What About GraalVM?

So there's this shiny new Java virtual machine called GraalVM I've already blogged about last Spring. In the meantime, Graal's managed to make an impact on the Java universe. Most prominently, there's the flagship project called Twitter. They're using GraalVM for some time now. They're running their Scala microservices on Graal.

The holy grail by Erich Ferdinand, published under a CC BY 2.0 license.

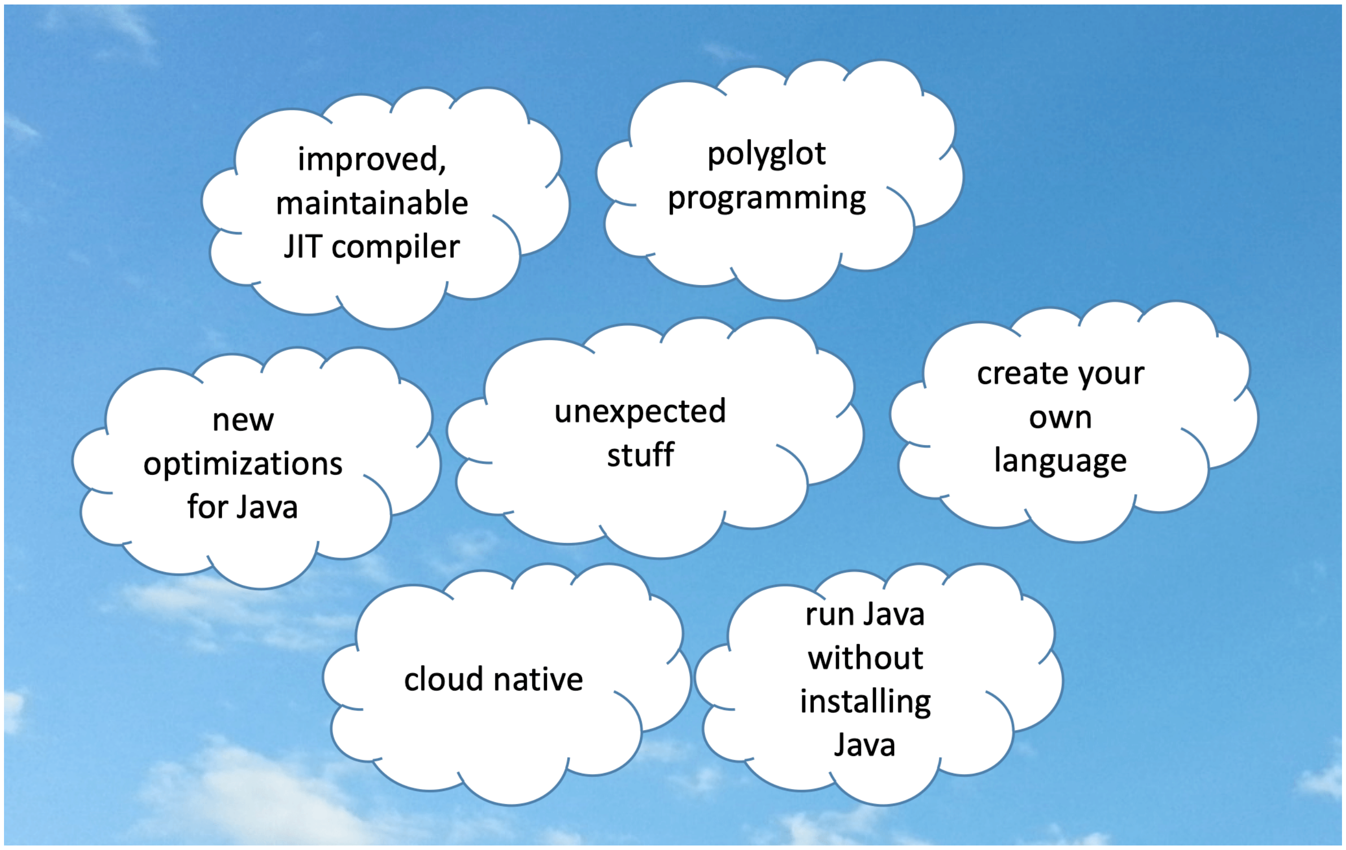
Graal has also arrived in a more conservative business world. It seems to be particularly attractive to cloud-native applications. Just think about Lambda functions. So it's time to revisit GraalVM. Mind you; it's 2020. Is it time to abandon your good old Java virtual machine and to move to something new?

Buzzwords

Let's have a look at the definition coined by my co-author Karine:

> *GraalVM is a universal virtual machine running applications written in languages like JavaScript, Python, Ruby, R. It also supports languages like C and C++. GraalVM runs any language with an LLVM compiler. That even includes FORTRAN. Not to mention the entire Java universe, including Scala, Kotlin, and (guess what!) Java itself. GraalVM promises to unleash true interoperability between different programming languages. Maybe we'll see polyglot applications using a common instance of a virtual machine soon. There's even more. The GraalVM can also run in embedded mode, as the Oracle Database and MySQL show.*

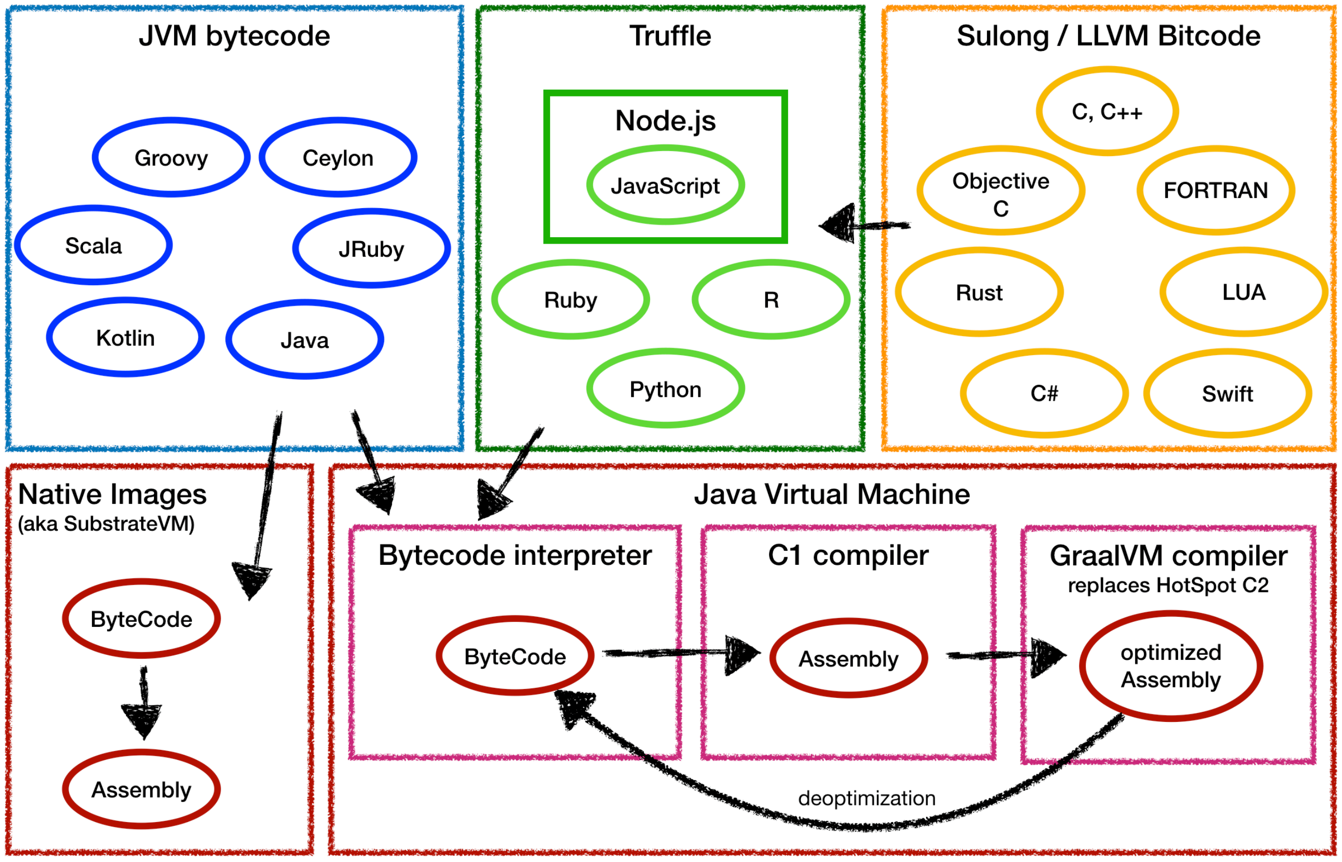
Wow. That's quite a chunk to swallow. GraalVM has a lot in store for you.


improved, maintainable JIT compiler
polyglot programming
new optimizations for Java
unexpected stuff
create your own language
cloud native
run Java without installing Java

Background of the image by Richard Barboza, published under a Pixabay license.

Let's examine it piece by piece. That'll take a while. This article is a high-level introduction to an eight-part series. Nine parts, if you count our December article on Truffle. And we didn't even start to cover the database bit. It's fascinating, but we're not sure we'd like to propagate programming Java in the database. It doesn't seem a best practice in the age of microservices and cloud-native applications. Nonetheless, it's a powerful tool, so maybe we'll explore it soon.

What is Graal?



First of all, Graal is a research project hosted at the Oracle labs. Since 2012, the development team has published 60+ papers on GraalVM. So the first answer to our question is: GraalVM is simply a remarkably long-running academic project. A successful one, at that.

Much of it revolves around the Futumara projection, making it a real and useful thing. That's the Truffle engine we've mentioned above. Truffle, in turn, is the polyglot part of GraalVM. It's the realm of JavaScript, Ruby, R, and all the LLVM languages. If you're like me, you've never heard about LLVM before. Simplifying things a bit, LLVM is a virtual machine for languages like C/C++, FORTRAN, and many others. The LLVM compiler converts your C code into LLVM bitcode, which runs on the GraalVM without further ado.

In other words, if you were able to get your hands on the source code of DOOM, you could play the game on a Java VM. Too bad it's so hard to get these source codes because of copyright reasons. It'd be a fascinating project. I (Stephan) fondly remember Daniel Kurka's legendary presentations of JavaScript, which used to include a JavaScript port of DOOM. At the time, his team used emscripten to translate the C source code to JavaScript. I'd like to see this accomplishment repeated with the GraalVM, so we'd be able to compare the two approaches. My guess: using emscripten was sort of a painful process, and GraalVM and LLVM take the sting out of it.

Did you notice our list of languages didn't include Java? That's not a mistake. Truffle's written in Java, but it's not meant to run Java.

But. There's an exciting project in the GraalVM universe doing just that. It's called Espresso, and for some reason, the blogosphere doesn't talk much about it. So I'm not sure the project is still alive. But if it is, and if it's going to be a success, Java will be a first-class citizen in the Truffle universe. Putting it in simpler words: Espresso is the promise of a genuinely polyglot world. It promises we can call Java from JavaScript, from Ruby, from R, and all the other languages supported by Truffle. We don't think that includes the LLVM languages, but maybe - just maybe - it's possible to provide an API for C++ to call Java, too.

GraalVM as a plug-in replacement to the JVM

Then there's the other part of the GraalVM project. It's the part currently stirring a lot more excitement. GraalVM is a plug-in replacement for the Java virtual machine, running Java, Scala, Kotlin, and all the other languages running on Java bytecode. Since 2019, GraalVM is production-ready on Linux. Version 20.1.0 also claims to support Windows fully. So we can the high claims to the test. We'll do that in the third part of the series.

Often people claim GraalVM offers superior performance. For example, the GraalVM implementation of Ruby runs up to 30 times faster than the original implementation, at least according to https://www.youtube.com/watch?v=iXCVaQzXi5w&t=715s and https://www.youtube.com/watch?

v=GinNxS3OSi0&t=726s. There's that. We don't know how reliable these pieces of information are. We've tested a few cases: JavaScript, Ruby, and Java.

Ruby is a pleasant surprise, indeed. GraalVM runs our synthetic Ruby benchmark 30% faster than Ruby 2.7.0. Of course, there are two other Ruby implementations out there, and we didn't test them yet. Nonetheless, 30% is a promising start. Plus, we're positive there's headroom for improvements.

From a Java programmer's perspective, things are a bit more down-to-earth. Generally speaking, the performance of the GraalVM is roughly on par with Oracle's HotSpot compiler. Not bad, but not that exciting, either. By the time you're reading this, things may have improved - we're talking about GraalVM 20.1.0. In the long run, we're positive GraalVM is going to run circles around the traditional JVM. But it's not going to happen in the short term. Java is one of the most heavily optimized programming languages, and it's still being optimized, so beating it is a hard call. We believe it's possible because GraalVM is a fresh, new take, which doesn't have to carry the burden of 20+ years of optimization. Currently, the source code of GraalVM is pretty straight-forward - at least compared to the source code of the C++ JVM. At least, that's what we're told, and that's what fuels our optimism.

The same applies to JavaScript. If you're to believe the conference talks, GraalVM currently runs JavaScript programs with the same speed as Google's V8 compiler, give or take some margin. That's a remarkable achievement, but nothing to get excited about if you're a manager. We also had to learn that's nothing to get excited about if you're a developer. When we ran a test, we learned that only a few JavaScript programs reach this performance level. These programs exist, we've seen them. But many other applications run at 10% speed. In particular, the Angular CLI is remarkably slow on GraalVM 20.1.0.

To be fair, we're impressed that the Angular CLI works flawlessly at all. It requires the entire ecosystem of node.js. GraalVM ships with a working copy of node.js and npm. Even better, it's compatible with ECMAScript 2019. Even if it may be slow today, that's a remarkable achievement. Plus, the project team has declared a bold goal: they want to reach the same peak performance as node.js offers.

Being technicians, we're impressed with all that. Try as we might, we couldn't do that! And we don't suffer from a lack of self-esteem. There's that. But all this technological excellence won't convince your CEO to run your cash-cow application on Graal. If GraalVM is going to be a thing, it has to prove its value.

Nonetheless, GraalVM raised a lot of attention among the media. What are the drivers behind the success of GraalVM?

Drivers: performance and maintainability

One of the key drivers is the superior performance of GraalVM. Granted, that's not a bit deal in the case of Java or JavaScript. We believe (or hope?) that'll change soon. GraalVM is an open-source project, written in a popular language. Nowadays, everybody and their grandma fluently talk Java, so it's easy to contribute to the GraalVM. OK, maybe that's exaggerating a bit, but it's a lot easier than it is to contribute improvements to the HotSpot compiler. Mind you; the HotSpot compiler carries two decades of history on its shoulders. Adding insult to injury, the HotSpot compiler is written in C/C++, a language that's still popular - but not among Java programmers.

Rewriting the JVM compiler in Java opens a new window of opportunity. The bet is that countless Java programmers are going to spend some time with the GraalVM, find some weaknesses, and contribute bug fixes and improvements. The young age of the GraalVM makes that a walk in the park. By contrast, both the V8 engine and the HotSpot compiler suffer from decades of optimizations. We're told it's not easy to improve anything without breaking something else. If you've already worked with a large enterprise application, you know the problem. In the business world, microservices come to the rescue. In the JVM worlds, it's GraalVM. Based on new ideas, it's a fresh take on an old problem. Plus, it has been written with optimization and with being extended in mind.

That's probably the reason why Twitter adopted GraalVM. They're running their production code on Scala. That's a language compiling to the JVM bytecode. Too bad, the bytecode has been developed specifically for Java. Scala introduces new concepts that don't match perfectly to the Java bytecode. Of course, Twitter still uses the same bytecode, but they've tweaked GraalVM with Scala-specific optimizations.

Things are more evident in the case of Ruby and R. These languages aren't JVM languages. Of course, there's the JRuby project compiling to Java bytecode. That's a remarkably successful project. However, the Graal implementation of JRuby tackles the problem from a different angle. It's an interpreter running on the Truffle framework, which is an implementation of the Futumara projection.

Truffe - the Futumara projection come true

We've explained the Futumara projection in much detail in an article dedicated to Truffe. At this point, suffice it to say it's a clever approach to writing a compiler, taking the sting out of writing a compiler. In a nutshell, the idea is once your base language uses an optimizing Just-in-Time compiler, it compiles and optimizes everything. If you write an interpreter, this interpreter is compiled and optimized by the Just-in-Time compiler, too. And if both you and the compiler are clever, your interpreter runs your application every bit as fast as if you'd written a traditional compiler. It's just a lot simpler. Everybody and the grandma can write an interpreter. Writing a good compiler is a piece of art.

The beauty of this approach is that it's both an interpreter and what they call a compiler-compiler. After a warmup phase, Graal compiles Ruby to native machine code that's more or less on par with hand-optimized Assembly code. The downside is you can always fine-tune the Assembly code far beyond anything Graal can do, no matter how much it evolves in the future.

However, in the real world, things look a bit different. Optimizing code - and Assembly code in particular - takes a lot of developer effort. It's more efficient to write an interpreter everybody can understand, maintain, and optimize, and to have it compiled to Assembly code by Truffle and the GraalVM. Both have been written in Java, so chances are many people contribute optimizations to these projects, too.

Oh, and don't confuse the Futumara projection with Futurama. Yoshihiko Futamura is a Japanese scientist who's invented his projection in the early 70s. Roughly fifty years after his scientific groundwork, the idea materializes in the business world.

Update February 04, 2020:

Thomas Würthinger, GraalVM's project lead, reports in an interview in December 2019 that the team is currently installing an advisory board populated by company representatives. On the one hand, this sounds like a bureaucratic process making it difficult for individual developers to provide a pull request. On the other hand, this indicates big companies are interested in improving GraalVM. Maybe this is a good omen, backing our hope GraalVM has more in store for us.

... and carbon footprint

By the way, raw performance is one thing. It's what we used to look at when computers were slow. Now we've run into another barrier. From an economic perspective, energy consumption costs money, and from a more global perspective, it adds to the carbon footprint of your company. That, in turn, has become important to marketing. It's easier to sell you're product if it's climate-friendly.

Write your own programming language!

In the academic world, there's probably another driver to the success of Graal. Truffle enables you to write a new programming language. We're curious where this leads.

During the last decade, we've already seen a plethora of new languages exploring new concepts. If you're working in the industry, you've probably missed this due to the dominance of Java, JavaScript, and C/C++. But many of the new concepts introduced to the mainstream languages have been tried in minor languages.

For example, dynamic typing has been made famous by Groovy and Ruby. Streams and functional programming become a real thing with the advent of Scala and Groovy. Later, Kotlin and Ceylon picked up the idea, before it finally made its way into Java 8. Compile-time null-safety came to the Java universe with Kotlin and Ceylon. Scala demonstrated how to use the immutability pattern without the pain. That, in turn, enabled Akka to unleash reactive programming and the power of your multi-core CPU.

Academic and experimental programming languages as innovation labs

We don't know much about the early discussions about enums, value types, and autoboxing. Still, by now, you see the pattern: before introducing a game-changer to a language used by millions of developers, it's a good idea to play around with the idea in a less known language. Developers are much more tolerant of breaking changes if they know they're working on the bleeding edge. They feel honored to be able to contribute their ideas and their input to the new language.

Mainstream-languages are a completely different piece of cake. For example, try to get rid of type erasure in Java. We daresay, that's impossible. You know, type erasure is one of the great ideas - and it's been great indeed! - that solved a lot of problems at the time. The problem is it removed a lot of headaches from the language designers. After publishing the feature, it turned out to cause problems in the wild. Most developers won't even notice, but type erasure is painful to framework designers. It'd been better to try this idea in a small language, migrate something like Gson or Jackson to it, and learn about the disadvantages of type erasure. As far as we can see, that's what the Java language designers are doing nowadays.

The ease of implementing a language with Truffle makes playing around with new concepts a lot easier. Truffle cuts the time it takes to write a decent language in half. Plus, Truffle's successful implementation of the Futumara projection gives your pet language a significant performance boost. That, in turn, is a key to convince developers to pick up your language.

Cloud computing, the main driver

And yet, we've still didn't mention the biggest key driver. The unique selling point of the GraalVM is its AOT compiler. After more than two decades, we can finally compile Java to native machine code. Even more important, nowadays we've got a use-case requiring native code. The most important being Amazon Lambda functions (and their siblings offered by other cloud providers).

The exciting bit about Lambda functions is you only pay-per-use. You only pay if your code runs. If nobody calls your service, you don't pay for it. To make this a compelling business case for the cloud provider, they have to shut down your virtual machine when nobody uses it. That, in turn, means your customers often suffer from a cold start delay. According to many slides and conference talks we've seen, that amounts to three seconds for a simple Spring Boot service, give or take a few. Too much in a world dominated by search engines penalizing sluggish web sites. Big webshops report a juice effect on sales if their web page slows down one-tenth of a second. Where does this leave your cold-starting Spring Boot service?

The AOT compiler solves that. Traditionally, Java starts slow because it has to load several thousand classes. However, your simple CRUD application only uses a small fraction of these classes. You don't need the full power of Spring Boot to respond to a GET request. Three lines of code do the trick just as well, as frameworks like Helidon, Micronaut, or Express.js (in the Node.js world) show.

The AOT compiler boils down your code to what's necessary and emits pre-compiled machine code. If you're lucky, that's just the three lines of Helidon code, plus the infrastructure required to support these three lines. But nothing beyond that.

Even better, that code is ready-to-use, without requiring any expensive infrastructure to load and initialize. Add a cloud-native framework like Quarkus or Helidon to the equation, and you end up with Lambda functions responding withing 0.005 seconds, according to some marketing slides we've seen. We'll put this to test in the final part of this series. Stay tuned!

Unexpected stuff

During the last year, a few developers began to embrace GraalVM. For instance, Michiel Borkent has created babashka. It's a Clojure interpreter, compiled as a native image by GraalVM. It integrates seamlessly in your Linux or macOS shell, allowing you to replace the bash commands with Clojure code. Judging by the project's popularity - 1500 GitHub stars in roughly a fourteen months - it's hit a sweet spot.

Being able to compile to native executables also seems to be a thing in the JavaFX community. You don't have to install Java, nor do you have to install JavaFX to run a JavaFX application compiled natively. That takes the sting out of installing such an application on thousands of PC in your company.

GraalVM also seems to fuel a whole new wave of microservice frameworks. For instance, there's Quarkus. We'll cover it in some detail in a dedicated article. In a nutshell, Quarkus is a collection of industry-standard frameworks, plus extensions enabling these frameworks to generate native binaries.

Wrapping it up

We'll examine these bold claims in this series. At this point, we can already say that GraalVM is a fresh new take on the decades-old struggle to optimize the JVM. It unleashes the power of polyglot programming. As my (Stephan's) previous article shows, polyglot programming works several magnitudes faster than older approaches like Rhino or Nashorn. So when we complained about performance, we were a bit unfair: we've compared the JavaScript performance to the V8 engine instead of comparing it to Nashorn. To our disappointment, multilingual programming with GraalVM still uses the same string-based approach, so we doubt it's fun to write a program using more than one language. Maybe we'll see that in the future.

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.

Be that as it may, we expect GraalVM to have a significant impact both in the realm of cloud-native computing and in the field of programming languages like R, Python, and Ruby. We're also excited about integrating low-level C/C++ code in our Java or JavaScript application. GraalVM is an opportunity to benefit from the experience of many developer communities that used to be cut off from the Java universe.

Dig deeper

Chris Thalinger on why Twitter is interested in GraalVM https://www.youtube.com/watch?v=iXCVaQzXi5w&t=715s and https://www.youtube.com/watch?v=GinNxS3OSi0&t=726s.

Bytecode @ BeyondJava

Assembly language @ BeyondJava.net

Introduction to LLVM bitcode

Post navigation

⟵ A Hurried Programmer's CSS Survival Guide

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