Modularization Based Code Optimization Technique for Ensuring Software Product Quality

Manikandan N [1] Senthilkumar M[2] Senthilkumaran U[3]
1 Assistant Professor(Senior), SITE, VIT University, vellore, Tamilnadu, india
2 Assistant Professor, SITE, VIT University, vellore, Tamilnadu, india
3 Assistant Professor(Senior), SITE, VIT University, vellore, Tamilnadu, india
manikandan.n@vit.ac.in senthilkumar.mohan@vit.ac.in u.senthilkumaran@vit.ac.in

Abstract-**A segment of a code or a program segment displays different or multiple behaviors when it is executed for multiple numbers of times or when the program is on fly. This divergence of behavior associated within the runtime behavior of a program segment implies that we can apply different optimization strategies on the same piece of code or even for a bunch of code which are collectively referred as modules which together makes a system or a component that aims in proper functioning of a system. In this paper we focus on an aggressive optimization strategy that can be applied to any part of a code segment to identify the number of actors (operators, modules, loops) and also to analyze statically the involvement of those actors over the program and to reduce the program complexity and memory.**

Keywords: Optimization, modularization, software quality

1.INTRODUCTION

Researchers of the optimization community has always been trying to develop some powerful and efficient optimization techniques over the past few decades which are aiming towards the improvement of the code. Well, still there are more promising avenues of improvement where we can work on the existing techniques and we can introduce some modified and improved optimization technique that will improve the present scenario. This work on the optimization will be a small step towards the coming improvements in the field of optimization. Most fortunately the development of optimization research in the recent past indicates that still there are enough possibilities for significant performance improvement by making use of new efficient designs and modification of traditional designs.

A segment of a program or a code segment may behave differently when the program is on fly or when it is executed multiple number of times by the user in a different fashion. This divergence of behavior is basically due to multiple invoking of a program segment [2]. This behavior difference is not only reflected when the program is executed multiple number of times but also during the interaction between the architectural features and code segments [2]. Optimizing a program when it is being executed is quite possible and often can lead to some significant improvements. Research study has always provided evidence supporting that applying optimization technique during the execution of a program provides much advantage instead of applying optimization technique to a program statically. The behavior of a program which is on fly can change over its lifetime thus it leaves traces of evidence stating that different optimization strategies are possible and are advantageous and hence can be applied at any point during the program execution. For example, a program which has a 'switch case' statement is often executed based on the condition a user wants the program to be executed, the user decides the sequence of the execution, which code segment will be executed first and which will be followed. The same program can be executed with the use of looping statements where the user will design the sequence of instructions to be executed at the compile time and there will be any user feedback at the run time but in such cases loop unrolling must be prevented.

It is always useful to have exact information at runtime as it not only helps to determine and identify which optimization technique to apply and when to apply but also it enables more effective application of certain benefits of optimization. For example, if we have the information which part of the program, let us say which if-else-statement is executed more frequently over the other if-else-statements at different points in the execution of a particular program and that information will help us to make a most proper organization or placement of code.

The behavior divergence of a particular code segment happens not only when that particular code segment is invoked by a different preceding code but also happens with the multiple invocation of the code segment by the same preceding code [2]. The main reason behind such behavior difference is due to the execution of the preceding code of a code segment probably change the context of execution of the respective code segments which implies that the code segment is actually executed with different set of architectural resources [2]. For example, a preceding block of code may allocate and need higher amount of memory space which can result

into a cache miss in the next code segment, whereas another block of code segment may not need to allocate much higher memory space and thus will not result into cache miss for the next code segments. The difference in behavior of code segments supports new opportunities for code optimization. Code Optimization research states that if there exist a code segment which shows behavior difference which can be optimized according to the divergence of behavior based on various contexts, possibility of improvement of overall performance is high.

The behavior divergence of code segment indeed exists in real-world programs. Figure1 shows the profile of cache misses of functions, [3] [4]. The functions are all invoked more than twice during the execution of the whole program. Figure1 (a) shows the distribution of those records of cache miss of one each function .Figure1 (b) shows the relative difference between max cache miss and min cache miss. The figures clearly show that program segments in real-world programs not only have a widely distributed spectrum of behaviors, but also possess a large difference between behavior extremes [2].

The traditional code optimization technique which tries to take the maximum advantage of the behavior change is *Hotspot* Optimization [5] which applies more aggressive code transformation. The traditional optimization techniques (which will be discussed later) normally operate in a reactive manner and are attempted at the receiver end only because of which high compression efficiency is not achieved at the code level. Existing optimization techniques have a limited view of the program due to which we face difficulty in utilizing the benefits of the actual run-time behavior of the programs [1] and hence increase the computational efficiency thereby hampering the quality terms.

2. RELATED WORKS

Optimization is defined as an act, process or methodology of making something (a design, system, or decision) as partially perfect, functional or effective as possible [5]. It is also a characteristic of processes that focuses on continually improving process performance through both incremental and innovative technological changes along with improvements [6]. Optimization is the field where most compiler research is done today [7].

The term optimization usually presumes the system to retain the same functionality. Optimization will generally focus on reducing execution time, memory usage, disk space or bandwidth or some other resource without hindering much the expected result. Optimization specifically means to modify the code and sometimes it allows compilation settings on computer architecture to produce more efficient software [5].

It is also necessary to determine when to apply optimization strategies as it can lead to **premature** optimization which later can result in a design that is not as clean as it could have been. Premature optimization occurs when a programmer lets performance considerations affect how they designs a piece of code, before getting the design right. Optimization must be approached with caution. It is important to first have sound algorithms and a working prototype [5]. "PREMATURE OPTIMIZATION IS THE ROOT OF ALL EVIL" - Tony Hoare.

Optimization can be done manually as well as it can be done with the use of automated tools [5]. Automated optimization tools are called Optimizers [7]. Optimization must focus on correctness as of when and where to optimize. Optimization can be performed at several levels and by various parties [7].

Optimization can be done at various levels by different parties.

I. Design Level.
II. Source Code Level.
III. Compile Level.
IV. Assembly Level.
V. Runtime Level.

In software engineering point of view and according to the Capability Maturity Model (CMMi) optimization stands at the final level of software process development which primarily focuses on deliberate process improvements. Optimization in this case of process development does not restrict to only code optimization.

Code Optimization is the process of transforming a piece of code to make more efficient (in terms of time & size) without changing its expected output [8]. Code optimization can also be termed otherwise as modification or process of transformation [9]. Code optimization involves the application of rules and algorithms to program code with the goal of making it faster, smaller and more efficient [10]. Code Optimization aims to satisfy one of many desirable goals in software engineering, and is often antagonistic to other important goals such as stability, maintainability and portability [11]. There are a variety of tactics for attacking optimization. Some techniques are applied to the intermediate code, to streamline, rearrange, compress, etc. in an effort to reduce the size or shrink the number instructions. Others are applied as part of final code generation choosing which instructions to emit, how to allocate registers and when & what to spill [12].

Common Sub-expression Elimination: Two operations are common if they produce the same result [7]. If the value resulting from the calculation of a sub-expression is used multiple times, then we perform the

calculation only once and substitute the result for each individual calculation [13]. The problem lies when it creates an excessive number of temporary values which may take longer than simply re-computing arithmetic results when it is needed [15].

Constant propagation: If a variable is assigned a constant value, then subsequent use of that variable can be replaced by the constant [7]. Constant propagation can also cause conditional branches which then increase the compile time.

Copy propagation: It is all about replacing multiple variables that use the same calculated value with one variable [13]. The assignment operation may get eliminated sometimes.

Dead Code Elimination: If an instruction's result is never used, the instruction is considered "dead" and can be removed from the instruction stream [7]. Code that never gets executed can be removed from the object file [13].

Global Register Allocation: Variables that do not overlap in scope may be placed in registers, rather than remaining in RAM [13].

Local Optimization: Optimizations performed exclusively within a basic block are called "local optimizations" [7]. These are typically the easiest to perform. Local optimizations may include instruction substitutions [13].

Global optimization: Global optimization allows the compiler/optimizer to look at the overall program and determine how best to apply the desired optimization level [13].

Peephole Optimization: Peephole provides local optimizations, which do not account for patterns or conditions in the program as a whole [13].

Inline calls: A function that is fairly small can have its machine instructions substituted at the point of each call to the function, instead of generating an actual call [13].

Instruction scheduling: Instructions for a specific processor may be generated, resulting in more efficient code for that processor but possible compatibility or efficiency problems on other processors. This optimization may be better applied to embedded systems, where the CPU type is known at build time [13].

Lifetime analysis: A register can be reused for multiple variables, as long as those variables do not overlap in scope [13].

Loop counting using CTR: The Count Register (CTR) on the PowerPC is intended to be used for counting, such as in loop iterations. There are some branch instructions that specifically use the value in the CTR [13].

Loop invariant expressions (code motion): Values that do not change during execution of a loop can be moved out of the loop, speeding up loop execution [13].

Loop transformations: Some loop constructs can be rewritten to execute more quickly [13].

Loop unrolling: Statements within a loop that rely on sequential indices or accesses can be repeated more than once in the body of the loop. This results in checking the loop conditional less often [13].

Strength reduction: Certain operations and their corresponding machine code instructions require more time to execute than simpler, possibly less efficient counterparts [13] or just replacing an operator by a "less expensive" one [7].

Code Motion: Code motion unifies sequences of code common to one or more basic blocks to reduce code size and potentially avoid expensive re-evaluation [13]. The most common form of code motion is loop-invariant code motion [7].

Data-Flow Analysis: The additional analysis that an optimizer must do is to perform optimizations across basic blocks is called data-flow analysis [7].

Machine Optimizations: Specific machines features (specialized instructions, hardware pipeline abilities, register details) are taken into account to produce code optimized for this particular architecture [7].

In general other than the individual shorcomings all the above mentioned traditional optimization techniques suffer from the following shortfalls.

- Code readability Decreases [14].
- Program Complexity Increases [14].

Considering the above pitfalls associated with the traditional optimization technique, this paper will focus on a modularization based code optimization technique that will work at the code level and its main focus will be process improvement.

3. PROPOSED TECHNIQUE

In this paper we propose an aggressive optimization technique which tries to take the advantage of the behavior change in code segment during its runtime which thus improve code performance and reduce code size. The technique will primarily focus on the following few aspects, described below.

- ✓ Minimum cost & reduction of computational complexity.
- ✓ Achieving the performance goals without much computational time consumption.

Architecture

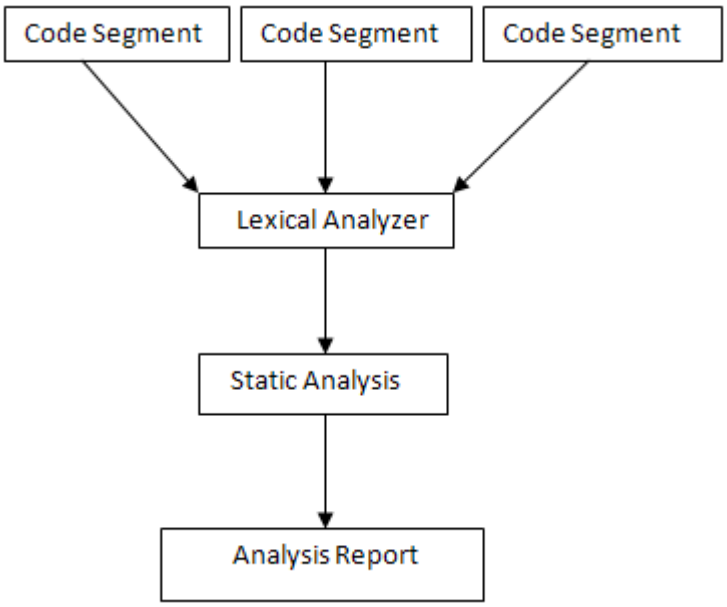


Fig. 1 Basic Architecture

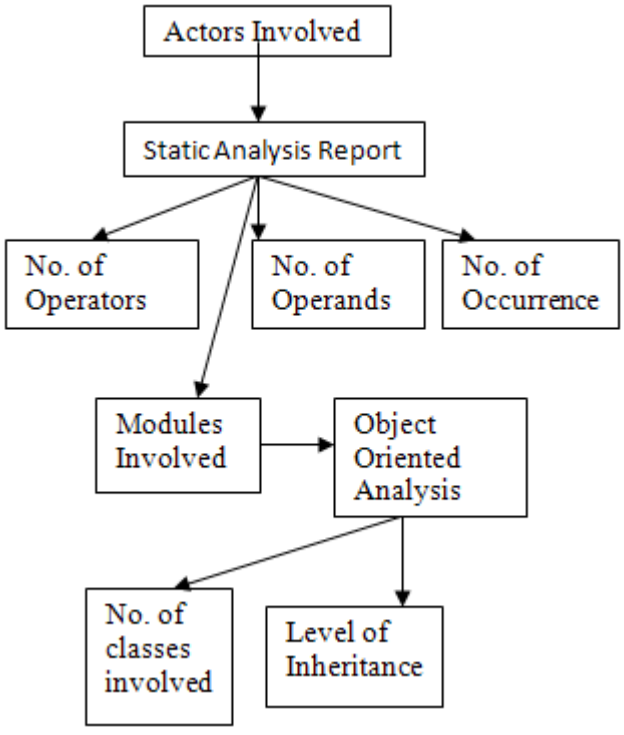


Fig 2. Detailed Architecture

Code segments are here used an input for the optimization technique to work. Code segments can be a piece of a large program which can be of any programming language. The code segment will be scanned as quite like the lexical analysis phase of the compiler which will be followed by static analysis and that will generate a static analysis report.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a,b,c,d,e;
    a=10;
    b=5;
    c=a+b;
    cout<<"\n the result is"<<c;

}
```

The above figure can be an example for one of the other code segments.

The lexical analyzer designed will scan the code segment and indentify the tokens involved in it. The few parts of the static analysis report will be based on the scanning of code segment by lexical analyzer and the later part will be designed in order to identify the other areas of the work.

In the later part the analysis report, will comprise of few more aspects which will include number of Lines of Code, the execution time of the particular code segment, the number of modules associated in the code segment, and the tokens which are identified will be stated as actors involved in the program.

The **algorithm** for proper working of the lexical analyzer and token separation is shown below:

1. Store the program for lexical analysis in a file.
2. Read the content of the file line by line.
3. Store all the operators, keywords of the language.
4. Check if the character is an alphabet or an operator or a digit.
5. If the next character read, is also an alphabet or a digit then store it in an array token until the next character becomes a non-alphabet.
6. Compare token with the list of keywords.
7. Check if the character string is header format prints it as Header.
8. If present, then print that it is a keyword else print that is an identifier.
9. If the character read is an operator, then print it is an operator.
10. Check if the character is a digit if so print it is digit.
11. Display all the details in a separate file.

The output of the lexical analyzer will provide the details of token which are actually termed as actors associated the code segment. The static analysis report will be stored in a textual format.

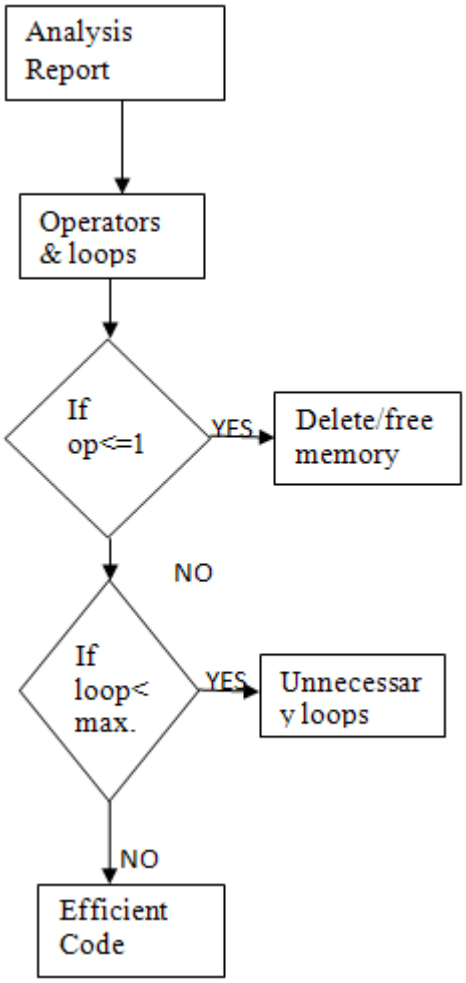
```
# - It is an operator
include
- It is a key word
< - It is an operator
stdio.h is a header file
> - It is an operator
# - It is an operator
include
- It is a key word
< - It is an operator
conio.h is a header file
> - It is an operator
```

The next part of the work will be including a little bit of software measurement related work, count of number of lines of code, the occurrences of each actors in the code segment, the number of modules count, which will be important for modular programming.

Modularization is normally a technique related to software design that increases the extent to which software is composed of separate inter-changeable components, called modules [16]. Counting the number of actors associated with the code segment in respective modules will be an important step towards the work. The each actor involved for example, an operator, so the occurrence of each operator in a particular module will be calculated for the respective module and the number of total occurrences will checked in that particular module. In the similar fashion the number of occurrences of loops in a particular module will also be calculated for the optimization technique to proceed.

The following flowchart diagram shows the various conditions that are checked in every module for appropriate optimization to work. The main aim of this work is to reduce the complexity involved in a program where more number of modules is involved. The basic function will be to identify the operators which are unused in a particular module and thus to remove such operators from that particular module will reduce the complexity as well as the execution time will be reduced. In the similar way the number of loops used in particular program will also be checked and if more number of loops are used in a particular loop then the unnecessary loop used will be removed.

Flow Diagram




```
fun modl 1


No.of.variables: 5


variable: a      in modl: 1
variable: b      in modl: 1
variable: c      in modl: 1
variable: d      in modl: 1
 remove d variable
variable: e      in modl: 1
 remove e variable _
```

Fig. 3 The output displaying the optimization technique

The technique will be concentrating on the following three aspects;

- To take the advantage of local computation (communication between the modules) as much as possible ;
- To control communication in order to reduce the number of unwanted or unnecessary communications between the module thus increasing the number of lines of code and making the program lengthy which increases the size and execution time as well;
- To reduce the message length in a communication process.

The technique proposed here is much useful when related to large programs where the modules are independent of each other and also they show much divergence of behavior when executed in a different fashion and based

on that we can apply this technique which will identify the occurrence of each operators and loops in individual module and thereby reducing the number of unused operators and loops which thus reduces the program complexity and also the execution time and does not hamper the code readability. In evolutionary computing also this technique must be useful. Normally uncertainties in this particular filed are too difficult to avoid or it can`t be stopped from occurring corresponding to the designs unless we use an optimal (most advantageous) or reliable optimization solution [17]. We encounter with the design issues whenever the design proposed for a particular problem fails to achieve the desired output.

To deal with such design issues we can focus on reliable optimization techniques. One popular approach is Monte Carlo Simulation technique which deals with uncertainties by creating number of samples with the help of probability distribution.

Conclusion

Optimization techniques can improve code size as well as speed. It is mandatory to identify when to apply optimization as premature optimization is the root cause for evil. Optimization must have to applied to appropriate conditions otherwise the code readability will not be achieved thereby hampering the performance.

References:

[1] Jason D. Hiser, Naveen Kumar, Min Zhao, Shukang Zhou, Bruce R. Childers, Jack W. Davidson, Mary Lou Soffa: Techniques and Tools for Dynamic Optimization, Proceedings of the international symposium on Code generation and optimization, 2006. IEEE Computer Society.
[2] Murat Bolat, Xiaoming Li: Context Aware Code Optimization, Proceedings of the International Conference on Electrical & Computer Engineering, Newark, USA, 2009. IEEE Society.
[3] SPECCPU2000V1.3.http://www.specbench.org/cpu2000.
[4] SPECCPU2006.http://www.spec.org/cpu2006.
[5] Online material about “Definition of Optimization” from wordiq.com.
[6] Online information about “Code Optimization” taken from SEO Consultants Directory.
[7] Handout written by Maggie Johnson about Code Optimization, Handout 20, August 04, 2008.
[8] Online Material about Efficient Code Optimization from Wikipedia, the free encyclopedia.
[9] Online material about Program Optimization from Wikipedia, the free encyclopedia.
[10] Online material from workshop, “Code Warrior Workshop”, volume 15 (1999), issue number 4, by Andrew Downs.
[11] Online material about program Optimization by Paul Hsieh.
[12] Handout about Code Optimization (handout 20) written by Johnson, August 4, 2008.
[13] Online material from workshop, “Code Warrior Workshop”, volume 15 (1999), issue number 4, by Andrew Downs.
[14] Online article about Optimization and Code-Instruction Expansion by NEC Electronics.
[15] Online article about Common Sub-Expression Elimination from Wikipedia, The free Encyclopedia. http://en.wikipedia.org/wiki/Common_subexpression_elimination.
[16] Online material about Modular Programming from Wikipedia, the free encyclopedia.