Static Analysis of Designing a Compiler Tool that Generates Machine Codes for the Predicted Execution Path : *A Bypass Compiler*

Manjunath T.K
Department of Computer Science & Engg.
SEA College of Engg. and Technology
Bangalore, India
manju_mist@rediffmail.com

Sharmila Chidaravalli
Department of Information Science & Engg.
PES Institute of Technology
Bangalore, India
c.sharmila11@gmail.com

***Abstract*—Generally Compilers translate the high level programming languages to optimized machine understandable codes for the entire source program. This paper proposes a new static analysis compiler Tool to perform an effective code optimization by predicting execution path during compilation process. The Proposed Tool identifies the execution path by accepting the user inputs during compilation process itself. Instead of generating the object code for the entire program, this Tool helps the compiler to generate the machine codes only for the predicted execution path. Thus automatically codes are reduced and will be executed faster.**

Keywords-Compilers;optimization ;source program;machine codes; execution path ;Basic Blocks;languages;Tool;Static Analysis;

I. INTRODUCTION

In this competitive world everybody looking for optimized solutions and optimized programming codes with respect to save execution time and space. Generally compilers translate high level programming language to optimized machine understandable language. During the process of compilation the source language undergoes all the phases of compiler step by step.

Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side-effects. The only difference visible to the code's user should be that it runs faster and/or consumes less memory. It is really a misnomer that the name implies you are finding an "optimal" solution— in truth, optimization aims to improve, not perfect the result.

Optimization is the field where most compiler research is done today. The tasks of the front-end (scanning, parsing, semantic analysis) are well understood and unoptimized code generation is relatively straightforward. Optimization, on the other hand, still retains a sizable measure of mysticism. High-quality optimization is more of an art than a science. Compilers for mature languages aren't judged by how well they parse or analyze the code—you just expect it to do it right with a minimum of hassle—but instead by the quality of the object code they produce.

Optimization can be very complex and time-consuming, it often involves multiple subphases, some of which are applied more than once.

Most compilers allow optimization to be turned off to speed up compilation (**gcc** even has specific flags to turn on /off individual optimizations.)

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 of the abstract syntax tree or shrink the number of Three Address Codes instructions. Others are applied as part of final code generation—choosing which instructions to emit, how to allocate registers and when/what to spill, and the like. And still other optimizations may occur after final code generation, attempting to re-work the assembly code itself into something more efficient.

Finally compilers generate optimized machine understandable codes for the entire source program and the same will be loaded for execution, but any executable program consists of number of execution paths. Based on the user inputs any one of execution path will be selected and only those codes will be executed. So it's a waste of loading the entire machine codes which are generated by the compiler. So compilers simply translate the source language statements to machine language formats, but all these statements may not execute at run time because execution also depends on user inputs. During compilation process compiler will not know the user inputs, it cannot predict the execution path of the program, and simply it generates object codes for the entire program.

In this paper we proposed an optimization technique for a compiler tool which generates the machine codes for the predicted execution path by taking user inputs during the compilation process itself using static analysis techniques.

II. PROBLEM STATEMENT

Predict the execution path from CFG, by performing Data Flow analysis on Abstract Syntax Tree during compilation process using static analysis by accepting the user inputs. Then finally front end of a compiler generate intermediate codes for the predicted execution path.

Backend of a compiler will generate the object codes for the same execution path and same codes will be loaded in the memory for execution. Thus program codes are reduced and automatically program occupies less memory.

978-1-61284-486-2/11/$26.00 ©2011 IEEE

A. *Existing System*

Available compilers translate source languages to object codes for all executable paths and it will not predict the execution path of a program during compilation process. It simply checks syntax semantics and does code optimization of the intermediate codes. Finally generates the optimized target object codes for each statements of the source language.

B. *Drawbacks of the Existing System*

Object codes will be generated for each statements of a source language irrespective of which codes are going to execute. Generally program will contain N number of execution paths, in each path there will be M number of Basic blocks. At the time of execution based on the input given by the user only one execution path will be executed, but object codes for N-1 execution paths unnecessarily loaded into memory, it is waste of time and space.

C. *Proposed System*

Proposed system will identify the execution path during compilation process itself by accepting the input from the user and target object codes will be generated for the identified execution path. Proposed system will help the compiler to eliminate the generation of object codes for each statement of a source language. Finally compiler generates the target codes only for predicted execution path and same will be loaded for execution. It will reduce loading time, space and execution time.

D. *Application of the Proposed System*

The proposed system will be applicable for the following applications

- Software development & testing modules.
- Software modules debugging systems.
- Load and go compilers etc.

E. *Limitations*

The proposed system will generates the object codes based on the input, if the user wants to give different inputs the whole program has to be compiled again, it is waste of time.For each time of execution compilation of whole program is compulsory.

III. A BYPASS COMPILER

Compilers bridge source programs in high-level languages with the underlying hardwares. A compiler requires 1) recognizing legitimacy of programs, 2) to generate correct and efficient code, 3) run-time organization, 4) to format output according to assembler or linker conventions. An Existing compiler consists of three main parts: frontend, middle-end, and backend.

Frontend checks whether the program is correctly written in terms of the programming language syntax and semantics. Here legal and illegal programs are recognized. Errors are reported, if any, in a useful way. Type checking is also performed by collecting type information. Frontend generates IR (Intermediate Representation) for the middle-end.

The back end of a compiler generates the actual target codes with respect to the machine architecture and here machine dependent code optimization will be performed, as shown in Figure 1. The generated codes will be executed by the linkers and loaders. Code optimization at both front and back end of a compiler will be performed with respect to instructions level, register level , operands memory level etc. whatever the object codes generated by the compiler, whole codes will be loaded into main memory for execution. But each and every codes of a program may not be executed.

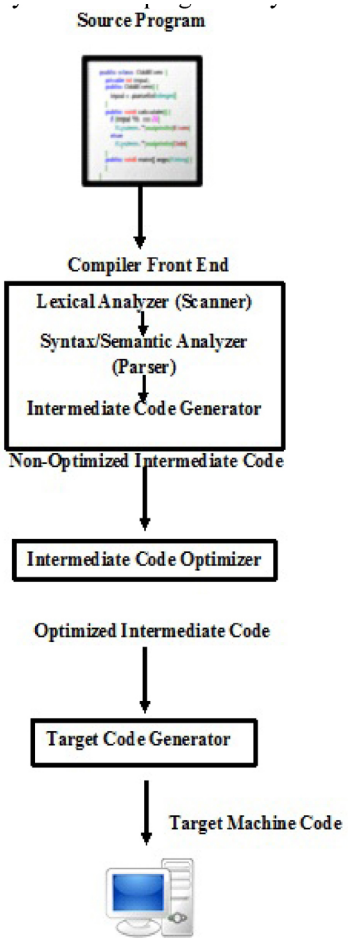

Figure 1. Existing Compiler

Execution of a program mainly depends on machine architecture, machine status and finally the inputs. Inputs to the program mainly decides the execution path, generally a program may have N number of execution path. Inputs decide the execution path, and then computer executes only the instruction which comes under this path. Instructions which come under other execution paths will not be

executed, then loading these codes in the memory it is a waste of time and space.

The proposed system will identify the execution path during compiling process itself, and then compiler generates the object codes only for the identified execution path. These codes will be loaded in to main memory for execution, this will executes faster and requires less memory. This way we can achieve application level optimization as shown in Figure 2.

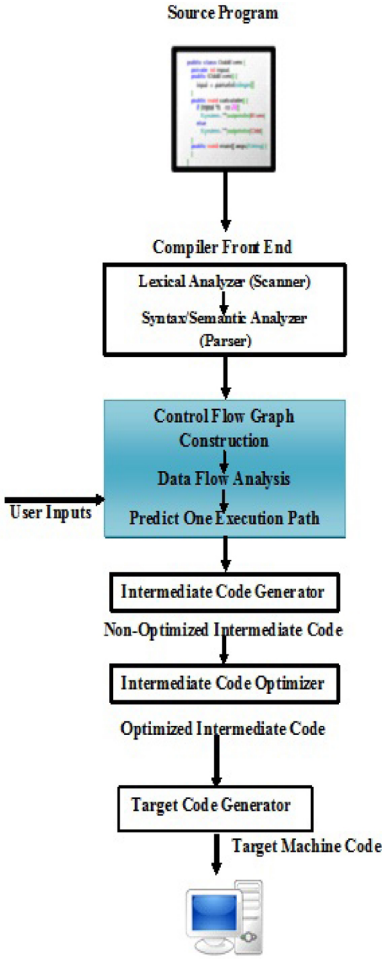

Figure 2. Proposed Tool-A Bypass Compiler

At the end of front end of a compiler one additional phase is added to identify the execution path by accepting the input from the user. By using these inputs it is possible to identify the execution path and creates the intermediate file which contains only one execution path and collections of Basic blocks which comes under this path. Back end of a compiler accept the input from this file and generates the target codes. These codes will be loaded into main memory and it will take less time to load and executes faster because of less codes and less space.

IV. STATIC ANALYSIS OF BYPASS COMPILER

The number of static analysis tools and techniques grows from year to year and this proves the growing interest in static analyzers. The cause of the interest is that the software under development becomes more and more complex and, therefore, it becomes impossible for developers to check the source code manually. This chapter gives a brief description of the static analysis process and analysis techniques for Bypass Compiler.

Static analysis process consists of two steps: Abstract Syntax Tree creation and abstract syntax tree analysis.

In order to analyze source code, a static analysis tool should "understand" the code, that is, parse it and create a structure, describing the code in a convenient form. This form is named abstract syntax tree (referred as AST).

An abstract syntax tree captures the essential structure of the input in a tree form, while omitting unnecessary syntactic details. ASTs can be distinguished from concrete syntax trees by their omission of tree nodes to represent punctuation marks such as semi-colons to terminate statements or commas to separate function arguments. ASTs also omit tree nodes that represent unary productions in the grammar. Such information is directly represented in ASTs by the structure of the tree. ASTs can be created with hand-written parsers or by code produced by parser generators. ASTs are generally created bottom-up.

During the process of software development and testing, particular modules will be selected and the same will be executed, but compilers generate complete object codes for the entire modules of the software, during testing only one module is selected. So remaing modules are simply compiled and loaded in the memory for execution. The proposed Static Analysis tool predicted the execution path during compilation process and helps the compiler to generate object codes only for the execution path. So codes will be reduced automatically.

Consider for example a source code consisting of switch statement to calculate area of a geometrical shape as shown below:

```
1.  switch(exp)
2.  {
3.    case 1:
4.          printf("To find Area of rectangle\n");
5.          printf("enter length and Breadth\n");
6.          scanf("%d%d",&l,&b);
7.          area=l*b;
8.          break;
9.    case 2:
10.         printf("To find Area of circle\n");
11.         printf("enter radius\n");
12.         scanf("%d",&r);
13.         area=3.14*r*r;
14.         break;
15.   case 3:
16.         printf("To find Area of Triangle\n");
17.         printf("enter height and base\n");
```

```
18.             scanf("%d%d",&h,&b);
19.             area=0.5*h*b;
20.             break;
21.     case 4:
22.             printf("To find Area of square\n");
23.             printf("enter length\n");
24.             scanf("%d",&l);
25.             area=l*b;
26.             break;
27.  }
```

Compiler generates object codes for the entire statements for the above considered code, irrespective of which case statement is going to execute (total size of the object program is 1,46kb).Compilers unnecessarilly generates object codes for entire switch case block its simply wastage of time and memory. By analysing the codes statically it is possible to determine executable path by accepting user inputs at the time of compilation and generate the object codes only for the predicted execution path and load the same for execution. This will reduce the time of execution and space effeciently.

A. Static Analysis

There are many different Static Analysis techniques, such as Abstract Syntax Tree (AST) walker analysis, Data Flow Analysis, Path-Sensitive Data Flow Analysis, etc. Concrete implementations of these techniques vary from tool to tool. Static analysis tools for different programming languages can be based on various analysis frameworks. AST walker analysis is performed by walking the AST and checking whether it fulfills the coding standard, specified as a set of rules and guidelines. This is the analysis performed by compilers.

The proposed tool accepts inputs from the user during compilation process itself, Data flow analysis can be performed as a process to collect information about the use, definition, and dependencies of data in programs. The data flow analysis algorithm operates on a control flow graph (CFG), generated from the source code AST.

B. Construction of Control Flow Graph (CFG)

During the process of compilation, compilers will not predict the user inputs, so CFG will be created for all possible execution paths. The CFG and Basic Blocks for the above example C program code is shown in Figure 3. CFG represents all possible execution paths of a given computer program, the nodes represent pieces of code and the edges represent possible control transfers between these code pieces. Since the analysis is performed by accepting the actual inputs from the user during the process of compilation, it is possible to determine the exact output of the program, by predicting the execution path in the control flow graph. In terms of control flow graphs, static analysis means that all possible execution paths are considered to be actual execution paths.

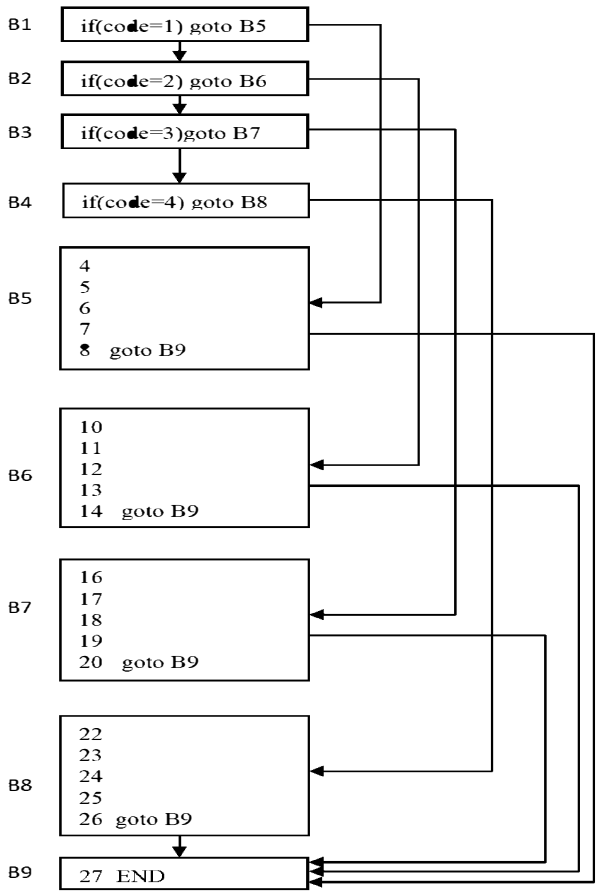

Figure 3. Control Flow Graph for the above considered switch case statement

C. Data Flow Analysis

Perform data flow analysis to predict the path-sensitive execution path of CFG, whether feasible or infeasible path, to a solution by the user inputs. However, programs execute only a small fraction of their potential paths and, moreover, execution time and cost is usually concentrated in a far smaller subset of hot paths. Therefore, it is natural to reduce the analyzed CFG and, therefore, to reduce the amount of calculations so that only subsets of the CFG paths are analyzed.

V. PREDICTED RESULTS

For the example sample considered above i.e. C program fragment, written to calculate Area of a geometrical shape, generally the compiler will generate the object codes for the entire switch case block and size will be approximately **755 bytes**, because compiler will not be knowing the value for the variable “exp”, this variable will be assigned by the value given by the user during execution of the program. Once value has been given by the user, particular case statement will be selected and executed. Remaining case statements unnecessarily were loaded in the memory but

these statements are not executed. To avoid this headache our proposed tool will predict the particular case statement which is going to execute during compilation process by accepting the value for the expression and construct CFG for the executable path and load the same into memory for execution.

Suppose user has selected the case 2, during compilation the value of expression of the switch statement will be 2, so our tool selects case 2 of switch statement and object code is generate for the execution path which contains the basic block B1, B2, B6 and B9 as shown Figure 4.

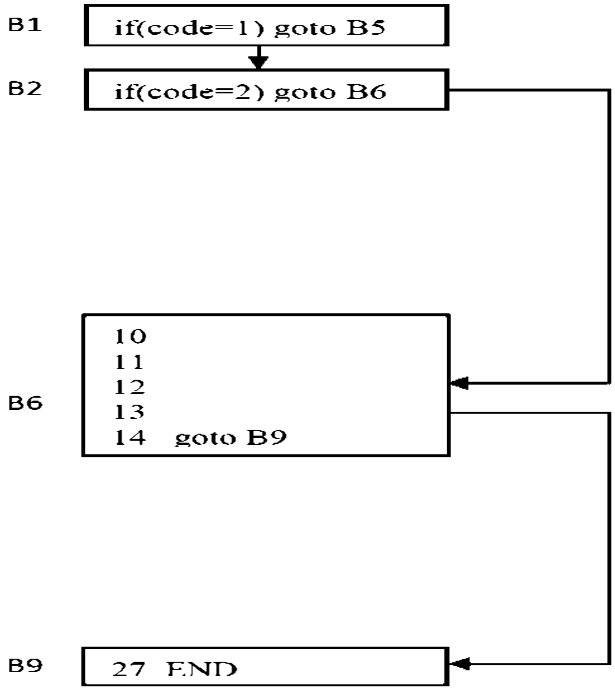

Figure 4. Control Flow Graph for the predicted execution path i.e. case 2 of the switch statement

Object codes will be generated only for case 2 and size of the complete program will be 1010 **bytes**, same object code is going to load in the memory for execution.

case 2:

```
printf("To find Area of circle\n");
printf("enter radius\n");
scanf("%d",&r);
area=3.14*r*r;
break;
```

The snapshot shown in Figure 5 is taken for the predicted execution path of the program. Total size is of the complete program which contains only case 2 block will be 1010 bytes.

No. of bytes reduced=Total Size-Size of execution path

=1496 -1010

=**486 Bytes**

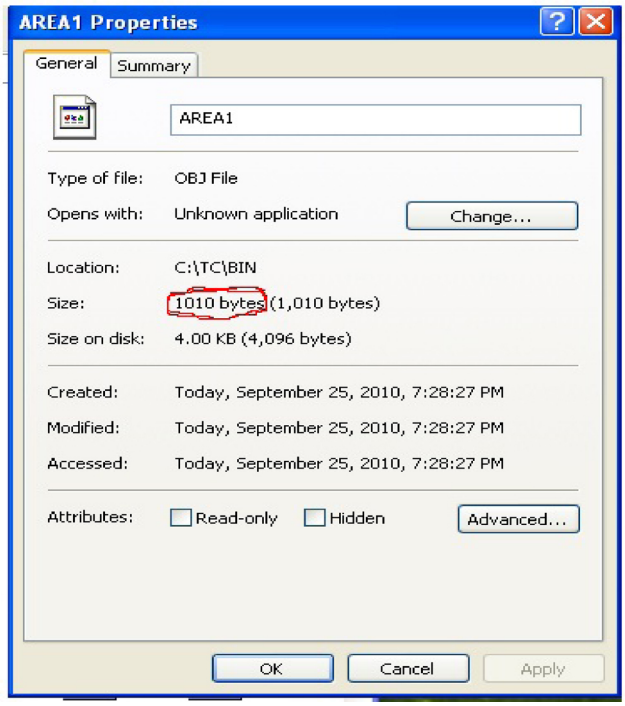

Figure 5. Shows the size of the object code of the case 2 in switch statement

VI. CONCLUSIONS

Static analysis tools and techniques grow from year to year and this proves the growing interest in static analyzers. The cause of the interest is that the software under development becomes more and more complex. The proposed tool based on static analysis and techniques, it predicts the execution path by accepting user inputs during compilation process and generates the object codes for the predicted execution path. Thus automatically reduces the memory size and execution time.

REFERENCES

[1] Dirk Giesen, "Philosophy and practical implementation of static analyzer tools",Dirk Giesen, cop. 1998. -Access mode: http://www.viva64.com/go.php?url=63

[2] Joel Jones, "Abstract syntax tree implementation idioms",Proceedings of the 10th Conference on Pattern Languages of Programs 2003, cop 2003.

[3] Leon Moonen, "A Generic Architecture for Data Flow Analysis to Support Reverse Engineering", Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, cop. 1997.

[4] A. Aho, R. Sethi, J.D. Ullman, "Compilers: Principles, Techniques, and Tools", Reading, MA:Addison-Wesley, 1986.

[5] R. Mak, "Writing Compilers and Interpreters", New York, NY: Wiley, 1991.

[6] S. Muchnick, "Advanced Compiler Design and Implementation" , San Francisco, CA: Morgan Kaufmann, 1997.

[7] A. Pyster, "Compiler Design and Construction", Reinhold, 1988.