Computation Pattern Classification for Compiler Optimization

Nathan Harmon*, Qing Yi† and Jugal Kalita‡
University Of Colorado
at Colorado Springs
Colorado Springs, Colorado 80918
Email: *nharmon@uccs.edu, †qyi@uccs.edu, ‡jkalita@uccs.edu

Abstract—**The first step towards automated code optimization is program comprehension. Develops can manually identify computation patterns, but it consumes more than half of their time [2]. Once the compiler knows the computation model of a program, optimizations can be applied that are likely to perform well. We present an extensible machine learning approach using dynamic features to determine the computation patterns with 94% accuracy.**

I. Introduction

Different computation models require different types of optimizations. Stencil code requires vastly different optimizations then Sparse Matrix code. In the pursuit of automated optimization, extracting computation patterns such as categories of code (Stencil, Sparse Matrix, etc.) is the first step. Comprehension of the code can be done by the developer, but consumes more than half of the developers time [2]. The introduction of the human element can also introduce additional errors from incorrectly tagged code. Once the type of computation is identified to a compiler, it can select optimizations that will likely be effective. Code comprehension is the first of many steps towards automated optimization of code.

Machine learning has been used to solve many difficult problems. Before machine learning can be used to address a problem, the problem must first be constructed in a way that machine learning is suited to solve, such as classification. To exploit the power of machine learning, we formulated the problem as a classification problem. We construct a finite number of classes representing different computation patterns and an Open Set for all other computation patterns. For the case study, we focus on separating various types of Stencil code. The features were extracted at runtime, as a time series. The uses of dynamic features has not been explored in previous work. We then use a Dynamic Recurrent Neural Network (LSTM) to classify different types of stencils. We must use dynamic networks since the length of the features extracted from the programs varies for each program. We will show that this approach has promise for future work in program compression.

II. Related Work

Machine learning has been used to improve compiler optimization. Much of the past work is focused on heuristics to improve the selection of optimization. Heuristics created by hand or typically designed to operate on a particular platform and do not generalize to other platforms. Machine learning allows the heretics to be optimized for each platform, so it selecting better optimizations for that platform.

Monsifrot, et al. [5] used decision trees and boosting to create new compiler heuristics for loop unrolling. The features were the number of statements, number of arithmetic operations, minimum number of iterations, number of array accesses, the amount of array element reuses and the number of if statements. Using the features they constructed a tree with the probability of loop unrolling speeding up the computation. The decision tree was used to replace the existing heuristic.

Stephenson et al. [6] used genetic programming to learn priority functions – cost function describing the efficacy of a heuristic. The priority function must be learned for each application. The genetic algorithm creates a priority function that selects the heuristic that will choose the best optimization for the application. They show that new heuristic can be created from the result of the genetic algorithm.

Cavazos, et al. [1] also used machine learning (logistic regression) to select the best optimizations at compile time. Rather than using static features, as most others works did, they used performance counters as inputs to the model. They determined that the performance counters gave more information on the behavior of a program then static code features.

Fursin et al. [3] created a machine learning based compiler. Which reduced the number of iterative compilation, needed to select good optimizations for unknown platforms. They used static program features (variables, types, instructions, basic blocks, temporary variables, etc.) along with runtime behavior to train various machine learning models.

III. Feature Selection

Many past approaches used static features including variable names, comments, and documentation. Variable names and comments contain meaningful information, yet they depend on the programmer. They do not generalize well across programs, reducing their usefulness as features. In contrast, we collect runtime features and only characteristics that are present in all programs. The goal of the feature is to encapsulate the inherent pattern in the computation. We do this by focusing on the memory reference in the program. For each memory reference, we create a feature containing six elements. The items are: read or write, data type, dependents, modification, scope, and relative distance from last access (arrays only).

A. Read or Write

Knowing whether the reference is being read or written to, is essential for understanding what role a reference plays in the computation. If a memory reference is being read, it indicates that the value will be used to calculate a value yet to be seen. Whereas a write is the result of computation.

B. Data type

Secondly, the data type. The data type limits how a memory reference is used, which in turn limits how the reference is used in the computation. Often structures such as loops can be identified by examing the alternation is the data types.

C. Dependents

Variable dependents lends information about the computation while remaining general across different types of computation.

D. Modification

We define modification as any arithmetic operation (+, -, /, *). The modification directly captures the computation that is occurring.

E. Scope

The scope is crucial since it affects what can be accessed. The representation of the scope is only the changes in the scope, not a particular block. Each time the scope changes a boolean is negated, creating an alteration of 0 and 1. The change in scope generalizes better across different programs, then indicating a particular block.

F. Relative Distance

If the reference was to an array, then the relative distance from the last access to that array is included in the feature. The relative offset captures the array access pattern.

IV. Feature Creation

The programs are instrumented to output the above features at runtime. The test programs for this paper are input independent. If the program is input dependent, then it should be run multiple time with different inputs to collect a variety of possible outputs. The features that are output by the programs are time series data. Each feature by itself includes limited information about the computation, but the sequence of features encapsulates essential characteristics of the computation.

V. Data Set

The data set consists of 9565 unique generated stencil codes. The stencil code contains 1-point through 11-point stencils. Each of the stencils was instrumented to output the features. Then run and the features were collected.

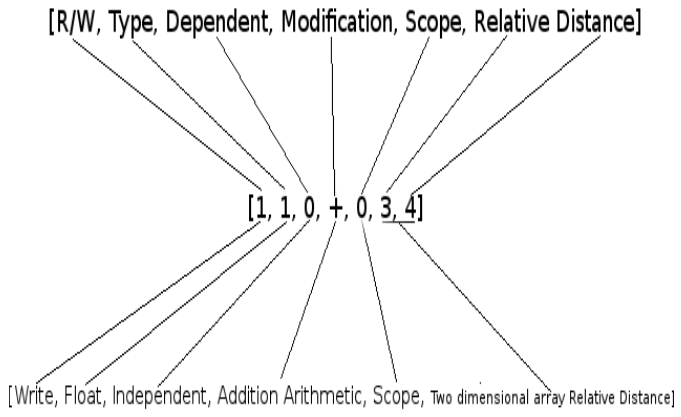
[R/W, Type, Dependent, Modification, Scope, Relative Distance]
[1, 1, 0, +, 0, 3, 4]
[Write, Float, Independent, Addition Arithmetic, Scope, Two dimensional array Relative Distance]

Fig. 1. Example Feature

VI. Program Classification

To take advantage of the power of machine learning, we constructed a classification problem. There are an infinite number of possible computation patterns, so the classes will never cover all possibilities. With that said it is feasible to cover all possibilities that have known optimizations. In this case study, we will separate stencil code into classes. The classes will represent the number of points in each stencil. There will also be an Open Set for stencil code that we do not have a class for. Since we have a good sampling of the types of stencil code that belong in the Open Set, we did not do any additional work in the Open Set filed.

VII. Neural Network

Since the features make variable length time series, we use a standard implementation of a Dynamic Recurrent Neural Network (LSTM)[4]. We use a dynamic version of the networks since the number of features from each program varies. The numbers that represent each element in are features do not hold meaning, so we found that we obtained better results when we used learned embeddings of the features. The features are input to the network using one-hot encoding, and the learned embedding is looked up for the feature. The current implementation requires a minimum of 25 features in each series and tunicates the series if more than 100 features exist. The network has six outputs; the first five outputs are for a 1-point stencil, 8-point stencil, 9-point stencil, 10-point stencil and 11 point stencil. These are the most common types of stencil code in the test set. The network has also been evaluated with different configurations of output classes with

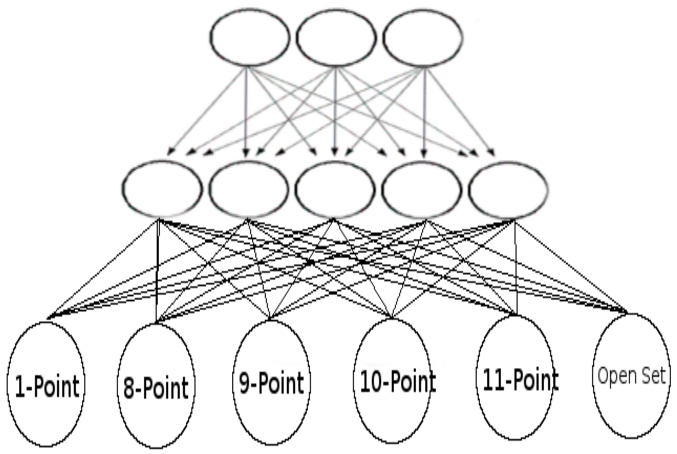

Fig. 2. Network Outputs

similar results. The last class is an Open Set for all other types of stencils.

TABLE I
NETWORK RESULTS

Learned Embeddings	Accuracy
No	74%
Yes	94%

VIII. EVALUATION

We evaluated the network using tenfold validation across the data set. Without the learned embeddings, the accuracy of the network was 74%. Next, we tested the network using learned embeddings which obtaining 94% accuracy. Changing which classes of stencil code we classified and which belonged to the Open Set did not change the accuracy. This result shows promise of this being a valid approach to identifying computation patterns in code. For this case study, we had good training data for the types of computation that belonged in the Open Set. In an actual application, we would not have good examples of what belongs in the Open Set. When randomly generated features were input into to the network to simulate unseen data, which should be classified into the Open Set, we saw poor accuracy. This is expected since there was no designed to ensure the unseen data would be classified as Open Set.

IX. FUTURE WORK

Additional effort will be put into classifying unseen computation patterns as Open Set. This will make this approach valid for use in the selection of optimization. More evaluation will be done with additional types of computation outside of stencil code.

REFERENCES

[1] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197. IEEE, 2007.

[2] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[3] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[4] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[5] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50. Springer, 2002.

[6] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices*, volume 38, pages 77–90. ACM, 2003.