Java Compiler Technology and Java Performance*

JI Zhen-yan CHENG Hu

(Institute of Software The Chinese Academy of Sciences Beijing 100080)

E-mail: chenghu@126.com

Abstract This paper summarizes Java's compiler technology, and sorts all kinds of Java compilers into five categories: compilers with interpreter technology, compilers with JIT compiler technology, compilers with adaptive optimization technology, native compilers and translators. Their architectures and working principles are described and analyzed in detail. The authors also analyze the effect that compiler technology has on Java performance.

Key words Java programming language, compiler, performance.

The biggest drawback of Java is low runtime speed. This drawback limits the rapid development of Java. The low runtime speed is mainly caused by Java's compiler architecture. To solve the problem, people adopt many kinds of compiler technology.

1 Compiler System with Interpreter Technology[1~4]

1.1 Architecture and working process

Figure 1 shows the architecture of the Java compiler system with an interpreter. The Java "compiler", javac, translates Java source codes into bytecodes and puts them into a .class file. Java bytecodes can be seen as the machine code instructions for the JVM (Java virtual machine). The bytecodes then run on the JVM. First, JVM loads .class file and verifies the bytecodes. Then after verification, the interpreter begins to execute the bytecodes. As shown in Fig. 1, it runs in a loop, gets one instruction (viz. bytecode) each time and then executes the machine code associated with the instruction.

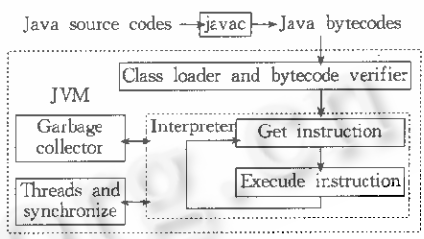

Fig. 1 The architecture

1.2 Advantages

To distinguish the interpreter in JVM from the usual interpreter such as the interpreter of Basic or Lisp, we call the former byte-code interpreter and the latter language interpreter. A byte-code interpreter doesn't need to do tokenizing and parsing(that have been finished in the translation stage with javac), so a byte-code interpreter is faster than a language interpreter.

With this architecture, Java has cross-platform portability. Bytecodes can be run on any computer that has

---

* **JI Zhen-yan** was born in 1972. She received a Ph. D. degree from Institute of Software, the Chinese Academy of Sciences in 1999. Her research interests are programming languages, compilers, localization of software (Chinese), AI and neural network. **CHENG Hu** was born in 1938. He is a professor and doctoral supervisor of the Insititute of Software, the Chinese Academy of Sciences. His current research areas include programming langages, compilers, software engineering, AI and neural network.

Manuscript received 1998-11-27, accepted 1999-06-22.

a JVM. They are machine-independent codes. With the architecture, the compiler system also has a good portability. During transplanting the compiler system, the platform-independent "compiler", javac, doesn't need any modification. What need to be modified are just platform-dependent parts of JVM.

The resulting bytecode programs are more compact than the fully compiled programs because bytecodes are more compact than native codes. The architecture supports Java's dynamic characteristic. It doesn't reduce Java's security.

1.3 Disadvantages

Although translating Java source codes into bytecodes saves the time for tokenizing and parsing, interpreting bytecodes is still much slower than executing the fully-compiled native codes. Additionally, very little optimization can be done during interpreting bytecodes.

2 Compiler System with JIT Compiler Technology[1,2,5,6]

2.1 Architecture and working process

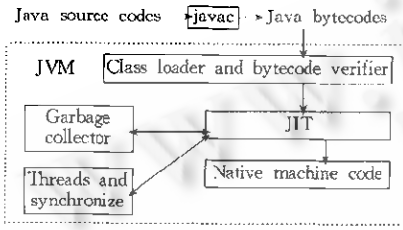

Fig. 2 The architecture

Figure 2 shows the architecture of the Java compiler system with JIT (just in time) technology. A JIT compiler translates incoming bytecodes into the machine instructions for the platform on which it is running, then executes the machine instructions instead of interpreting the bytecodes. The machine instructions are saved in memory. The results of the compilation are not kept between runs. The next time the program is run, the bytecodes are translated into machine codes once again.

2.2 Advantages

A JIT compiler compiles bytecodes rather than executing one instruction at a time, so it can do some optimization. Compared with an interpreter, a JIT compiler improves Java's performance greatly, especially it does very well with computational programs that execute the same series of instructions many times. Programs run much faster with a JIT than with an interpreter——as much as 50 times faster. Additionally, the results of the compilation are not kept between runs, so loading time and storage space are saved.

The architecture still supports Java as a platform-independent, dynamic language. And it also doesn't break up Java's security architecture.

2.3 Disadvantages

A JIT compiler optimizes everything it sees, so it will spend a lot of time in uselessly optimizing initialization code and other methods just executed one time. The time is inevitably wasted.

Since all optimizations must be performed rapidly, the optimization techniques that can be performed are very limited, and the speed-up is also limited.

3 Compiler System with Adaptive Optimization Technology[1,2,4,7]

3.1 Architecture and working process

The representative of the compiler systems with adaptive technology is Sun's HotSpot technology. Its architecture is shown in Fig. 3. HotSpot includes a dynamic compiler and a virtual machine to interpret bytecodes. The bytecodes produced by the Java compiler (javac) are first interpreted in the virtual machine. As they run, the profiler keeps track of performance information. When a method is found to be taking a lot of time, HotSpot compiles and optimizes it. Compiled methods are stored in a cache of native machine code. When a method is invoked, the native machine-code version is used, if it exists. Otherwise, the bytecodes are reinterpreted.

3.2 Advantages

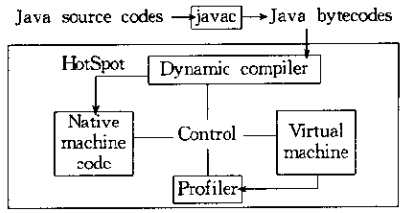

Fig. 3 The architecture

Compared with JIT, HotSpot has more time to optimize codes than JIT because it spends time on less codes (hot spots in the codes). In most cases, HotSpot is faster than JIT technology and much faster than JVM interpretation.

HotSpot's dynamic compiler needn't deal with exceptions and the hard-to-optimize cases, because HotSpot can turn to the interpreter to interpret the bytecodes. So HotSpot can rapidly produce highly optimized codes for the codes that are most likely to run. For a static compiler, it must handle all the unusual cases during optimizing, which is very difficult and time-consuming.

Dynamic compilation can utilize runtime information to find the bottleneck of a program, such as a method invoked a lot of times, and then compile and optimize it. A static compiler can't make such judgments because it can't obtain runtime information. Dynamic compilation can continually make adjustments as a program runs.

Inlining frequently-called small methods is one of the important optimizations performed by HotSpot. A little method spends a large part of its excution time in entering and exiting, while inlining the method will save the overhead of the method call. Inlining saves the run time of a program but increases the size. To gain the most improvements in performance with an acceptable size penalty, HotSpot utilizes runtime information to look for frequently-called methods and inlines them.

The architecture supports Java as a platform-independent, dynamic language. And it doesn't break up Java's security architecture.

3.3 Disadvantages

With dynamic compilation technology, Java's applications are still not up to the performance of a compiled C program. Additionally, it is possible to occur that the dynamic compiler decides to optimize a method when the method finishes executing for the last time, which obviously wastes the time for optimizing.

In some cases such as a 100 percent computing-intensive program, a good JIT compiler will provide better performance than the dynamic compiler because the JIT doesn't wait before optimizing and it optimizes everything preemptively.

Compared with native compilers, the optimization technology adopted by a dynamic compiler is limited because the optimization must be performed rapidly.

4 Compiler System with Native Compiler Technology[1,2,8,9]

Client's and server's requirements are different: clients usually execute Java applets or small applications, while servers usually execute large, sophisticated enterprise application with high performance. Due to Clients' platform variety, Java applets or applications that are downloaded to execute on client-side should be platform-independent, while the applications executed on servers should fully exploit advanced capabilities of server machines.

Forenamed Java compiler architectures all provide platform-independent and dynamic load characteristics. JIT and HotSpot technology are fit for processing client applications. Their speed-ups are limited because the optimization techniques that are available are limited, so they can't meet the performance requirement of sophisticated enterprise server-side applications. To meet the requirements of server-side Java applications, one proven way is to adopt native compiler technology.

4.1 Architecture and working process

According to the difference of the input, we can divide native compilers into two kinds: source code native

compilers and bytecode native compilers. Source code native compilers are high performance compilers that convert Java source codes into static executables, while bytecode native compilers are high performance compilers that convert Java bytecodes into native executables. According to whether they support dynamic loading or not, we can divide native compilers into two kinds: static native compilers and dynamic native compilers. Static native compilers don't support Java's dynamic loading characteristic, while dynamic native compilers do.

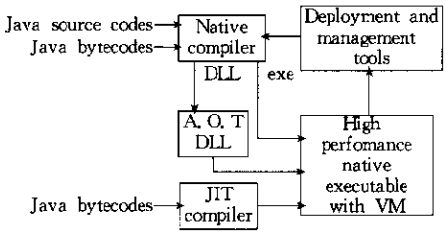

Fig. 4 The architecture

Figure 4 illustrates the architectures of a Java native compiler system. It's a mixed architecture. We can say that it includes a source code native compiler and a bytecode native compiler, or includes a static native compiler and a dynamic native compiler. The input is Java source codes or bytecodes. The output is the combination of the following:

(1) highly optimized native executables;

(2) "Ahead Of Time"(AOT) dynamic load libraries (which are native code in the form of DLLs in Windows or shared libraries in Unix).

To improve performance, static native compilers can be improved in four aspects:

(1) Executing front-end object dispatch analysis;

(2) Adopting machine independent front-end optimization techniques;

(3) Adopting machine-specific back-end optimization techniques;

(4) Improving run-time performance including garbage collection.

Object dispatch analysis is very difficult to implement, but it is very effective because performance can be improved 200% to 400% with it.

Static native compilers have drawbacks. They don't support Java's dynamic characteristic. Programmers have to stop, recompile and reload the applications in order to make changes. Dynamic native compilers can solve the problem. They can support dynamic updating of executables via Java bytecodes or AOT dynamic load libraries. They have three means to perform dynamic compilation, and the three means can be combined to perform on a single system:

(1) fully optimized native code with hooks for possible updating;

(2) system updates via precompiled native code libraries (or DLLs);

(3) JIT-style system updates via bytecode loading.

4.2 Advantages

Compared with JIT compilers, native compilers avoid the overhead of loading bytecodes and on-the-fly compilation time. Because native compilers optimize codes statically before the application is executed, they have enough time to perform a lot of optimizations that can't be performed by JIT or the compilers with adaptive optimization technology. With native compilers, the Java program execution speed could be essentially equivalent to the C++ program.

Native compilers can hot-start applications. With native compilers, an application can start running at full speed because it has been optimized. With HotSpot, the application will start off cold and then dynamically adapt itself and get faster as it locates the hot spots.

Compiling to native form can also protect intellectual property because bytecodes are much easier to be decompiled into source codes.

4.3 Disadvantages

Source code native compilers ruin Java's portability. Native compilers also reduce Java's security.

Additionally, native compilers can't perform dynamic optimizations instructed by runtime information like HotSpot, so it can't adjust speed performance during execution.

5 Compiler System with Translator Technology[1,2,9~11]

5.1 Working process

Many higher-level language compilers, especially C compilers, are very mature, and they can perform various optimizations and yield fully optimized native executables now. To utilize the existing higher-level language optimization compilers to compile Java programs, a translator is needed to translate Java language to the higher-level language. Translators that translate bytecodes to C source codes are typical representatives. The compilation process is shown in Fig. 5. The reason of making bytecodes as the input of translators is to keep Java's portability.

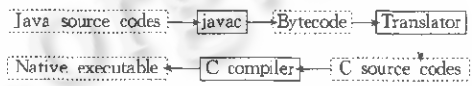

Fig. 5 The compilation process

Translation of a class file (bytecodes) produces a .h file and a .c file. The .h file contains function prototypes, structure definitions, and a class initializer macro. The .c file contains structure initializations and executable method code. The .c file references its own .h file and the .h files of other classes referenced by the class file. Each Java method produces a C function with corresponding parameters. Instance methods also include a parameter corresponding to the instance object. Method names are modified if necessary (to produce names that are legal in C language). Positions on the Java Virtual Machine (JVM) stack are mapped to sets of local variables. Stack operations of the JVM code become assignments in the generated C code. Control flow is handled using conditional and unconditional goto statements. Generated labels correspond to each point of the JVM code that is a jump target. Exceptions are handled using setjmp and longjmp. Caught exceptions are dispatched by a switch statement at the head of the method. The JVM ret operation, actually a computed goto, is also handled using the switch statement. Thread support is built on the operating systems' threads implementation.

Then .c files and .h files generated by a translator are compiled to generate native executable programs by the existing C compilers.

5.2 Advantages

The compiler architecture supports Java's portability. Because it translates Java class files into C source codes, it can make use of C optimizing compiler to perform sophisticated optimization to produce high performance native executables of Java applications, and it can avoid the overhead of interpreting or JIT compiling Java bytecodes by directly generating native executable programs. With the compiler systems, the execution speed of Java applications is close to that of equivalent programs written in C or C++ and the performance requirement of server-side applications can be met. The compiler systems raise performance far above that of compilers with interpreters or JIT. Compiling to native form can also protect intellectual property.

5.3 Disadvantages

The compiler architecture can't compile applets. It does not support dynamic loading, and all needed classes must be linked into the executable file. It ruins Java's security architecture, and reduces Java applications' security. It is inconvenient for debugging Java programs.

6 Discussion

Besides the compilation means adopted by Java, garbage collection and thread synchronization are also important factors that affect Java's performance. Garbage collection and multithread are superiorities of Java over C/C++, but they consume a lot of resource and quantities of runtime. It is estimated that garbage collection

takes up about 20 percent of the average application's run time, while thread synchronization takes up about 20 percent. The time can't be reduced by adopting different compilers and optimization technology[5,7]. To reduce the time spent in garbage collection and thread synchronization, we should design more effective algorithms of garbage collection and thread synchronization, which will make a drastic improvement in Java program's runtime performance.

At present, developers are mainly committed to study the optimization technology during bytecodes' runtime, but bytecodes generated by javac of JDK are not highly optimized codes because there is little optimization such as inlining static, final, private methods to be performed[1]. In fact, highly optimized bytecodes can be generated through adopting all kinds of optimization during generating .class file, and they will improve Java programs' runtime performance greatly.

To sum up, Java is a very promising programming language. Java will have wider and wider application with the improvement of performance.

References

1 David Flanagan. Java in a Nutshell. O'Reilly & Associates, 1996
2 Louden K C. Compiler Construction Principles and Practice. PWS Publishing Company, 1996
3 Lindholm T, Yellin F. The Java Virtual Machine Specification. Addision-Wesley, 1997
4 Armstrong E. HotSpot: a new breed of virtual machine. Javaworld on Line, 1998
5 Java and the Java Runtime Environment. http://www.sun.com/solaris/java/wp-java/2.html
6 Cramer T *et al*. Compiling Java just in time. IEEE Micro, May/June 1997,17(3),36~43
7 Smith G S. Java's new virtual machine. Javaworld on Line. 1997
8 Howard R R. Developing and Deploying Server-hosted Applications with Java. TowerJ's White-paper, 1997
9 Gosling J, Joy B, Steele G. The Java Language Specification. Addision Wesley Publishing Company, 1996
10 Yukio Andoh. j2c, 1996. URL: http://www.webcity.co.jp/info/andoh/java/j2c.html
11 Toba. URL: http://www.cs.arizona.edu/sumatra/toba/

Java 编译程序技术与 Java 性能

冀振燕 程虎

(中国科学院软件研究所 北京 100080)

摘要 概述了 Java 编译程序技术,把 Java 编译程序分成 5 类:具有解释技术的编译程序;具有及时(JIT)编译技术的编译程序;具有自适应优化技术的编译程序;本地编译程序和翻译程序.详细描述和分析了它们的体系结构和工作原理.同时也分析了编译程序技术对 Java 性能的影响.

关键词 Java 程序设计语言,编译程序,性能.

中图法分类号 TP314