code, it could determine an equation for working set size based on the blocking coefficients. These coefficients will be only variables to the predictive model, but, to the compiler, an assignment of these variables represents a combination of optimizations. The job of the performance model at runtime is to find the assignment to these tuning parameters which maximizes performance on the host system. By using MARS we have created a predictive model that is differentiable, and by restricting the compiler to send polynomial equations, which are both typical and expressive, we have a set of equations to calculate predictive parameters that are also differentiable. Since both the predictive model and the constraint equations are differentiable we can use Lagrange Multipliers to find a finite set of possible local maxima to test, making the process of finding the best assignment of the tunning parameters at compile time faster and more accurate than methods of iterative improvement. Importantly, this method will allow us to make ensure that we make the best choice of parameters that our model can predict and will avoid being stuck at a local maximum.

III. Related Work

In the field of static performance prediction much has been attempted to mixed results. There have long been techniques for predicting cache miss rates [7], however, since these results were achieved, much has changed in computer architectures, and even with an accurate prediction of miss rates, the results cannot necessarily be generalized to an accurate prediction of performance. Others have tried to predict the performance directly, though their methods were only partly static, requiring expensive profiling at compile-time and were only able to achieve average error of around 20% [1]. Yet others have narrowed their focus to scientific applications and were able to create models that had good performance prediction across multiple architectures [4]. However, their methods were still dependent on dynamic analysis. A large variety of machine learning algorithms have been used at compile time with different heuristics to tune compiler optimizations with a 3 fold improvement from unoptimized code, but requiring 10 minute iterative compilation process [6]. This compile time overhead has been avoided by others, instead using machine learning off line to tune the search heuristics of the compiler, with positive, but less significant results, only achieving a 2 fold speedup over unoptimized code on standard benchmarks [5].

IV. Future Work

Continued improvements will come in two forms. First, the current models could be made more precise. The error of our model is larger than the average error from the mean measured from samples within each of our vectors. This observation leads us to conclude that our model does not predict as well as our feature vector will allow, and thus there is room for improvement in the creation of the model. To realize this potential improvement is a matter of improving our model creation process. By changing machine learning parameters, changing code generation patterns to be more representative of real code, and choosing to get more data from specifically misclassified sections of the model, we can work to improve the accuracy of the model for a specified feature set. Fine tunning the initial suit of tests is essential to creating an accurate model. If irregularities are not found initially, they could be missed by later steps as well. To insure an initially good set of tests we can employ the standard Roof Line model [3] to generate initial areas of interest for the first round of data collection. This model is already used to great effect in performance analysis, and though it is not meant to guarantee the degree of precision we desire, it will provide a baseline that is close and will only improve after the tests have been performed.

To achieve improvements past the mean measured standard error of a specific feature set would require a change to the features them selfs. This option has the possibility of greatly improving the performance of the model, giving it the ability to distinguish different utilization patterns of the host machine that were previously indistinguishable. However, it is important that the features that were added are both useful, they provide new information, and raw, directly observable from the source code.

The second form of improvement is to begin to utilize the created models for actual optimization prediction. After the previously mentioned improvements produce a model capable of accurately predicting performance of a specific type of code we can imagine an optimization that transforms the code by keeps it within the set if codes capable of being predicted by the performance model. Given a mathematical description of the parameterized optimizations in the form of equations describing how changing these parameters changes the features of the code, it is possible to use the trained mathematical model to find the predicted optimum choice of parameters without ever running the program. If we accept as given that the model is accurately predicting changes in performance, such a system has the possibility of giving better and more reliable performance gains than current methods. These methods could be further improved by supporting more optimizations or supporting compositions and permutations of supported optimizations.

V. Conclusion

Choosing the best or even choosing a good set of optimizations for an arbitrary program on an arbitrary architecture is a problem so difficult that standard manual analysis techniques prove impractical. Common methods for dealing with this complexity, such as iterative compilation, relearn with each new program certain key facts about the structure of not only the optimization but how the machine performs. It is our hope that creating a mathematical performance model at compiler-tune time will avoid this relearning process, creating a single reliable model that can be trained when time is less important.

It is clear from our initial work that creating mathematical models for significantly complex sets of codes will require larger sets of quality features. However, it's also clear that for well structured code and with enough features, the creation of mathematical models with moderate accuracy is possible. Continuing to improve the process of creating the prediction models, and increasing their accuracy has promise to greatly improve compiler optimizations. Utilizing the completeness of the mathematical description of the machine, it may be possible to provide greater confidence in the improvement of the chosen optimizations. Work will continues to expand model creation to be accurate for more types of codes, adding new features when necessary to counteract increases in complexity. As these results are improved, this work will allow compilers to utilize the structure of the machine and optimizations to perform more effective searches that may lead to quicker compilation of more efficient programs.

REFERENCES

[1] Calin Cascaval, Luiz De Rose, David A. Padua, and Daniel A. Reed. Compile-time based performance prediction. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 365–379, London, UK, UK, 2000. Springer-Verlag.

[2] Jerome H Friedman. Multivariate adaptive regression splines. *The annals of statistics*, pages 1–67, 1991.

[3] YuJung Lo, Samuel Williams, Brian Van Straalen, TerryJ. Ligocki, MatthewJ. Cordery, NicholasJ. Wright, MaryW. Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science*, pages 129–148. Springer International Publishing, 2015.

[4] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 2–13, New York, NY, USA, 2004. ACM.

[5] Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. In *Software Automatic Tuning*, pages 335–351. Springer, 2010.

[6] Keith Seymour, Haihang You, and Jack Dongarra. A comparison of search heuristics for empirical code optimization. In *Cluster Computing, 2008 IEEE International Conference on*, pages 421–429. IEEE, 2008.

[7] Y. Zhong, S.G. Dropsho, Xipeng Shen, A. Studer, and Chen Ding. Miss rate prediction across program inputs and cache configurations. *Computers, IEEE Transactions on*, 56(3):328–343, March 2007.