CHAPTER 4

Working with the JIT Compiler

The just-in-time (JIT) compiler is the heart of the Java Virtual Machine. Nothing in the JVM affects performance more than the compiler, and choosing a compiler is one of the first decisions made when running a Java application—whether you are a Java developer or an end user. Fortunately, in most situations the compiler needs little tuning beyond some basics.

This chapter covers the compiler in depth. It starts with some information on how the compiler works and discusses the advantages and disadvantages to using a JIT compiler. Then it moves on to which kinds of compilers are present within which versions of Java: understanding this and choosing the correct compiler for a situation is the most important step you must take to make applications run fast. Finally, it covers some intermediate and advanced tunings of the compiler; these tunings can help get those last few percentage points in the performance of an application.

Just-in-Time Compilers: An Overview

Some introductory material first; feel free to skip ahead if you understand the basics of just-in-time compilation.

Computers—and more specifically CPUs—can execute only a relatively few, specific instructions, which are called assembly or binary code. All programs that the CPU executes must therefore be translated into these instructions.

Languages like C++ and Fortran are called compiled languages because their programs are delivered as binary (compiled) code: the program is written, and then a static compiler produces a binary. The assembly code in that binary is targeted to a particular CPU. Complementary CPUs can execute the same binary: for example, AMD and Intel CPUs share a basic, common set of assembly language instructions, and later versions of CPUs almost always can execute the same set of instructions as previous versions of that

CPU. The reverse is not always true; new versions of CPUs often introduce instructions that will not run on older versions of CPUs.

Languages like PHP and Perl, on the other hand, are interpreted. The same program source code can be run on any CPU as long as the machine has the correct interpreter (that is, the program called `php` or `perl`). The interpreter translates each line of the program into binary code as that line is executed.

There are advantages and disadvantages to each of these systems. Programs written in interpreted languages are portable: you can take the same code, drop it on any machine with the appropriate interpreter, and it will run. However, it might run slowly. As a simple case, consider what happens in a loop: the interpreter will retranslate each line of code when it is executed in the loop. The compiled code doesn't need to repeatedly make that translation.

There are a number of factors that a good compiler takes into account when it produces a binary. One simple example of this is the order of the binary statements: not all assembly language instructions take the same amount of time to execute. A statement that adds the values stored in two registers might execute in one cycle, but retrieving (from main memory) the values needed for the addition may take multiple cycles.

Hence, a good compiler will produce a binary that executes the statement to load the data, executes some other instructions, and then—when the data is available—executes the addition. An interpreter that is looking at only one line of code at a time doesn't have enough information to produce that kind of code; it will request the data from memory, wait for it to become available, and then execute the addition. Bad compilers will do the same thing, by the way, and it is not necessarily the case that even the best compiler can prevent the occasional wait for an instruction to complete.

For these (and other) reasons, interpreted code will almost always be measurably slower than compiled code: compilers have enough information about the program to provide a number of optimizations to the binary code that an interpreter simply cannot perform.

Interpreted code does have the advantage of portability. A binary compiled for a SPARC CPU obviously cannot run on an Intel CPU. But a binary that uses the latest AVX instructions of Intel's Sandy Bridge processors cannot run on older Intel processors either. Hence, it is common for commercial software to be compiled to a fairly old version of a processor and not take advantage of the newest instructions available to it. There are various tricks around this, including shipping a binary with multiple shared libraries where the shared libraries execute performance-sensitive code and come with versions for various flavors of a CPU.

Java attempts to find a middle ground here. Java applications are compiled—but instead of being compiled into a specific binary for a specific CPU, they are compiled into an idealized assembly language. This assembly language (know as Java bytecodes) is then run by the `java` binary (in the same way that an interpreted PHP script is run by the

php binary). This gives Java the platform independence of an interpreted language. Because it is executing an idealized binary code, the `java` program is able to compile the code into the platform binary as the code executes. This compilation occurs as the program is executed: it happens "just in time."

The manner in which the Java Virtual Machine compiles this code as it executes is the focus of this chapter.

Hot Spot Compilation

As discussed in Chapter 1, the Java implementation discussed in this book is Oracle's HotSpot JVM. This name (HotSpot) comes from the approach it takes toward compiling the code. In a typical program, only a small subset of code is executed frequently, and the performance of an application depends primarily on how fast those sections of code are executed. These critical sections are known as the hot spots of the application; the more the section of code is executed, the hotter that section is said to be.

Hence, when the JVM executes code, it does not begin compiling the code immediately. There are two basic reasons for this. First, if the code is going to be executed only once, then compiling it is essentially a wasted effort; it will be faster to interpret the Java bytecodes than to compile them and execute (only once) the compiled code.

But if the code in question is a frequently called method, or a loop that runs many iterations, then compiling it is worthwhile: the cycles it takes to compile the code will be outweighed by the savings in multiple executions of the faster compiled code. That trade-off is one reason that the compiler executes the interpreted code first—the compiler can figure out which methods are called frequently enough to warrant their compilation.

The second reason is one of optimization: the more times that the JVM executes a particular method or loop, the more information it has about that code. This allows the JVM to make a number of optimizations when it compiles the code.

A number of those optimizations (and ways to affect them) are discussed later in this chapter, but for a simple example, consider the case of the `equals()` method. This method exists in every Java object (since it is inherited from the `Object` class) and is often overridden. When the interpreter encounters the statement `b = obj1.equals(obj2)`, it must look up the type (class) of `obj1` in order to know which `equals()` method to execute. This dynamic lookup can be somewhat time-consuming.

Registers and Main Memory

One of the most important optimizations a compiler can make involves when to use values from main memory and when to store values in a register. Consider this code:

```
public class RegisterTest {
    private int sum;

    public void calculateSum(int n) {
        for (int i = 0; i < n; i++) {
            sum += i;
        }
    }
}
```

At some point in time, the `sum` instance variable must reside in main memory, but retrieving a value from main memory is an expensive operation that takes multiple cycles to complete. If the value of `sum` were to be retrieved from (and stored back to) main memory on every iteration of this loop, performance would be dismal. Instead, the compiler will load a register with the initial value of `sum`, perform the loop using that value in the register, and then (at an indeterminate point in time) store the final result from the register back to main memory.

This kind of optimization is very effective, but it means that the semantics of thread synchronization (see Chapter 9) are crucial to the behavior of the application. One thread cannot see the value of a variable stored in the register used by another thread; synchronization makes it possible to know exactly when the register is stored to main memory and available to other threads.

Register usage is a general optimization of the compiler, and when escape analysis is enabled (see the end of this chapter), register use is quite aggressive.

Over time, say that the JVM notices that each time this statement is executed, `obj1` is of type `java.lang.String`. Then the JVM can produce compiled code that directly calls the `String.equals()` method. Now the code is faster not only because it is compiled, but also because it can skip the lookup of which method to call.

It's not quite as simple as that; it is quite possible the next time the code is executed that `obj1` refers to something other than a `String`, so the JVM has to produce compiled code that deals with that possibility. Nonetheless, the overall compiled code here will be faster (at least as long as `obj1` continues to refer to a `String`) because it skips the lookup of which method to execute. That kind of optimization can only be made after running the code for a while and observing what it does: this is the second reason why JIT compilers wait to compile sections of code.

Quick Summary

1. Java is designed to take advantage of the platform independence of scripting languages and the native performance of compiled languages.
2. A Java class file is compiled into an intermediate language (Java bytecodes) that is then further compiled into assembly language by the JVM.
3. Compilation of the bytecodes into assembly language performs a number of optimizations that greatly improve performance.

Basic Tunings: Client or Server (or Both)

The JIT compiler comes in two flavors, and the choice of which to use is often the only compiler tuning that needs to be done when running an application. In fact, choosing your compiler is something that must be considered even before Java is installed, since different Java binaries contain different compilers. That will get sorted out in just a bit; first, let's figure out which one should be used in which circumstances.

The two compilers are known as `client` and `server`. These names come from the command-line argument used to select the compiler (e.g., either `-client` or `-server`). JVM developers (and even some tools) often refer to the compilers by the names `C1` (compiler 1, client compiler) and `C2` (compiler 2, server compiler). The names imply that the choice between them should be influenced by the hardware on which the program is running, but that's not really true: especially today, some 15 years after the terms were first utilized, and your "client" laptop has four to eight CPUs and 8 GB of memory (which is more processing power than a midrange server had when Java was first developed).

Compiler Flags Are Different

Unlike most Java flags, the flags to select a compiler are different: most of them do not use `-XX`. The standard compiler flags are simple words: `-client`, `-server`, or `-d64`.

The exception here is tiered compilation, which is enabled with a flag in the common format: `-XX:+TieredCompilation`. Tiered compilation implies that the server compiler must be used. The following command silently turns off tiered compilation, because it conflicts with the choice of the client compiler:

```
% java -client -XX:+TieredCompilation other_args
```