Strategies Towards Improving Software Code Quality in Computing

Ekbal Rashid[1]
Department of Comp. Sc. & Engg. Siksha 'O' Anusandhan University,Bhubaneshwar, Orissa

Srikanta Patnayak [2]
Department of Comp. Sc. & Engg. Siksha 'O' Anusandhan University,Bhubaneshwar Orissa

Vandana Bhattacherjee [3]
Department of Comp. Sc. & Engg. Birla Institute of Technology, Mesra, Ranchi

Abstract: This paper is an attempt to address the problem of quality code generation in terms of asymptotic time complexity. It aims towards the gradual development of an automated system that can analyze any given code, calculate its time complexity and suggest or take corrective steps towards improvement of the code. The preliminary work has been initiated with the 'c' language and also for programs written in 'c'. However in due course of time, as the research enters newer phases, other programming languages shall be targeted. The work is expected to benefit people from various walks of life. While the ordinary programmer, largely inefficient with tasks such as complexity calculation, will be able to generate quality code, at the same time, the expert will also reduce tedious workload and concentrate on higher productive skills and development tasks. The corporate world will be armed with a new automated technology that will reduce cost of production and the ordinary business people may also benefit by getting quality code from not-so-expert people. The research world may get newer dimensions to work on and the learning community may benefit from learning easy and innovative alternatives of improved code, nevertheless. The work is mainly a compiler building project. It may also be viewed as an attempt towards building of an optimizing tool. The project will consist of three phases – the analysis, the comparison and the suggestive/corrective phases. This software is compiled using Turbo C++ 3.0 and hence it is very compact and standalone, It can be readily deployed on any low configuration system and it would not impact its performance.

Keywords : complexity, compiler, optimization, code-analysis, code-improvement

I .Introduction:

A machine serves its purpose if it fulfils the task of generating quality products in short time. The dream of any engineer is to build machines fitted for this purpose. Researchers have engaged themselves with such tasks and this leads to the development of newer gadgets, devices and machines. Moreover after the advent of computational machines, the task has also been to develop software that can handle different situations. So a newer field of research has developed for software development. This field of research has grown rapidly over the last decade and many sub-branches have emerged. There is the task of developing newer software for newer tasks. Then there is the important area of development of computational languages. Among several others, there is also the need of working with the methodology of software development. This field in particular deals with the notion that what we need should not only be workable software, but should also be efficient. Efficiency amounts to a balance between using least possible space and employing minimum run-time. So algorithms and complexity theories emerged rapidly. Then there also emerged the urge to develop tools for programming and thus reducing the work load and stress of both the adept and ordinary programmer. This work is focused on the last two areas mentioned above. That is, on one hand developing a device that can be used by scientists as well as ordinary programmers to judge and produce software of the best quality, while on the other hand reduce the workload of the man and transfer his burden on the machine.

The rest of the paper is organized as follows: section II gives a overview of the problem description, section III describes the related work. In section IV we describe the significance of study. Section V present Methodology and in section VI conclusion is presented

II. Problem description:

The task of programming has evolved through many stages. One chain contains stages in the development of higher programming languages. The other chain has focused on developing tools for programming. Today programming has reached to a height where the use of GUI tools enables even a non technical person to write programs with comparative ease. A good deal of research is focused on the need to develop advanced GUI tools and also programming languages of higher levels to reduce the workload of the programmer.

However as vices often accompany virtues; the development of GUI tools has confronted us with newer problems, or better call them inadequacies.

1. We have seen GUI tools generate codes that could have well been avoided and perhaps would have never been used if the programmer had chosen to hard-code the program. Extra lines often amount to increase in the complexity of the program. Increased complexity involves greater runtime cost and therefore lacks in quality. Although the principles of data structuring and problem solving have been laid down quite adequately, it appears as if reducing complexity does not appeal greatly to common programmers. They feel more or less satisfied with generating runnable codes. The task of analyzing complexity and improving the quality of software seems to be the task of engineers and scientists at large. So, the observation is that the GUI tools and high level languages have made programming easy for non-engineers and non-scientists, that is, programming, or rather say coding has become easy for the common programmers, but, as they lack the ability to analyze the complexity of programs, their work suffers in quality.
2. The majority of the consumers get their work done by such programmers who care little about the complexity of the program. So what the consumers get for their money is software that can give the desired output, but at the same time be low in efficiency. This results in low market output and the economy directly or indirectly suffers.
3. As far as the corporate are concerned, they have to work with quality products and so have to hire scientists and engineers to analyze and supply quality software that balances most effectively between space and time complexities. This increases the cost of their software ultimately making the benefits from their products lesser.
4. Scientists and engineers are too much engaged analyzing software and testing methods to produce greater quality. They could have done more useful things if relieved from such routine tasks. The society as such would be benefited greatly if they would devote their precious time and energy to more useful research activities.

Thus there arises a necessity to develop something that can automatically improve the quality of any program, whether hard coded or developed using GUI tools. Something needs to be developed that can give the ordinary programmer the power to produce quality code. Just as the GUI tools have made programming easier and have given even the non technical hand power to write program, similarly the device or tool in question can again strengthen this power towards generation of quality code.

III. Related work

Grace Hooper used the term ‘compiler’, perhaps in 1952, when he wrote the first compiler for the A-0 language.[1] John W. Backus led the FORTRAN team, which worked over a span of eighteen years to build what is generally considered to be the first complete compiler in 1957.[2] Friedrich L. Bauer, in 1958 completed the first ALGOL compiler. Probably a FORTRAN program was compiled for different architectures in the mid 1960s.[3]. The first high level language which could work in different architectures was probably COBOL. It was done by a team led by Grace Hopper in 1960.[1] In 1962, Tim Hart and Mike Levin of MIT wrote the first self hosting compiler (written in the language it compiles) for Lisp. The Lisp compiler was written in Lisp.[4, 5] In mid 1950s, Noam Chomsky developed the formalism of context-free grammar that allowed the construction of efficient parsing algorithms. In a paper titled “On the translation of Languages from Left to Right” Donald Knuth in 1965 detailed the construction of the LR parser.[6] In 1969 Frank DeRemer in his PhD thesis published the SLR and the LALR techniques which were considered to be important breakthroughs as the LR(k) translators designed by Knuth were too large to be implemented for computer systems of those days.[7, 8] In an article published in 1986 named "Very fast LR parsing", Thomas Pennello was the first to describe Recursive ascent, a technique that implemented an LALR parser by using mutually recursive functions instead of tables.[9] In 1988, G.H. Roberts advanced the technique further[10] and in 1992, in an article published in the journal Theoretical Computer Science, three scientists Leermakers, Augusteijn, Kruseman Aretz also expanded the same theory further.[10] Several other methods like the LL parsing and the Earley parsing were developed in this period. The other type of compiler constructed since 1960s was the compiler compiler which is basically a parser generator. Then there was the cross compiler which runs in one environment and produces object code for another environment. Then came in the era of computer optimization which aims towards producing quality code. John Cocke and Jacob Schwartz in 1970 published a book titled 'Programming Languages and their Compilers'. The book contains a large number of optimization algorithms.

So we can say that the development of assembly language was an important milestone in this direction. Then there developed many other languages, of which many became very popular, and served the purpose adequately. The origin of the 'c' language was another milestone in the development of compilers. It gave enormous power to the programmer both at the low and the high level, opening a new world altogether. Subsequently compilers were designed for yet higher level languages and this touched a new dimension when the object oriented programming came into existence.

The development of GUI tools and the subsequent research in compiler design followed. Today, researchers are working on developing 3D GUI tools and also speech recognized programming and complicated compilers are being developed. The work in question is in continuity to all this and many other related works. However as there is a continuity, there is also a break. Compiler design here is not only to ease programming but to enhance the quality as well. And in this area perhaps nothing concrete has yet developed.

IV. Significance of Study

The significance can be many sided.

1. The application of mathematics to the development of this software can lead to the development of mathematics as a subject as well. This is because implementation of mathematical methods to achieve the target would definitely enrich the understanding of the mathematical aspects of the issue. Usually development of theoretical and fundamental science precedes the development in technology. However, there have been instances when the development in technological fields gave rise to enrichment of fundamental science and mathematical theories. The present work also has the potential to explore such new avenues.
2. The study of compiler design can be considered from a newer perspective. Since compiler design has more or less been targeted to achieve error free code and the translation of higher languages into lower languages, we can say that this will throw new light altogether upon the theory of compiler writing.
3. This study is also an attempt to explore new data structural designs, which may in future lead to different forms of data structure, hitherto unknown.
4. It may add a new dimension to software engineering concepts, as the software would enable the ordinary programmer generate quality code, the ability of which was confined to only scientist and engineers.
5. The quicker development of quality programs and the implementation would lead to newer commercial and economic equations.
6. The engineers and the scientific community engaged in developing quality code will be relieved and then they will be able to devote their time to more productive activity, introducing whole new areas for research and knowledge.

V. Methodology :

The first step in the development of the software is :
Analysis of code and determining the complexity:
This will consist of the following steps;
step 1. Checking of syntax, compilation and testing its working.

step 2. Identifying the components of the code – type and number.
step 3. Identifying number of iterations, number of recursive calls.
step 3. Calculating the time complexity.
step 4. (at advanced stages) Calculating the space complexity.
step 5. Trying to decide what type of program is being analyzed.

The programming language that is being used is the ‘c’ itself. Till today, the program being developed is able to count the number of comment lines in the program, able to count the lines of code and the number of while and for loops in the program. We are giving the The some snapshots of the results in the Appendix.

VI. Conclusion :

The checking of the complexity is the first step towards building up of this compiler. We rather call it a compiler optimizer or a program optimizer. After the first step has been achieved, the following two steps will be targeted, and the complete compiler may be built up that can suggest or replace the existing code with another one having a superior algorithm.

2. Checking with standard models:
This will consist of the following steps:
step 1. preparing a database of quality code of different types of program.
step 2. check within the existing database whether the code analyzed in stage 1 is better than the possible codes of the same type. The database will consist of the identities of available standard codes along with their complexities and comparison will be made between the complexity calculated in stage 1 and the complexities of the available standard codes.
step 3. if the code analyzed in the stage 1 is of higher complexity than what is available in the database, then flag mark the condition.
step 4. if the code analyzed in the stage 1 is at par or better in efficiency than what is previously available, it would be included in the database.

3. Suggestive/Corrective measures:
This stage would in suggestive form report the result of the analysis and comparisons and if necessary suggest an alternative algorithm or code to produce the desired results.
In the corrective stage, the software would replace the existing code (if lower in efficiency) with the standard code available.

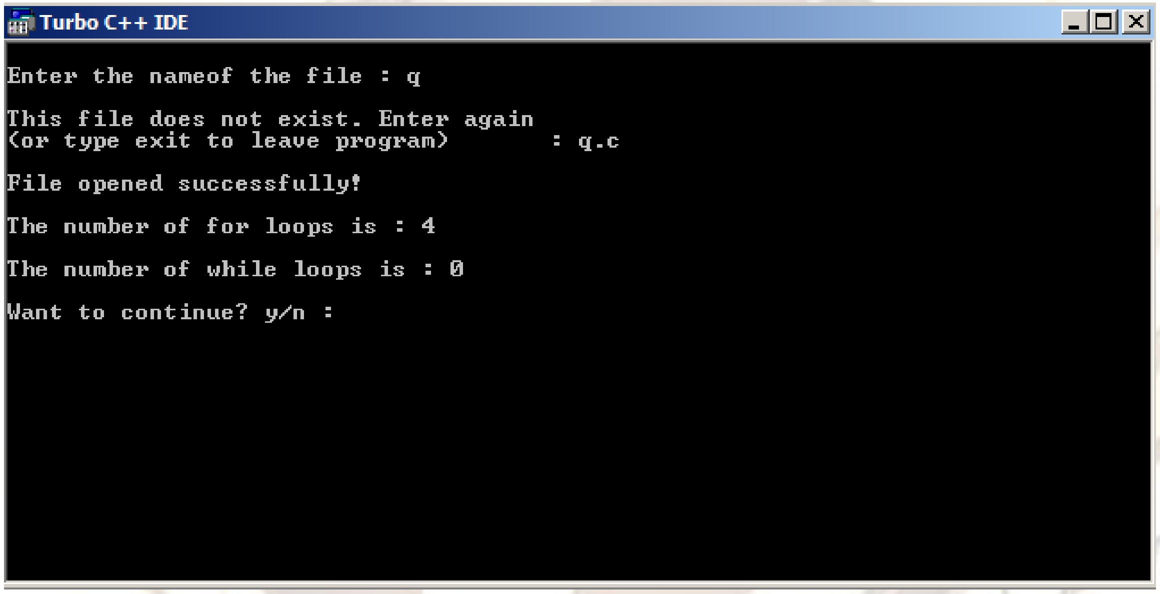
The language used to design the software would be primarily ‘c’ and the file system of ‘c’ would be employed. Later, if necessary, assembly language may be employed.

Reference:

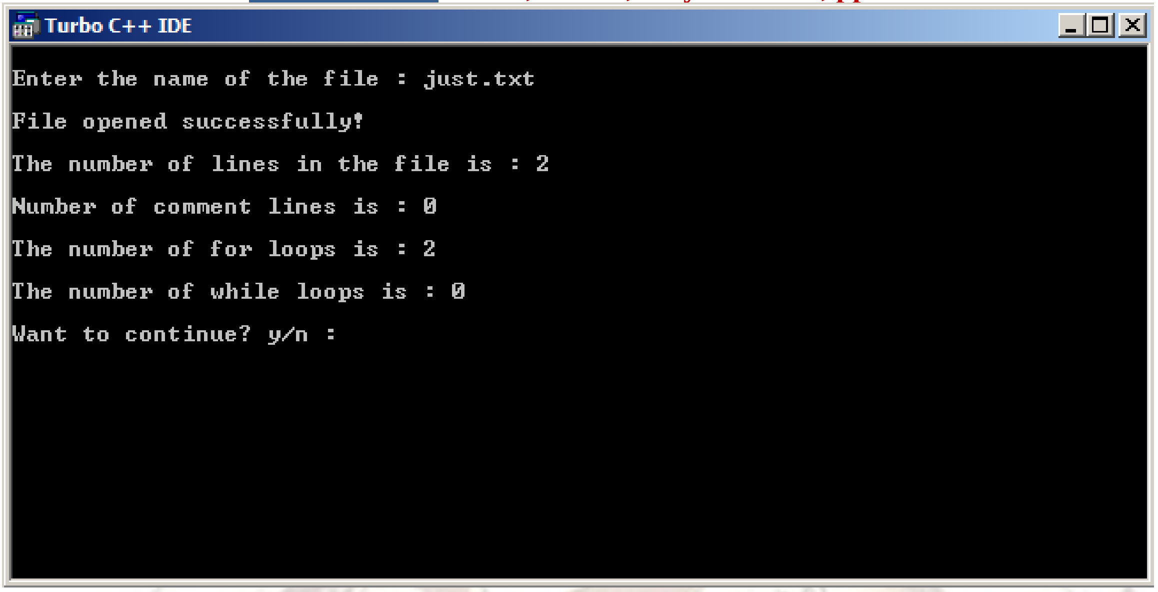
1. http://www.computerhistory.org/events/lectures/cobol_06121997/index.shtml
The world’s first COBOL Compilers
2. Backus et al. “The FORTRAN automatic coding system”, Proc, AFIPS 1957, 188-198
3. Rosen, Saul. ALTAC, FORTRAN, and compatibility. Proceedings of the 1961 16th national ACM meeting
4. T. Hart and M. Levin "The New Compiler", AIM-39 CSAIL Digital Archive - Artificial Intelligence Laboratory Series
5. Tim Hart and Mike Levin. "AI Memo 39-The new compiler". Retrieved 2008-05-23.
6. Knuth, Donald. "On the Translation of Languages from Left to Right". Retrieved 29 May 2011.
7. DeRemer, F. Practical Translators for LR(k) Languages. Ph.D. dissertation, MIT, 1969.
8. DeRemer, F. “Simple LR(k) Grammars,” Communications of the ACM, Vol. 14, No. 7, 1971.

9. Thomas J Pennello (1986). "Very fast LR parsing". *ACM SIGPLAN Notices* **21** (7).
10. G.H. Roberts (1988). "Recursive ascent: an LR analog to recursive descent".

Appendix:



```
Enter the nameof the file : q

This file does not exist. Enter again
(or type exit to leave program)        : q.c

File opened successfully!

The number of for loops is : 4

The number of while loops is : 0

Want to continue? y/n :
```



```
Enter the name of the file : just.txt
File opened successfully!
The number of lines in the file is : 2
Number of comment lines is : 0
The number of for loops is : 2
The number of while loops is : 0
Want to continue? y/n :
```