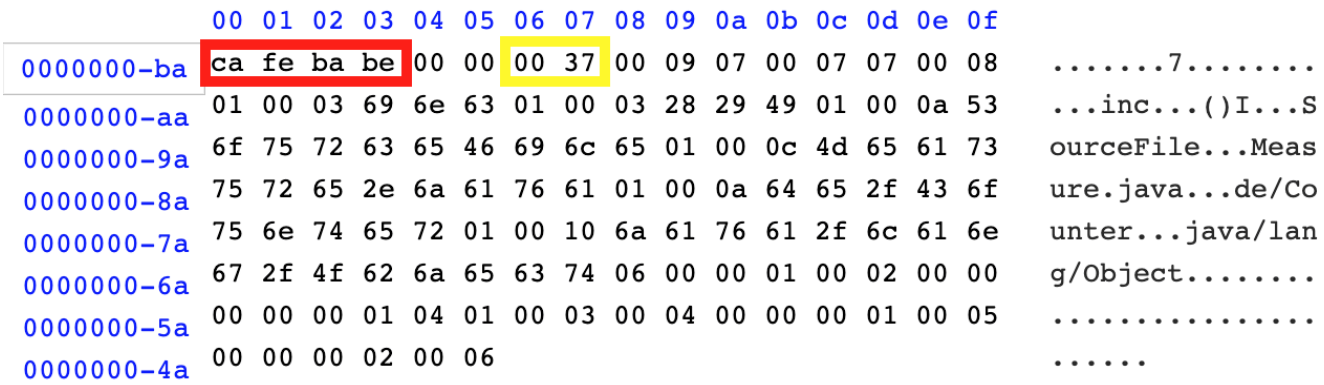
GraalVM Dictionary: Bytecode, Interpreters, C1 Compiler, C2 Compiler, CPUs, and More

Let's talk about what a compiler does. More specifically, what a Java compiler does. Or any compiler, for what it's worth. Most languages use similar concepts so that this article won't be lost on C# developers or JavaScript aficionados. Google's V8 compiler works more or less the same way as the Java compiler does (at least from a high-level perspective). It only omits the bytecode phase. C#, in turn, uses an "IR" code resembling Java bytecode. That's one of the reasons why so many different languages peacefully co-exist and even co-operate in the .NET universe. All these languages compile to the same IR code, so there's a common basis. That's pretty much the same with Kotlin and Java: both compile to Java bytecode, so there's a common basis allowing you to write an application using both languages side-by-side.

Anatomy of a Java file

Most likely, you've already heard about bytecode. Probably you also know that a Java `*.class` consists of bytecode, and maybe you've even tried to inspect such a file. If you've used a Hex editor, chances are you've even seen the famous "CAFEBABE" signature:



The first four bytes are the hallmark of Java bytecode files. The next four bytes - in particular, the two bytes we've marked yellow - are the version number of the class file format. The

1. High-level introduction to the GraalVM series

hexadecimal number 37 indicates we've compiled this file with Java 11. You JDK refuses to run any files without the "CAFEBABE" signature, and it'll refuse to run it if it's not compatible to Java 11.

The screenshot also shows some readable text. That's the map of variables. If you've ever wondered how Spring can use a variable name as the qualifier string to disambiguate multiple beans implementing the same interface - that's the explanation. Unlike most compilers, `javac` doesn't remove every variable or method name from the compiled code. Most names vanish because they aren't necessary to run the application, but some of them survive.

If you're interested in the class file format, Wikipedia has an exhaustive article for you.

Disassembling a Java class file

There's more. Today, we'll look at what you can't read in the hex editor. The stuff the most hex editors print as dots.

But fear not, you can convert the dots into human-readable code easily. Java ships with a tool helping you to read the `*.class` file. Only, if you're like us, it'll confuse you. What to make of this output?

```
class de.Counter1 implements de.Counter {
  private int x;
    descriptor: I

  de.Counter1();
    descriptor: ()V
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 53: 0

  public int inc();
    descriptor: ()I
    Code:
       0: aload_0
       1: dup
       2: getfield      #2                  // Field x:I
       5: dup_x1
       6: iconst_1
       7: iadd
       8: putfield      #2                  // Field x:I
      11: ireturn
    LineNumberTable:
      line 57: 0
}
```

Spoiler: this is the corresponding Java sourcecode:

```
class Counter1 implements Counter {
 private int x;

 public int inc() {
 return x++;
 }
}
```

What's Java bytecode?

Let's go back to the start. We assume you're a programmer, so you're editing source code all day long in your IDE.

That's just text. From an editor's point of view, a Java class is just a long string. We humans are good at reading such a long string, especially if it's nicely formatted, but to the CPU is just meaningless clatter. It uses an entirely different programming language, optimized to run at speed, but barely comprehensible to human beings.

We need an interpreter translating human language to machine code. Or a compiler, which is also a translator, just working a bit differently. We'll come back to that in a minute.

Most of the time, you won't notice that your IDE converts this string to something much more to the liking of your CPU. It calls the `javac` command to convert your Java source code to Java bytecode. That's an intermediate runtime code (aka IR code). It's something in between. Humans can read it with some effort, and CPUs can - well, they can't read it, but most IR codes are designed to be compiled to Assembly code easily. The Java bytecode, in particular, has an interesting twist. It's a stack-oriented engine that strongly resembles the good old programming language FORTH. That's great because stack-oriented bytecode is sort of a linearized version of the AST tree we'll meet later in this article series. Granted, that's simplifying things a bit, but you can also apply many tools and techniques developed for AST trees to bytecode.

There's that. Bytecode is the closest approximation to what happens under the hood in your computer most developers will ever meet. It looks a bit weird, enough to drive away the average programmer who cares about generating revenue. You've already seen some of it: `dup_x1`, `iconst_1`, `aload_0`, Take a break from that. Our journey has barely begun.

Did you notice we're cheating? We've introduced you to Java bytecode without telling you what it is. It's beyond the scope of this article. If you're interested, we invite you to read Stephan's exhaustive article about Java bytecode.

Why is bytecode so great?

So far, we've only seen the disassembled version of the bytecode. In the class file, every instruction fits into a single byte. In total, there are 202 instructions, most of them without parameters, the vast majority of the rest with a single-byte parameter. Only a few bytecode instructions take multi-byte parameters.

That's a very compact and concise representation of your source code. As a rule of thumb, it takes less memory than your source code.

But why should we care? Memory is cheap, isn't it?

There's the nasty little secret called "CPU cache." Your computer is fast - especially if it's cheating. Most of the time, it avoids accessing your spacious main memory. Modern CPU can run hundreds of instructions in the time it takes to read a single byte of data from the main memory.

To alleviate the pain, modern CPUs have at least two caches. The L1 cache is high-speed and very small, usually only a few dozen kilobytes per CPU core. The L2 cache is several times slower and much larger, often reaching one megabyte per core. But it's still a lot faster than the L3 cache, which is faster than the main memory. Superuser.com has a great in-depth explanation of CPU caches, including a photography of CPU without cover. The picture shows that the L1 cache - even if it's small in turns of bytes and megabytes - takes a huge area on the CPU die. Chip designers can create fast memory, or they can't create a huge memory, but it's impossible to achieve both goals at the same time.

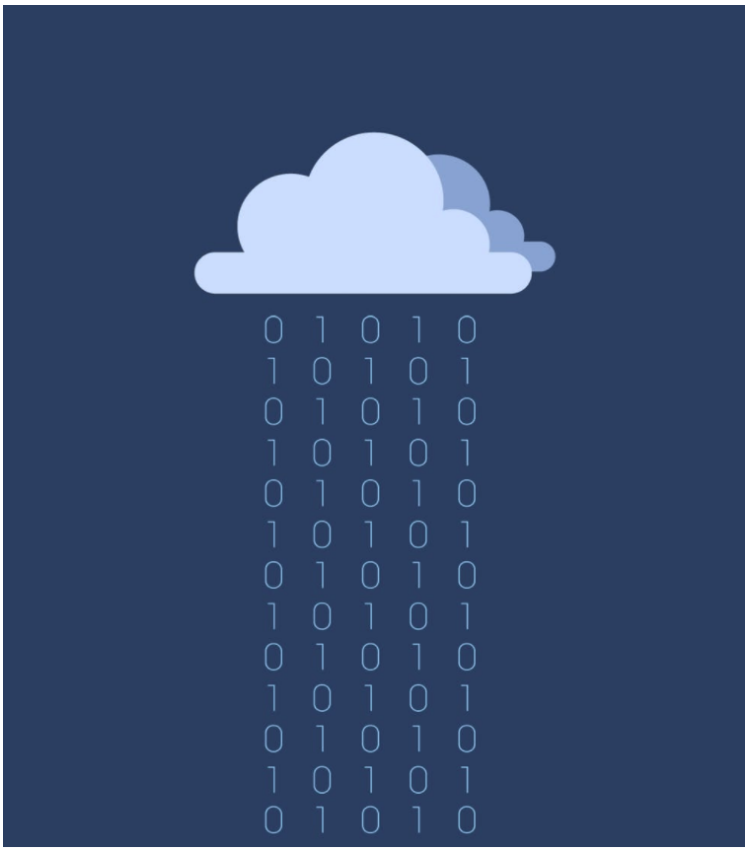
Hard disk, main memory, C3 cache, C2 cache, c1 cache, CPU registers: ever shrinking memories, stacked into each other like Russian dolls. Image published at pxhere under a CC0 license. Unknown photographer.

So the art of efficient programming is to keep your application in the CPU cache. Performance drops sharply if your application leaks into the next level cache. My (Stephan's) article a child's garden of cache effects demonstrates how easily you can find out the cache size of your CPU using a Java program.

A small memory footprint is why Java bytecode is so great. It's small enough to live in the super-fast L1 cache. Both the source code you're writing and the machine code the compiler is generating is a lot larger. A simple, innocuous-looking Java instruction may compile to dozens or hundreds of lines of Assembly code. That's a good reason not to generate machine code.

Can your CPU run Java bytecode?

Java bytecode is a low-level language. It looks a lot like an Assembly language. So it should be possible to create a CPU running Java bytecode natively.



GraalVM: all the way from cloud to Assembly by "BadBoy", published under a Pexels license.

People created such CPUs. The first attempt was SUN's picoJava project, as early as 1997. At the time, this project had remarkable success. It ran Java 20 times faster than the JVMs of that age. However, it was doomed because this processor couldn't run anything but Java. So it required an all-new operating system. Something like Android - but Android came decades too late to make picoJava a commercial success. Adding insult to injury, no company ever produced the CPU. It would've required roughly 440.000 gates, a moderate figure compared to the 7.5 million transistors of the Pentium II chip which also landed in 1997.

PicoJava was meant to be implemented in an CISC processor. A follow-up project, picoJava-II, can be implemented in an FPGA processor, giving the hardware designer more flexibility to embed it into a greater system.

Later, there was the Jazelle project. It was an extension of ARM processors. The processor could run both native ARM Assembly code and Java bytecode. There were even a few commercial CPUs, but they didn't gain enough market share to make a difference.

There are several other projects, most of them in the academic realm. For example, there's the JOP project. That's an open-source project allowing you to run native bytecode in an FPGA chip. It was maintained until 2013.

Obviously it's possible to implement a Java bytecode processor as a processor or co-processor. However, these chips never gained traction, not even in embedded computing. Progress in compiler technology made these approaches obsolete. In particular, the invention of the just-in-time compiler (aka JIT) was a game-changer.

Running byte code without hardware support, part one: An interpreter

The easiest way to write a programming language is to write an interpreter. Putting it in a nutshell, an interpreter is a key-value map. If you encounter bytecode `x`, execute machine code `y`. That's a simple programming model, great for programmers and newbies in particular. The GraalVM team chose to implement JavaScript and Ruby as an interpreter precisely because of the simplicity of the programming model (plus the the attractive features of the Futamara projection.) It's just a bit slow because of the lookups. A good compiler emits Assembly code running one hundred to one thousand times faster than interpreted code.[1]

When the JVM starts an application, it uses an interpreter to run the bytecode. The advantage of this approach is that it doesn't waste time compiling. Looking up the machine code in a hash map may be slow, but compiling the same code to machine code requires much more time.

Nonetheless, looking up the key-value maps adds up over time. In the long run, there's a major performance penalty if the same code is run time and again. Just imagine a `for` loop. If it's run a hundred times, you'll look up the same instructions a hundred times.

... a compiler

Instead of writing a program looking up instructions to execute them, we could also write a program collecting all these instructions before executing the program. That's the idea of a compiler. That's the way C and C++ programmers used to work for ages. Write a program, call the compiler, run the program, edit the source code, and so on. That's precisely what we Java programmers do when editing source code in our IDE. The IDE generates byte code each time we save our source code. `javac`, the program converting source code to byte code, is also a compiler.

We could implement a compiler looking up all your bytecode instructions and assembling them into a target file. This approach has many advantages. That's the idea behind the native-image compiler of the GraalVM. It's also what compilers did in the age before Java.

As things go, the Java VM does just that, but in a sophisticated way. It takes into account that compiling code comes at a cost. So it doesn't compile your bytecode light-heartedly. Before firing the compiler, it observes what your application is doing. There's no point in compiling code that's run only once. Granted, the machine code generated by the compiler is fast, but that doesn't pay unless the code runs several times. So the JVM tracks if part of the program "runs hot." If it does, the JVM starts to compile it.

... called "client compiler"

When researching this article, I (Stephan) remembered that this compiler used to be called the "client compiler." At the time, the theory was there's a difference between code run on the client and server-side code. Usually, every server is running a dedicated application. Plus, it's running the program for days, weeks, and months, serving countless users along the time. It pays to invest extra effort in optimizing this code.

Client code tends to be much more variable. Several JDKs ago, collective wisdom said it doesn't pay to optimize client code because it doesn't run long enough for a thorough analysis. It's more important to optimize startup time. When multi-core CPUs became popular, the compiler was shifted to a separate thread. Hence the comprehensive analysis stopped being a performance penalty, and the server compiler was activated on clients, too — time to rename the compilers. Funny thing is people stopped using Java on their PC roughly at the same time, the only exceptions being IDEs like Netbeans, Eclipse, and IntelliJ.

Nowadays, the client compiler is simply called C1. The server compiler is called C2.

The C1 compiler is a fascinating achievement in itself. The C1 compiler is running in a second CPU thread. The main thread runs your application, and the second thread optimizes it. When the compiler has finished, the JVM stops executing the interpreted code in favor of the compiled code. The compiled code picks up all the

intermediate results and all the local variables. A `for` may run the first couple of iterations in interpreted mode, and the remaining iterations run in fast Assembly code.

Pushing the compilation to a separate CPU thread took the sting out of compiling the code. Quite an impressive achievement!

... and just another compiler

What happens when even the C1 code runs hot?

Now things get interesting. The bytecode is recompiled. This time for real. The JVM already knows - or suspects - the code is going to be called frequently in the future. It becomes obvious it pays to invest even more effort in optimizing the code.

The C2 compiler doesn't kick in immediately. We've already seen the interpreter and the C1 compiler come first. During that time, the C2 compiler - more specifically, a module called "profiler" - observes the behavior of your application. It observes if the "then" or the "else" part of an `if` statement is taken. How often loops are repeated. How many implementations an interface has.

The C2 compiler employs the profiler data to optimize the machine code. Probably the most exciting bit is speculative optimization. Speculative optimization exploits the fact that programmers love certain features - without hardly ever using them. Mind you; if you've adopted the object-oriented mindset, you'll probably like overriding methods. If we were to take this feature away from you, you'd cry out in pain. Here's the catch: How often do you use it?

The vast majority of classes and method turns out never to be overridden. That, in turn, opens a large window of opportunity for optimizations.

So the C2 compiler generates machine code reflecting how you're using the programming language instead of reflecting how you could use it, as traditional ahead-of-time compilers do.

That's the reason why Java programs are often faster than C++ programs. Granted, there's no way a Java program will ever overtake a hand-crafted C program, let alone a carefully optimized Assembly program. Nonetheless, optimizing code to

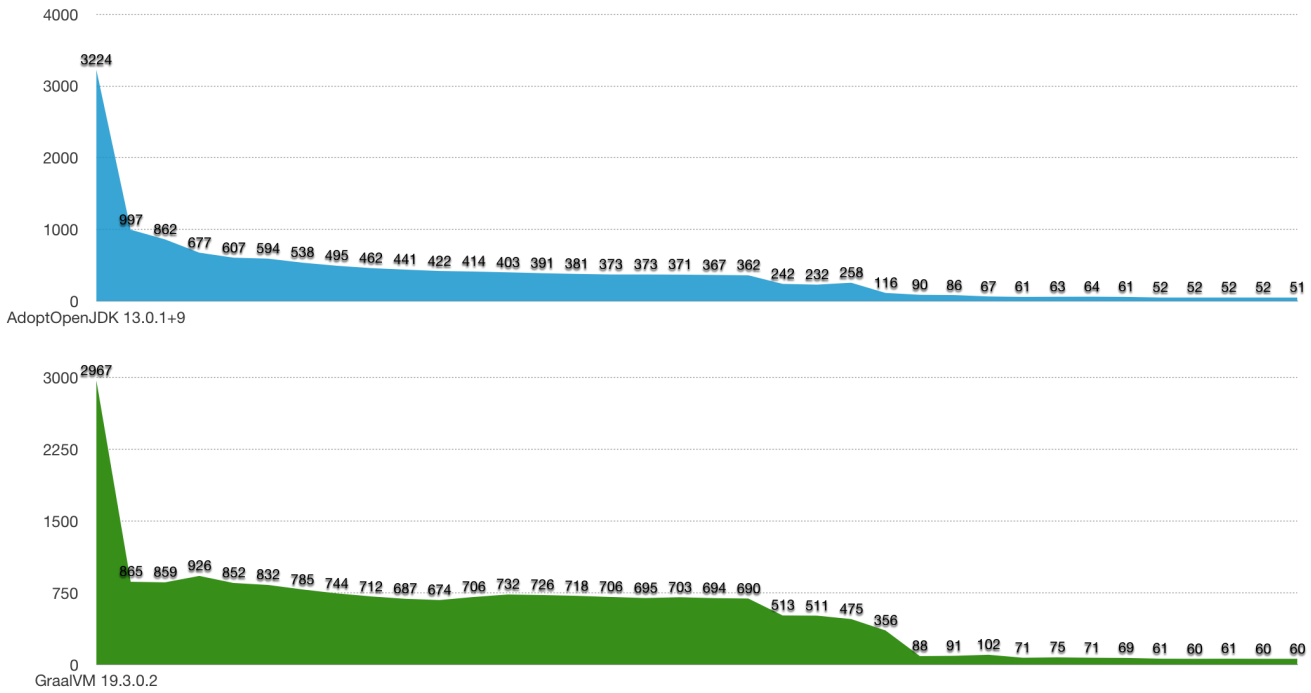
such a level takes a lot of time. Modern Java compilers offer a remarkably good combination of decent speed of development and decent application performance.

The next installment of this series explains some of the optimizations of the HotSpot compiler and the GraalVM compiler in depth.

Putting the theory to test

By the way, that's not just theory. Putting this to the test only takes a few minutes. A couple of years ago, I (Stephan) had published a benchmark demonstrating the speed-up of the JVM with each run. As the commentators at the time mentioned, my approach was not without faults. But even today, I'm pretty convinced the bottom line is correct. The more often you run your algorithm, the faster it gets. Usually, this comes to an end after 10.000 repetitions. You'll observe two major speed-ups. You've already seen the first one. It's when the C1 compiler replaces the interpreter. As you can see, it start almost immediately:

GraalVM 19.3.0.2 and AdoptOpenJDK 13 | Java 5, 6, and 7



For some reason, I (Stephan) didn't manage to add the x-axis when I re-ran the tests. So I've added the original statistics from 2012. The x-axis are the same. I've

uploaded both the benchmark and the results to GitHub.

After 100 iterations - give or take a few - the benchmark accelerates a second time. That's when the C2 compiler performance it magic. The benchmark also shows minor speed-ups all the time, until 10.000 iterations. This is the effect of the profiler and the optimizer.

A weird observation is that my 2012 figures are slightly faster than the numbers I've taken in 2020. It shows that my laptop is five years old. Plus, I've taken the 2012 figures on a Windows desktop PC running at almost the double clock speed. Advances in compiler technology seem to compensate the slower speed of my laptop's CPU and memory.

Another surprising observation is that in this particular benchmark, GraalVM picks up speed later than AdoptOpenJDK. However, the peak performance is almost identical: GraalVM is 15% slower. The initial performance of the GraalVM is 10% better than the performance of AdoptOpenJDK. Please take these numbers with a grain of salt: there are better benchmarks out there.

The next part of this series is going to cover the performance in more detail, and we'll shed light on some of the tricks the JIT compilers use.

Wrapping it up

About the co-author
Karine Vardanyan occupies herself with making her master at the technical university Darmstadt, Germany. Until recently, she used to work at OPITZ CONSULTING, where she met Stephan. She's interested in artificial intelligence, chatbots, and all things Java.

Java has come a long way since it's modest beginnings. There's been progress on all levels. A large part of the progress happened unnoticed by most in the realm of compilers. We've shed some light on this part of the Java success story. Granted, it's low-level stuff. None of this knowledge makes you a better business programmer. But having dug a bit deeper, we can't help but feel honored to stand on the shoulders of the giants.

Dig deeper

Wikipedia describing the class file format

Introduction to Java bytecode

Introduction to Assembly code

in-depth explanation of CPU caches, including a photography of CPU without cover

A child's garden of cache effects

SUN's picoJava project running bytecode in hardware

picoJava-II in FPGA including a lot of background information and history

JOP project allowing you to run native bytecode in an FPGA chip

---

1. At least that held until the Truffle framework implemented a working Futamara projection and unleashed the power of the JIT compiler.↩

Post navigation

← What About GraalVM?

Optimization Strategies of the GraalVM →