iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 3 **Issue:** X **Month of publication:** October 2015

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)

Survey on JIT Java Compiler

Kamini Patil
Computer department, Vadodara Institute of Engineering College, GTU University

Abstract- This paper I present a survey on JIT Java Compiler. This paper provides an overview of JIT Java compiler and optimization of code. This compiler translates bytecode on demand into native code.
Keywords- JIT Java compiler, bytecode, Java virtual machine, optimizing code, inlining, local optimizations, Control flow optimizations, global optimizations, native code generation

I. INTRODUCTION

Java is a widely used programming language largely due to the portability and machine independent nature of bytecode. These bytecode can be interpreted, compiled to native code or directly executed on a processor whose Instruction Set Architecture is the bytecode specification. Interpreting the bytecode which is the standard implementation of the Java Virtual Machine (JVM) makes execution of programs slow. To improve performance, JIT compilers interact with the JVM at run time and compile appropriate bytecode sequences into native machine code. When using a JIT compiler, the hardware can execute the native code, as opposed to having the JVM interpret the same sequence of bytecode repeatedly and incurring the penalty of a relatively lengthy translation process. This can lead to performance gains in the execution speed, unless methods are executed less frequently. The time that a JIT compiler takes to compile the bytecode is added to the overall execution time, and could lead to a higher execution time than an interpreter for executing the bytecode if the methods that are compiled by the JIT are not invoked frequently. The JIT compiler performs certain optimizations when compiling the bytecode to native code. Since the JIT compiler translates a series of bytecode into native instructions, it can perform some simple optimizations. Some of the common optimizations performed by JIT compilers are data- analysis, translation from stack operations to register operations, reduction of memory accesses by register allocation, elimination of common sub-expressions etc. The higher the degree of optimization done by a JIT compiler, the more time it spends in the execution stage. Therefore a JIT compiler cannot afford to do all the optimizations that is done by a static compiler, both because of the overhead added to the execution time and because it has only a restricted view of the program.

II. RELATED WORK

Previous research has mainly concentrated on analyzing the interpreted behavior of Java execution. Romer investigated the interaction of interpreters with modern architectures and demonstrated that interpreter performance is relatively independent of the application[6]. They also concluded that specialized hardware support for interpreted environments was not essential and that performance improvements could be achieved through software means. Newhall and Miller developed a tool based on a performance measurement model that explicitly represents the interaction between the application and the interpreter. This tool measures the performance of interpreted Java applications and is shown to help application developers tune their code to improve performance. Hsieh, et.al. studied the impact of interpreters and Java compilers on microarchitectural resources such as the cache and the branch predictor. This study is important to provide hints in the development and improvement of Hotspot Java compilers. The HotSpot compilers choose between the JIT and interpreter mode of execution selectively to improve performance.

III. OVERVIEW

The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time. Java programs consists of classes, which contain platform neutral bytecode that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecode into native machine code at run time.
The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecode of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and

memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.

JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

IV. HOW THE JIT COMPILER OPTIMIZES CODE

When a method is chosen for compilation, the JVM feeds its bytecode to the Just-In-Time compiler (JIT). The JIT needs to understand the semantics and syntax of the bytecode before it can compile the method correctly. To help the JIT compiler analyze the method, its bytecode are first reformulated in an internal representation called *trees*, which resembles machine code more closely than bytecode. Analysis and optimizations are then performed on the trees of the method. At the end, the trees are translated into native code. The remainder of this section provides a brief overview of the phases of JIT compilation. For more information, see JIT and AOT problem determination. The JIT compiler can use more than one compilation thread to perform JIT compilation tasks. Using multiple threads can potentially help Java™ applications to start faster. In practice, multiple JIT compilation threads show performance improvements only where there are unused processing cores in the system.

The default number of compilation threads is identified by the JVM, and is dependent on the system configuration. If the resulting number of threads is not optimum, you can override the JVM decision by using the XcompilationThreads option. For information on using this option, see JIT and AOT command-line options.

The compilation consists of the following phases:

Inlining
Local optimizations
Control flow optimizations
Global optimizations
Native code generation

A. *Inlining*

It is the process by which the trees of smaller methods are merged, or "inlined", into the trees of their callers. This speeds up frequently executed method calls.

Two inlining algorithms with different levels of aggressiveness are used, depending on the current optimization level. Optimizations performed in this phase include:

Trivial inlining
Call graph inlining
Tail recursion elimination
Virtual call guard optimizations

B. *Local Optimizations*

Local optimizations analyze and improve a small section of the code at a time. Many local optimizations implement tried and tested techniques used in classic static compilers.

The optimizations include:

Local data flow analyses and optimizations
Register usage optimization
Simplifications of Java™ idioms

C. *Control Flow Optimizations*

Control flow optimizations analyze the flow of control inside a method (or specific sections of it) and rearrange code paths to improve their efficiency.

The optimizations are:

Code reordering, splitting, and removal
Loop reduction and inversion
Loop striding and loop-invariant code motion

Loop unrolling and peeling
Loop versioning and specialization
Exception-directed optimization
Switch analysis

D. Global Optimizations

Global optimizations work on the entire method at once. They are more "expensive", requiring larger amounts of compilation time, but can provide a great increase in performance.
The optimizations are:
Global data flow analyses and optimizations
Partial redundancy elimination
Escape analysis
GC and memory allocation optimizations
Synchronization optimizations

E. Native Code Generation

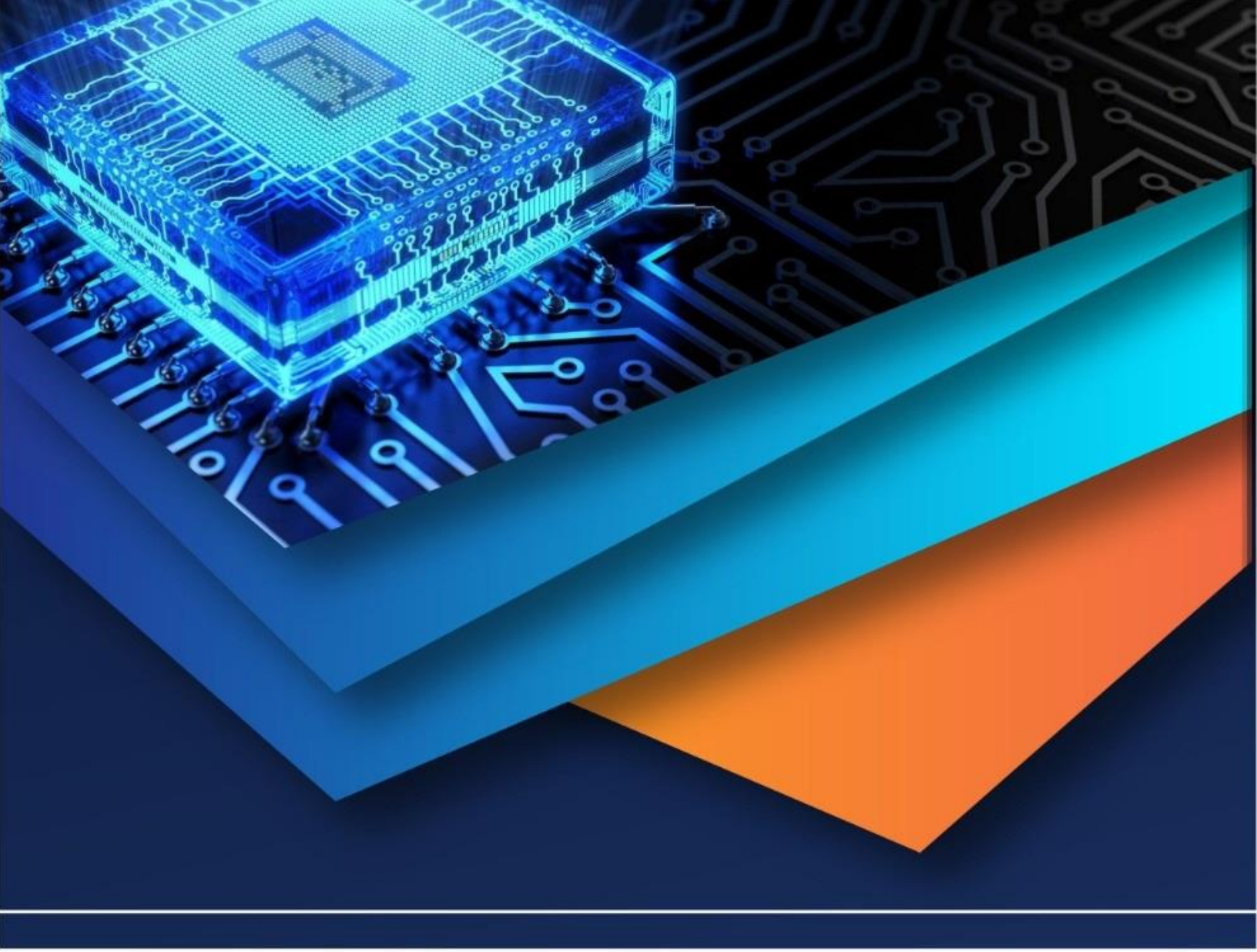
Native code generation processes vary, depending on the platform architecture. Generally, during this phase of the compilation, the trees of a method are translated into machine code instructions; some small optimizations are performed according to architecture characteristics. The compiled code is placed into a part of the JVM process space called the *code cache*; the location of the method in the code cache is recorded, so that future calls to it will call the compiled code. At any given time, the JVM process consists of the JVM executable files and a set of JIT-compiled code that is linked dynamically to the bytecode interpreter in the JVM.

V. CONCLUSION

This paper has provided how Just-in-time compiler works. Just-in-time compiler improves performance of java application. The name itself tells us that it compiler application ‘just in time’.

REFERENCES

[1] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, Reading, Mass. 1996.
[2] R. Jones and R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management,John Wiley & Sons, New York, 1996.
[3] L.P. Deutsch and A.M. Schiffman, ìEf?cient Implementation of the Smalltalk-80 System,î Proc. 11th ACM Symp. Principles of Programming Languages, Assoc. Computing Machinery, New York, 1984, pp. 297302.
[4] A.V. Aho, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley1988.
[5] CaffeineMark(TM) Version 2.01, Pendragon Software, www.webfayre.com/pendragon/cm2/.
[6] W.G. Griswold and P.S. Phillips, UCSD Benchmarks for Java
[7] U. Hˆlzle and D. Ungar, ìA Third-Generation Self Implementation: Reconciling Responsiveness with Performance,î Proc.
[8] ACM OOPSLA (Object-Oriented Programming Systems, Languages, and Applications) 94 Conf.,ACM, 1994, pp. 229-243. [14] C. E. Perkins and P. Bhagwat.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY