Dynamic Compilation

By Mohit Kumar

Dynamic compilation -- a brief history

- 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.

Dynamic compilation -- a brief history

- 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

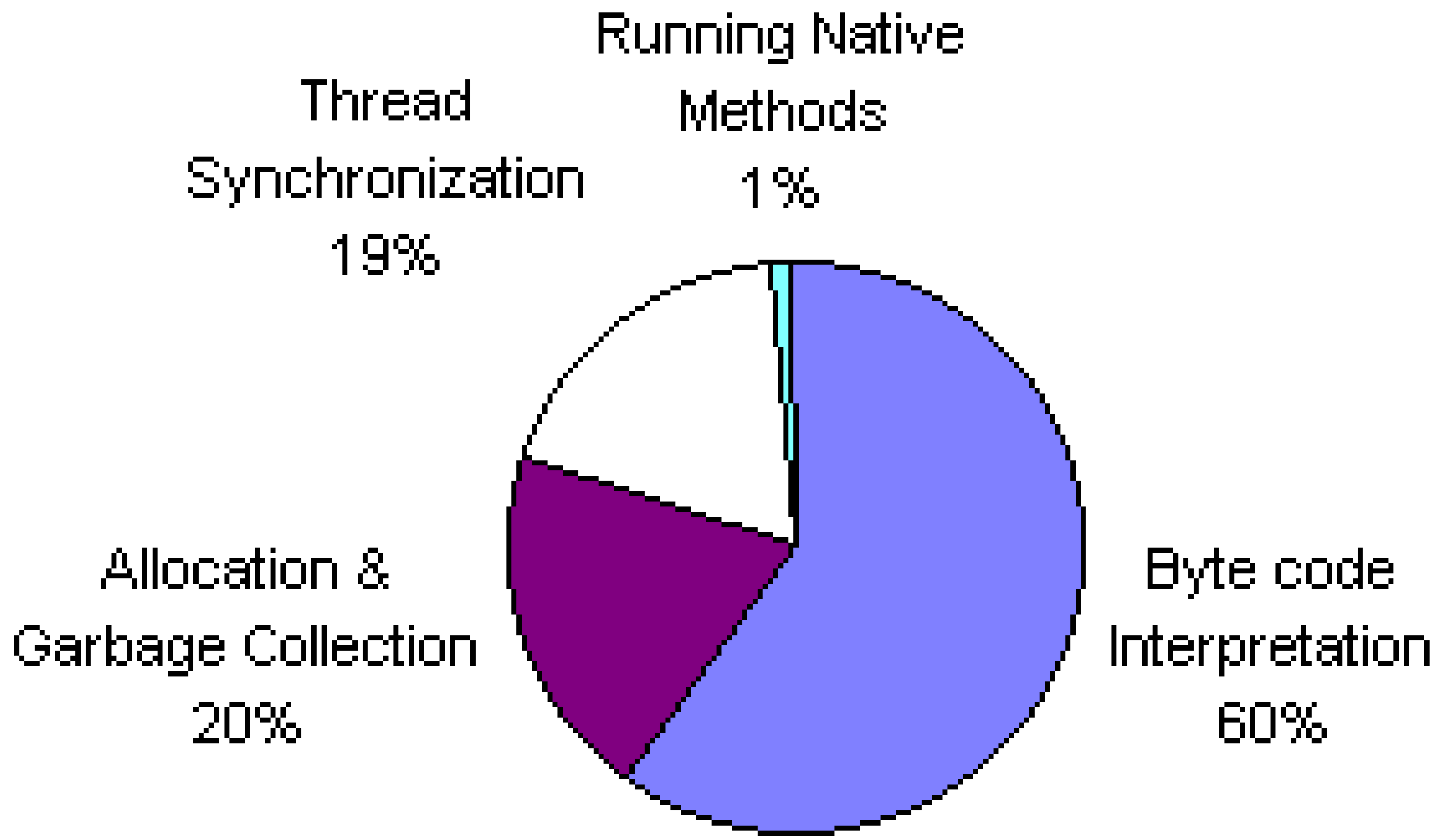
Dynamic compilation -- a brief history

- **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.

Dynamic compilation -- a brief history

- 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.

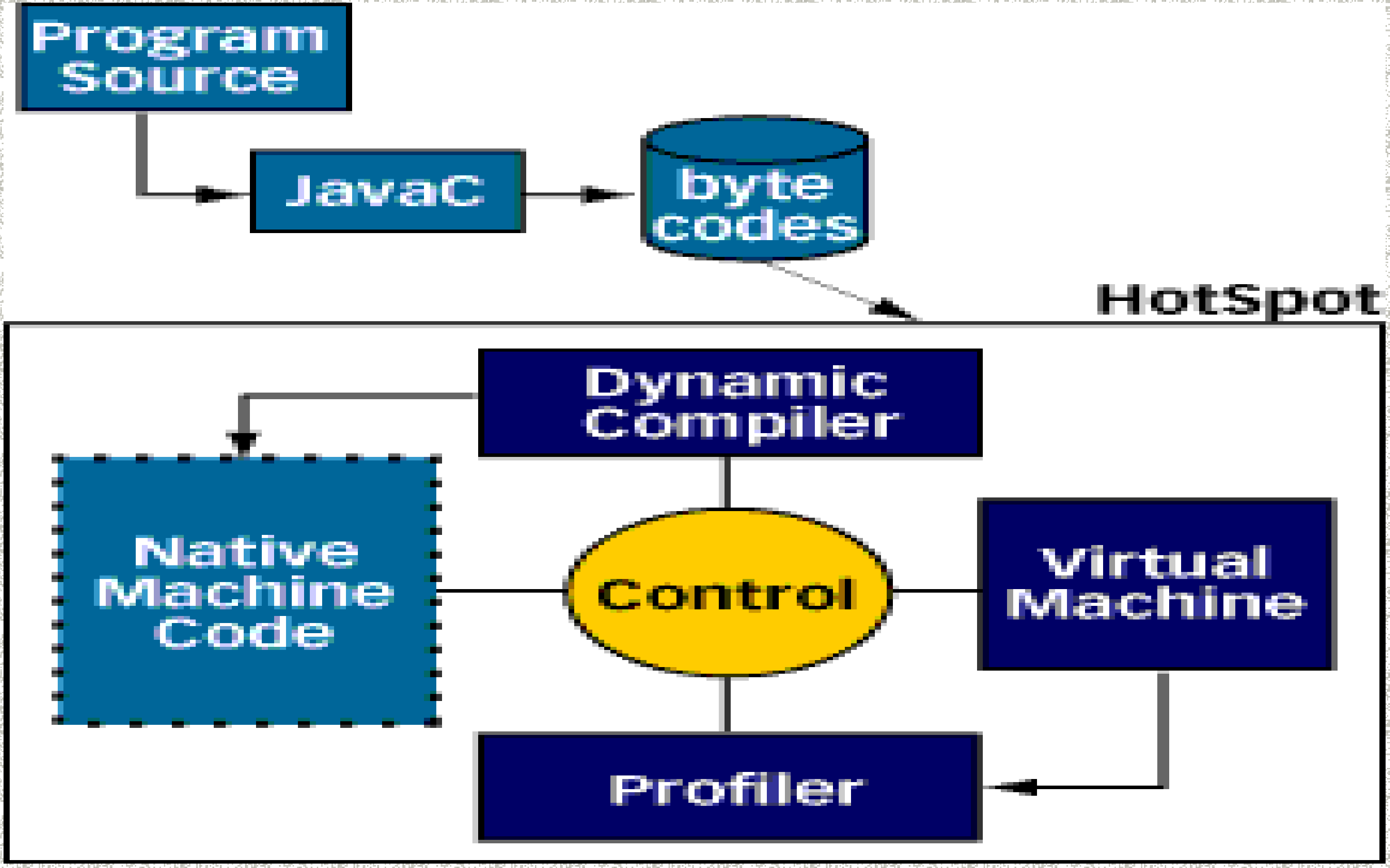
Dynamic compilation -- a brief history



Dynamic compilation -- a brief history

- **HotSpot dynamic compilation**
- The HotSpot execution process combines interpretation, profiling, and dynamic compilation. Rather than convert all bytescodes 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.

Dynamic compilation -- a brief history



Dynamic compilation -- a brief history

- 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.

Dynamic compilation -- a brief history

- 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.

Dynamic compilation -- a brief history

- **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.

Dynamic compilation -- a brief history

- **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.

Dynamic compilation -- a brief history

- 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.

Dynamic compilation-Details

- **Profiling and compiling heuristics**
  - The most fascinating aspect of the whole system (to anyone interested in the subject of machine reasoning, at least) is how the profiler decides which methods to optimize.
  - At present, the system uses a very good heuristic that does involve some sort of artificial intelligence.
  - Although the exact details are proprietary, it's reasonable to suppose that the system predicts the time it will take to optimize a given method.
  - Once the time spent in that method reaches a predefined threshold set at, say, 70 percent of the predicted optimization time, it becomes a candidate for optimization. (The exact threshold is undoubtedly the result of a considerable amount of research.)

Dynamic compilation-Details

- Fortunately, say the folks at Sun, it's unnecessary for such a system to be perfect. As long as it's right most of the time, it can make an astonishing difference in how a program runs.
- And, who knows? There is always the possibility that some day more advanced artificial intelligence heuristics can be used. There are plans to pursue this possibility -- the company expects to have its hands full refining its current implementation over the next year or two.

Dynamic compilation-Details

- **The advantages of dynamic compilation**
  The beauty of integrating a compiler with an interpreter is that it can do things a normal static compiler simply can't do. In particular, it can perform *optimistic compilation* and take advantage of *runtime information* to do robust *inlining*.

Dynamic compilation-Details

- ***Optimistic compilation***
  - Because the interpreter is always available to run the bytecodes, HotSpot's dynamic compiler can assume that exceptions and other hard-to-optimize cases don't happen. If they do, HotSpot can revert to interpreting the bytecodes. For a static compiler, optimizing in a way that handles all the unusual cases is a very difficult and time-consuming process.
  - By ignoring those cases, HotSpot can rapidly produce highly optimized code for the code that is most likely to run. That produces a lot of bang for the buck -- significant performance improvement for a small investment in optimization time.

Dynamic compilation-Details

- ***Runtime information***
  - The second major advantage of dynamic compilation is the ability to take into account information that is known only at runtime. Again, the details are proprietary. But it's not hard to imagine, for example, that if a method is invoked in a loop that has an upper index value of 10,000, it's an immediate candidate for optimization.
  - If, on the other hand, the upper loop index value is 1, the optimizer will know to ignore that method and let it be interpreted. Because a static compiler has no way of knowing for sure what the values might be, it can't make such judgements.

Dynamic compilation-Details

- ***Inlining***
  - One of the important optimizations HotSpot performs is *inlining* frequently-called methods. That means that instead of being invoked with the overhead of a method call, the method's code is copied into the calling routine and executed as though it had been coded there.
  - The smaller the method, the more sense inlining makes. A one-line method might spend more than twice as much time entering and exiting the routine as it does executing.

Dynamic compilation-Details

- **Inlining provides a tremendous boost in performance, but it has a size penalty.** If every method in a program were copied to every place it was called from, the program would balloon to five or ten times its original size.
- But HotSpot optimizes only the *critical* sections of a program. The 80/20 rule says that 80 percent of a program's runtime is spent in 20 percent of the code. By inlining as much as possible of that 20 percent, HotSpot can achieve a significant boost in performance with an acceptable size penalty. Here, runtime information makes a big difference, because HotSpot knows where to find that critical 20 percent.

Dynamic compilation-Details

- As with all good things, of course, there are tradeoffs. If a method is very large, the time spent entering and exiting it is only a small fraction of its execution time.
- In such a case, the space required to duplicate the method may not compare favorably to the time saved by inlining. So HotSpot undoubtedly has an upper limit on how large an inlined method can be. (The exact details, however, are proprietary.)

Dynamic compilation-Details

- **CompileThreshold**
  - Default Value: 1500 ,10000
  - Example Usage:

    java -XX:CompileThreshold=1000000 <yourclass> The current implementation of HotSpot usually waits for a method to be executed a certain number of times before it is compiled. Not compiling every method helps startup time and reduces RAM footprint. This option allows you to control that threshold.
  - By increasing the number, you can trade slight reductions in RAM footprint in exchange for a longer period of time before your program reaches peak performance.

Dynamic compilation-Details

- **Thread Synchronization**
  - Another big attraction of the Java programming language is the provision of language-level thread synchronization, which makes it easy to write multithreaded programs with fine-grained locking.
  - Unfortunately, older JVMs' synchronization implementations are highly inefficient relative to other micro-operations in the Java programming language, making use of fine-grain synchronization a major performance bottleneck.
  - HotSpot incorporates a unique synchronization implementation that boosts performance substantially. The synchronization mechanism provides its performance benefits by providing ultra-fast, constant-time performance for all uncontended synchronizations, which dynamically comprise the great majority of synchronizations.

Dynamic compilation-Details

- **Other Optimizations**
  - The optimizer performs all of the classic optimizations such as dead code elimination, loop invariant hoisting, common subexpression elimination, and constant propagation. It also features optimizations more specific to Java technology, such as null-check elimination.
  - The register allocator is a global graph coloring allocator and makes full use of large register sets.

Flawed Benchmarking

- **A bizarre result**

```
public class Tester {
   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;
         }
      }
   }
```

Flawed Benchmarking

```
public static void main(String[] args) throws InterruptedException {
    doSomeStuff();

    int nThreads = Integer.parseInt(args[0]);
    Thread[] threads = new Thread[nThreads];
    for (int i=0; i<nThreads; i++)
      threads[i] = new Thread(new Runnable() {
        public void run() { doSomeStuff(); }
      });
    long start = System.currentTimeMillis();
    for (int i = 0; i < threads.length; i++)
      threads[i].start();
    for (int i = 0; i < threads.length; i++)
      threads[i].join();
    long end = System.currentTimeMillis();
    System.out.println("Time: " + (end-start) + "ms");
  }
}
```

Flawed Benchmarking

Number of threads	Client JVM runtime	Server JVM runtime
10	43	2
100	435	10
1000	4142	80
10000	42402	1060

Flawed Benchmarking

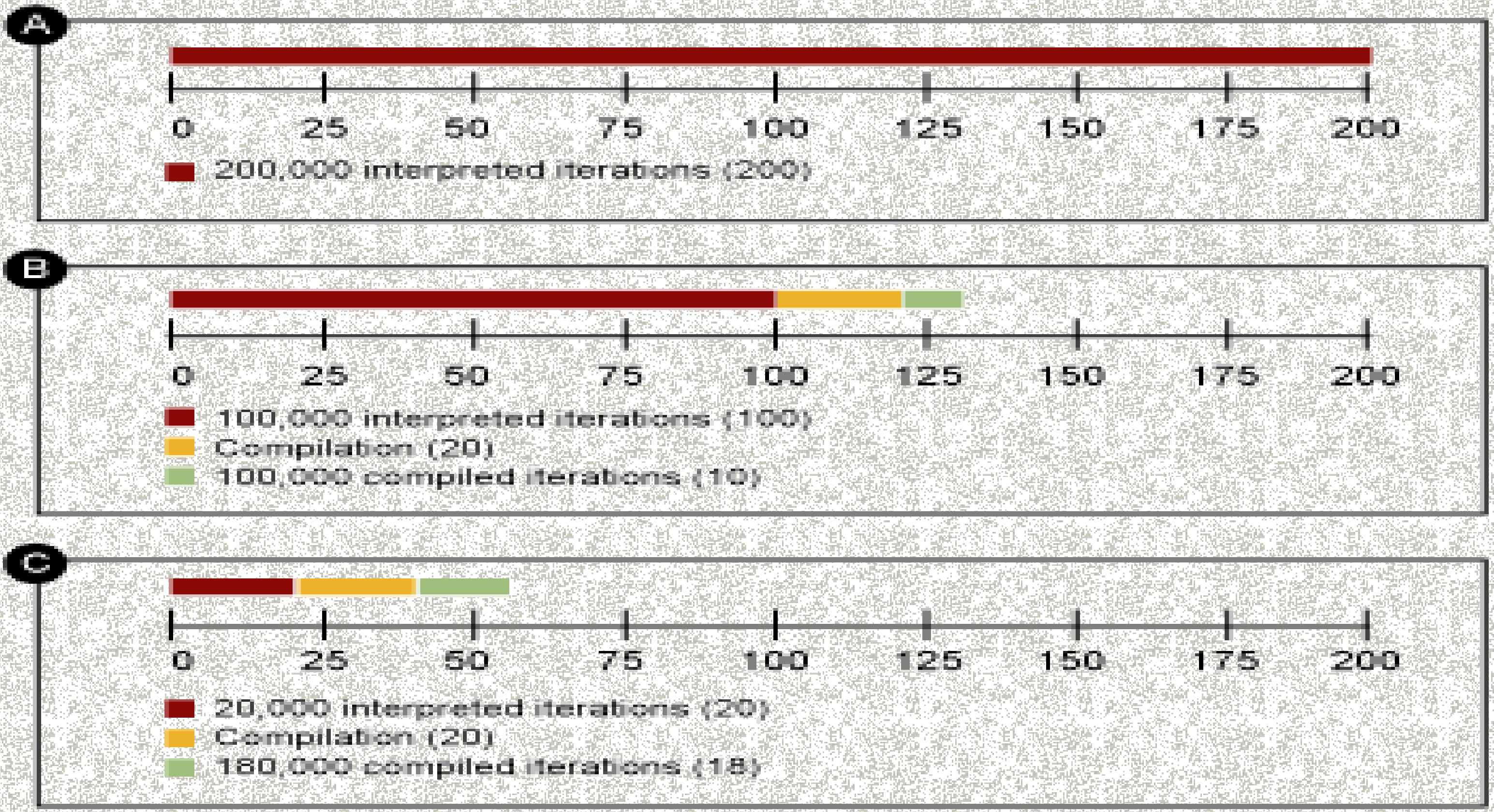
- The doSomeStuff() method is supposed to give the threads something to do, so we can infer something about the scheduling overhead of multiple threads from the run time of StupidThreadBenchmark.
  - However, the compiler can determine that all the code in doSomeStuff is dead, and optimize it all away because uselessSum is never used. Once the code inside the loop goes away, the loops can go away, too, leaving doSomeStuff entirely empty
  - 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.

Flawed Benchmarking

- 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.**

Flawed Benchmarking

- **Warmup**



Flawed Benchmarking

- Don't forget garbage collection
- Dynamic deoptimization

```
//How inlining can lead to better dead-code optimization
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);
        }
}
```

Flawed Benchmarking

- 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.

Flawed Benchmarking

- 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.
- 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:

```
Foo foo = getFoo();
foo.doSomething();
```

Flawed Benchmarking

- 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*.)

Flawed Benchmarking

- 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.**

Flawed Benchmarking

- **Another Bizarre Result**

```
public class StupidMathTest {
   public static class SimpleAdder implements Operator {
      public double operate(double d) {
         return d + 1.0;
      }
   }
   public static class DoubleAdder implements Operator {
      public double operate(double d) {
         return d + 0.5 + 0.5;
      }
   }
   public static class RoundaboutAdder implements Operator {
      public double operate(double d) {
         return d + 2.0 - 1.0;
      }
   }
}
```

Flawed Benchmarking

- StupidMathTest first tries to do some warmup (unsuccessfully), then measures the run time for SimpleAdder, DoubleAdder, and RoundaboutAdder.
- 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**.)

Flawed Benchmarking

- 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.**

Flawed Benchmarking

- Ask the wrong question, get the wrong answer
- **The scary thing about microbenchmarks is that they always produce a number, even if that number is meaningless. They measure something, we're just not sure what. Very often, they only measure the performance of the specific microbenchmark, and nothing more.**
- **But it is very easy to convince yourself that your benchmark measures the performance of a specific construct, and erroneously conclude something about the performance of that construct.**

Legends of Performance

- Synchronization is really slow
- Declaring classes or methods final makes them faster
- Immutable objects are bad for performance
- Allocation is an Alligator
- Collection is slow

Legends of Performance

- **Lessons learned**
  - There are several common themes across all of these performance legends. All of them are rooted in performance claims made in the very early days of Java technology, before significant effort was invested in improving the performance of the JVM.
  - Some of them were true when they were first stated, but the performance of JVMs has improved considerably since then. While we certainly shouldn't ignore the performance impact of synchronization and object creation, we should not elevate them to the status of constructs to be avoided at all cost.

Legends of Performance

- **Optimize with concrete performance goals in mind**
  - For each of these myths, the danger is the same: compromising good design principles -- or worse, program correctness -- to achieve a questionable performance benefit.
  - Optimization always carries risk, such as breaking code that already works, making code more complicated and therefore introducing more potential bugs, limiting the generality or reusability of the code, introducing constraints, or just making the code harder to understand and maintain.
  - Until there is a demonstrated performance problem, it is usually best to err on the side of clarity, clean design, and correctness. Save optimizations for situations where performance improvements are actually needed, and employ optimizations that will make a measurable difference.