Design aspects and trends among the Recent Compilers and Interpreters

Dharmendra Kumar[1]

[1]*M. Tech. Student with Rajiv Gandhi Proudyogiki Vishwavidyalaya, Bhopal*

Abstract: Mapping from high to low i.e. simple mapping of a high level language program to machine or assembly language produces inefficient execution. Higher the level of abstraction more will be the inefficiency towards CPU communication. If not efficient then High-level abstractions are useless. Hence the designers need to provide a high level abstraction with performance of giving low-level instructions. It requires the translation of a high level language program to the corresponding low level or machine level instructions. This research paper discusses the most fundamental and critical design aspects between the Compiler and Interpreters. Compiler construction is a course competence seldom needed in the industry. Yet we claim that compiler construction is a wonderful subject that benefits from virtually all the computer-science topics. In this paper we show in particular why Compiler Construction is a killer example for Interpreted programming languages, providing a unique opportunity for students to understand what it is, what it can be used for, and how it works.

Keywords-- Lexical Analysis, Syntax Analysis, Semantic Analysis, Code Optimization, Target Code, Complexity, CPU, Architecture.

I. INTRODUCTION TO COMPILER

A Compiler is a program (system software) that accepts or reads the statements of a program written in one programming language called the source programs, and if these statements make sense in that programming language, it (compiler) translates that program into statements of a semantically equivalent code in another language. This resultant code produced after translation is called as the target language code or target code. Examples of compiled languages are C, C++, PASCAL, FORTRAN, COBOL, ADA, ALGOL60, etc.

We have the compiler for converting the following source code to the indicated target code:

- Modula-2 to C
- Java to byte code
- C language to assembly language
- COOL to MIPS code

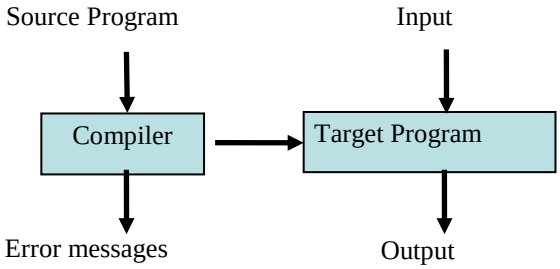

Figure 1: Compilation Process

Compilers generate code that is reasonably fast, but is target specific (it only runs on a particular computer system).

A *compiler* translates a high level language, which is architecture independent, into assembly language, which is architecture dependent. While an *assembler* translates assembly language programs into executable binary codes. For fully compiled languages like C and Fortran, the binary codes are executed directly by the target machine. Java stops the translation at the byte code level. The *Java virtual machine*, which is at the assembly language level, interprets the byte code (hardware implementations of the JVM also exist, in which Java byte codes are executed directly.)

A compiler may stop short of generating actual target code instead and generate some form of assembly to be processed further by a translator and/or the language processor provided with the operating system or bundled with the compiler.

1.1 Target Language

Compilers may generate many types of target codes depending on machine while some compilers make target code only for a specific machine. The target language may be another programming language or the machine language of another computer between a microprocessor and the super computer. It is generally the object code for the target machine i.e. the code in the machine instructions of the computer. The object code, are then linked with standard libraries by the Linker to produce an executable file. As the entire program is converted to machine code, it runs very quickly.

The target program for most of the compilers is normally the:

- equivalent program in machine code – the re-locatable object file
- same machine language that is the target for assemblers

Re-locatable machine code is generally the separately compiled modules of a program. Linker is a program which combines the re-locatable machine code into a form suitable for execution.

Indeed the purpose of compilers is to ease the process of creating the program in machine language, but most of the early compilers and even the modern compilers compile the source program into the Assembly language program first, and then let an assembler to finish the translation to machine language. The example of such an compilers are the C language Compilers. These compilers produce the assembly code as its target language that is passed to an assembler for further processing to translate it into the object code. Other compilers perform the job of the assembler, producing the re-locatable machine code that can be passed directly to the linker/loader.

1.2 Inputs to the Compiler

The input to the Compiler is the standard high level imperative programming language like Java, C, C++ etc. The following constructs from the source program are accepted by the compiler as an input:

(i) State: The state defines the following constructs from the source program:.

- Variables (Local, global, static, register, extern, reference, address, instance etc.)
- Record and Structure
- Class
- Enumeration
- Union
- Array and Lists

(ii) Computation: The computation denotes the following:

- Expressions (arithmetic, logical, relational etc.)
- Assignment statements
- Control flow structures (conditionals, loops etc.)
- Procedures, subroutines, functions, methods

After accepting the input, the Compiler does the translation producing some output as described under.

1.3 Output of the Compiler

The output from the Compiler is a set of low level assembly or machine instructions. Translator generates codes to allocate storage for the variable and uses the address of allocated storage wherever the code references the variable. Compiler produces following constructs as the output, which are occurring in the corresponding object code (or some other code produced by the compiler).

(i) State: This corresponds to the following:

- Registers
- Memory with Flat Address Space (Not necessarily absolute)

(ii) Machine Code/ Target Code: They can be of the following form:

- Load, Store instructions
- Arithmetic, logical operations on registers
- Branch instructions

A compiler involves the six phases as given under:

1. Lexical analysis
2. Syntactic analysis
3. Semantic analysis
4. Intermediate code generation
5. Code optimization
6. Code generation

1.4 General Tasks Performed by a typical Compiler

A Compiler in general performs the following tasks:

- Reads and understands the source program
- Precisely determines what actions it require
- Figure out how to faithfully carry out those actions
- Put the translated instructions for the computer to carry out those actions into a file called as the object file (in most of the cases but excluding some exceptions)

- Cooperate with the debugger for error handling

1.5 Cost of Designing a Compiler

Cost of Designing a Compiler is proportional to the following:

- Complexity of the source code (Lesser be the complexity, more easily the source code be converted into the target code)
- Complexity of the architecture of the target machine (includes addressing modes, instruction set etc.)
- Flexibility of the available instruction set (closeness of the instructions with the machine architecture). An instruction directly invokes some of the hardware of the CPU/machine. An itn directly invokes some of the Hw of the CPU/Machine

1.6 Desirable Features of the Compilers: An Ideal Compiler

We desire the following features from the good compilers:

- Smaller in size
- Better understanding of programming languages
- Correctness - preserve the meaning of the code after translating them
- Takes less time for compilation
- Better speed of compilation (translation) and generation of target code
- Cooperation with the debugger - Good error reporting/handling
- Support for separate compilation
- Written in a high level language
- Produces the target codes that are smaller in size and executes faster
- Portable w. r. t. the machine architecture
- Modular (separate compilation) i.e. entire operation should be divisible into subroutines
- Compilation time is proportional to the size of the program. Hence the time complexity is O(n); where n is the measure of the program size (usually the number of characters)

II. INTRODUCTION TO INTERPRETER

An Interpreter is a translator in that it reads a source program and translates it immediately, just as a human interpreter makes a verbal translation that is hard and understood immediately. An Interpreter executes the source program immediately as it is read, rather than generating the machine dependent object code. An Interpreter bridges an execution gap without generating a machine language program but appears to execute a source program as if it were a machine language.

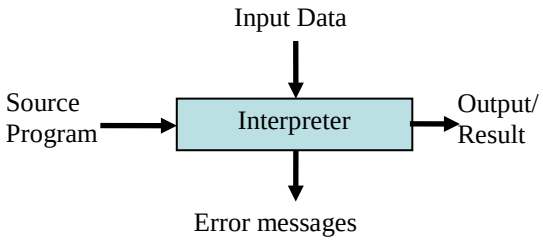

Figure 2: Interpreter

The Interpreter looks at and executes program on a line-by-line basis or one statement at a time rather than producing object code. As opposed to the Interpreter a Compiler is a program which takes some form of source program as input and produces the corresponding code called as the target code in general. Compiled languages can achieve the greater efficiency while Interpreted language can offer a higher degree of flexibility.

A language is interpreted if source code is translated only into an intermediate form, which can't be executed directly but must be interpreted at run time. Typically imperative languages are compiled and typically applicative languages and functional languages tend to be interpreted. Examples of interpreted languages are Unix shells (sh, csh, ksh, etc.), BASIC (Beginners All Purpose Symbolic Instruction Code) and Java (Though Java is both, compiled as well as Interpreted.) BASIC, QBASIC, Perl, JavaScript, Python, (pure) LISTS and (though not applicable) APL and SNOBOL.

An Interpreter generally has three phases (or components.) as given under:

(i) Symbol Table and other Tables: Interpreter holds the information concerning entities in the source program. The other tables are like the compiler.

(ii) Data Store: It contains the values of the data items declared in the program being executed.

(iii) Data Manipulation routines: It contains a routine for every legal data manipulation action in the source language. There are a number of routines.

III. TRANSLATION

3.1 Compiler

A Compiler translates the whole program into the machine code at once. It translates the source code into an executable program that can be run at a later time.

Translation and execution are the separate activities. Process is slow as the compilers are concerned with the target machine language and features. A Compiler converts (translates) the entire program into machine code, only when all the errors are removed. It do not give any feedback until the whole program is compiled and processed.

3.2 Interpreter

Instead of translating, it interprets the source program statements one by one or line by line. It read through source code from a program and, turns (translate) it directly into actions. Here translation and execution are the combined activities. This process is fast as the Interpreter does not have to be concerned with the target machine language and features. It interprets the program statements into an executable form immediately as it (statement) is read, directly without creating the object code. It gives rapid and direct feedback.

IV. TRANSLATION SPEED AND TIME

In one sense, the Interpreter never really completes the translation process. Because the Interpreter does not have to be concerned with the target machine features, it can often process a line of source program much faster than the same code is compiled and interpreted. An Interpreter reads its input program over and over to compute the result but the compiler translates it once.

Compiler takes longer to get the output from the first time a program is run, but subsequent runs by the compiler are faster than that from the Interpreter because no additional translation is required. In contrast the compiler creates a target code which will be executed at a later time. It takes longer to run a program under an Interpreter than to run the compiled code but it (Interpreter) generally takes less time (time only needed by the Interpreter only, excluding the time taken by the programmer) to interpret it than the total time required to compile and run the same program.

V. ANALYSIS PROCESS

Interpreting the code is slower than running the compiled code because the Interpreter must analyze each statement in the program each time it is executed and then perform the desired action whereas the compiled code just performs the action. Both compiler and Interpreter analyze a source program to determine its meaning. An Interpreter might well use the Scanner and Parser like the compiler and then interpret the resulting Abstract syntax tree (AST). Hence each line of the source program sent to the interpreter is scanned, parsed and then executed directly. The next source line is then fetched from memory and the same process is repeated for it, till the entire program has been executed. Hence whenever a program need to be executed repeatedly, the source code has to be interpreted every time.

In case of Compiler the analysis of a statement of the source program is followed by the synthesis process. For any number of run of the same program, analysis is performed only once to generate the target code. While in case of Interpreter, the analysis of a statement of the source program is followed by the actions which implement its meaning. For every run of the same program, analysis is performed the number of times the program is run.

VI. MEMORY REQUIREMENT

In the process of Interpretation, the entire source code needs to be present in memory until the execution is complete, as a result of which the memory space required is more when compared to that of a compiled code. Hence the Interpreter, besides being slow in execution, will require more memory. It will however, permit dynamic changes in the specification as only the changed portion of the program need to be Interpreted.

Access to variables is also slower in an Interpreter because the mapping of Identifiers to storage locations must be done repeatedly at run time rather than at compile time. There are various compromises between the development speed when using an Interpreter and the execution speed when using a compiler.

It is the machine dependent (generally object) code (created after the compilation) which are executed. Hence, neither the source program nor the compiler is required for execution process. Hence during the execution process neither the source program nor the compiler is present inside the memory.

As no target code is created in case of Interpreter, so each time the program is executed, every line of the source program is first analyzed, checked for syntax error and then the code is executed. Hence, both the source program and the Interpreter are needed for the execution process. For this purpose the source program and the interpreter are present inside the memory.

VII. COMPLEXITY

7.1 Compiler

Compiler is a complex program, generally big in size and generates the code for the particular platform. Hence it requires more main memory itself. Hence the development complexity is more. It is less expensive in terms of CPU time, as complete program is subjected to the simultaneous translation and this translated program (generally object code) is executed by the CPU.

7.2 Interpreter

Interpreting is more expensive in terms of CPU time, as each line is subjected to the interpretation cycle which involves the analysis also. Hence CPU basically switches

from the translation to other task, if other task is there. Interpreter is a simple and comparatively smaller program which does not involve the code generation phase. Hence development complexity is low. Platform dependency is almost nil.

VIII. CONCLUSION

The term *Compiler* is generally reserved for more complex languages, where there is no intermediate and direct relationship between the source language and the target language. Most of the compiled languages do not use the line numbers.

Designers involved for writing the Compilers undergo good Software engineering experience. Also the Compiler design involves an application of a wide range of theoretical techniques as: Data Structures, Theory of Computation, Algorithms, System software and Computer Architecture.

IX. FUTURE SCOPE

The future compiler techniques can be used to create novel and enhance existing dependability mechanisms to create a wider range of cost/dependability tradeoffs than is currently available. Similarly, compilers can assist in the area of error detection by expanding the range of errors that can be detected. New compiler techniques, can be used to dynamically guide a system as it makes choices, with cost, dependability, and performance tradeoffs, in response to the occurrence of faults and changes in the environment.

REFERENCES

[1] Stevan S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann
[2] D. M. Dhamdhere. System Programming and Operating Systems. Tata cGraw-Hill
[3] Y.N. Srikant and Priti. Shankar. The Compiler Design Handbook: Optimizations and Machine Code Generation. CRC Press, 2002
[4] PyPyTechnology:http://doc.pypy.org/en/latest/interpreter-optimizations.html#introduction
[5] J. E. Hopcroft, R. Motwani, and J.D. ulman Compilation Techniques. Pearson Education Asisa, India
[6] Self Optimizimg AST: http://www.christianwimmer.at/Publications/Wuerthinger12a/
[7] Optimizing Indirect Branch Prediction Accuracy in Virtual Ma-chine Interpreter https://www.scss.tcd.ie/David.Gregg/papers/toplas05.pdf
[8] Andrew W. Appel. Modern Compiler Implementation in Java. 2nd edition Cambridge University Press
[9] Keith D. Cooper and Linda Torczon. Engineering a Compiler. Morgan Kaufmann
[10] Flemming Neils, Hanne Riis Neilson, and Chris Hankin, Principles of Program Analysis Springer.

AUTHOR'S PROFILE

Dharmendra Kumar has received his Bachelor of Engineering and Technology degree in Computer Science and Engineering from Harcourt Butler Technological Institute (HBTI), Kanpur in the year 2003. At present he is pursuing M.Tech. in Computer Science and Engineering from gargi institute of science & technology, Bhopal (Affiliated to *Rajiv Gandhi Proudyogiki Vishwavidyalaya, Bhopal.*) His area of interest includes Compiler Design, Operating System, System Programming, Computer Architecture, Data Structure and C/C++ programming languages.