Optimization Techniques in C – A Survey

Pratibha Singh
M.Tech (I.T) Student
Banasthali Vidyapith
Rajasthan, India

Smriti Sonker
M.Tech (C.S) Student
Banasthali Vidyapith
Rajasthan, India

Prabhat Verma
Assistant Professor
CSE Department
HBTI, U.P, India

ABSTRACT
Programmers spent most of their time in speeding up a program. Sometimes, speeding up a program leads to increase in the code size that adversely affects the readability as well as the complexity of the program, which makes the code less efficient. Thus, in-order to make the code efficient to work, optimization of the code is needed. There are a number of compilers available which can automatically optimize the code which dominates the kind of manual optimization. In this work, we have studied the different techniques which can be applied to improve the way of writing program in C language, such that the code becomes more efficient. Also, the term optimization is being explained, along with when and where the optimization is to be applied**.**

General Terms
Machine dependent techniques, Machine independent techniques, Optimization levels.

Keywords
Optimization, Complexity, Efficient code, Function.

1. INTRODUCTION
Sometimes, it is difficult to find out the part of the program which consumes most of the resources and results into an un-efficient code. In compiler design, there is one of the technique in which a piece of code is being transformed to make it more efficient as well as to improve the performance such that the output remains same, termed as optimization. Code optimization aims to make high quality code with best complexity (time and space) such that it should not affect the exact result of the code. It is mainly based on the criterion to preserve the semantic equivalence of the program, such that the algorithm must not be modified. On an average, the transformation should speed up the execution of the program. Optimization includes finding a bottleneck, a critical part of the code which is the primary consumer of the needed resources. Basically Code optimization concerns on correctness, it means the correctness of the generated code should not be changed.

On using different optimization techniques, the code can be optimized without affecting the original (actual) algorithm and final output with the intent of high performance. When performance is to be considered, then there is need to choose an algorithm which runs quickly and the available computing resources are being used efficiently. Basically, Code optimization involves the employment of rules and algorithms to the program segment with the goal such that the code becomes faster, smaller, more efficient and so on.

Optimization is classified as high level optimization and low level optimization. High level optimization are usually performed by those programmers who handles abstract entities and also keeps in mind the general framework of the task to optimize design of a system. On the other hand, low level optimization is performed at the stage when source code is compiled into a set of machine instructions.

2. WHEN AND WHERE IT IS NEED TO BE OPTIMIZED?
There are a number of strategies to achieve optimization. Some of the techniques are applied to the intermediate code in-order to reduce the size of the TAC instructions. And some other techniques are implemented as final code generation which involves selection of the instructions that are need to be omitted as well as allocation of registers. There are also some techniques that could be applied after the final code generation to make it more efficient.

3. OPTIMIZATION TECHNIQUES IN C
There are different optimization techniques through which an un-optimized code will be easily transformed to optimized one, in which some of the techniques are machine independent while others are machine dependent techniques. There also exist different optimization tools. After going through the different papers the techniques which are being used to optimize the codes in C are :

3.1 Machine Dependent Techniques
Machine dependent optimizations are those which requires knowledge of target machine architecture. Some of the strategies used are :

3.1.1 Minimize local variable:
On minimizing local variables in a function, the compiler will be able to fit them into registers, and hence the frame pointer operations on local variables that are kept on stack will be avoided. If all the variables are mentioned as register, then the performance will be improved since it will be accessed from register instead of memory which will always be faster. Also, when no local variables required to be saved on the stack, it will not incur any overhead of setting up and restoring frame pointer [7].

3.1.2 Number of parameters to be minimized
With large number of parameters function calls may be expensive because a large number of parameter pushes on stack on each call. Hence for this reason, to pass complete structures as parameters is being avoided and in this case pointers and references are to be used [7].

3.1.3 Ignore defining a return value if not used:
The return value is being always passed by the called function, because the called function does not "know" if the return value is being used or not, so it is being advisable that this return value may be avoided by not defining a return value which is not being used [7].

3.1.4 Prefer int instead of char and short:
Always it is being preferred to use int instead of char and short because in C, all operations of char is being performed with integer. If char is used in operations like passing char to

a function or any arithmetic operation, first compiler will convert the value of char into integer and after performing the operations it will again be converted into char. If a single char is used then it may not be efficient but if the same operations are performed number of times in a loop then the efficiency of a program may be decreased [7].

3.1.5 Optimizing switch statement:

Switch statement is being translated in different ways .If case labels are in a narrow range, an if-else-if cascade for the switch statement is not generated, instead if-else-if a jump table of case labels is originated and the originated code is faster in comparison to if-else-if cascaded code, also, performance of a jump table based switch statement is not dependent of the number of case entries in switch statement. If case labels are far apart, by placing the frequent case labels first, number of comparisons can be reduced. It means the class label which is being used least number of times should be at last and the frequently used label to be placed first.

In some cases the above techniques will not work where the compiler not originate the cascade of if-else-if, in such condition to get the same effect nested switch statement can be used.

In case of big switch statements, to minimize the number of comparisons being performed, big switch statements is to be break down into nested switch statements. In this condition the more frequently occurring case labels is placed into one switch and the rest are in other switch [7].

3.1.6 Prefer pre increment/decrement over post increment/decrement:

When pre increment/decrement and post increment/decrement are used for same operation, using pre increment/decrement will be more efficient. When post increment/decrement is used a copy of the object is produced, then increment/decrement is executed and the value is copied to the variable location. Instead using pre increment/decrement, the value is incremented first and the value is copied to variable location [7].

3.2 Machine Independent Techniques

Machine Independent optimization can be performed independently of the target machine for which the compiler is generating the code.

3.2.1 Inlining:

Inlining is one of the commonly used optimization technique which is used to" inline" the contents of function –basically instead of traditional call to that function, because it eliminates the need to jump, creation of a new stack frame and to reverse the process at the end of the function[5]. Inlining can be easily understand with the below given piece of code.

Old code

```
int fn(x , y)
{ x=x-y;
  y++ ;
  x=x*y ;
  return x;
}
```

New code

```
#define fn(x,y) (((x)-((y)) * ((y) + 1))
```

Fig 1: Example of Inlining

3.2.2 Code Motion:

This optimization technique is also known as code hoisting which unifies sequences of code that are same to one or more basic blocks such that code size be reduced and hence expensive re-evaluation will be potentially avoided. Loop invariant code motion is most common form of code motion, in which if a computation inside a loop produces same results for all iterations, it may be possible to move the computation outside the loop [1]. An example is shown in Fig.2

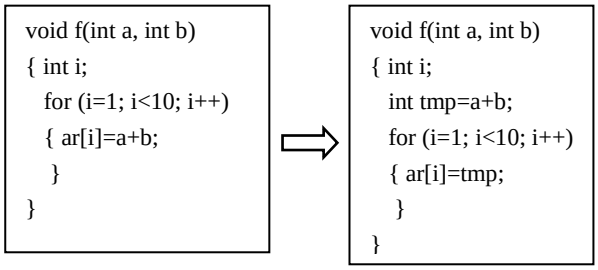

Fig 2: Example of Code Hoisting

3.2.3 Constant propagation and Constant folding:

Constant propagation is an optimization technique in which, if a constant value is assigned to a variable, then the subsequent use of that variable is replaced by the constant, i.e. in this integer constants are move to the place they are used [1]. Advantage of this technique is that both the number of registers and instructions executed reduces. On the other side, Constant Folding is the replacement of expressions that can be evaluated at compile time by their computed values. This technique is mostly applied to the expressions which have constant operands and can be evaluated at time. By this technique, the runtime performance will be increased because the code size will be returned by avoiding evaluation at compile time. Folding does not require additional pass in the analysis phase and hence it can be applied (used) to the entire program, most of the time it is preferred to perform folding during the production of intermediate language. Folding can be easily performed w.r.t to the basic block which is passed and so it is commonly thought of as a kind of local optimization technique.

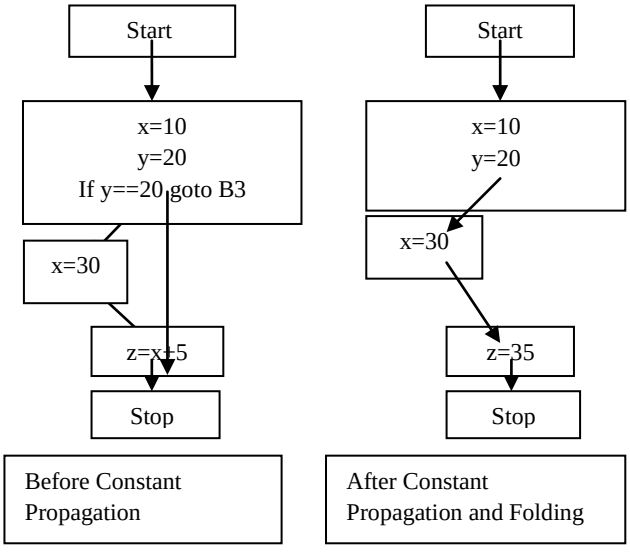

Fig 3: Example of Constant Propagation and Folding

3.2.4 Strength Reduction:

Is an optimization technique in which a type of operation is replaced by another type of operation, or it can be understood as those operations which are computationally expensive are replaced by the simpler ones having an equivalent effect. As an example of this, addition takes less time in comparison to multiplication operator, hence the multiplication operation is being replaced by the addition operation which is a classical example of strength reduction.

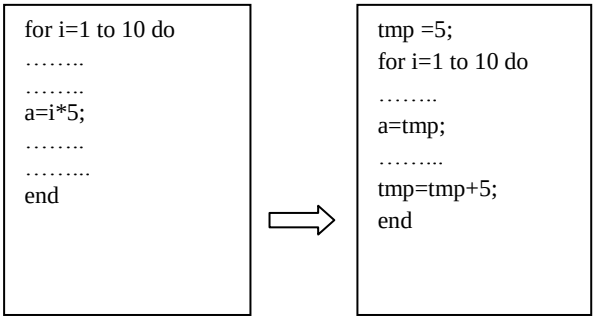

Fig 4: Example of Strength Reduction

3.2.5 Common Sub-Expression Elimination:

Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it. The operands which is used to calculate the expression, have not been altered, then the expression will be alive. An expression that is no longer alive is dead [1].

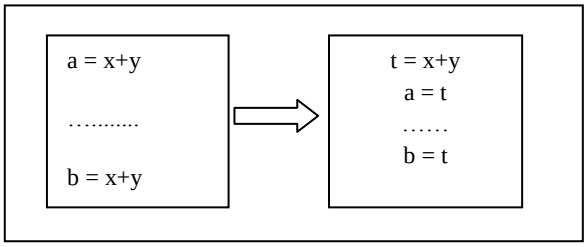

Fig 5: Example of Common Sub-Expression Elimination

3.2.6 Dead Code Elimination:

This consists of eliminating the instruction or code that is never used in a program and so it is considered as "dead" [1].

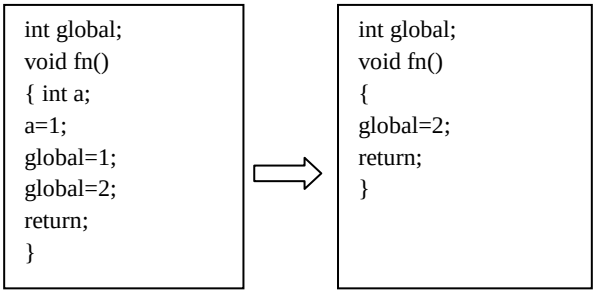

Fig 6: Example of Dead Code Elimination

3. CONCLUSION

This paper describes about the optimization and the different techniques along with how and where to optimize. The techniques can be applied by the programmer to make the code work better and hence to increase its efficiency. As the main section of the paper (Section 3) is divided into two parts in which first section focuses on machine dependent techniques. The second part describes about the different machine independent techniques. Hence, going through all the section of the paper, need of optimization can be easily understood, also when and where it is needed and finally how to apply different optimization techniques to a piece of source code. Further in future two or more techniques can be used together in order to make the optimization more fruitful.

4. ACKNOWLEDGMENTS

We are thankful to Mr. Prabhat Verma, Assistant Professor, CSE Department, HBTI Kanpur, who have contributed towards the successful completion of this paper.

5. REFERENCES

[1] Michael E. Lee, "Optimization of Computer Programs in C", Ontek Corporation, USA "Code Optimization" article. Available: http://leto.net/docs/C-optimization.php Forman, G. 2003.

[2] Maggie Johnson, "Code Optimization", Handout 20, August 04, 2008.

[3] Mohammed Fadle Abdulla, "Manual and Fast C Code Optimization", Anale. Seria Informatica. Vol. VIII fasc. I-2010.

[4] Mr. Chirag H. Bhatt, Dr. Harshad B. Bhadka, "Peephole Optimization Technique for analysis and review of Compile Design and Construction", IOSR Journal of Computer Engineering (IOSR-JCE), Volume 9, Issue 4 (Mar. - Apr. 2013).

[5] "Optimization Techniques in C", Fall, 2013. Available: http://cs.brown.edu/courses/cs033/docs/guides/c_optimization_notes.pdf.

[6] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, "Source–Level Execution Time Estimation of C Programs", Proceedings of the ninth international symposium on Hardware/software code design.

[7] The EventHelix website. [Online] "Optimizing C and C++ code" Available: http://www.eventhelix.com/realtimemantra/basics/optimizingcandcppcode.htm#.VCpsxfmSyFM.

[8] Tips for "Optimizing C/C++ Code". Available: http://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf

[9] "Writing Efficient C and C Code Optimization". Article: http://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization

[10] "Optimizing C++/ Code Optimization/ Faster operations" Available: http://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Code_optimization/Faster_operations.

[11] "Research on code optimization when develop highway network monitoring software based on Trimedia". Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6843485&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D6843485

[12] "Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation". Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1213375&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1213375\

[13] S. K. Srivastava, Deepali Srivastava, "C in Depth" 3rd edition.