ISSN:0975-9646

Code Optimization using Code Purifier

Neeta Malviya[#1], Dr. Ajay Khunteta[*2]

[#]*M.Tech (Software Engineering)*
Department of Computer Science & Engineering, Rajasthan Technical University
Poornima College of Engineering, Jaipur, India

[*]*Professor (Department of Computer Science & Engineering)*
Poornima College of Engineering, Jaipur, India

***Abstract*- In this paper we have implemented the concept of Inlining, Folding, Dead Code Removal and Common Sub expression removal for optimizing the code.**

The improvement in the quality of code remains a big issue from the earlier days. Sometimes it is difficult for a programmer to find out which part of code consumes more resources and hence lead to an inefficient code. Previously most of the optimization were done manually or can be said as statically which leads to number of problems to the programmer and also had some of the limitations. However, these days several compilers are available which makes the optimization to be performed dynamically. In this dissertation work, an attempt has been made to design and implement a system that can automatically optimize the code in order to minimize the complexity of the code such that the code becomes more efficient. To accomplish this, a system has been developed to implement the two machine independent techniques- dead code elimination and common sub-expression elimination, which dynamically optimizes the code. The thesis basically moves around these two techniques, how these techniques have been implemented, and how does it works along with the calculation of the code complexity. For this, the codes were taken randomly. Hence the optimization of code could be possible using different optimization techniques. Instead of focusing on developing a new algorithm or to improve the existing one, this dissertation attempts to understand the existing compiler techniques clearly and to analyze the result after implementing the techniques and also to compare the complexities based on different metrics of the code.

I. INTRODUCTION

In compiler design, there is one of the technique in which a part of code is being transformed to produce 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. The main aim of the code optimization is to make high quality code with improved complexity (time and space) without affecting the exact result of the code.

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. So, it can be said that the objective of optimization is to write a code in such a way that can reduce both the memory as well as speed. Basically, Code optimization involves the employment of rules and algorithms to the program segment with the aim such that the code becomes faster, smaller, more efficient and so on.

In theoretical perspective, the compiler optimization basically refers to the program optimization to achieve performance in the execution. Source code optimization refers to the three aspects, a programming language code (front code), an assembly language code which is generated by the compiler to the appropriate programming language (intermediate code), the object code which is generated from the assembly language code for the execution of the actual work. This work involves the implementation of two different techniques, dead code elimination and common sub expression elimination with the use of different tools in order to optimize the code.

II. IMPORTANCE AND RELEVANCE OF THE STUDY

Optimization is that the field wherever most of the analysis is completed looking the various papers and article a number of the relevant data are gathered that makes the optimization method doable.

Chirag[1] describes concerning the hole optimization technique using completely different pattern matching approaches that forms AN regular expression and conjointly has explores the previous and current analysis problems in term of "optimizing" compilers using optimization rules that area unit imagined to be matched through that the redundant instruction of intermediate code are often investigated and replaced. For this completely different pattern matching approaches are mentioned like string based mostly, tree manipulation, object based mostly etc. The paper is being divide into four sections wherever specific rules and pattern rules are laborious coded that explains concerning the machine dependent hole optimizers, the inspiration of retarget ready hole optimisation the combination of code generation and optimisation into one part and concerning completely different pattern matching methods severally.

Brandolese[2] introduced a comprehensive methodology for software package execution time estimation, that is supported by rigorous mathematical models of C statements in terms of elementary operations. During this the complete analysis flow has been performed inside an epitome toolset that are presently being employed for supportive and adapting model parameters.

Johnson[3] in his article describes concerning the optimization that is procedure of changing a bit of code such the code becomes additional economical and therefore the output remains same. It's being conjointly declared that a lot of the issues were NP complete and therefore most of the optimization formula depends on approximation and heuristics. Conjointly he declared that when the code has been written, let the compiler do the optimization using compiler processes.

In this paper the outline is given concerning the optimization that focuses thereon the collection method ought to be like because the correctness of the generated code mustn't be modified, conjointly strategy is defined like "when" and "where" to optimize for playing optimization. a number of the techniques applied to intermediate code, others area unit applied to final code generation and even a number of the techniques may occur once the final code generation within which the try is created to remodel the assemble code itself to additional economical one. It conjointly describes concerning the "local optimization" that is outlined because the optimization that area unit solely enforced inside a basic block, therefore these sort of optimization area unit easy to implement as a result of any reasonably management data isn't required solely the optimization is to be performed inside a block. A number of the native optimization like constant folding, constant propagation also are explained. Optimizations that may eliminate useless instructions using algebraically identities area unit like operator strength reduction, copy propagation, dead code elimination except for native optimizations, the same reasonably optimization that might be applied across the essential blocks makes them world optimization. Conjointly concerning machine optimization is explained, one among the machine optimization that is of specific importance is register allocation, another most significant optimization is instruction programming.

Michael E Lee[4] narrate techniques which were used to optimize the code of C. The concentration is on reducing time spent by the CPU and provides sample source code transformation which often yield improvement. The transformation or optimization is performed such that the developers of the application programs have the responsibility to design programs in order to make use of limited and expensive resources.

The article also describes some of techniques which were developed for C and C++ code which were developed for real systems. Detail is also given as how to work with the big switch statements in order to reduce the number of comparisons, some of the techniques are described which can be used to minimize local variable declaring and reducing number of parameters. Also use of " int over char and short " is preferred. The document also presents some of the techniques that may enhance performance.

Huang Zhijun[5] in his paper discuss about different code optimization such that the system resources can be fully optimize in order to maximize the efficiency of code. Using different techniques of optimization such as data flow optimization, loop invariant code, the operation of data etc., a new product on TI-DSP is being developed. Code optimization can greatly improve the calculation speed, to some of the specific software implementation processes, other methods of optimization can be integrated such that maximum optimization can be achieved.

Keith[6] developed a modified algorithm for Operator strength reduction termed as OSR. OSR actually is an improvement over a prior/previous algorithm given by Allen, Cocke, and Kennedy. OSR depends on prior optimization and properties of SSA graph such that the algorithm developed should be easy to understand and implement, it should avoid instantiation of sets of induction variable and region constants which are required by other algorithm. The algorithm depends on dominance data which are calculated during the construction of SSA, the resultant algorithm is easy to recognize, teach, and implement. Also, discussion about previous work which are been done in the field of strength reduction.

Koushik Ghosh[7] in his article have collected all the experience and information that can be used to speed up the execution as well as memory to make a C code optimized. In this article the author had also presented the number of guidelines using which the optimization in C can be made possible. It also discusses that which part of the code need to be optimized. Going through the article, different techniques of optimization are well understood such as about inlining, integers, boolean expression etc.

Tom Erkinen[8] presents Model based design capabilities and tools which support verification of optimized fixed point ECU software. It also tells that while implementing production software it is important to consider about the code optimization and code verification strategies for embedded software. It also covers the developed technologies that further enable organizations to adopt MBD for embedded system deployment and verification.

Paul Hsieh[9] have presented an elusive subject of program performance optimization. This article outline the general task of Code optimization, Code architecture which is about the simple mathematical analysis, understanding of technological performance, optimization, also many of the examples is been given by the author.

Keith D. Cooper [10] presented a part of survey in which transformation of code is intended to enhance the running time of programs on the uniprocessor machines. Instead of analysis methods, the aim of transformations is to enhance the quality of code. Analytical techniques and specific data flow problems are described which are necessary to understand transformations. Author had individually discussed many code transformations. Each of the transformations was discussed in depth which makes broad understanding.

Reg. Charney[11] describes about how the code complexity can be determined with complexity metric and presented his own metric known as Pymetric. The author

also described about maintenance metrics which are also known as static metric and has also subdivided maintenance metrics into formatting and logical, where formatting metric concerns with indentation conventions, naming conventions, comment forms, whitespace usage, and so on whereas logical metric concerns with number of paths through a program, conditional statements and blocks, level of parenthesization, number of terms and factors in an expression etc. Also, the column of author outlines factors which can be described while calculating the complexity such as MCB.

Bruce Childers[12] describes a new method which is based on a constant compilation system, by which the application of code is constantly improved when the aggressive and adaptive code optimization is applied at all times from static to dynamic optimization. Author has described a general approach and procedure for continuous compilation of application code, also in this paper it is shown that for loop optimization prediction framework has high accuracy.

Doeppner[13] in his article described about the different optimization technique such as code motion, loop unrolling, inlining etc. Also the author has discussed about writing cache friendly code that is, code which can be organized and designed such that it utilizes the machine's cache in the most efficient way possible. Also, about writing pipeline friendly cache is described.

Mohammed FadleAbdulla[14] documented the experiences which they have collected, that can be used to enhance the style of writing programs in C language even an implementation of Intel VTune profiler is presented using manual optimization. The document also shows how different small changes made to program can affect the performance of program. It is being also stated if the developers were using good compilers and have some knowledge about the optimization techniques, then they can much easily develop applications with high performance[15].

III. PROPOSED SOLUTION

We have suggested a solution which works in the four main concepts.

A. Dead Code Elimination

Dead code elimination is one of the technique used for the transformation of code. The idea behind this technique is that if the variable holds a value at a point which is not used later anywhere in the program, then the variable is said to be dead at that place. The assignment made to a dead variable is a dead assignment which can be safely removed from the program. Dead code is a code which is never executed or that does nothing useful or if an instruction result is never used, then that is considered as "dead" and can be removed from instruction stream.

In this dissertation work, the technique is implemented which search for the function which is never used and then eliminate the unused function. The dead code elimination/technique works in following steps:

1) First of all the code which is to be optimized is loaded and when a remove dead code button is clicked, the code is extracted from the rich text line by line and places into the array.
2) After the code is being placed to the array, the code or program is then passed to the Cppcheck tool which will compile and return the result by identifying the function names which are not used in the program code and hence are treated as dead code.
3) Once the unused function is identified then the search is made to the original program and the unused functions which were treated as dead code in previous code are removed from the original i.e. from the unoptimized code.
4) Finally the new optimized code is saved with the name dump.cpp in debug folder.

B. Common Subexpression Elimination

Common sub expression elimination is a type of machine independent optimization. The idea behind this compiler optimization technique is to find redundant expression evaluation and replacing them with a single computation. Basically the idea behind this is that two operations are common if they produce same result. Many of the times it happens that in a same program the same expression is evaluated at different places and the values of operands does not change i.e. the value of the operand remains same in the expression, hence in this case the redundant expression can be easily replaced with one variable. For example, the program may evaluates a*b in the starting and end of the code, if the values of a and b does not change in between these two expression, then in this condition instead of computing these expression again and again it is better to save the value of a*b in temporary variable and then use it in the end. Using this technique the redundant computations in the program is eliminated, also the time overhead which is required to compute the expression more than once is reduced.

The criteria used in our work comprises of the following steps:

1) First the source code is loaded which aims to be optimized.
2) When the common sub expression removal button is clicked, only those lines are extracted which contains equal to (=) sign and also ends with a semicolon (;)
3) Then the common expression is searched from those statements.
4) When the common expression has been found, its value is assigned to a variable 'SUB1'.
5) This variable then substitutes all other instances of the common expression.
6) The new modified code is then saved to the desired folder using the save result button.

C. Inlining

Inlining is that the method by that the contents of a function are "inlined", basically, derived and affixed rather than a traditional decision thereto function. This manner of optimisation avoids the overhead of function calls, by eliminating the requirement to leap, produce a brand new

stack frame, and reverse this method at the top of the function.

Advantages: -

1) It doesn't need function line of work overhead.
2) It conjointly save overhead of variables push/pop on the stack, whereas function line of work.
3) It conjointly save overhead of come back decision from a function.
4) It will increase locality of reference by utilizing instruction cache.
5) When in-lining compiler also can apply intra procedural improvement if nominal. this can be the foremost vital one, during this method compiler will currently specialize in dead code elimination, will provide additional stress on branch prediction, induction variable elimination etc..

Old code:

```
int foo(a, b)
{
    a = a - b;
    b++;
    a = a * b;
    return a;
}
```

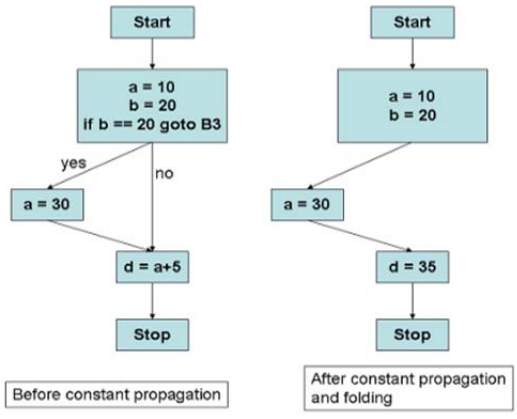
New code:

```
#define foo(a, b) (((a)-(b)) * ((b)+1))
```

D. Constant Propagation and Folding

Constant folding refers to the analysis at compile-time of expressions whose operands are identified to be constant. It involves determinant that every one of the operands in AN expression are constant-valued, acting the analysis of the expression at compile-time, so exchange the expression by its worth. If AN expression like ten + a pair of * three is encountered, the compiler will compute the result at compile-time (16) and emit code as if the input contained the result instead of the initial expression.

If a variable is appointed a continuing worth, then ensuant uses of that variable is replaced by the constant as long as no intervening assignment has modified the worth of the variable. it's known as constant propagation.



IV. IMPLEMENTATION

The system interface is this work has been developed in visual studio 2010 using visual basic programming language. It has been designed such that the user feel ease to understand the basic layout of the work. Its architecture is such that it could easily be navigated among the various options, and performs the analysis work as desired.

There are various menus provided for the user as per the requirements that are placed on the top of the window, and they are:

1. Inlining
2. Dead Code
3. Common Sub Expression
4. Folding
5. Analysis

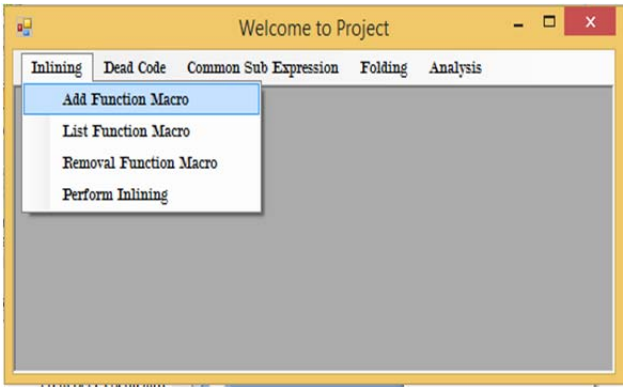


Fig. System Layout

REFERENCES

[1] 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).

[2] 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 codesign.

[3] Maggie Johnson, "Code Optimization", Handout 20 "August 08,2004.

[4] Michael E. Lee, "Optimization of Computer Programs in C ", Ontek Corporation, USA.

[5] Huang Zhijun, Liu Weiming,HeZengzhen, "Research on code optimization when develop highway network monitoring software based on Trimedia" available at http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6843485&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D6843485

[6] Keith D. Cooper, L. Taylor Simpson, Christopher A. Vick" "Operator Strength reduction" available at :http://www.cs.rice.edu/~keith/EMBED/OSR.pdf.

[7] Koushik Ghosh, "Writing Efficient C and C Code Optimization" available at: http://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization.

[8] Tom Erkinen,"Fixed point ECU code optimization and verification with model based design" available at http://in.mathworks.com/tagteam/59064_2009-01-0269.New.pdf.

[9] Paul Heish, "Programming Optimization: Techniques, examples and discussion" available at: http://www.azillionmonkeys.com/qed/optimize.html

[10] Keith D. Cooper, Kathryn S. Mckinley, and Linda Torczon,"Compiler-Based Code-Improvement Techniques"

available at http://www.cs.tufts.edu/~nr/cs257/archive/keith-cooper/survey.pdf

[11] Reg. Charney, "Programming Tools: Code Complexity Metrics" available at http://www.linuxjournal.com/article/8035

[12] Bruce Childers, Jack W. Davidson, Mary Lou Soffa, "Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation" available at http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1213375&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1213375\

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

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

[15] Code Optimization" article available at: http://www.viva64.com/en/t/0084/