A Hybrid Approach of Compiler and Interpreter

Achal Aggarwal, Dr. Sunil K. Singh, Shubham Jain

Abstract— This paper essays the basic understanding of compiler and interpreter and identifies the need of compiler for interpreted languages. It also examines some of the recent developments in the proposed research. Almost all practical programs today are written in higher-level languages or assembly language, and translated to executable machine code by a compiler and/or assembler and linker. Most of the interpreted languages are in demand due to their simplicity but due to lack of optimization, they require comparatively large amount of time and space for execution. Also there is no method for code minimization; the code size is larger than what actually is needed due to redundancy in code especially in the name of identifiers.

Index Terms— compiler, interpreter, optimization, hybrid, bandwidth-utilization, low source-code size.

1 Introduction

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining some very simple commands into a program in what is called machine language. Since this is a tedious and error prone process most programming is, instead, done using a high-level programming language.

Programs are usually written in high level code, which has to be converted into machine code for the CPU to execute it. This conversion is done by either a compiler (or a linker) or an interpreter, the latter generally producing binary code, machine code, that can be processed to be directly executable by computer hardware but compilers will proceed usually by first producing an intermediate binary form called object code. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler and interpreter come in.

2 Compiler

A compiler [1] translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

- Compared to machine language, the notation used by programming languages closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.
- Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

3 The Phases of Compiler

Since writing a compiler is a nontrivial task, it is a good idea to structure the work. A typical way of doing this is to split the compilation into several phases [2] with well-defined interfaces. Conceptually, these phases operate in sequence (though in practice, they are often interleaved), each phase (except the first) taking the output from the previous phase as its input. It is common to let each phase be handled by a separate module. Some of these modules are written by hand, while others may be generated from specifications. Often, some of the modules can be shared between several compilers.

A common division into phases is described below. In some compilers, the ordering of phases may differ slightly, some phases may be combined or split into several phases or some extra phases may be inserted between those mentioned below.

- *Achal Aggarwal is currently pursuing bachelors degree program in computer science and engineering in Bharati Vidyapeeth's College of Engineering, New Delhi, India. E-mail: theachalaggarwal@gmail.com*
- *Dr. Sunil K. Singh is professor at computer science and engineering department in Bharati Vidyapeeth's College of Engineering, New Delhi, India. E-mail: sunil.singh@bharatividyapeeth.edu*
- *Shubham Jain is currently pursuing bachelors degree program in computer science and engineering in Bharati Vidyapeeth's College of Engineering, New Delhi, India. E-mail: spvr.shubham@gmail.com*

Lexical analysis: This is the initial part of reading and analyzing the program text. The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

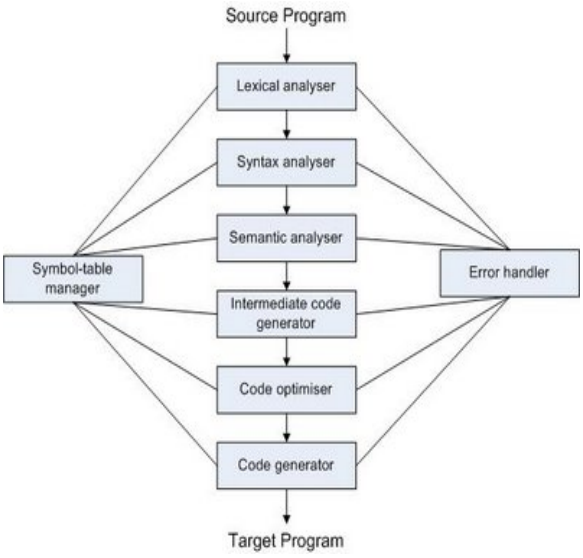
Syntax analysis: This phase takes the list of tokens produced by the lexical analysisand arranges these in a tree-structure (called the syntax tree) that reflects thestructure of the program. This phase is often called parsing.

Type checking : This phase analyses the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared or if it is used in a context that does not make sense given the typeof the variable, such as trying to use a boolean value as a function pointer.

Intermediate code generation: The program is translated to a simple machine independent intermediate language. The symbolic variable names used in the intermediate codeare translated to numbers, each of which corresponds to a register in thetarget machine code.

Machine code generation: The intermediate language is translated to assembly language (a textual representation of machine code) for specific machine architecture.

Assembly and linking: The assembly-language code is translated into binary representationand addresses of variables, functions, etc., are determined.



The first three phases are collectively called the frontend of the compiler and the last three phases are collectively called the backend. The middle part of the compiler is in this context only the intermediate code generation, but this often includes various optimizations and transformations on the intermediate code.

Each phase, through checking and transformation, establishes stronger invariants on the things it passes on to the next, so that writing each subsequent phase is easier than if these have to take all the preceding into account. For example, the type checker can assume absence of syntax errors and the code generation can assume absence of type errors.

Assembly and linking are typically done by programs supplied by the machine or operating system vendor, and are hence not part of the compiler itself, so we willnot further discuss these phases in this book.

4 INTERPRETERS

An interpreter is another way of implementing a programming language. Interpretation shares many aspects with compiling. Lexing, parsing and type-checking are in an interpreter done just as in a compiler. But instead of generating code from the syntax tree, the syntax tree is processed directly to evaluate expressions and execute statements, and so on. An interpreter may need to process the same piece of the syntax tree (for example, the body of a loop) many times and, hence; interpretation is typically slower than executing a compiled program. But writing an interpreter is often simpler than writing a compiler and the interpreter is easier to move to a different machine, so for applications where speed is not of essence, interpreters are often used.

5 COMPILER VS INTERPRETER

A Compiler and Interpreter both carry out the same purpose – convert a high level language (like C, Java) instructions into the binary form which is understandable by computer hardware. They are the software used to execute the high level programs and codes to perform various tasks. Specific compilers / interpreters are designed for different high level languages. However both compiler and interpreter have the same objective but they differ in the way they accomplish their task

Compilation and interpretation may be combined to implement a programming language. The compiler may produce intermediate-level code which is then interpreted rather than compiled to machine code. In some systems, there may even be parts of a program that are compiled to machine code, some parts that are compiled to intermediate code, which is interpreted at runtime while other parts may be kept as a syntax tree and interpreted directly. Each choice is a compromise between speed and space. Compiled code tends to be bigger than intermediate code, which tends to be bigger than syntax, but each step of translation improves running speed.

Using an interpreter is also useful during program development, where it is more important to be able to test a program modification quickly rather than run the program efficiently.

And since interpreters do less work on the program before execution starts, they are able to start running the program more quickly. Furthermore since an interpreter works on a representation that is closer to the source code than is compiled code, error messages can be more precise and informative.

Of course, in the real world there is actually more of a spectrum of possibilities available to the implementer of the language system, lying somewhere between these two poles. Various tradeoffs between compilation speed, runtime speed, space usage, interactivity, and other factors all contribute to a rich spread of implementations in practice.

6 RECENT DEVELOPMENT IN PROPOSED AREA

6.1 PyPy

It is a standard interpreter designed for optimizing the source code written in Python language. One of the advantages – indeed, one of the motivating goals – of the PyPy [3] standard interpreter (compared to CPython) is that of increased flexibility and configurability.

One example of this is that it can provide several implementations of the same object (e.g. lists) without exposing any difference to application-level code. This makes it easy to provide a specialized implementation of a type that is optimized for a certain situation without disturbing the implementation for the regular case.

Most of them are not enabled by default. Also, for many of these optimizations it is not clear whether they are worth it in practice for a real-world application (they sure make some micro benchmarks a lot faster and use less memory, which is not saying too much). Alternative object implementations are a great way to get into PyPy development since you have to know only a rather small part of PyPy to do them.

6.2 Self-Optimizing AST Interpreters

An abstract syntax tree (AST) interpreter [4] is a simple and natural way to implement a programming language. However, it is also considered the slowest approach because of the high overhead of virtual method dispatch. Language implementers therefore define bytecode to speed up interpretation, at the cost of introducing inflexible and hard to maintain bytecode formats. It presents a novel approach to implementing AST interpreters in which the AST is modified during interpretation to incorporate type feedback. This tree rewriting is a general and powerful mechanism to optimize many constructs common in dynamic programming languages. The system is implemented in Java and uses the static typing and primitive data types of Java elegantly to avoid the cost of boxed representations of primitive values in dynamic programming languages.

6.3 Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters

Interpreters [5] designed for efficiency execute a huge number of indirect branches and can spend more than half of the execution time in indirect branch mispredictions. Branch target buffers (BTBs) are the most widely available form of indirect branch predictions; however, their prediction accuracy for existing interpreters is only 2%–50%. There are two methods for improving the prediction accuracy of BTBs for interpreters: replicating virtual machine (VM) instructions and combining sequences of VM instructions into super instructions These techniques can eliminate nearly all of the dispatch branch mispredictions, and have other benefits, resulting in speedups by a factor of up to 4.55 over efficient threaded-code interpreters, and speedups by a factor of up to 1.34 over techniques relying on dynamic super instructions alone.

6.4 Google Clousure Compiler

The Closure Compiler [6] is a tool for making JavaScript download and run faster. Instead of compiling from a source language to machine code, it compiles from JavaScript to better JavaScript. It parses your JavaScript, analyzes it, removes dead code and rewrites and minimizes what's left. It also checks syntax, variable references, and types, and warns about common JavaScript pitfalls. The Closure Compiler reduces the size of your JavaScript files and makes them more efficient, helping your application to load faster and reducing your bandwidth needs.

7 CONCLUSION

It can be concluded that both Compiler and Interpreter have their own usage, merits and demerits. Both can function independently, depending on the type of language it is working on, the usage, the requirements etc. Also it can be said that some phases of compiler like optimization [7] which the interpreter lacks can be worked upon and can be included in the interpreter to get the optimized results with low space usage and greater efficiency. The approach is based on the following research i.e. trying to make a compiler for interpreting languages like java script, python, perl etc. The project aims at optimizing the interpretation process. This includes significant reduction in both size and time complexity.

Most of the interpreted languages are in demand due to their simplicity but due to lack of optimization, they require comparatively large amount of time and space for execution. Also there is no method for code minimization. The code size is larger than what actually is needed due to redundancy in the code especially in the name of identifiers. These problems are not encountered while compiling the code, so we aspire to design a Compreter which adds optimization phases of a compiler in production pipeline of interpreted code and then produces the optimized source code for the interpreted language which will be optimized code in terms of running time

and memory space.

For proof of concept, we intend to make a Compreter for subset of JavaScript [8] and compare the performance with already existing technologies which includes the interpreter for JavaScript, which are already been designed to solve the purpose. In terms of hybrid technology [9] our proposed solution to bridge the gap between compilers and interpreters is surely to increase power of computation at hardware level.

REFERENCES

[1] Dragon Book-Code Optimizations http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/20-Optimization.pdf. 2014.

[2] Compiler construction lecture notes http://www.personal.kent.edu/~rmuhamma/Compilers/compnotes.html

[3] PyPyTechnology:http://doc.pypy.org/en/latest/interpreter-optimizations.html#introduction

[4] Self Optimizimg AST: http://www.christianwimmer.at/Publications/Wuerthinger12a/

[5] Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreter https://www.scss.tcd.ie/David.Gregg/papers/toplas05.pdf

[6] Google Closure Compiler Introduction https://developers.google.com/closure/?csw=1.

[7] Original slides from Computer Systems: A Programmer's Perspective by Randal E. Bryant and David R. Randal E. Bryant and David R. O'Hallaron O'Hallaron "Code Optimization: Machine Independent Optimizations"

[8] How JavaScript compilers work http://creativejs.com/2013/06/the-race-for-speed-part-2-how-javascript-compilers-work/

[9] Sunil Kr. Singh, R. K. Singh, M.P.S. Bhatia "Performance Evaluation of Hybrid Reconfigurable Computing Architecture over Symmetrical FPGA" IJESA, doi:10.5121/ijesa.2012.2312, Vol.2, No.3, page 107-116, September 2012.