JAVA VIRTUAL MACHINE

Manjeet Saini[1], Abhishek Jain[2], Ashish Chauhan[3]

[1, 2, 3]Department Of Computer Science And Engineering, Dronacharya College Of Engineering Khentawas, Farrukh Nagar, Gurgaon, Haryana

ABSTRACT

*This machine or the '***Simulated computer within the computer***' is known as the "***Java Virtual Machine***" or* ***JVM****. It is an abstract machine. It is a specification that provides runtime environment in which java byte code can be executed. JVM loads, verifies and executes the code. It provides a runtime environment. It provides definitions for the Memory area, Class file format, Register set, Garbage-collected heap, fatal error reporting etc. The Java Virtual Machine forms part of a large system, the Java Runtime Environment (JRE). Each operating system and CPU architecture requires a different JRE.*

Keywords: Just-In-Time, Interpreter, Compiler, Ahead Of Time Compilation, Transpiler, Cross Compiler, De-Compiler, Low Level Language, Assembly Language, High Level Language, Object Code, Translator.

I. INTRODUCTION

Java virtual machine is a virtual machine which is used to execute the java byte code files. It is the code execution component of java development kit.

A Java virtual machine (JVM) works as an interpreter. It interprets compiled java byte code also known as java binary code for computer processor or hardware platform so that it can perform java program's instruction. James Gosling introduces JVM concept to make java platform independent language. Java allows programmers to write code that could run on any platform without the need for rewriting and recompilation for each separate platform. Java use just in time compilation, not interpreting to achieve faster speed.

Creating a Virtual Machine within our computer's memory requires building every major function of a real computer down to the very environment within which programs operate. These functions can be broken down into seven basic parts:

- A set of registers
- A stack
- An execution environment
- A garbage-collected heap
- A constant pool
- A method storage area
- An instruction set

II. COMPILER

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code).The most common reason for transforming the source code is to create an executable program. The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a de-compiler. A program that translates between high-level languages is usually called a source-to-source compiler or transpiler. A language rewriter is usually a program that translates the form of expressions without a change of language. More generally, compilers are sometimes called translators.

III. INTERPRETER

Interpreter is a computer program that directly executes or performs instructions written in programming or scripting language without previously compiling them into a machine language program while in java JVM uses JIT for purpose of compiling byte code into machine executable code.

An interpreter generally uses one of the following strategies for program execution:

1. Parse the source code and perform its behavior directly.
2. Translates source code into some efficient intermediate representation and immediately execute this.
3. Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

Source programs are compiled ahead of time and stored as machine independent code, which is then linked at run-time and executed by an interpreter and/or compiler (for JIT systems).

While interpretation and compilation are the two main means by which programming languages are implemented, they are not mutually exclusive, as most interpreting systems also perform some translation work, just like compilers. The terms "interpreted language" or "compiled language" signify that the canonical implementation of that language is an interpreter or a compiler, respectively.

IV. C TO BYTE-CODE COMPILER

From the viewpoint of a compiler, the Java virtual machine is just another processor with an instruction set, Java byte-code, for which code can be generated. The JVM was originally designed to execute programs written in the Java language. However, the JVM provides an execution environment in the form of a byte-code instruction set and a runtime system that is general enough that it can be used as the target for compilers of other languages. Because of its close association with the Java language, the JVM performs the strict runtime checks mandated by the Java specification. That requires C to byte-code compilers to provide their own lax machine abstraction, for instance producing compiled code that uses a Java array to represent main memory (so pointers can be compiled to integers), and linking the C library to a centralized Java class that emulates system calls. Most or all of the compilers listed below use a similar approach.

Several C to byte-code compilers exist:

- Nested-VM translates C to MIPS machine language first before converting to Java byte code.
- Cybil works similarly to Nested-VM but targets J2ME devices.
- LLJVM compiles C to LLVM IR, which is then translated to JVM byte-code.
- C2J is also GCC-based, but it produces intermediary Java source code before generating byte-code supports the full ANSI C runtime. Available as a Win32 binary or as a Java executable.
- Java Backend for GCC, possibly the oldest project of its kind, was developed at The University of Queensland in 1999.
- Javum is an attempt to port the full GNU environment to the JVM, and includes one of the above compilers packaged with additional utilities.
- Compilers targeting Java byte-code have been written for other programming languages, including Ada and COBOL

V. JVM (JAVA VIRTUAL MACHINE)

5.1 Just-In-Time Compilation

Just in time compilation is used by java JVM which is also known as dynamic translation. Just in time compilation is done during execution of a program at run time rather than prior to execution. This consists of translation to machine code which is directly executed. JIT compilation is a combination of two traditional approaches to translation to machine code- ahead of time compilation (AOT) and interpretation and combines some advantages and drawbacks of both. JIT compilation is a form of dynamic compilation and allows adaptive optimization such as dynamic recompilation thus in principle JIT compilation can yield faster execution than static compilation. Interpretation and JIT compilation are particularly suited for dynamic programming languages, as the runtime system can handle late-bound data types and enforce security guarantees. JIT compilation fundamentally uses executable data, and thus poses security challenges and possible exploits. Implementation of JIT compilation consists of compiling source code or byte code to machine code and executing it. This is generally done directly in memory – the JIT compiler outputs the machine code directly into memory and immediately executes it, rather than outputting it to disk and then invoking the code as a separate program, as in usual ahead of time compilation. In modern architectures this runs into a problem due to executable space protection – arbitrary memory cannot be executed, as otherwise there is a potential security hole. Thus memory must be marked as executable; for security reasons this should be done after the code has been written to memory, and marked read-only, as writable/executable memory is a security hole

5.2 AOT (Ahead of Time Compilation)

Dynamic recompilation is a feature of some emulators and virtual machines, where the system may recompile some part of a program during execution. By compiling during execution, the system can tailor the generated code to reflect the program's run-time environment, and potentially produce more efficient code by exploiting information that is not available to a traditional static compiler. Adaptive optimization is a technique in computer science that performs dynamic recompilation of portions of a program based on the current execution profile. With a simple implementation, an adaptive optimizer may simply make a trade-off between Just-in-time compilation and interpreting instructions Ahead-of-time (AOT) compilation is the act of compiling a high-level programming language such as C, or an intermediate language such as Java byte-code, .NET Common Intermediate Language (CIL), IBM System/38 or IBM System "Technology Independent Machine Interface" code, into a native (system-dependent) machine code. Most languages with a managed code runtime that can be compiled to an intermediate language take advantage of just-in-time (JIT). This, briefly, compiles intermediate code into machine code for a native run while the intermediate code is executing, which may decrease an application's performance. Ahead-of-time compilation eliminates the need for this step by performing the compilation before execution rather than during execution.

AOT compilation is mostly beneficial in cases where the interpreter (which is small) is too slow or JIT is too complex or introduces undesirable latencies. In most situations with fully AOT compiled programs and libraries it is possible to drop considerable fraction of runtime environment, thus saving disk space, memory and starting time. Because of this it can be useful in embedded or mobile devices. AOT in most cases produces machine optimized code, just like a 'standard' native compiler. The difference is that AOT transforms the byte-code of an existing virtual machine into machine code. AOT compilers can perform complex and advanced code optimizations which in most cases of JIT will be considered much too costly. On the other hand AOT can't usually perform some optimizations possible in JIT, like runtime profile-guided optimizations, pseudo-constant propagation or indirect/virtual function in lining.

V1. EXECUTION OF INSTRUCTION

The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. These activities are performed in a strict order:

a. Loading: finding and importing the binary data for a type
b. Linking: performing verification, preparation, and (optionally) resolution
c. Verification: ensuring the correctness of the imported type
d. Preparation: allocating memory for class variables and initializing the memory to default values
e. Resolution: transforming symbolic references from the type into direct references.
f. Initialization: invoking Java code that initializes class variables to their proper starting values.

6.1 Startup Delay And Optimizations

JIT typically causes a slight delay in initial execution of an application, due to the time taken to load and compile the byte-code. Sometimes this delay is called "startup time delay". In general, the more optimization JIT performs, the better the code it will generate, but the initial delay will also increase. A JIT compiler therefore has to make a trade-off between the compilation time and the quality of the code it hopes to generate. However, it seems that much of the startup time is sometimes due to IO-bound operations rather than JIT compilation (for example, the rt.jar class data file for the Java Virtual Machine (JVM) is 40 MB and the JVM must seek a lot of data in this contextually huge file). One possible optimization, used by Sun's Hot-Spot Java Virtual Machine, is to combine interpretation and JIT compilation. The application code is initially interpreted, but the JVM monitors which sequences of byte-code are frequently executed and translates them to machine code for direct execution on the hardware. For byte-code which is executed only a few times, this saves the compilation time and reduces the initial latency; for frequently executed byte-code, JIT compilation is used to run at high speed, after an initial phase of slow interpretation. Additionally, since a program spends most time executing a minority of its code, the reduced compilation time is significant. Finally, during the initial code interpretation, execution statistics can be collected before compilation, which helps to perform better optimization. The correct tradeoff can vary due to circumstances. For example, Sun's Java Virtual Machine has two major modes—client and server. In client mode, minimal compilation and optimization is performed, to reduce startup time. In server mode, extensive compilation and optimization is performed, to maximize performance once the application is running by sacrificing startup time. Other Java just-in-time compilers have used a runtime measurement of the number of times a method has executed combined with the byte-code size of a method as a heuristic to decide when to compile. Still another uses the number of times executed combined with the detection of loops. In general, it is much harder to accurately predict which methods to optimize in short-running applications than in long-running ones. Native Image Generator by Microsoft is another approach at reducing the initial delay. Native Image Generator pre-compiles byte-code in a Common Intermediate Language image into machine native code. As a result, no runtime compilation is needed. .NET framework 2.0 shipped with Visual Studio 2005 runs Native Image Generator on all of the Microsoft library DLLs right after the installation. Pre-jitting provides a way to improve the startup time. However, the quality of code it generates might not be as good as the one that is jitted, for the same reasons why code compiled statically, without profile-guided optimization, cannot be as good as JIT compiled code in the extreme case: the lack of profiling data to drive, for instance, inline caching. There also exist Java implementations that combine an AOT (ahead-of-time) compiler with either a JIT compiler (Excelsior JET) or interpreter (GNU Compiler for Java.)

VII. SUMMARY

In a byte-code-compiled system, source code is translated to an intermediate representation known as byte-code. Byte-code is not the machine code for any particular computer, and may be portable among computer architectures. The byte-code may then be interpreted by, or run on, a virtual machine. The JIT compiler reads the byte-codes in many sections (or in full, rarely) and compiles them dynamically into machine language so the program can run faster. Java performs runtime checks on various sections of the code and this is the reason the entire code is not compiled at once. This can be done per-file, per-function or even on any arbitrary code fragment; the code can be compiled when it is about to be executed (hence the name "just-in-time"), and then cached and reused later without needing to be recompiled. In contrast, a traditional interpreted virtual machine will simply interpret the byte-code, generally with much lower performance. Some interpreters even interpret source code, without the step of first compiling to byte-code, with even worse performance. Statically compiled code or native code is compiled prior to deployment. A dynamic compilation environment is one in which the compiler can be used during execution. For instance, most Common Lisp systems have a compile function which can compile new functions created during the run. This provides many of the advantages of JIT, but the programmer, rather than the runtime, is in control of what parts of the code are compiled. This can also compile dynamically generated code, which can, in many scenarios, provide substantial performance advantages over statically compiled code, as well as over most JIT systems. A common goal of using JIT techniques is to reach or surpass the performance of static compilation, while maintaining the advantages of byte-code interpretation: Much of the "heavy lifting" of parsing the original source code and performing basic optimization is often handled at compile time, prior to deployment: compilation from byte-code to machine code is much faster than compiling from source. The deployed byte-code is portable, unlike native code. Since the runtime has control over the compilation, like interpreted byte-code, it can run in a secure sandbox. Compilers from byte-code to machine code are easier to write, because the portable byte-code compiler has already done much of the work.

JIT code generally offers far better performance than interpreters. In addition, it can in some cases offer better performance than static compilation, as many optimizations are only feasible at run-time:

1. The compilation can be optimized to the targeted CPU and the operating system model where the application runs. For example JIT can choose SSE2 vector CPU instructions when it detects that the CPU supports them. However there is currently no mainstream JIT that implements this. To obtain this level of optimization specificity with a static compiler, one must either compile a binary for each intended platform/architecture, or else include multiple versions of portions of the code within a single binary.
2. The system is able to collect statistics about how the program is actually running in the environment it is in, and it can rearrange and recompile for optimum performance. However, some static compilers can also take profile information as input.
3. The system can do global code optimizations (e.g. in lining of library functions) without losing the advantages of dynamic linking and without the overheads inherent to static compilers and linkers. Specifically, when doing global inline substitutions, a static compilation process may need run-time checks and ensure that a virtual call would occur if the actual class of the object overrides the in lined method, and boundary condition checks on array accesses may need to be processed within loops. With just-in-time compilation in many cases this processing can be moved out of loops, often giving large increases of speed.
4. Although this is possible with statically compiled garbage collected languages, a byte-code system can more easily rearrange executed code for better cache utilization.

VIII. FUTURE SCOPE

Continuing porting more of Perl to the JVM via Kawa is the most open area for future work . Currently, only a small subset of Perl is supported, but the path is clear. The task of porting Perl to the JVM becomes much more feasible via the Kawa's architecture. The Kawa/JVM environment can be a real competitor to Microsoft's .NET system. Kawa/JVM system has the added advantage of being completely open and free software, while Microsoft's .NET will remain proprietary. This advantage can surely carry a Kawa/JVM-based language system, along with a Perl port to Kawa/JVM, to success for users and programmers alike.

IX. REFERENCES

[1] Compiler textbook references a collection of references to mainstream Compiler Construction Textbooks.
[2] How the Java Virtual Machine Works by Bikash Shaw.
[3] Compiler textbook references A collection of references to mainstream Compiler Construction Textbooks.
[4] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D. (1986). Compilers: Principles, Techniques, and Tools (1st Ed.). Addison-Wesley. ISBN 9780201100884.
[5] Allen, Frances E. (September 1981). "A History of Language Processor Technology in IBM". IBM Journal of Research and Development (IBM) **25** (5). (Subscription required (help)).

[6] Allen, Randy; Kennedy, Ken (2001). Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers. ISBN 1-55860-286-0.

[7] Appel, Andrew Wilson (2002). Modern Compiler Implementation in Java (2nd Ed.).Cambridge University Press. ISBN 0-521-82060-X.

[8] Apple, Andrew Wilson (1998). Modern Compiler Implementation in ML. Cambridge University Press. ISBN 0-521-58274-1.

[9] Bornat, Richard (1979). Guide. MacMillan. ISBN 0-333-21732-2.

[10] Cooper, Keith D.; Torczon, Linda (2004). Engineering a Compiler. Morgan Kaufmann. ISBN 1-55860-699-8.

[11] Leveret, Bruce W.; Catelli, R. G. G.; Newcomer, Joseph M.; Hobbs, S.O.; Reiner, A.H.; Schatz, B.R.; Wulf, W.A. (August 1980). "An Overview of the Production – Quality Compiler – Compiler Project". Computer (Carnegie-Mellon University) 13 (8): 38–49.Doi: 10.1109/MC.1980.1653748. ISSN 0018-9162. (Subscription required (help)).

[12] McKee-man, William Marshall; Horning, James J.; wort-man, David B. (1970). A Compiler Generator. Englewood Cliffs, NJ: Prentice-Hall. ISBN 0-13-155077-2.

[13] Much-nick, Steven (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers. ISBN 1-55860-320-4.

[14] Scott, Michael Lee (2005). Programming Language Pragmatics (2nd ed.). Morgan Kaufmann. ISBN 0-12-633951-1.

[15] Srikant, Y. N.; Shankar, Priti (2003). The Compiler Design Handbook: Optimizations and Machine Code Generation. CRC Press. ISBN 0-8493-1240-X.