**Available Online at www.ijcsmc.com**

International Journal of Computer Science and Mobile Computing

**A Monthly Journal of Computer Science and Information Technology**

ISSN 2320–088X

*IJCSMC, Vol. 3, Issue. 5, May 2014, pg.557 – 561*

**RESEARCH ARTICLE**

# Code Optimization in Compilers using ANN

**Jay Patel[1], Prof. Mahesh Panchal[2]**

[12]Computer Science & Engineering, Kalol Institute of Technology and Research Centre, Kalol, Gujarat, India

[1] jaypatel8853@gmail.com; [2] mkhpanchal@gmail.com

***Abstract***— ***Today's compilers have a plethora of optimizations to choose from, and the correct choice of optimizations can have a significant impact. Furthermore, choosing the correct order in which to apply those optimizations has been a long standing problem in compilation research. Traditional compilers typically apply the same set of optimization in a fixed order to all functions in a program, without regard the code being optimized. Understanding the interactions of optimizations is very important in determining a good solution to the phase ordering problem. We will develop a system that automatically selects good optimization orderings on a per method basis within a dynamic compiler by including profiles of programs.***

***Keywords***— ***Compilers; Code Optimization; Artificial Neural Networks; Feature set; Profiles***

## I. Introduction

In computing, an optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program [1]. The most common requirement is to minimize the time taken to execute a program a less common one is to minimize the amount of memory occupied. The growth of portable computers has created a market for minimizing the power consumed by a program. Compiler optimization is generally implemented using a sequence of optimizing transformations, algorithms which take a program and transform it to produce a semantically equivalent output program that uses fewer resources. It has been shown that some code optimization problems are np-complete, or even undecidable. In practice, factors such as the programmer's willingness to wait for the compiler to complete its task place upper limits on the optimizations that a compiler implement or might provide. Optimization is generally a very CPU- and memory-intensive process. In the past, computer memory limitations were also a major factor in limiting which optimizations could be performed. Because of all these factors, optimization rarely produces optimal output in any sense, and in fact an optimization may impede performance in some cases; rather, they are heuristic methods for improving resource usage in typical programs.

## II. Code Optimization techniques

There are basically two times of code optimization, Machine independent and machine dependent. Here machine independent means optimization regardless of compiler and processor and machine dependent optimization we must consider some important attributes of compiler and processor e.g. cycles per element.

**Code Motion**
Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop. An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and evaluates the expression before the loop .' Note that the notion "before the loop" assumes the existence of an entry for the loop, that is, one basic block to which all jumps from outside the loop go.

**Reduction in strength**
The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as strength reduction. But induction variables not only allow us sometimes to perform a strength reduction; often it is possible to eliminate all but one of a group of induction variables whose values remain in lock step as we go around the loop.
**Common Sub expression Elimination**
An occurrence of an expression E is called a common sub expression if E was previously computed and the values of the variables in E have not changed since the previous computation. We avoid recounting E if we can use its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim.

**Loop Unrolling**
In region-based scheduling, the boundary of loop iteration is a barrier to code motion. Operations from one iteration cannot overlap with those from another. One simple but highly effective technique to mitigate this problem is to unroll the loop a small number of times before code scheduling.

**Instruction Level Parallelism**
To implement instruction level parallelism we must need a pipelining architecture for calculations.in this two multiplies within loop should not dependent on each other.in parallel execution it needs multiple registers to hold sums/products.it needs it needs 6 usable registers and 8 fp registers. When enough registers are not available we must spill temporaries onto stack

## III. ARTIFICIAL NEURAL NETWORKS

Artificial neural networks are computational models inspired by animal central nervous systems (in particular the brain) that are capable of machine learning and pattern recognition [2]. They are usually presented as systems of interconnected "neurons" that can compute values from inputs by feeding information through the network. For example, in a neural network for handwriting recognition, a set of input neurons may be activated by the pixels of an input image representing a letter or digit. The activations of these neurons are then passed on, weighted and transformed by some function determined by the network's designer, to other neurons, etc., until finally an output neuron is activated that determines which character was read.

Mainly three types of ANN models are present single layer feed forward network, Multilayer feed forward network and recurrent network. Single layer feed forward network consist of only one input layer and one output layer. Input layer neurons receive the input signals and output layer receives output signals. Multilayer feed forward network consist of input, output and one more addition than single layer feed forward is hidden layer. Computational units of hidden layer are called hidden neurons. In Multilayer Feed Forward Network there must be only one input layer and one output layer and hidden layers can be of any numbers. There is only one difference in recurrent network from feed forward networks is that there is at least one feedback loop. In neurons we can input vectors taken as input and weights are included. With the help of weights and input vectors we can calculate weighted sum and taking weighted sum as parameter we can calculate activation function.

## IV. LITERATURE REVIEW

**A. Phase Ordering of optimization techniques.**
Authors used a technique that selects the best ordering of optimizations for individual portions of the program, rather than applying the same fixed set of optimizations for the whole program[3] . It develops a new method-specific technique that automatically selects the predicted best ordering of optimizations for different methods of a program.

**B. Phase ordering with genetic algorithm**
Two different experiments done using GAs. The first experiment consisted of finding the best optimization sequence across our benchmarks [4]. Thus, we evaluated each optimization sequence (i.e., chromosome) by compiling all our benchmarks with each sequence. We recorded their execution times and calculated their speedup by normalizing their running times with the running time observed by compiling the benchmarks at the O3 level. That is, we used average speedup of our

benchmarks (normalized to opt level O3) as our fitness function for each chromosome. This result corresponds to the "Best Overall Sequence". The purpose of this experiment was to discover the optimization ordering that worked best on average for all our benchmarks. The second experiment consisted of finding the best optimization ordering for each benchmark. Here, the fitness function for each chromosome was the speedup of that optimization sequence over O3 for one specific benchmark. This result corresponds to the "Best Sequence per Benchmark". This represents the performance that we can get by customizing an optimization ordering for each benchmark individually.

## C. Motivation

Predict the current best optimization: This method would use a model to predict the best single optimization (from a given set of optimizations) that should be applied based on the characteristics of code in its present state. Once an optimization is applied, we would re-evaluate characteristics of the code and again predict the best optimization to apply given this new state of the code. For this we can apply Artificial Neural Network in this method and we will also include profiles for better prediction of optimization sequence for particular program.

## D. Automatic Feature generation

Automatic Feature generation system is comprised of the following components: training data generation, feature search and machine learning [5]. The training data generation process extracts the compiler's intermediate representation of the program plus the optimal values for the heuristic we wish to learn.

Once these data have been generated, the feature search component explores features over the compiler's intermediate representation (IR) and provides the corresponding feature values to the machine learning tool. The machine learning tool computes how good the feature is at predicting the best heuristic value in combination with the other features in the base feature set (which is initially empty). The search component finds the best such feature and, once it can no longer improve upon it, adds that feature to the base feature set and repeats. In this way, we build up a gradually improving set of features.

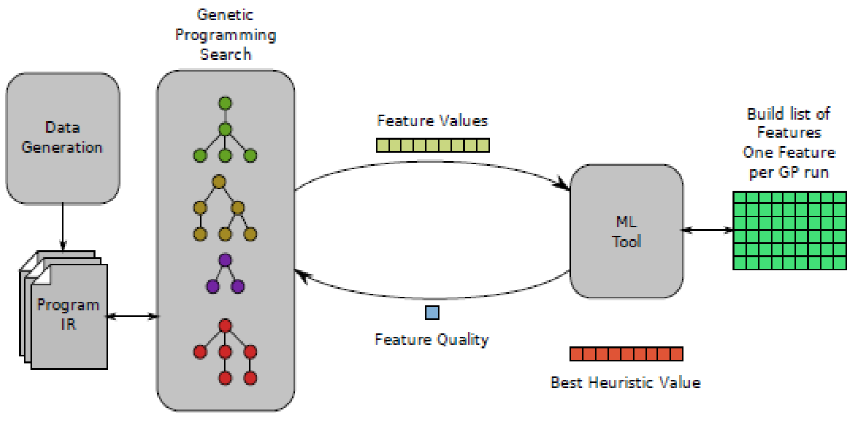

Figure 1 Automatic Feature generation System

### a. Data Generation

In a similar way to the existing machine learning techniques (see section II) we must gather a number of examples of inputs to the heuristic and find out what the optimal answer should be for those examples. Each program is compiled in different ways, each with a different heuristic value. We time the execution of the compiled programs to find out which heuristic value is best for each program. We also extract from the compiler the internal data structures which describe the programs. Due to the intrinsic variability of the execution times on the target architecture, we run each compiled program several times to reduce susceptibility to noise.

### b. Feature Search

The feature search component maintains a population of feature expressions. The expressions come from a family described by a grammar derived automatically from the compiler's IR. Evaluating a feature on a program generates a single real number; the collection of those numbers over all programs forms a vector of feature values which are later used by the machine learning tool.

### c. Machine Learning

The machine learning tool is the part of the system that provides feedback to the search component about how good a feature is. As mentioned above, the system maintains a list of good base features. It repeatedly searches for the best next feature to add to the base features, iteratively building up the list of good features. The final output of the system will be the latest features list.

Our system additionally implements parsimony. Genetic programming can quickly generate very long feature expressions. If two features have the same quality we prefer the shorter one. This selection pressure prevents expressions becoming needlessly long.

### E. Motivation

They have developed a new technique to automatically generate good features for machine learning based optimizing compilation. By automatically deriving a feature grammar from the internal representation of the compiler, we can search a feature space using genetic programming. We have applied this generic technique to automatically learn good features.

### F. Optimization Performed

The register allocator has been modified to use frequencies to compute the expected cost for choosing proper register classes and for computing priorities for the allocation itself [13]. The experimental runs have shown that the current register allocator correctly incorporates the new information and works considerably better than with the original set of heuristics, especially on register starved architectures. The reg-stack pass has been enhanced to optimize the common paths of code at the expense of uncommon paths. This decreased random peaks in the benchmark results, but did not bring as large improvements as the previous change. Code alignment decisions are now based on profile information, avoiding useless alignment of infrequently executed regions, e.g. loops that iterate only a few times.

## V. PROPOSED METHOD

For ordering of different optimization techniques using ANN we must need to implement that in 4Cast-XL [14] as it is a dynamic compiler. 4Cast-XL constructs an ANN, Integrate the ANN into Jikes RVM's optimization driver than Evaluate ANN at the task of phase-ordering optimizations [6]. For each method dynamically compiled, repeat the following two steps

i. Generate a feature vector of current method's state [15]
ii. Generate profiles of program[9]
iii. Use ANN to predict the best optimization to apply

Use ANN to predict the best optimization to apply. Run benchmarks and obtain feedback for 4Cast-XL Record execution time for each benchmark optimized using the ANN. Obtain speedup by normalizing each benchmark's running time to running time using default optimization heuristic.

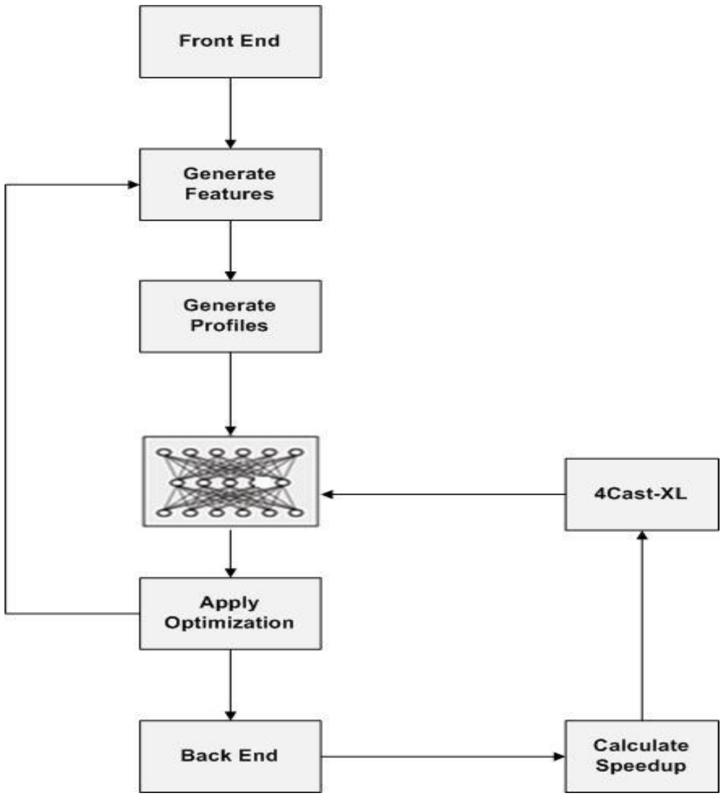

Figure 2 Steps to implement Method of code optimization using ANN

## VI. Conclusion and Future Work

Research work is aimed for optimizing code using artificial neural networks. In order to make this precise, better profiles generated from given set of features using Milepost GCC compiler with ten different programs. Experimental results demonstrate that profiles of program can be used for optimization of code. For further work different features can also be used. Static compiler can be used to get different profiles and through that optimization in static compiler can be performed.

## References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compilers Principles, Techniques & Tools*, Pearson Publication

[2] S. Rajasekaran, G. A. Vijayalakshmi Pai, *Neural Networks, Fuzzy Logic and Genetic Algorithm: Synthesis and Application*, PHI learning Pvt Ltd

[3] Sameer Kulkarni, John Cavazos, *Mitigating the Compiler Optimization Phase Ordering Problem Using Machine Learning*, 14th April 2012

[4] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, Todd Waterman, *Finding Effective Compilation Sequences*, 11-13 June 2004

[5] Hugh Leather, Edwin Bonilla, Michael O'Boyle, *Automatic Feature Generation for Machine Learning Based Optimizing Compilation*

[6] K. O. Stanley and R. Miikkulainen. *Efficient reinforcement learning through evolving neural network topologies*. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002), page 9, San Francisco, 2002. Morgan Kaufmann

[7] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[8] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," pp. 41–50, 2002.

[9] Benchmark. *Specjvm 98*, URL http://www.spec.org/jvm98/.

[10] Benchmark. *Java grande benchmarks*, http://www2.epcc.ed.ac.uk/computing/research activities/java grande/sequential.html.

[11] Jan Hubicka, Richard Henderson, Andreas jaeger, *Infrastructure for Profile Driven optimization*, 29 August 2001

[12] Maggie Johnson, *Code Optimization*, 4 August 2008

[13] Code Profiling in Linux http://www.linuxforu.com/2011/06/code-profiling-in-linux-using-gprof/

[14] Tool. 4Cast-XL http://www.xlpert.com/software-4Cast-XL.html

[15] Milepost GCC Feature Extractor http://ctuning.org/wiki/index.php/Special:CPredict?request=extract_features