Compiler Construction Optimization Techniques

Dhruvil Shah
Institue of Technology
Nirma University
19BCE248

Jurin Vachhani
Institue of Technology
Nirma University
19BCE286

***Abstract*—Without altering the program's meaning, a compiler converts code written in one language to another. Additionally, it is assumed that a compiler will make the target code effective and space and time optimal. Basic error detection and translation methods are part of the compilation process. The accuracy of these assessments is heavily constrained by complicated target architectures and unforeseen optimization interactions, which results in performance loss as a result of suboptimal optimization choices. There are hundreds of optimizations available in contemporary compilers, and this number will only increase in the future. The fact that each optimization would attempt to target particular code constructs and boost their efficiency by implementing particular templates is the main reason for the increase in the number of optimizations that the compiler could perform. A growing number of optimizations that are being applied to a certain code may not have been intended for the code that is now being compiled as the optimizations target increasingly specific code constructs. The iterative compilation strategy, which encourages investigating several optimization alternatives and choosing the optimal one a posteriori, can help prevent this performance loss. The newest optimization techniques can be used by compiler suites to produce code 20–30% faster than typical compilers, which has a variety of advantages for developers. Without having to rely on faster and more expensive processors, improved optimization would also enable developers to reduce costs and improve products. In this paper, we discuss various methods for improving compiler optimization, such as caching techniques, dynamic optimization methods, and compiler optimization for dynamic languages like JavaScript. Additionally covered are machine learning approaches that can be used to improve compiler optimization.**

***Index Terms*—compiler optimization, dynamic, machine learning, cache.**

I. Introduction

In today's fast-paced world everything is expected to be available faster to people as well as machines. Considering domain like compiler which plays an important role in compiling the code in any language in any form. Without compilation we can't make our process available to computer because they are the one's which are going to run user based code in from Low-level language to High-level. The paper is broadly divided into topics like Cache Optimization, Dynamic Language and Machine Learning techniques. Cache can be very helpful at times because here we store frequent accessed memory by compiler. If we assign some registers and other basic memory units we can improve the performance to a great extent. Dynamic optimization consist of parallelization of multiple component of code in a simultaneous manner. Some of the examples can be taken from languages like JavaScript. In the present time we have Multi-core CPU/GPU which makes these thing possible. Machine learning technique basically make use of ensemble models which consist of both supervised as well as unsupervised learning such as Support Vector, K-Means clustering and Decision Tree algorithm.

II. Related Work

Kyle Dewey et al. demonstrated how to parallelize a JavaScript abstract interpreter. Because of JavaScript's complicated semantics and dynamic behaviour, static analysis of JavaScript is difficult. The paper presents STS, an alternative to the traditional DFA approach, for program analysis [1]. Zheng Wang et al. described how to optimise compilers using ensemble machine learning models. The ensemble model was a combination of supervised and unsupervised machine learning techniques. The proposed ensemble model included algorithms like Decision Trees, Support Vector Machines, and K-means clustering. The ensemble model was tested and found to produce accurate and efficient results [2]. Jay Patel et al. propose an artificial neural network-based method for code optimization in compilers. A feature vector of the current method's state is generated for each compiled method, along with program profiles. The best optimization is then predicted using Artificial Neural Networks. Instead of using the same optimization for the entire program, the prediction assists in determining the best ordering of the optimizations. The phases are then reordered using genetic algorithms. An automatic feature generation model with training data generation, feature search, and machine learning components is also used [3].

III. Approach Using Cache Optimization

These approach basically aims at creating a framework for improving register reuse in regular loop computations This was accomplished through the use of operation commutativity and associativity. Optimization frameworks can be used to improve performance in a subset of critical computations known as stencils. The study also demonstrated how register reuse can be used to reduce the number of loads/stores by optimising stencil operations.

Major goals is to reduce Miss Rate. McFarling in 1998 reported 50% reduction in cache misses when memory was

increased to 2KB while 75% when size increased to 8KB. These was based on the concept of increasing the resource. But these things doesn't work always as it not feasible to be cost effective. Another technique is to reorder the instruction to reduce conflict misses.

Major Techniques involved in reducing miss rate are as follow:

1) Merging Arrays:
   As the name suggest we merge two arrays which were explicitly declared before into a single one. If we talk in terms of compiler application we have Key-Value pair format for various use cases like semantic , syntactic analysis.

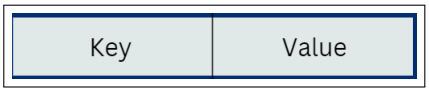

Fig. 1: Before Merging

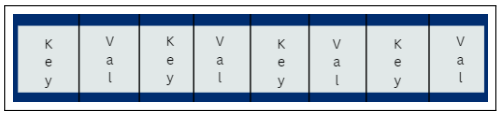

Fig. 2: After Merging

So from the given two figure we can see the clear difference between them. Here in the first one all the keys are arranged first and then all the values. So if the length of them is large then condition like all the keys may fit but not the values which arises a hit case/ But in the second architecture it is different which is similar to an architecture of Map Data Structure which reduces the percentage of miss rate.

2) Loop Interchange:
   In these particular method we change the orientation of executing loops in case of nested loops.

```
for (j = 0; j < 100; j = j+1)
.      for (i = 0; i < 5000; i = i+1)
              x[i][j] = 2 * x[i][j];
```

Fig. 3: Before Interchange

After :-

```
for (i = 0; i < 5000; i = i+1)
     for (j = 0; j < 100; j = j+1)
           x[i][j] = 2 * x[i][j];
```

Fig. 4: After Interchange

As we know that the memory are stored in a row format because majorly all the computer languages supports 1D array. Thus the computation seeming to be 2D in nature is actually 1D. So in our case before the interchanging the misses would be 5000 as *j* was dependent on *i*. But in the second case it would be 100 only.

3) Loop Fusion:
   Fusion of two loops doing the same computation can be merged because as we can see from the above example that there are some accessible which are common to both the loops like *a[i]*

```
int acc = 0;
for (int i = 0; i < n; ++i) {
      acc += a[i];
      a[i] = acc;
}
for (int i = 0; i < n; ++i) {
      b[i] += a[i];
}
```

Fig. 5: Before Fusion

```
Int acc = 0;
for (int i = 0; i < n; ++i) {
      acc += a[i];
      a[i] = acc;  // Will DCE pick this up?
      b[i] += acc;
}
```

Fig. 6: After Fusion

Now if there is miss on *a[i]* then it will be twice as they are in two different loops considering memory allocation for each loop to be different. If we merge them into one then it becomes only once reducing the ratio to 50% down.

4) Blocking:
   Whenever we need to do operations like matrix multiplication we do the operations by taking submatrix of each of them and then reconstruction the solution from already computed values of smaller parts. These improves the performance over traditional approach as it was based on going through whole row-col at the same time which increases cache miss percentage.

IV. Optimization for dynamic languages

A programming language that requires less rigid coding on the part of the programmer. It typically includes "dynamic typing" ,which allows the programmer to pass parameters at runtime without having to define them ahead of time. A dynamic language may also perform some self-checking at runtime, which would normally be done before hand during the compilation stage. Although dynamic languages provide greater runtime flexibility and are generally easier to programme, they do not absolve the programmer of responsibility for understanding the interactions that will occur.

JavaScript, VBScript, Lisp, Perl, PHP, Python, Ruby, and Smalltalk are examples of dynamic languages. Here we will take some instances in which we can improve performance in language like Javascript by changing some functionality or inclusion of some other feautures.

- JS has a concept of not declaring any variable with it's predefined type which makes it uncomfortable in many scenarios. Also concept of hoisting also makes confusing for the compiler because it states that you can use the variable before it's use. To some extent these things can be neglected with the use of strict typed JavaScript. But w.r.t to variable type it still remains ambigious. More in the V8 Chrome version it has provided two functions (1) The inherited prototype object has been incorporated into the current object definition. and (2) Method bindings are also part of type definition . These requirements are what make type assignment unpredictable.

- Because of JavaScript's complicated semantics and dynamic behaviour, static analysis of JavaScript is difficult. STS, an alternative to the traditional DFA approach divides the analysis into two distinct components. A method for selectively merging states during reachability computation and an embarrassingly parallel reachability computation on a state transition system. The STS framework makes it simple to experiment with various parallelization strategies. In addition, it is more applicable to languages with difficult control flow than DFA.

- Major concept than can have an impactful behaviour is JIT. JIT stands for just-in-time compiler. Here it applies the concept of dynamic loading of components as per needs. Languages like Java and C have a great importance for achieving better performance. The most effective JIT compilation policy for any given application can be identified by analysing all of the available JIT compilation policies. It has been demonstrated that utilising all of the free compilation resources are being used aggressively to compile more programs methods reaches a point of diminishing returns. At the same time, using free resources to reduce the backup queue significantly improves performance, particularly in slower JIT compilers.

V. Optimization using Machine Learning

Compilers perform two functions: translation and optimization. First, they must correctly translate program into binary. Second, they must find the most efficient translation. The vast majority of engineering and research practises are traditionally focused on this second performance goal. Using the traditional method of optimizing which can be considered as Hard-coded,Hand-crafted and not responsive.
Given a program, compiler writers want to know which compiler heuristic or optimization to use to improve the code. Better often means faster execution, but It could also mean a smaller code footprint or less power. Machine Learning can be used to create a model that will be used by the compiler to make such decisions for any given program. There are two major stages to consider: learning and deployment. The first stage teaches the model using training data, while the second stage applies the model to previously unseen data. We require a method of representing programmes in a systematic manner during the learning stage. This illustration is referred to as the program features. Machine Learning has major three components which are being executed in a sequential manner which are as follows:

1) Feature Engineering

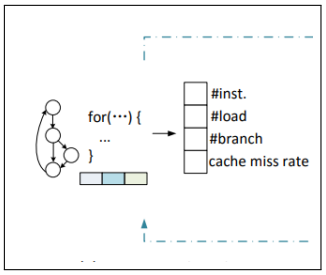

Fig. 7: Feature Engineering

We need to be able to characterise programmes before we can learn anything useful about them. Machine learning characterises programmes using a set of quantifiable properties, or features . There are numerous various features that can be used such as number of instruction or branches.Because standard machine learning algorithms work with fixed length inputs, the selected properties will be summarised into a fixed length feature vector. Each vector element can be an integer, real, or Boolean value. Feature engineering refers to the process of selecting and fine-tuning features. This process may need to be repeated several times to find a set of high-quality features from which to build an accurate machine learning model.

2) Model Training

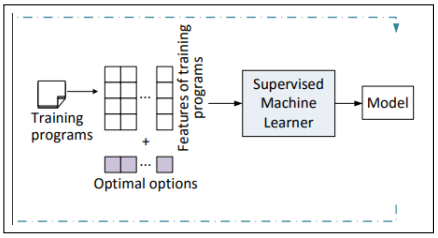

Fig. 8: Training

The next step is to use training data to create a model with a learning algorithm. Unlike other machine learning applications, we typically generate our own training data from existing applications or benchmarks. The compiler developer will choose training programmes that are representative of the application domain. We calculate the feature values for each training programme before compiling it with various optimization options and running and timing the compiled binaries to find the best performing option. This process generates a training instance for each training programme that consists of the feature values and the best compiler option for the programme.

3) Model Deployment

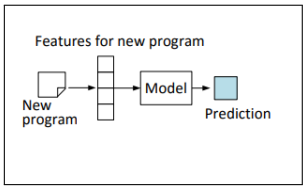

Fig. 9: Deployment

In these particular phase we deploy our model so that it could be useful to real-world. So we can assume that data would be a very for the model. Generally we make it happen by deploying it inside our server or making it to public by some API. Also one naive way is to export pre-trained model and then import it for further use. The compiler extracts the features of the input program before feeding the extracted feature values to the learned model to make a prediction.

A. *Example*

We can take an example where our main goal is to predict whether given OpenCL code should run on CPU or GPU. For the same we will use deep learning model named LSTM . The further proceeding is as below.

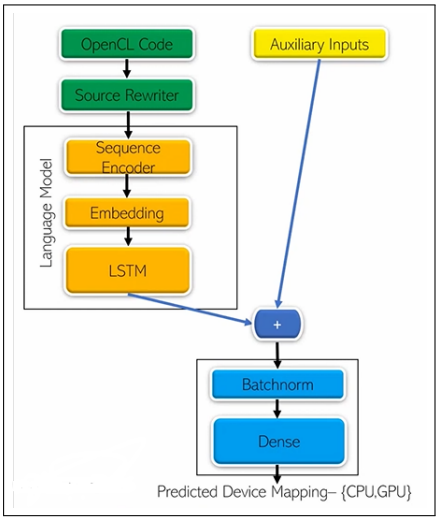

Fig. 10: Complete Process

As we can see from the figure that there are majorly 3 component before applying model namely :

1) Source Rewriter
   Source Rewriter just makes tasks easy for models to do things in a simplified way. It takes OpenCL code as an input and performs some actions like converting variable names to a simplified one, making it space-optimized as well as more readable for the computer.
2) Language Model
   It is nothing but a set of data available to a model for prediction. It can be used for training as well as testing. OpenCL code can be written to enable parallel computing on hardware such as a CPU, GPU, or FPGA. These codes are human-made, and as our goal is to make it execute with the minimum resources available, we will be using them as our dataset. But as we are using LSTM, we generally term these as Language Models.
3) Sequence Encoder
   As we all know, all ML/DL models work on numeric data rather than text data. As a result, the programmer must first convert it to machine readable format before applying the model. We are performing similar things here by using either of the methods like Word2Vec, Bag of Words, TF-IDF, etc. The below shown table shows the output for the same.

```
__kernel void memset_kernel(__global char *mem_d,
                            short val, int num_bytes) {
    const int thread_id = get_global_id(0);
    mem_d[thread_id] = val;
}
```

Fig. 11: Source Code

```
__kernel void A(__global char *a, short b, int c) {

    const int d = get_global_id(0);
    a[d] = b;
}
```

Fig. 12: After Rsewriting

idx	token	idx	token	idx	token
1	'__kernel'	10	','	19	'const'
2	' '	11	'short'	20	'd'
3	'void'	12	'b'	21	'='
4	'A'	13	'int'	22	'get_global_id'
5	'('	14	'c'	23	'0'
6	'__global'	15	')'	24	';'
7	'char'	16	'{'	25	'['
8	'*'	17	'\n'	26	']'
9	'a'	18	' '	27	'}'

Fig. 13: Word Embedding

The complete processes is shown below in pictorial format.

Steps followed :-

- First two parts (a) and (b) are already explained in above section.
- Once these steps are performed, we will arrive at word embedding, where each word will be represented by some numeric format. Here we will use the BOW (Bag of Words) technique. These arrays are passed into the LSTM architecture with hyperparameter tuning.
- Now we are going to pass this input into a CNN model, which is just a simple combination of dense layers. But before that, we would normalise it with the Batch Normalization technique. Batch normalisation is a technique for training very deep neural networks in which the contributions to a layer are normalised for each mini-

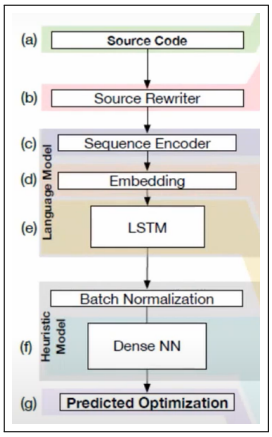

Fig. 14: Flowchart of whole process

batch. This has the effect of settling the learning process and significantly reducing the number of training epochs needed to train deep neural networks.

- The last layer of architecture will give output as either 0/1 which indicates which hardware component to use.

These was one of the applications where ML/DL techniques can be put into action for compiler optimization.

VI. Approach Using Dynamic optimization

Three major dynamic systems for automatic, profile-driven adaptive compilers have been developed. These are roughly divided into two categories: the Intel research compiler and the JUDO compiler. system, as well as the Jalapefio JVM, adhere to a compile-only approach, whereas HotSpot includes an interpreter as in our system, to allow for a mixed execution environment with interpreted and compiled code.

HotSpot is a Java Virtual Machine product that implements an adaptive optimization system. It immediately runs a program using an interpreter, as in our system, and detects critical "hot spots" in the program as it runs.It continuously monitors program hotspots as the program runs so that the system can adapt to changes in tile program behaviour. However, there is no detailed information available about the program monitoring techniques or the system structure for recompilation. Jalapefio is yet another Java-based JVM. They developed a multilevel reeompilation framework that included a baseline and an optimising compiler with three optimization levels which demonstrated significant performance improvements in both startup and constant state. They first proposed a strategy where the programmer needs to annotate the part of the code to be examined for optimization by the compiler. At compile time, it would react to only those highlighted. Then the JIT concept came into existence, where programmers didn't need to interrupt in between , all the things would be automatically done by the compiler.

VII. Conclusion

A review of current research on compiler optimization techniques revealed how various subsystems are being worked on in order to improve compiler efficiency. We have discussed various techniques for improving compiler performance, including cache optimization, dynamic optimization, optimization in dynamic languages, and the use of machine learning techniques. Several different techniques for improving these subsystems in the current literature were discussed, as well as their overall impact on compiler performance. The survey paper is intended to be a compilation of current research in the field of compiler optimization. It is intended to serve as both a reference and a benchmark for researchers developing new techniques to improve compiler efficiency.

REFERENCES

[1] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. "A Parallel Abstract Interpreter for JavaScript," in Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO'15)

[2] Wang, Zheng, and Michael O'Boyle. "Machine learning in compiler optimization."

[3] Jay Patel and Mahesh Panchal, International Journal of Computer Science and Mobile Computing.

[4] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler optimizations for improving data locality.

[5] Amogh S. Inamdar, Sindhuja V. Rai, Anagha M. Rajeev, Sini Anna Alex. Compiler Optimization using Machine Learning Techniques.

[6] Aman Raghu Malali, Ananya Pramod, Jugal Wadhwa, Sini Anna Alex. A Survey of Compiler Optimization Techniques.

[7] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Kornatsu, Toshio Nakatani. A Dynamic Optimization Framework for a Java Just-in-Time Compiler.

[8] M. Kandemir I, Kolcu I, Kadayif.Experimental Evaluation of A Compiler-Based Cache Energy Optimization Strategy.

[9] Byron Hawkins and Brian Demsky, Derek Bruening and Qin Zhao. Optimizing Binary Translation of Dynamically Generated Code.