Writing and interpreting performance benchmarks for dynamically compiled languages, such as Java, is far more difficult than for statically compiled languages like C or C++. In this installment of Java theory and practice, Brian Goetz explores a few of the many ways in which dynamic compilation can complicate performance testing.

I set out this month to write an article dissecting a badly written microbenchmark. After all, we programmers are obsessed with performance, and we all like to understand the performance characteristics of the code we write, use, or criticize. As I occasionally write on the subject of performance, I often get e-mails from people saying, "This program I wrote shows that dynamic frosternation is faster than static blestification, contrary to your last article!" Many of the so-called "benchmark" programs accompanying such e-mails, or the manner in which they were run, show a substantial lack of understanding of how the JVM actually executes Java bytecodes. So before I write the article I set out to write (that'll come in a future column), let's take a look under the hood of the JVM. Understanding dynamic compilation and optimization is the key to understanding how to tell a good microbenchmark (and there are woefully few of these) from the bad ones.

Dynamic compilation -- a brief history

The compilation process for a Java application is different from that of statically compiled languages like C or C++. A static compiler converts source code directly to machine code that can be directly executed on the target platform, and different hardware platforms require different compilers. The Java compiler converts Java source code into portable JVM bytecodes, which are "virtual machine instructions" for the JVM. Unlike static compilers, javac does very little optimization -- the optimizations that would be done by the compiler in a statically compiled language are performed instead by the runtime when the program is executed.

The first generation of JVMs were entirely interpreted. The JVM interpreted the bytecodes rather than compiling them to machine code and executing the machine code directly. This approach, of
course, did not offer the best possible performance, as the system spent more time executing the interpreter than the program it was supposed to be running.

**Just-in-time compilation**

Interpretation was fine for a proof-of-concept implementation, but early JVMs quickly got a bad rap for being slow. The next generation of JVMs used just-in-time (JIT) compilers to speed up execution. Strictly defined, a JIT-based virtual machine converts all bytecodes into machine code before execution, but does so in a lazy fashion: The JIT only compiles a code path when it knows that code path is about to be executed (hence, the name, *just-in-time* compilation). This approach allows the program to start up more quickly, as a lengthy compilation phase is not needed before any execution can begin.

The JIT approach seemed promising, but it had some drawbacks. JIT compilation removed the overhead of interpretation (at the expense of some additional startup cost), but the level of code optimization was mediocre for several reasons. To avoid a significant startup penalty for Java applications, the JIT compiler had to be fast, which meant that it could not spend as much time optimizing. And the early JIT compilers were conservative in making inlining assumptions because they didn't know what classes might be loaded later.

While technically, a JIT-based virtual machine compiles each bytecode before it is executed, the term JIT is often used to refer to any dynamic compilation of bytecode into machine code -- even those that can also interpret bytecodes.

**HotSpot dynamic compilation**

The HotSpot execution process combines interpretation, profiling, and dynamic compilation. Rather than convert all bytecodes into machine code before they are executed, HotSpot first runs as an interpreter and only compiles the "hot" code -- the code executed most frequently. As it executes, it gathers profiling data, used to decide which code sections are being executed frequently enough to merit compilation. Only compiling code that is executed frequently has several performance advantages: No time is wasted compiling code that will execute infrequently, and the compiler can, therefore, spend more time on optimization of hot code paths because it knows that the time will be well spent. Furthermore, by deferring compilation, the compiler has access to profiling data, which can be used to improve optimization decisions, such as whether to inline a particular method call.

To make things more complicated, HotSpot comes with two compilers: the client compiler and the server compiler. The default is to use the client compiler; you can select the server compiler by specifying the `-server` switch when starting the JVM. The server compiler has been optimized to maximize peak operating speed, and is intended for long-running server applications. The client compiler has been optimized to reduce application startup time and memory footprint, employing fewer complex optimizations than the server compiler, and accordingly requiring less time for compilation.

The HotSpot server compiler can perform an impressive variety of optimizations. It can perform many of the standard optimizations found in static compilers, such as code hoisting, common
subexpression elimination, loop unrolling, range check elimination, dead-code elimination, and data-flow analysis, as well as a variety of optimizations that are not practical in statically compiled languages, such as aggressive inlining of virtual method invocations.

Continuous recompilation
Another interesting aspect of the HotSpot approach is that compilation is not an all-or-nothing proposition. After interpreting a code path a certain number of times, it is compiled into machine code. But the JVM continues profiling, and may recompile the code again later with a higher level of optimization if it decides the code path is particularly hot or future profiling data suggests opportunities for additional optimization. The JVM may recompile the same bytecodes many times in a single application execution. To get some insight into what the compiler is doing, try invoking the JVM with the `-XX:+PrintCompilation` flag, which causes the compiler (client or server) to print a short message every time it runs.

On-stack replacement
The initial version of HotSpot performed compilation one method at a time. A method was deemed to be hot if it cumulatively executed more than a certain number of loop iterations (10,000 in the first version of HotSpot), which it determined by associating a counter with each method and incrementing that counter every time a backward branch was taken. However, after the method was compiled, it did not switch to the compiled version until the method exited and was re-entered -- the compiled version would only be used for subsequent invocations. The result, in some cases, was that the compiled version was never used, such as the case of a compute-intensive program, where all the computation is done in a single invocation of a method. In such a situation, the heavyweight method may have gotten compiled, but the compiled code would never be used.

More recent versions of HotSpot use a technique called on-stack replacement (OSR) to allow a switch from interpretation to compiled code (or swapping one version of compiled code for another) in the middle of a loop.

So what does this have to do with benchmarks?
I promised you an article about benchmarks and performance measurement, but so far all you've gotten is a history lesson and a rehash of Sun's HotSpot white papers. The reason for this lengthy detour is that without understanding the dynamic compilation process, it's almost impossible to correctly write or interpret performance tests for Java classes. (Even with a deep understanding of dynamic compilation and JVM optimization, it's still pretty hard.)

Writing microbenchmarks for Java code is far more difficult than for C code
The traditional way to determine if Approach A is faster than Approach B is to write a small benchmark program, often called a microbenchmark. This inclination is perfectly sensible. The scientific method demands no less than an independent investigation. Alas, the devil is in the details. Writing -- and interpreting -- benchmarks is far more difficult and complicated for dynamically compiled languages than for statically compiled ones. There's nothing wrong with trying to learn something about the performance of a given construct by writing programs that use it, but in many cases, microbenchmarks written in Java language don't tell you what you think they do.
With a C program, you can learn a lot about a program's likely performance characteristics without even running it -- simply look at the compiled machine code. The instructions generated by the compiler are the actual machine instructions that are going to be executed, and their timing characteristics are reasonably well understood in the common cases. (There are pathological examples where performance is far worse than would be expected from looking at the machine code because of persistent branch prediction misses or cache misses, but for the most part, you can learn a lot about the performance of a C program by looking at the machine code.)

If the compiler is going to optimize away a block of code because it reasons the code to be irrelevant (a common situation with benchmarks that don't actually do anything), you can see it in the generated machine code -- the code is not there. And you usually don't have to run C code for very long before you can make some reasonable inferences about its performance.

On the other hand, the HotSpot JIT is continuously recompiling Java bytecode into machine code as your program runs, and recompilation can be triggered at unexpected times by the accumulation of a certain amount of profiling data, the loading of new classes, or the execution of code paths that have not yet been traversed in already-loaded classes. Timing measurements in the face of continuous recompilation can be quite noisy and misleading, and it is often necessary to run Java code for quite a long time (I've seen anecdotes of speedups hours or even days after a program starts running) before obtaining useful performance data.

**Dead-code elimination**

One of the challenges of writing good benchmarks is that optimizing compilers are adept at spotting dead code -- code that has no effect on the outcome of the program execution. But benchmark programs often don't produce any output, which means some, or all, of your code can be optimized away without you realizing it, at which point you're measuring less execution than you think you are. In particular, many microbenchmarks perform much "better" when run with `-server` than with `-client`, not because the server compiler is faster (though it often is) but because the server compiler is more adept at optimizing away blocks of dead code. Unfortunately, that dead-code optimization that makes such short work of your benchmark (possibly optimizing it all away) is not going to do quite as well with code that actually does something.

**A bizarre result**

Listing 1 contains a benchmark with a block of do-nothing code, which was taken from a benchmark intended to measure concurrent thread performance, but that instead measures something completely different. (This example was borrowed from the excellent JavaOne 2003 presentation "The Black Art of Benchmarking." See Related topics.)

**Listing 1. Benchmark distorted by unintentionally dead code**

```java
public class StupidThreadTest {
    public static void doSomeStuff() {
        double uselessSum = 0;
        for (int i=0; i<1000; i++) {
            for (int j=0; j<1000; j++) {
                uselessSum += (double) i + (double) j;
            }
        }
    }
}
```
The `doSomeStuff()` method is supposed to give the threads something to do, so we can infer something about the scheduling overhead of multiple threads from the run time of `StupidThreadBenchmark`. However, the compiler can determine that all the code in `doSomeStuff` is dead, and optimize it all away because `uselessSum` is never used. Once the code inside the loop goes away, the loops can go away, too, leaving `doSomeStuff` entirely empty. Table 1 shows the performance of `StupidThreadBenchmark` with client and server. Both JVMs show roughly linear run times with the number of threads, which can be easily misinterpreted to suggest that the server JVM is 40 times faster than the client JVM. In fact, what is happening is that the server compiler does more optimization and can detect that the entirety of `doSomeStuff` is dead code. While many programs do see a speedup with the server JVM, the speedup you see here is simply a measure of a badly written benchmark, not the blazing performance of the server JVM. But it's pretty easy to mistake one for the other if you're not looking carefully.

**Table 1. Performance of StupidThreadBenchmark under client and server JVMs**

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Client JVM run time</th>
<th>Server JVM run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>43</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>435</td>
<td>10</td>
</tr>
<tr>
<td>1000</td>
<td>4142</td>
<td>80</td>
</tr>
<tr>
<td>10000</td>
<td>42402</td>
<td>1060</td>
</tr>
</tbody>
</table>

Dealing with overeager dead-code elimination is a problem for benchmarking statically compiled languages, as well. However, detecting that the compiler has eliminated a large chunk of your benchmark is a lot easier in a statically compiled language. You can look at the generated machine code and see that there's a chunk of your program missing. With dynamically compiled languages, that information is not easily accessible to you.

**Warmup**

If you're looking to measure the performance of idiom X, you generally want to measure its compiled performance, not its interpreted performance. (You want to know how fast X will be in the
field.) To do so requires "warming up" the JVM -- executing your target operation enough times that the compiler will have had time to run and replace the interpreted code with compiled code before starting to time the execution.

With the early JITs and dynamic compilers without on-stack replacement, there was an easy formula for measuring the compiled performance of a method: Run a certain number of invocations of it, start the timer, then execute the method a number of additional times. If the number of warmup invocations exceeded the threshold at which the method was compiled, the actual timed invocations would all be of compiled code, and all of the compilation overhead would be incurred before you start timing.

With today's dynamic compilers, it's a lot harder. The compiler runs at less predictable times, the JVM switches from interpreted to compiled code at will, and the same code path may be compiled and recompiled more than once during a run. If you don't account for the timing of these events, they can seriously distort your timing results.

Figure 1 shows some of the possible timing distortions due to unpredictable dynamic compilation. Let's say you're going to time 200,000 iterations through a loop, and the compiled code is 10 times faster than the interpreted code. If compilation kicks in at 200,000 iterations, you only measure interpreted performance (timeline (a)). If compilation kicks in at 100,000 iterations, your total run time is the time to execute 200,000 interpreted iterations, plus the compilation time (which you don't want to be including), plus the time to execute 100,000 compiled iterations (timeline (b)). If it kicks in at 20,000 iterations, it will be 20,000 interpreted iterations, plus compilation time, plus 180,000 compiled iterations (timeline (c)). Because you don't know when the compiler will run or for how long, you can see how your measurements could be badly distorted. Depending on the compilation time and how much faster the compiled code is than the interpreted code, small changes in the number of iterations can result in big differences in measured "performance."

Figure 1. Performance measurement distortions due to timing of dynamic compilation.

A

B

C

So, how much warmup is enough? You don't know. The best you can do is run your benchmarks with -XX:+PrintCompilation, observe what causes the compiler to kick in, then restructure your...
benchmark program to ensure that all of this compilation occurs before you start timing and that no further compilation occurs in the middle of your timing loops.

**Don't forget garbage collection**

So, you've seen that if you want to get accurate timing results, you must run the code to be tested even more times than you think to allow for JVM warmup. On the other hand, if the test code does any object allocation at all (and nearly all code does), it's going to create garbage, and eventually, the garbage collector is going to have to run. This is another element that can badly distort timing results -- a small change in the number of iterations could mean the difference between no garbage collection and one garbage collection, skewing the "time per iteration" measurement.

If you run your benchmarks with `-verbose:gc`, you can see how much time was spent in garbage collection and adjust your timing data accordingly. Even better, you can run your program for a long, long time, ensuring that you trigger many garbage collections, more accurately amortizing the allocation and garbage collection cost.

**Dynamic deoptimization**

Many standard optimizations can only be performed within a "basic block," and so inlining method calls is often important to achieve good optimization. By inlining method calls, not only is the method call overhead eliminated, but it gives the optimizer a larger basic block to optimize, with substantial opportunity for dead-code optimizations.

Listing 2 shows an example of the type of optimization that is enabled through inlining. The `outer()` method calls `inner()` with an argument of `null`, which will result in `inner()` doing nothing. But by inlining the call to `inner()`, the compiler can see that the `else` branch of `inner()` is dead code, and can optimize the test and the `else` branch away, at which point it can optimize away the entirety of the call to `inner()`. Had `inner()` not been inlined, this optimization would not have been possible.

**Listing 2. How inlining can lead to better dead-code optimization**

```java
public class Inline {
    public final void inner(String s) {
        if (s == null)
            return;
        else {
            // do something really complicated
        }
    }

    public void outer() {
        String s=null;
        inner(s);
    }
}
```

Inconveniently, virtual functions pose an impediment to inlining, and virtual function calls are more common in Java language than in C++. Let's say the compiler is trying to optimize the call to `doSomething()` in the following code:
The compiler can't necessarily tell from this fragment of code which version of `doSomething()` will be executed -- will it be the version implemented in class `Foo`, or in a subclass of `Foo`? There are a few cases where the answer is obvious -- such as if `Foo` is `final` or `doSomething()` is defined as a `final` method in `Foo` -- but most of the time, the compiler has to guess. With a static compiler that only compiled one class at a time, we'd be out of luck. But a dynamic compiler can use global information to make a better decision. Let's say that there are no loaded classes that extend `Foo` in this application. Now the situation is more like the case where `doSomething()` was a `final` method in `Foo` -- the compiler can convert the virtual method call into a direct dispatch (already an improvement) and, further, has the option to inline `doSomething()`, as well. (Converting a virtual method call to a direct method call is called `monomorphic call transformation`.)

Wait a second -- classes can be loaded dynamically. What happens if the compiler makes such an optimization, and later a class is loaded that extends `Foo`? Worse, what if this is done inside the factory method `getFoo()`, and `getFoo()` then returns an instance of the new `Foo` subclass? Won't the generated code then be incorrect? Yes, it will. But the JVM can figure this out, and will invalidate the generated code that is based on the now-invalid assumption and revert to interpretation (or recompile the invalidated code path).

The result is that the compiler can make aggressive inlining decisions to achieve higher performance, then back out those decisions later if they are no longer based on valid assumptions. In fact, this optimization is so effective that adding the `final` keyword to methods that are not overridden (a performance trick advised in the early literature) does very little to enhance actual performance.

A bizarre result

Listing 3 contains a code pattern that combines improper warmup, monomorphic call transformation, and deoptimization, yielding totally meaningless, but easily misinterpreted, results:

**Listing 3. Test program's results are distorted by monomorphic call transformation and subsequent deoptimization**
public double operate(double d) {
    return d + 2.0 - 1.0;
}

public static void runABunch(Operator op) {
    long start = System.currentTimeMillis();
    double d = 0.0;
    for (int i = 0; i < 5000000; i++)
        d = op.operate(d);
    long end = System.currentTimeMillis();
    System.out.println("Time: " + (end-start) + " ignore:" + d);
}

public static void main(String[] args) {
    Operator ra = new RoundaboutAdder();
    runABunch(ra); // misguided warmup attempt
    runABunch(ra);
    Operator sa = new SimpleAdder();
    Operator da = new DoubleAdder();
    runABunch(sa);
    runABunch(da);
}

StupidMathTest first tries to do some warmup (unsuccessfully), then measures the run time for SimpleAdder, DoubleAdder, and RoundaboutAdder, with the results shown in Table 2. It looks like it's much faster to add 1 to a double by adding 2, and then subtracting 1, than simply adding 1 directly. And it's marginally faster to add 0.5 twice than to add 1. Could that possibly be? (Answer: No.)

Table 2. Meaningless and misleading results of StupidMathTest

<table>
<thead>
<tr>
<th>Method</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleAdder</td>
<td>88ms</td>
</tr>
<tr>
<td>DoubleAdder</td>
<td>76ms</td>
</tr>
<tr>
<td>RoundaboutAdder</td>
<td>14ms</td>
</tr>
</tbody>
</table>

What happened here? Well, after the warmup loop, RoundaboutAdder and runABunch() had been compiled, and the compiler did monomorphic call transformation on Operator and RoundaboutAdder, so the first pass ran quite quickly. In the second pass (SimpleAdder), the compiler had to deoptimize and fall back to virtual method dispatch, so the second pass reflects the slower execution due to not being able to optimize away the virtual function call, and the time spent recompiling. In the third pass (DoubleAdder), there was less recompilation, so it ran faster. (In reality, the compiler is going to do constant folding on RoundaboutAdder and DoubleAdder, generating exactly the same code as SimpleAdder. So if there's a difference in run time, it's not because of the arithmetic code.) Whichever one ran first will be the fastest.

So, what can we conclude from this "benchmark?" Virtually nothing, except that benchmarking dynamically compiled languages is much more subtle than you might think.

Conclusion

The results in the examples here were so obviously wrong that it was clear something else must have been going on, but smaller effects can easily bias the results of your performance test.
programs without triggering your "There must be something badly wrong here" detector. And while the ones presented here are common sources of microbenchmark distortions, there are many others. The moral of the story: You're not always measuring what you think you're measuring. In fact, you're usually not measuring what you think you're measuring. Be very leery of the results of any performance measurement that does not involve a realistic program load over a long period of time.
Related topics

- This white paper details some of the tricks used to improve performance in the HotSpot JVM.
- This section on on-stack replacement explains the difference between early HotSpot compilation and current HotSpot JVMs.
- This section on inlining gives some details on monomorphic call transformation, deoptimization, and speculative virtual method inlining.
- This white paper goes into greater detail on the architecture of the HotSpot JVM.
- This piece offers an overview of the architecture of the IBM JIT compiler.