BEN EVANS AND
PETER LAWREY

BIO

BEN EVANS' PHOTOGRAPH BY
JOHN BLYTHE

Part 1

Introduction to JIT Compilation in Java HotSpot VM

Use the PrintCompilation switch to observe the effects of Java HotSpot VM compiling methods during runs.

This article is the first article in a two-part series about Java HotSpot VM and just-in-time (JIT) compilation.

Java HotSpot VM is the VM that Oracle acquired with the Sun acquisition, and it is the VM that forms the basis of both the Java Virtual Machine (JVM) and the open source OpenJDK.

Like all VMs, Java HotSpot VM's role is to provide an operating environment for bytecode. In practice, there are three major functions that need to be performed:

- Executing the instructions and computations that are requested by methods
- Locating, loading, and verifying new types (that is, class loading)
- Managing memory on behalf of application code

The last two functions are huge topics in their own right, so in this article we will focus purely on the execution of code.

JIT Compilation

Java HotSpot VM is a *mixed-mode VM*, which means that it starts off interpreting the bytecode, but it can (on a method-by-method basis) compile code into native machine instructions for faster execution.

BEST BET

Java HotSpot VM works best when it can accumulate enough statistics to make intelligent decisions about what to compile.

By passing the switch -XX:+PrintCompilation, you can see entries in the log file that show each method as it is compiled.

This compilation takes place at runtime—after the method has already been run a number of times. By waiting until the method is actually being used, Java HotSpot VM can make sophisticated decisions about how to optimize the code as it compiles the code.

If you're curious about how much difference the JIT makes, you can turn it off using -Djava.compiler=none and then look at the difference in your benchmarks.

Java HotSpot VM is capable of running in two separate modes: client or server. You can choose the mode by specifying the -client or -server switch to the JVM on startup. (This must be the first switch provided on the command line.) Each mode has different situations in which it is usually preferred. In this article, we'll be concerned only with the server mode.

The major difference between the two modes is that the server mode makes more-aggressive optimizations—based on assumptions that might not always hold. These optimizations are always protected with a simple *guard condition* to check whether the assumption is correct. If, for any reason, an assumption is not valid, Java HotSpot VM reverts the optimization and drops back to interpreted mode. This behavior means that Java HotSpot VM will never do the wrong thing due to an incorrect optimization assumption; it always checks the optimization first.

In server mode, by default, Java HotSpot VM runs a method in interpreted mode 10,000 times before compiling it. You can adjust this value by using the CompileThreshold switch. For example, passing -XX:CompileThreshold=5000 causes Java HotSpot VM to run methods only half as many times before compiling.

It can be tempting for new users to reduce the compile threshold to a very low value. However, you should resist this temptation, because it can reduce performance by spending time compiling methods that the VM doesn't run enough to recover the cost of compilation.

Java HotSpot VM works best when it can accumulate enough statistics to make intelligent decisions about what to compile. If you reduce the compile threshold, Java HotSpot VM might spend a lot of time compiling methods that are not very relevant to the code paths that are run the most. Some optimizations are performed only when enough statistics have been collected, so the code might not be as optimal as it could be if you reduce the compile threshold.

On the other hand, many developers want to achieve the better performance of the compiled mode as soon as possible for their important methods.

One standard way of solving this problem is to have a warm-up harness that exercises the code enough to force compilation (by sending test traffic into the system) after the process has started. For systems such as order management or trading systems, it's important to make sure that the warm-up harness doesn't generate real orders.

Java HotSpot VM provides a number of switches to increase the amount of information logged about JIT compilation. The most common is PrintCompilation—which we already met—but there are several others.

We're going to use PrintCompilation to observe the effects of Java HotSpot VM compiling methods during runs. First, however, we need to say a few words about the System.nanoTime() method for timing.

Timer Methods

Java gives us access to two main time values: currentTimeMillis() and nanoTime(). The former corresponds fairly closely to the time that we observe in the physical world (so-called *wall clock time*). Its resolution is enough for most purposes but not for low-latency applications.

The higher-resolution alternative is the nanosecond timer. This timer measures time in incredibly short intervals. One nanosecond is the time it takes light to move about 20 centimeters in a fiber-optic cable. By contrast, light takes around 27.5 milliseconds to travel between London and New York through fiber-optic cables.

Due to the very high resolution of nanosecond time stamps, uncertainty is inherent in them. Careful handling is required when working with them.

currentTimeMillis(), for example, is usually synchronized between machines reasonably well, and it can be used to measure network latencies, but nanoTime() is not useful between machines.

GET READY, GET SET

Getters and setters are simple methods that are much more expensive if they are not inlined, because the call is more expensive than the field access—a prime candidate for inlining.

To put some of the above theory into practice, we're going to look at a very simple (but extremely powerful) JIT compilation technique.

Method Inlining

One of the key optimizations the JIT compiler (but not javac) does is *method inlining*: copying the body of methods into the methods that call them and eliminating the call. This functionality can be important, because the cost of calling into a trivial method can be expensive compared to the work done in it.

The JIT compiler is able to progressively inline—that is, start by inlining simple methods and then move on to larger and larger blocks of code as other optimizations become possible.

The example shown in **Listing 1**, **Listing 1A**, and **Listing 1B** is a simple test harness that compares the performance of using a field directly or via a getter/setter method.

Getters and setters are simple methods that are much more expensive if they are not inlined, because the call is more expensive than the field access—a prime candidate for inlining.

If you run the test harness using java -cp . -XX:PrintCompilation Main, you can see the difference in performance (see **Listing 2**).

What does all this mean? The first column in **Listing 2** is the number of milliseconds since the program started. The second column is the method ID (for compiled methods) or the itera-

JVM Convergence

Oracle engineers are currently working to merge Java HotSpot VM and Oracle JRockit into a converged offering that leverages the best features of each. Oracle plans to contribute the results of the combined Java HotSpot and Oracle JRockit Java Virtual Machines (JVMs) into OpenJDK. Here are the key points:

- Oracle JRockit and HotSpot will be merged into a single JVM, incorporating the best features from both.
- The converged JVM will be based on HotSpot code, with features from Oracle JRockit included.
- The result will be contributed incrementally to OpenJDK.
- Some existing value-adds, such as those in Oracle JRockit Mission Control, will remain proprietary (and licensed commercially).
- Oracle will continue to distribute free JDK and JRE binaries, which include some closed source items.
- The JVM convergence will be a multiyear process.

For more details about the JVM merge, read "Oracle's JVM Strategy" by Henrik Ståhl, senior director of product management for the Java Platform Group at Oracle. To learn more about HotSpot, visit the OpenJDK HotSpot page. You can see a complete list of JDK Enhancement Proposals, including on the converged JVM, at the JEP Index. To follow development and reviews of the merged JVM, you can join the hotspot-dev@openjdk.java.net mailing list.

—Tori Wieldt

tion count (for the number of iterations performed so far).

Note: The String and UTF_8 classes are not used directly by the test, but compilation output still appears for them because they're used by the platform.

On the second line in **Listing 2**, you can see that both tests are very slow. This is because the first run of the code includes the time to load each class. The next line is much faster even though no code tested is compiled at this stage.

Also note the following:

- At 1,000 and 5,000 iterations, direct access to the fields is faster than via getter/setter methods because the getter and setter have not been inlined or even optimized. Even so, both are pretty fast.
- By 9,000 iterations, the getter is optimized (because it is called twice per loop), which gives a slight overall improvement to performance.
- By 10,000 iterations, the setter has been optimized. The extra time spent including the optimized code means the code briefly runs slower.
- Finally, both test classes are optimized:
  - DFACaller uses direct access to the fields, and GetSetCaller uses the getter and setter. This is the point at which the getter and setter are not just optimized; they are also inlined.
  - You can see that in the next iterations, the test times are still not optimal.
- After 13,000 iterations, the performance for each is as good as the final, much longer test. We've reached steady-state performance.

The important thing to note is that the steady-state performance for accessing fields directly or using getters and setters is basically the same because the methods have (finally) been inlined into GetSetCaller, meaning the callable code in viaGetSet is doing exactly the same work as the code in directCall (which accesses the fields directly).

The JIT compilation is performed in the background, and exactly when each optimization is available for execution varies from machine to machine and, somewhat, from run to run.

Conclusion

In this article, we've shown you the very tip of the JIT compilation iceberg. In particular, we haven't addressed some very important aspects of how to write good benchmarks and how to use statistics to ensure that the dynamic nature of the platform isn't fooling you.

The benchmark used here is very simple, and it isn't suitable for a real benchmark. In Part 2, we plan to show you how to handle a more realistic benchmark and also delve deeper into the code that the JIT compiler produces when it compiles your code. </article>

LEARN MORE

- Java HotSpot VM
- Just-in-time compilation

LISTING 1 / LISTING 1A / LISTING 1B / LISTING 2

```
public class Main {

  private static double timeTestRun(String desc, int runs, Callable<Double> callable)
throws Exception {
    long start = System.nanoTime();
    callable.call();
    long time = System.nanoTime() - start;
    return (double) time / runs;
  }

  // Housekeeping method to provide nice uptime values for us
  private static long uptime() {
    return ManagementFactory.getRuntimeMXBean().getUptime() + 15;
  // fudge factor
  }

  public static void main(String... args) throws Exception {
    int iterations = 0;
    for (int i : new int[]{ 100, 1000, 5000, 9000, 10000, 11000, 13000, 20000,
100000} ) {
      final int runs = i - iterations;
      iterations += runs;

      // NOTE: We return double (sum of values) from our test cases to
      // prevent aggressive JIT compilation from eliminating the loop in
      // unrealistic ways
      Callable<Double> directCall = new DFACaller(runs);
      Callable<Double> viaGetSet = new GetSetCaller(runs);

      double time1 = timeTestRun("public fields", runs, directCall);
      double time2 = timeTestRun("getter/setter fields", runs, viaGetSet);

      System.out.printf("%7d %,7d\t\tfield access=%.1f ns, getter/setter=%.1f ns%n",
uptime(), iterations, time1, time2);
      // added to improve readability of the output
      Thread.sleep(100);
    }
  }
}
```

Press to download source code

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US
Java .net
blog