EPISODE 644

[INTRODUCTION]

[0:00:00.3] JM: Java programs compline into Java bytecode. Java bytecode executes in the Java virtual machine, which is a runtime environment that compiles that bytecode into machine code and also optimizes the runtime by identifying hot code paths and keeping those hot code paths executing quickly. The Java virtual machine is a popular platform for building languages on top of. Languages like Scala and Clojure compile down to Java bytecode and they can take advantage of the garbage collection system and the code path optimizations of the JVM.

But when Scala and Clojure compile into Java bytecode, the code shape, which is a rough way of saying the way that the programs are laid out in-memory, that code shape is not the same as when Java programs compile into Java bytecode. Executing bytecode that comes from Scala will have certain performance penalties relative to a functionally identical program written in Java, and that's because Scala code, when it compiles down to Java bytecode, takes a certainly different code shape than the Java code.

GraalVM is a system for interpreting languages into Java bytecode that can run efficiently on the JVM. Any language can be interpreted into an abstract syntax tree that the GraalVM can execute using the JVM. Languages that run on GraalVM include JavaScript, R, Ruby, and Python.

Thomas Wuerthinger is a senior research director at Oracle and the project lead for GraalVM. He joins the show to explain the motivation for GraalVM and the architecture of the project and the future of language interoperability. It was an exciting discussion and I learned a lot about the Java ecosystem. I personally am still a little bit intimidated by GraalVM. I hope I explained it correctly in this preamble. But if I didn't explain it well, then Thomas will certainly explain it well in this episode that you're about to hear.

Again, I really find this episode interesting, exciting. It made me really curious about some other elements of the Java ecosystem, which we probably have not covered on this show as much as we should.

Before we get started I want to announce that we're hiring a creative operations lead. If you're an excellent communicator and you are looking to get a job in engineering communications, check out softwareengineeringdaily.com/jobs. This is a great job for somebody who just graduated a coding boot camp and they feel like they're in between their previous life and an engineering life and they want another job that doesn't require coding quite yet, or they want to learn a little bit about coding before they start their coding job, or somebody who has a background in the arts who's making their way into technology. I think there's a lot of people who fit that bill, and if you want to be creative and you want to learn more about engineering, check out this position at softwareengineeringdaily.com/jobs.

[SPONSOR MESSAGE]

[0:03:10.8] JM: DigitalOcean is a reliable, easy to use cloud provider. I've used DigitalOcean for years whenever I want to get an application off the ground quickly, and I've always loved the focus on user experience, the great documentation and the simple user interface. More and more people are finding out about DigitalOcean and realizing that DigitalOcean is perfect for their application workloads.

This year, DigitalOcean is making that even easier with new node types. A $15 flexible droplet that can mix and match different configurations of CPU and RAM to get the perfect amount of resources for your application. There are also CPU optimized droplets, perfect for highly active frontend servers or CICD workloads, and running on the cloud can get expensive, which is why DigitalOcean makes it easy to choose the right size instance. The prices on standard instances have gone down too. You can check out all their new deals by going to do.co/sedaily, and as a bonus to our listeners, you will get $100 in credit to use over 60 days. That's a lot of money to experiment with. You can make a hundred dollars go pretty far on DigitalOcean. You can use the credit for hosting, or infrastructure, and that includes load balancers, object storage. DigitalOcean Spaces is a great new product that provides object storage, of course, computation.

Get your free $100 credit at do.co/sedaily, and thanks to DigitalOcean for being a sponsor. The cofounder of DigitalOcean, Moisey Uretsky, was one of the first people I interviewed, and his

interview was really inspirational for me. So I've always thought of DigitalOcean as a pretty inspirational company. So thank you, DigitalOcean.

[INTERVIEW]

[0:05:17.5] JM: Thomas Wuerthinger, you are the senior research director at Oracle and the GraalVM project lead. Welcome to Software Engineering Daily.

[0:05:26.5] TW: Hi. Nice to meet you.

[0:05:28.0] JM: It's great to meet you as well, and I'm excited to talk about GraalVM. This is a somewhat complicated topic. So it requires some background. I want to start by discussing the JVM. The JVM has historically been used to run Java bytecode. Can you give a brief explanation of the compilation and execution path of a Java program?

[0:05:52.1] TW: Sure. A Java program is first interpreted. It runs into an interpreter, and the interpreter is looking at the bytecodes and executing them. But if certain message get hot, they will start just-in-time compilation to make the hot parts that are frequently executed faster. This gives the JVM this unique speedup. Sometimes Java applications can even be faster than, let's say, C++ applications, because of this profiling feedback that gets to the compiler to make the programs faster.

[0:06:25.3] JM: So the Java code that I might write gets compiled into Java bytecode. Then when does the bytecode get interpreted into machine code?

[0:06:35.6] TW: That is when the message get hot. So that is when a certain message is executed very frequently, usually about 10,000 times. Then the compiler turns that bytecode into machine code.

[0:06:46.6] JM: So I run Java C on my machine and I compile a Java program into Java bytecode, and when my code actually executes, it uses a just-in-time compiler that interprets that bytecode into machine code. Is that right?

[0:07:03.8] TW: Yes, but first, like when you first use the Java comment, the Java bytecodes are still not turned into machine code. It's just interpreted. Only when they run for some time, then you have to apply the just-in-time compiler to machine code.

[0:07:18.7] JM: Is that just-in-time compiler written in C++?

[0:07:21.8] TW: In current virtual machines, in current production virtual machines, it is written in C++. Yes.

[0:07:27.5] JM: Why is that? Why is the Java just-in-time compiler written in C++?

[0:07:31.1] TW: Because I think historically people saw that system level software should be written in C++ because of the more tight control of memory management and the more type controller of performance characteristics.

[0:07:43.3] JM: What are the problems with C++?

[0:07:45.4] TW: Well, one issue with C++ is that C++ code is not as secure as managed code, because you have all sorts of security vulnerabilities, like buffer overflow and similar that can occur in C++ code. The other thing is that – Well, high-level languages are sometimes just more productive because it's faster to write code in them.

[0:08:03.8] JM: So meaning it's more of a relative question. It's not that C++ is necessarily problematic. Although we could talk about some problems. But relative to higher level programming languages, it may be hard to be as productive in C++ as in, say, Ruby, or Python, or Java itself.

[0:08:25.3] TW: Yes, absolutely. I think that is why people more and more move to such dynamic languages or at least managed languages.

[0:08:34.4] JM: Yeah. What's the difference there? What's a dynamic versus a managed language? Can you define those terms?

[0:08:39.3] TW: So a managed language is a language where you have a runtime as a garbage collector. So you have like somebody who is looking after the way the objects get allocated or you have like automatic memory management. A dynamic language is one that data types are not defined at development time, but are defined at runtime. So when you say a certain value in, let's say, JavaScript, you don't say that the value will always carry a number, for example, but you just declare it as some value. Then at runtime, it can have any possible time.

[0:09:19.1] JM: The traditional compilers for Java have been around for a while. We've been developing in Java for a long time and the just-in-time compiler that we're talking about here – Actually, there is multiple types of compilers, and we could get into that a little bit later on. You've got client compilers and server compilers. But we're just talking about the idea of a compiler for Java specifically. Those compilers have been around for a while. Have they built up any technical debt over that period of time?

[0:09:49.7] TW: Well, I mean, natural way I would say, any software product that's around and in production for a longer period of time becomes harder to maintain. I think it's almost natural or any software product in production to [inaudible 0:10:02.5] a technical debt.

[0:10:04.1] JM: What are the kinds of – I don't know if technical debt is even the right term, but what are the issues that come with age that have been built up in the Java just-in-time compiler?

[0:10:15.3] TW: So, I mean, one thing with compilers in general actually is that they're usually optimized for certain workloads and for certain benchmarks. Then once you optimize them for certain benchmarks, then they have optimization phases that makes those things faster. But the way people program are changing overtime.

So the way people are nowadays using the JVM – Like the way they're using is very different from, let's say, 20 years ago. Nowadays, people run on the JVM other languages, like Kotlin, and Scala, or Groovy. So there is different constructs that reach the just-in-time compiler. Some of these things, some of the optimization phases that were originally put in, they are not taking into account the complexity of the new constructs. This is something where if you look at,

nowadays, workloads that are running on the JVM, one can do better on by writing a new optimization phase for the new type of program.

[0:11:18.3] JM: That's so interesting. Let's go into that in a little more detail. So when I compile a Java program I guess to Java bytecode, versus compiling a Scala program to Java bytecode, versus compiling a Groovy program to Java bytecode. So these are all different languages. They're compiled down to Java bytecode, and then the Java bytecode runs on the JVM. I think what you're saying there is that the in-memory object representations that those languages produce are in different formats. The Java objects looks different than the Scala object. Is that what you were saying?

[0:11:57.1] TW: It's not the object representations themselves, but the types of patterns you see in the Java bytecodes being used. So when you look at, for example, Scala, it has a lot of functional components. Then the bytecode shapes, sort of the sequence of bytecodes that would be common for Scala programs is different than for the average Java program. We see on average more allocations and more abstractions in some ways than in a Java program.

So it's not the representations of the objects themselves, but it's the code shape. It's kind of the sequence of bytecodes that gets executed that has just a very different pattern, just different patterns for different languages. For a language like Ruby, for example, it's more complex. The patterns are more complex. So there's more optimization potentially.

[0:12:53.9] JM: So we're talking about the just-in-time compiler optimizations that can be made. Yeah, I think you're saying that it's not problematic that Scala code compiles into a different code shape than Java code. What you're saying is that the JVM just-in-time compiler was created with just Java code in mind originally. Now we're in this world where you've got all of these different code shapes that are going to be hitting the just-in-time compiler and there's room to optimize for this new paradigm that we're in. Am I understanding correctly?

[0:13:29.6] TW: Yes, that's 100% correct. Yes.

[0:13:31.5] JM: Okay. What kinds of optimizations can you make? If you have a just-in-time compiler that's ready just for Java versus a just-in-time compiler that's ready for Java, and

Scala, and Groovy, whatever else, what kinds of different just-in-time compiler implementations do you want?

[0:13:50.4] TW: So in terms of the optimization phases, one that is very important is the optimization phase that we call partial escape analysis, and it is about removing object allocations. So in object-oriented languages, typically people allocate a lot. So the code is full with allocations when new objects get created. But very often, one can prove that an object is only necessary temporarily, which means the object is just created and a couple of instructions later, the object is no longer referenced.

These are scenarios where a just-in-time compiler, like the Graal compiler, can remove the object allocation. It can just virtually create the object and not physically create the object in-memory. So this is one of the optimizations where that is very good for modern code shapes, which includes shapes created by Scala or Groovy, but also by modern type of Java code. For example, Java code that uses the latest the string API expressions is also – It creates also such code shapes that are good for virtualization.

[0:15:00.4] JM: Can you define that term a little bit more? I think I didn't quite understand what you said. So what do you mean by virtualization there?

[0:15:06.4] TW: I mean, I virtualize an object. Usually, when you create an object, like let's say I have a point class and it's a new point in Java, and I assign two values, X and Y in the constructor of that object. Usually what would happen is that somewhere on your memory there is space reserved for that object and there is an [inaudible 0:15:29.4] instruction to write to memory the values of X and Y. But writing to memory is kind of a slow operation and it is not something you would like to do. You would ideally keep values in registers for as long as possible.

So when the compiler figures out that this point object actually doesn't escape, meaning it's not stores in some cache somewhere or it's not like – It's only temporary necessary. Then the compiler decides to not really allocate this object in-memory, but just keep the values, the X value and the Y value of my point object in two registers. Never store them to memory and only

remember that my point object is not actually physically existing on my memory, but just virtually existing in the form of those two registers.

[0:16:23.4] JM: I see. Okay, that's pretty interesting. So I think we should start to get into GraalVM, because we've given some examples of optimizations that can be made in the just-in-time compiler. GraalVM is an extension on the JVM. It's built to support more languages and to have a higher overall performance. Describe the motivation for GraalVM.

[0:16:53.2] TW: So for the motivation of GraalVM, we need to look a little bit into my background. Before I joined Oracle Labs, I was working at Google on the V8 JavaScript compiler. After my work at Google, I joined Oracle and mostly working on a Java compiler. Then I did that. I was always thinking like why are we doing always a new compiler for every new language? Because the languages look fairly similar, like Java, JavaScript, Scala, Ruby, Kotlin. Those languages have all big similarities from the point of view of a just-in-time compiler.

Then I started to think of ways how I could build just one compiler that runs a lot of different languages at high performance and including languages that are somewhat similar, but also currently run by different just-in-time compiler, like Ruby, Python or Java.

[SPONSOR MESSAGE]

[0:17:56.8] JM: In today's fast-paced world, you have to be able to build the skills that you need when you need them. With Pluralsight's leaning platform, you can level up your skills in cutting edge technology, like machine learning, cloud infrastructure, mobile development, devops and blockchain. Find out where your skills stand with Pluralsight IQ and then jump into expert-led courses organized into curated learning paths.

Pluralsight is a personalized learning experience that helps you keep pace. So get ahead by visiting pluralsight.com/sedaily for a free 10-day trial. If you're leading a team, discover how your organization can move faster with plans for enterprises. Pluralsight has helped thousands of organizations innovate, including Adobe, AT&T, VMWare and Tableau.

Go to pluralsight.com/sedaily to get a free 10-day trial and dive into the platform. When you sign up you also get 50% off of your first month. If you want to commit, you can get $50 off an annual subscription. Get access to all three, the 10-day free trial, 50% off your first month and $50 off a yearly subscription at pluralsight.com/sedaily.

Thank you to Pluralsight for being a new sponsor of Software Engineering Daily, and to check it out while supporting Software Engineering Daily, go to pluralsight.com/sedaily.

[INTERVIEW CONTINUED]

[0:19:36.1] JM: Where exactly does GraalVM slot in to this compilation process that we have described so far, where my Java program gets compiled into Java bytecode and then the Java bytecode gets executed by a just-in-time compiler. Where is the GraalVM slotting in and what are the responsibilities of the GraalVM?

[0:19:57.1] TW: So GraalVM itself is not necessarily – First, one of the things is that GraalVM and Java are two different project. So GraalVM sees itself as a way to run a lot of different languages on a lot of different platforms. Then we mentioned the language part that runs so many languages, and in terms of platforms, we can run on the JVM platform, and there it's kind of connected with Java. But it also can run on the Node.js platform and it can run on other platforms like the Oracle database or MySQL.

So one of the things when we run – I will talk now about the set up when we run in the context of the JVM platform. In this set up, GraalVM replaces the just-in-time compiler with the Graal compiler. The input to the Graal compiler is still Java bytecodes. But one of the things it manages to do is that it allows the compiler to very efficiently create compiled code for other languages that are built as interpreters in Java.

So what we did is, for example, for JavaScript, is that we built a JavaScript interpreter in Java, and then Graal compiler, when it executes this JavaScript interpreter, is able to understand the semantics expressed by this interpreter to also create efficient machine code for the JavaScript that interpreter is executing. This gets a little complicated, and it's like real hard to describe.

[0:21:26.9] JM: That's okay. That's okay. No. No. No. No. No. you're describing it quite well. I'm just thinking about what the best direction to go in. Before we dive into GraalVM as a part of the JVM a little bit more, you touched on a fact that this is portable and you can use it, for example, in an Oracle database. If we think of it as just an extension of the JVM, that's one thing. But if we think of it as something that can also be run in an Oracle database, what would it do if you ran the GraalVM in an Oracle database? What functionality would it serve for you there?

[0:22:02.0] TW: It would run JavaScript as one language. It would run Python as another language. We are also working in a way that it would run Java in the context of the Oracle database. So our goal is that GraalVM provides a way to extend existing applications with multilingual capabilities, because GraalVM, we can think of it as something that can run any language and you can embed it in any environment.

These environments, like Oracle database, or Java virtual machines, which is a tool of the environments we embed it in, they're really very different. When we embed in the Oracle database, we are embedding in a native application. When we embed in the JVM, we run more as a Java application, or we can extend Java applications.

[0:22:51.9] JM: So the Oracle database idea – So that, if I understand it correctly, it would let me – Instead of just executing SQL queries on my database, I could express a query in JavaScript or Python?

[0:23:04.6] TW: You could use JavaScript as part of your query, yes. You could run stored procedures in JavaScript. One of the demos we have there is where you use a Node.js module for validating email addresses, and you can use this module as part of your database query.

[0:23:23.5] JM: So this sounds kind of like an interpreter, but I guess it's a very portable interpreter. Do I understand that correctly?

[0:23:31.9] TW: Yes. It's not just – It's an interpreter plus a just-in-time compiler. So the JavaScript that you run in the Oracle database is also just-in-time compiled, because the crowd compiler that is part of this deployment will also just-in-time compile JavaScript there.

[0:23:48.3] JM: Okay. So I didn't prepare for this Oracle database example. I had read about that, but I would like to go a little bit deeper on that. What would be an application? What would cause a programmer to want to write JavaScript or Python against their Oracle database? What would be the practical example for doing that?

[0:24:08.2] TW: So, I mean, one of the examples is when you want to reuse an existing Node.js module that has a certain functionality, a certain filter functionality, for example. You want to reuse this as part of your theory or as part of a database trigger.

The advantage of running this piece of code inside the database is that, first of all, if you filter things, you will reduce the load that it needs to be sent from the database to the middleware. The other aspect in general is that keeping data in the database instead of taking it out and running it in the middleware can also have security advantages. As the database keeps the data more protected and you do not need to worry about potential leakage of data when you copy it to some other environment.

I mean, just to Oracle database, is one example. But another prototype we have internally is to integrate the Spark Platform, where this is something where you would make a Spark theory and then run GraalVM as part of this theory. There're a Spark extensions for Python and for R, for the R programming language where people are invoking these new engines as part of the query, and GraalVM can integrate the engine with the Spark engine and thus making programs that use Spark and R or Spark and Python faster, because there's less indirection and the integration is closer.

[0:25:44.4] JM: Okay. Let's go on that example again. So Spark code, if I remember correctly, if you write a Spark query, that is in Scala?

[0:25:52.7] TW: Yes.

[0:25:53.8] JM: Okay. So your Scala code in the "normal path", what most people are doing are your Scale code compiles down to Java bytecode. Your Java bytecode executes on the JVM and through the just-in-time compiler and queries your Spark working set of the RDD, the

resilient distributed dataset. But what you're saying is that if you were using GraalVM to instead do the just-in-time compilation to machine code, it's going to compile faster or run faster?

[0:26:33.4] TW: It's going to run faster. So first, GraalVM can run the Scala code, of course. But then GraalVM can also run – For example, if you want to use Python as part of your Spark theory, GraalVM can compile that Python code as well. Because for GraalVM, all languages are kind of similar. So GraalVM allows to even [inaudible 0:26:56.7] and cross language. So when you have a Scala code and it calls a Python function, then we can compile those two functions together into one compilation unit.

[0:27:07.6] JM: Oh, wow!

[0:27:08.3] TW: So it's really a tight integration.

[0:27:10.5] JM: Wow! Are people doing that? Or if you just done an experimental context?
[0:27:13.4] TW: Yes.

[0:27:14.6] JM: People are doing.

[0:27:16.1] TW: Yes, people are combining these languages. We have at the moment in terms of production deployments of GraalVM, only the deployment Twitter, where Twitter is run using GraalVM to run Scala code. We have at the moment a no production deployment of multiple languages that are run together. But we have demos on our website and we hope to also get the production deployment step.

[0:27:40.6] JM: By the way, just because this is a complex topic and I'm not sure I completely understand it. When you say GraalVM runs the code. So Scala code gets compiled down into Java bytecode, and GraalVM runs the Java bytecode. What does that mean? What is it mean that it's running it? Does that means it compiling it to machine code and then managing the hot code paths and doing that stuff? What is that term run mean?

[0:28:07.2] TW: Yes. It means it compiles to machine code and then runs the machine code.

[0:28:11.9] JM: Does it do the hotspot stuff or like funding the hot code paths, or is that still delegated – It does. Okay. Cool. Why is this an extension on the JVM? It sounds like this could be – If it can do all the things, like managing the code paths and doing – I don't know if you mentioned garbage collection, but why is it an extension on the JVM? Why isn't it just a replacement?

[0:28:33.6] TW: So the JVM has a lot of great applications for workloads that require, for example, very large heaps or just traditional Java workloads. So we do not see as a replacement of the JVM. We see ourselves as technology that can enhance the JVM. But it's similar with Node.js, for example. We see ourselves as not a replacement for Node.js, but as a technology that can enhance Node.js.

When GraalVM is integrated into another platform, it doesn't try to replace that platform. It tries to enrich that platform with the ability to run more languages, and that's also what it does in the JVM case. When we are running standalone, like when we run embedded in the database or any run embedded in Node.js, we use different architecture running on the JVM that has different tradeoffs. Meaning it's an architecture that is using less memory footprint, but it is an architecture that is not performing as well if you have very large heaps, for example.

[0:29:47.9] JM: So let's talk about the Twitter use case. How does Twitter use GraalVM?

[0:29:52.9] TW: So Twitter uses GraalVM to execute their Scala code, and Chris Thalinger, who is the lead compiler engineer there give a lot of presentations over the last year on how it is speeding up their micro-services effectively to use less CPU cycles per tweet and to reduce garbage collection times most likely due to the reduction in object allocations from the optimization I described in the beginning.

[0:30:24.6] JM: In Twitter's case, they got a reduction in CPU cycles and garbage collection – Do you say garbage collection time or garbage collection memory?

[0:30:36.4] TW: Garbage collection time, like time spent in garbage collections.

[0:30:39.6] JM: Right.

[0:30:40.2] TW: Or CPU time spent for garbage collection.

[0:30:43.8] JM: That's because when the Scala code goes from its Java bytecode representation to the machine code, being executed in machine code and being managed in machine code, the GraalVM is doing things like the object virtualization that you mentioned, here you could have more efficient representations of objects, for example.

[0:31:08.1] TW: Yes.

[0:31:08.5] JM: So this is really cool. There's a lot of stuff that you can build with that GraalVM idea that we have now, I think, reiterated like from three or four different angles. But just to take one contrasting angle. So interpretation on the Java platform has been done in other ways. Before, I've worked at a couple of places that have used J Ruby, for example. So they use Ruby. They'll build a Ruby on Rails app, and then when they get to the point where they need to really get that Ruby on Rails at performant, they will sometimes switch to J Ruby, which is a Java implementation of J Ruby.

But because this is a complicated subject, I want to just illustrate another contrasting example. Could you contrast the compilation model of J Ruby with what GraalVM can do?

[0:31:55.8] TW: Sure, absolutely. So J Ruby is written on the JVM in a way where the Ruby source code is at runtime converted to Java bytecodes, and then that Java bytecodes executed by the Java virtual machine.

In our version of Ruby that we call Truffle Ruby, the Ruby source code is interpreted by an interpreter written in Java. When a certain Ruby method gets hot, the Graal compiler is doing this partial revelation magic to convert the Ruby method to machine code.

So the main technical difference here is – And this also leads then to the difference in performance characteristics you get is that the translation from Ruby source code directly to Java bytecodes, it's a little bit difficult to do, because Ruby is a very dynamic language. So the data types all do not have defined types, sort of the data structures do not have defined types.

But when you translate its Ruby code to Java bytecodes, then you need to assign types to those bytecodes, because Java bytecodes are originally designed for the Java programming language that is statically typed and you need to specify, for example, that a certain variable of field is of type integer, so it's a number.

This mismatch between Ruby's dynamic way of thinking about things and the static typing of the Java bytecodes is for great performance issues when you do such a translation. Those performance issues are also associated within the way that the Java JIT compiler is looking at those bytecodes in a way, it's like profiling those bytecodes.

In the context of GraalVM, we do not have that stack. So we are doing this automatic transformation from the definition of interpreters to machine codes without dynamically generating bytecodes. We still use bytecodes as the input format to GraalVM, to the Graal compiler. So the Graal compiler is still a compiler from bytecode to machine code, but we do not dynamically generate new bytecodes if you load a new Ruby function, let's say.

[0:34:28.1] JM: Okay. So what about Java code itself? So if I run my regular vanilla Java code in the traditional JVM, versus running it in a JVM with GraalVM as an extension, is my Java code going to run more efficiently?

[0:34:46.5] TW: It depends. It depends on the type of workload. It can run more efficient, or it can also run less efficient. Different JIT compilers – JIT compilers are very complex piece. They have a lot of different optimizations, optimization phases. In GraalVM we have like more than a hundred different optimization phases. Those optimization phases also have complex interactions. One optimization phase feeds into the other or hinders the other from performing.

So when you compare two JIT compilers, the answerer which one is faster is almost always it depends. There are some code shapes where GraalVM is doing better. In particular, those code shapes around object allocations or those code shapes around other languages, like Scala code or Kotlin code. But there are also some code shapes where we are doing worse in particular for code that is looking like very traditional type of Java code or other smaller things.

We would definitely encourage people to give GraalVM a try for their Java applications. We are actually interested to hear feedback and to hear from people who have benchmark setups, because our goal is to continuously improve the performance for GraalVM.

[SPONSOR MESSAGE]

[0:36:09.7] JM: Citus Data can scale your PostgreS database horizontally. For many of you, your PostgreS database is the heart of your application. You chose PostgreS because you trust it. After all, PostgreS is battle tested, trustworthy database software, but are you spending more and more time dealing with scalability issues? Citus distributes your data and your queries across multiple nodes. Are your queries getting slow? Citus can parallelize your SQL queries across multiple nodes dramatically speeding them up and giving you much lower latency.

Are you worried about hitting the limits of single node PostgreS and not being able to grow your app or having to spend your time on database infrastructure instead of creating new features for you application? Available as open source as a database as a service and as enterprise software, Citus makes it simple to shard PostgreS. Go to citusdata.com/sedaily to learn more about how Citus transforms PostgreS into a distributed database. That's citusdata.com/sedaily, citusdata.com/sedaily.

Get back the time that you're spending on database operations. Companies like Algolia, Prosperworks and Cisco are all using Citus so they no longer have to worry about scaling their database. Try it yourself at citusdata.com/sedaily. That's citusdata.com/sedaily. Thank you to Citus Data for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:37:55.0] JM: GraalVM itself is written in Java. As we discussed earlier, there are some issues with C++, but C++ is commonly regarded as the language that you use if you’re writing systems, or low-level, or high-performance systems, like a garbage collector, or an operating system. I think if not C++, people would be saying, “Oh, use Go, or use Rust.”

I am intrigued by the selection of Java as the language to implement GraalVM in. Why Java?

[0:38:32.5] TW: So why Java? Because we like Java, I guess. I mean, we were initially from the project, the project's history is coming out of a research virtual machine called Maxine Virtual Machine, who's idea it was to write the whole virtual machine in Java. This is why the original implementation for the Graal compiler, we pick Java as the implementation language.

I think having done that in the project has been an excellent choice, and the Graal compiler is a very good proof point that Java is efficient enough to run system software. We are very happy with Java in particular also because of the good IDEs around it. So we feel we are more productive in Java.

One interesting side effect of this, which is I think underestimated, is that because of all of our team is writing every day in Java, all of the team has a very good understanding of the code patterns and the Java programming language. So kind of writing your compiler in the language you're compiling is overall, I think, a great strategy, because also develop a good mindset for the patterns that occur in that language.

[0:39:46.3] JM: Dog fooding.

[0:39:47.3] TW: Yes, it's dog fooding from the beginning.

[0:39:49.3] JM: Cool. So GraalVM has Graal and Truffle as its main parts. You alluded to Truffle earlier when you're talking about Ruby. Truffle is used to allow you to run Ruby on Graal. Could you describe the functionality of both Graal and Truffle?

[0:40:08.1] TW: So in Graal we kind of refer to the overall project, and Truffle is a part of Graal. I would characterize it like that. Truffle is what we call this language implementation framework that our JavaScript interpreter, our Ruby interpreter is built upon. So as a regular user of GraalVM, one should not need direct contact with Truffle. But if you want to implement your own language on GraalVM, you will use Truffle for it.

Truffle in the end is more – It's just a set of APIs that you develop your language against such that it runs on GraalVM and such that it gets all the benefits of GraalVM's independent and

language agnostic tooling. Because one of the benefits of GraalVM is it's not just executing the languages, but because we have this common API that all of the languages are using. We do have, for example, a language agnostic profiler or a language agnostic debugger, kind of tools for languages that are then working across all languages that implement that API.

[0:41:17.1] JM: Okay. So if I understand it correctly, Truffle helps you write an abstract syntax tree for a language. So if you want to execute a language against GraalVM, you can use Truffle to build an abstract syntax tree from that language than Graal will be able to understand. Is that correct? Am I understanding that right?

[0:41:40.7] TW: Yes, that was correct.

[0:41:42.5] JM: That abstract syntax tree, that AST, does that mean that Graal is going to be evaluating an AST rather than just Java bytecode, or is this an AST that represented in Java bytecode?

[0:41:53.8] TW: Yes, it's the later, and this is a very important point and many people misunderstand that, because when you describe it, "Oh, that's an EST, and then that's run," then people feel like somehow have some – Yeah, direct parts. You're not like — it's a very important point, like it is an EST, but that EST, this abstract syntax tree is consisting of Java objects and the methods on that EST, on those Java objects, are all defined in Java bytecodes.

When the Graal compiler is doing this partial evaluation of this EST interpreter to create the compiled code from it, all it does it just look at the Java bytecodes of that EST interpreter. The way the semantics of Ruby is communicated to the Graal compiler is only via Java bytecodes, but it is via the Java bytecodes of the execute methods of the classes that form the Truffle EST.

[0:42:55.8] JM: Okay. Just as you mentioned, that Java code compiles to a certain code shape in Java bytecode. Scala code compiles to a particular code shape as well. I'm guessing that the Truffle abstract syntax tree that you could build for any language. Basically, Truffle is a way to interpret languages like Ruby, or Python, or R into an abstract syntax tree and that abstract syntax tree probably has a somewhat consistent code shape, and then you can optimize Graal for evaluating the abstract syntax tree. Is that right?

[0:43:31.8] TW: Yes, that is correct. One important aspect here is that this abstract syntax tree is like you kind of extend – We have like a node-base class for this abstract syntax tree. When you implement your own language, if you implement Ruby, then you extend that node-base class to add your own classes of nodes to that abstract syntax tree. Then you overwrite the execute method on that node and then thus execute method becomes the semantic of that operation.

That is very powerful, because in our system, for example, if you look at our JavaScript interpreter, in our system, if you want to change the way JavaScript edition works, for example, there's just a single point in the whole architecture where it is defined how to add two JavaScript numbers. It's a single point of definition for any piece of semantic. That's very different to other JavaScript engines, like let's say, the V8 engine, where there are all sorts of specialized machine code or duplicated definition for certain operations.

[0:44:37.2] JM: Okay. So this sounds like an intermediate representation. So when I've had conversations with people who work on LLVM, I think when people are trying to make their language compatible with LLVM, they compile whatever language their programming in, or whatever language they're trying to write an LLVM compilation system for, they compile it to an intermediate representation. So this abstract syntax tree in GraalVM, would you describe it as an intermediate representation?

[0:45:13.7] TW: I would describe it as something slightly more – It's kind of something more enhanced than an intermediate representation, because the thing is that – When you compile something to Java bytecodes or to LLVM bit code or something like that, you take your input, then you make your transformations, and the output is then that set of bit code or that set of bytecode.

But this is not the case as with Truffle EST's. With Truffle EST's, we convert the source code to Truffle EST's, yes. But then we execute those EST's. While those EST's are executing, they're profiling the application. Only once we profile the application and a certain EST is hot, then we convert this EST with this partial revelation to machine code.

This is an important additional step in particular when you want to execute a dynamic language, because they're having an EST that's transformable and able to react to how the program executes. It's very important for performance.

So it's kind of an intermediate representation, but it's an intermediate representation that can itself execute and change based on the program behavior before it is turned into machine code.

[0:46:30.8] JM: Okay. I think I understand. So have you looked at LLVM much? Would it be worthwhile to contrast LLVM to the GraalVM?

[0:46:40.0] TW: I mean, GraalVM, we do support LLVM bit code as input. This is actually the way how we execute C, C++, or Rust code. We execute that code by having built a Truffle interpreter for LLVM bit code. Then we can do just-in-time compilation for the Truffle interpreter to create compiled code for LLVM bit code.

We use this in particular for things like when we have, let's say, Ruby method, and then Ruby calls a C extension, like some module that's written in C. Then we can compile both of these parts of the program, the Ruby code and the C code to machine code with the Graal compiler. As with all GraalVM base languages, they can then be [inaudible 0:47:25.1], which can lead to performance optimizations on the boundary between, let's say, C code and Ruby code. That's how LLVM and GraalVM are currently sort of connected.

In general, from a very high level, the projects are similar in name maybe because they both have VM at the end. But in terms of the scope and the way they are thinking about things, it's very, very different. So LLVM is really great for static languages. When you compile a static language to machine code, then LLVM is a good way. That's also why in GraalVM we do support static languages via LLVM bit code. So we recommend to compile the static language to LLVM bit code and then we execute that LLVM bit code.

GraalVM itself sees itself more as a runtime, as a managed runtime with also built in runtime facilities like a garbage collector and just-in-time compilation and these things. This is kind of just from the history of this project, where LLVM was mainly focusing on static languages and to be a part of a static compiler.

[0:48:38.3] JM: I can't remember. Does LLVM have a garbage collector?

[0:48:40.4] TW: No. It does not.

[0:48:42.9] JM: I see. So do people that use LLVM, does that just not garbage collect? What's the garbage collection story there?

[0:48:51.1] TW: So there are projects that do add garbage collection functionality for the specific use case, but they had to do extra work. But in general, when you have, for example, a C program and you compile this LLVM, then you will not have a garbage collection, because C doesn't need one, right?

One of the projects that LLVM is used is in the Azul Compiler, but I use it in the context of the JVM. But we had to kind of do quite some work to add some amount of garbage collection support to the LLVM compiler. Then also write down garbage collector, or connect it with the JVM's garbage collector, right? So there's no garbage collector by default available from LLVM.

Not only that, there's also some functionality necessary in order to support garbage collected code. Some bookkeeping in the compiler, because the compiler needs to know which values are pointers and which values are integers or numbers. This functionality is also something that, at least, originally was in the LLVM. I think they are now adding some of these. But that's just a different – I think a very different use case, I think, LLVM versus GraalVM.

[0:50:10.3] JM: Okay. Let's begin to wrap up. This has been a really fascinating show. What's the biggest vision for what purposes GraalVM could solve? What could it enable beyond just improving the performance of programming languages in general and creating compatibility layers between things like R, and Ruby, and Java, which is that's ambitious enough. Tell me, what's the big vision? What motivates you to continue working on this project?

[0:50:39.2] TW: So I think the main part of the vision is we want to make sure that every language gets good performance and can run in any environment. I think that's our goal. Our goal is that programmer do not need to choose a language based on some performance

characteristics, or based on some tooling availability. They should use the best language for the task or the language they just like more.

GraalVM tries to be this universal runtime for any language and not have this isolated runtime where every language has its own compiler, their own garbage collector, their own way of doing things. We just think that it's more efficient and better overall to have this support of one compiler, one VM that runs many languages and that can be connected to many different environments. This is what we are hoping for, that this project will help us.

So far, we're very excited about the positive feedback we got over the last couple of months since our first release and we're looking forward to more feedback, more input. We're also looking forward to more people joining the GraalVM ecosystem and the GraalVM effort either by implementing their own language on top of GraalVM or by embedding GraalVM into their own engine or their own application, or also by implementing a language agnostic tool or other language agnostic functionality in the context of GraalVM.

[0:52:11.6] JM: I think this is really exciting. It's funny, because I feel like there are some kind of trend here between some other topics we've covered recently. We covered web assembly recently. We covered – We've done a lot of shows about React Native, and Flutter, and now GraalVM. People want flexibility in their choice of programming languages and tooling and UI frameworks, and they don't want to feel restricted by a specific operating system, a specific platform. Whether they're on the web or whether they're on mobile or whether they're designing a user interface for a car. These are not constraints that should exist, and it seems like they won't exist in 5 or 10 years for state of the art and development at least. People just don't want these things. So I think they're going away, and it's exciting to see you working on something that relates to that.

[0:53:08.8] TW: Yes, and I very much agree. I think these things should not exist and people should use the language of their choice and the language they feel more productive in, and we will do everything to help achieve that type of future.

[0:53:22.6] JM: Thomas, it's been great to talk to you, and I'm looking forward to covering GraalVM more in the future.

[0:53:26.9] TW: Thanks, Jeff. Great talking to you as well.

[END OF INTERVIEW]

[0:53:32.2] JM: Cloud computing can get expensive. If you're spending too much money on your cloud infrastructure, check out DoIt International. DoIt International helps startups optimize the cost of their workloads across Google Cloud and AWS so that the startups can spend more time building their new software and less time reducing their cost.

DoIt international helps clients optimize their costs, and if your cloud bill is over $10,000 per month, you can get a free cost optimization assessment by going to D-O-I-T-I-N-T-L.com/sedaily. That's a D-O-I-T-I-N-T-L.com/sedaily. This assessment will show you how you can save money on your cloud, and DoIt International is offering it to our listeners for free. They normally charge $5,000 for this assessment, but DoIt International is offering it free to listeners of the show with more than $10,000 in monthly spend. If you don't know whether or not you're spending $10,000, if your company is that big, there's a good chance you're spending $10,000. So maybe go ask somebody else in the finance department.

DoIt International is a company that's made up of experts in cloud engineering and optimization. They can help you run your infrastructure more efficiently by helping you use commitments, spot instances, rightsizing and unique purchasing techniques. This to me sounds extremely domain-specific. So it makes sense to me from that perspective to hire a team of people who can help you figure out how to implement these techniques.

DoIt International can help you write more efficient code. They can help you build more efficient infrastructure. They also have their own custom software that they've written, which is a complete cost optimization platform for Google cloud, and that's available at reoptimize.io as a free service if you want check out what DoIT International is capable of building.

DoIt International art experts in cloud cost optimization, and if you're spending more than $10,000, you can get a free assessment by going to D-O-I-T-I-N-T-L.com/sedaily and see how much money you can save on your cloud deployment.

[END]

