EPISODE 1536

> *"**SR:** We implemented a different garbage collection algorithm that is truly pause less, in that we don't need to pause the application threads by introducing what's called a read barrier. Every time you access an object, we can intercept that access and we can say, "Right, do we need to do anything to make sure that we're safe in terms of marking objects and in terms of relocating objects?" By doing that, what we can then do is say, "Right, we can do garbage collection concurrently with the application threads." That means you don't see any pauses as a result of the garbage collector doing its work in whilst the application threads are paused."*

[INTERVIEW]

[0:00:43] AD: Simon Ritter, welcome to Software Engineering Daily.

[0:00:46] SR: Hi. Thanks for having me on.

[0:00:47] AD: Absolutely. You are the Deputy CTO at Azul Systems. Just from looking at your background resume, it seems like you've been around the Java world quite a while, probably know about as much as there is to know about Java, Azul, Oracle, Sun Microsystems. For those who don't know, maybe just give us a bit of your background and what you do.

[0:01:06] SR: Yes. As you said, I've been doing Java really right from the very beginning. I joined Sun Microsystems way back in January 1996, which makes me feel really old. I literally joined about the same time that JDK 1.0 came out. I followed Java all the way through the Sun years. Got acquired by Oracle. Spent another five years at Oracle, and then joined Azul a little bit over seven years ago.

A lot of the time, I've been really focusing on helping people understand what Java is and promoting it and driving adoption of Java through various different ways. As you say, I've been heavily involved in various aspects of that. Currently I'm on the Java SE Expert Group for the standardization of Java. I'm also on the OpenJDK vulnerability group, in terms of how we

address vulnerabilities in Java and updates and so on. I'm on the JCP Executive Committee as well. I've got a good feel for what's going on in the Java world.

[0:02:06] AD: Yeah, absolutely. I'm sure you've seen a lot of changes and I want to save some time at the end just to see like, “Hey, what do you see that's new in the Java world that get you excited?” We'll talk about that. But I want to start basic. What is a JVM? Why do I need one?

[0:02:18] SR: Right. This is really one of the things that makes Java the platform so popular is the fact that you have the JVM. I would really describe the JVM as the crown jewels of the Java platform. What it allows you to do is to have a managed runtime environment for your applications. That has a number of advantages. If you go right back to the beginning of Java, one of the things that you would have seen is the catchphrase ‘write once, run anywhere.’ That was a big thing at the beginning, which was you could take the same Java application, you compile it once and then run it on Intel, Windows, or ARM and Linux, or wherever you wanted to. You didn't have to recompile. Didn't have to change any of your code.

Certainly, in terms of moving into the cloud, that's actually very beneficial, because you don't have to worry about exactly what platform you're going to be deploying on now. The other thing that the managed runtime environment gives you is it takes away some of the hard work that the developers have to do. If you look at languages like C and C++, then you're responsible for allocating memory for where you want to store things. Most importantly, you're responsible for figuring out when you don't need that space anymore and freeing it up and remembering to do that, so you don't get memory leaks and things like that.

One of the big things that you get in the JVM is this idea of automated memory management. Space is allocated in the heap by the JVM and it keeps a track of when you're using objects, so that when you finish with them, it can run this thing called the garbage collector and then reclaim that space and make it available for new objects as you create those. That's one of the running, really key things.

Then from a performance perspective, just the ability to take the code that's being compiled into the byte codes, the virtual instruction set and compile that at runtime. Rather than doing it ahead of time with a static compiler, what we do is we do it at runtime. A big advantage of doing it that

way is you can actually, in terms of the JVM, you can actually see exactly what that code is doing. You can profile it, see how it's working. Then when you compile it into native instructions, you can optimize it in the best way for what's actually happening in that code, versus if you do it ahead of time, you don't know what's going to be happening in the application at any given time, so you have to make certain assumptions and they may not be as good as when you're doing it at runtime. Those are the key things is really that idea of automated memory management, just-in-time compilation versus static compilation.

[0:04:50] AD: Very cool. I want to get into more details about JVM. One thing you mentioned is like, the benefits of Java is write once, run anywhere. We've seen a little bit more of that with Docker, too. How does Java and Docker interact? Do people run Java on Docker? Do they not really need it since Java is already running in the JVM?

[0:05:07] SR: Absolutely. I mean, as you say, Docker takes the idea of platform neutrality to one step further, so that you're saying, okay, let's put everything into this container that I need to run my service typically, or application. Then I can ship that around. I can put it wherever I want to. One of the big benefits of using containerization and things like Docker is that you can also have certain number of resources allocated to a specific container, and that's very good in a cloud environment.

You say, “Okay, I've got two V cores. I've got 8 gigabytes memory,” and you know that's what you're actually paying for. In terms of how Java plays in that, it plays very well, because you can put the JVM into the container, and especially now that we've got things since JDK9 with the introduction of modularity, what that allows us to do is to create a runtime, which is tailored specifically to the application that's going to run on that. Whereas before, we would have the whole JDK and we'd share that amongst many different applications. That would result in having somewhere in the region of 300, 350 megabytes of Java runtime.

Now by using JLink as a command and then stripping out all the bits you don't need, you can shrink that down, if you had the most basic application, you'd be talking about maybe 30 megabytes of runtime. Obviously, as you start doing things in your service, it'll be a bit bigger than that. It really does reduce it by an order of magnitude. Put that into your container, it's

tailored specifically for your application and everything works really nicely in that environment. Yes, Java does fit very well into this whole idea of Docker containers and so on.

[0:06:49] AD: Very cool. Okay, so back to JVMs, what are my options for choosing a JVM, if I need to choose one? What are my options there?

[0:06:57] SR: Right. I guess, the primary thing that most people look at is OpenJDK. OpenJDK was the project that was created back in 2006, I think it was, by Sun Microsystems when they decided to open-source Java. Most people will look at that and they'll say, “Oh, okay. That's the reference implementation for the Java SE standard. That's the one we're going to use.” Whether it's the Oracle implementation, which people have used a lot in the past, or from a now much wider variety of providers of OpenJDK distributions.

This is certainly something we've seen over the last few years with Oracle changing their licensing terms. There has been a real expanding of the number of distributions are available. You've got ones from Azul. Obviously, you've got ones from Amazon for running on AWS called Corretto. Even Microsoft have their own JDK distribution, which somebody who was involved in Java right at the very beginning and all the shenanigans that went on with Microsoft then, it's quite funny to see them come full circle and go back to creating an OpenJDK distribution.

In terms of alternatives to that, there's really two things that you could look at. One is IBM. They have their J9 implementation. The idea behind that was a complete clean room implementation of the specification. They didn't look at OpenJDK. They didn't use any of the source code. They went off completely, did it from scratch. That's still obviously used a lot for IBM software. If you're running things like WebSphere, then you'll be running IBM J9. They did open source it. That's one option.

The other option is what we do at Azul, where we take OpenJDK as a starting point, but then we make some significant changes in terms of the way that the JVM works. That's what we call Platform Prime. It's based on OpenJDK, fully conformant with the standard. We run all of the TCK tests, pass everything to make sure that it is a drop-in replacement for OpenJDK. Don't have to change any of your code. Don't have to compile any of your code. You can just use that and get different performance characteristics from that.

[0:09:13] AD: Got that. You mentioned the standards and the TCK tests and different things like that. Is that something where every implementation comply to that, or is it more like the SQL standard where it's like, “Hey, we comply, but maybe there'll be tweaks here and there that don't comply”? Or how well does everything fit that standard?

[0:09:32] SR: I'd have to say, everything fits that standard very, very well. One of the nice things about the Java platform is that we have managed to maintain that compatibility and standardization. Sun started the Java community process as a way of having an open standard for Java. We have all of these, what they call Java specification requests, or JSRs. Each version of Java has that specification. There's a very clear definition of what's in the language syntax, what the JVM has to do from a functional point of view, just not how it does it. Then the core class libraries that are always available for your application code.

By using the TCK, and literally everybody – there's probably a couple of exceptions in terms of distributions who don't pass the TCK, but almost everybody does pass the TCK. It gives you that level of confidence that if you're moving from one TCK-tested JDK to another TCK-tested one, your application is going to run in exactly the same way from a functional point of view. Different performance characteristics maybe, but the way the code actually works will be identical.

[0:10:40] AD: Yeah. Very nice. It even makes me think back to Python. There were like different Python implementations. I think there were some site things you couldn't do in one or the other, so that's pretty nice that it mostly works there. I want to talk a little bit more about just what Platform Prime is doing, but you mentioned Oracle subscription changes, billing model, and I've heard a lot of consternation I would say, in the Java industry about new billing model stuff. I don't know how much we’re going to get into that, but can you tell me what Oracle's new billing model is and how do you think that's going to affect the community, maybe open JDK, different things like that?

[0:11:14] SR: Yeah, sure. I mean, as I mentioned, Oracle changed the licensing that they use, and this is something they've done a couple of times now. When Oracle took over Sun Microsystems, they continued using pretty much the same license that they had from Sun,

which was what they then called the Oracle binary code license. That effectively meant you could use the Oracle JDK freely wherever you wanted to, unless it was in embedded systems, or on mobile phones. The history behind that was that was the one place where Sun was making money from Java, so they wanted to force people to license it separately for those types of environments.

For desktops, laptops, server-side applications, you could use the Sun and then Oracle JDK completely freely. There was no problem with that. Then when JDK9 came out, a little bit after that, Oracle decided that they changed the license and they would impose more restrictions in terms of where you could use it for free. Literally, if you were running Java in a commercial environment, in a server, web server, something like that, then you would need to purchase a Java SE subscription.

They've then introduced yet another license for JDK17, which is called the No-Fee Terms and Conditions, which sounds like it could be free, but it's a little bit – you have to be very careful about the licensing there. It's only for three years anyway. What they've done most recently is not change licensing again, but they have changed the way that they calculate the cost of the Java SE subscription. It used to be that you would look at all the machines you were running Java on, you count how many cores you've got in the processors. There was a little bit of complexity in terms of using core multipliers, which meant you had to multiply by a certain factor, depending on how fast your processor was and things like that.

It was really based on the number of machines you had and the number of processors, which was pretty logical. What they've done now, as they've said, "Well, we're going to make it much simpler. All we're going to do now is charge you based on the number of employees that you have in your company. That's full-time employees, part-time employees, contractors, and so on." Sure, if you're in an IT organization, that's not too bad, because most of your people are going to be using Java. If you're in a company where you've got lots of people doing manufacturing, lots of people doing deliveries or whatever, then to say that you're going to have to pay everybody as using Java, even though you may only have 5% of your employees using Java, it's quite a shift in terms of the way people are being billed for the use of the Oracle JDK.

[0:13:54] AD: Yeah. I mean, you think of just the company that comes to mind would be Amazon, or Walmart that have a ton of retail and warehouse employees, but not using Java directly, like their technicals.

[0:14:06] SR: Exactly. Yes.

[0:14:08] AD: Okay. When they were billing more on a processor per-core basis, were there weird architectural shenanigans to make that bill lower, or was it just not worth the hassle of some of that stuff?

[0:14:21] SR: I don't think it was worth a hassle, because most people would just look at it as like, “Okay, what do we need to do to run our applications?” They were more concerned about thinking, “Okay, we need this number of processors. We'll choose the latest Intel architecture, or we'll choose the previous generation of architecture, or whatever,” and that's where that multiplier came in to determine how many percentage you had to increase the value by for the processes you were using. Most people would focus just on, how do I deliver the services that I need to and then work out the cost from that.

[0:14:53] AD: Okay. Let's talk about Azul Platform Prime. Tell me about Prime and what it does and why people are choosing to use that over OpenJDK, over Oracle, things like that.

[0:15:03] SR: Yeah. As I said, the great thing about the JVM is you've got this managed runtime environment, garbage collection, just-in-time compilation, and so on. What we did originally was we looked at the way that the systems work and said, is there a way that we can improve things? One of the big issues that people have always had in the early days of Java and really up to now is the whole idea of garbage collection pauses. It was something that Java was famous for very much in the beginning and even now people are still trying to eliminate the problem from their application code.

It really comes down to the fact that for most algorithms that do garbage collection, in order to do it safely, meaning you're not going to corrupt any of your data, or miss things that are being used, you have to pause the application threads whilst you do the work of the garbage collector. That's obviously, if you're marking to figure out which objects are still in use, you want to do that

without people making changes. Same thing if you're going to move objects around within the heap to do compaction, you don't want people changing the objects as you copy them, because you might lose those changes. The safest way to do that is to just pause application threads. That works very well, as long as you don't have too much load on the garbage collector.

As you start seeing an increase in the use of the heap and place more stress on it, what you'll see is more and more garbage collection pauses, because the garbage collection has to kick in more frequently and you can go from having maybe a few milliseconds of garbage collection pause that can extend out into seconds of GC pause. The longest GC pause I ever came across was one customer who had a GC pause of one and a half days, which was just an insane thing to have. I remember saying to them, “Well, why don't you just restart the application? It's got to be quicker.” They said, “No, it isn't. Because the time it takes to load the data to start the application is actually longer than doing garbage collection pause.”

[0:17:04] AD: Can I get some details? How big was the heap for that with the day and a half pause?

[0:17:08] SR: That was terabytes. That was literally tens of terabytes of heap space. They had a very complex environment, which did result in this incredibly long garbage collection pause. I remember saying to them, “Well, how do you even know after an hour that it's still doing anything, that you just assume it's crashed?” They said, “Oh, no, no. We've got instrumentation. We can see it's still doing the garbage collection and we know that it will finish after a day and a half.”

[0:17:35] AD: Did they have control over when that garbage collection ran, given that it's a day and a half? Could they run it in –

[0:17:40] SR: No. That's a very important part about how garbage collection works is it is non-deterministic. You can't predict when it's going to happen. It really depends entirely on how much object allocation you do, how quickly you use the objects and release the links to them and so on. This is what we did. We said, okay, well, we look at that and this is one of the biggest problems that people see with Java. What we then did was say, okay, how can we take a different approach to that? We implemented a different garbage collection algorithm that is truly

pause less in that we don't need to pause the application threads by introducing what's called a read barrier.

Every time you access an object, we can intercept that access and we can say, “Right. Do we need to do anything to make sure that we're safe in terms of marking objects and in terms of relocating objects?” By doing that, what we can then do is say, “Right. We can do garbage collection concurrently with the application threads.” That means you don't see any pauses as a result of the garbage collector doing its work in, whilst the application threads are paused.

I mean, other algorithms have taken similar approach and tried to do similar things. The algorithm that gets used a lot in hotspot, the OpenJDK, is called G1. That does a good job, but at certain points, it will actually get so heavily loaded, it will say, “Right. I need to stop and do a full compacting garbage collection.” We don't have that. We don't have a full back situation where we need to do that. That does allow us to scale up to literally tens of terabytes of heap space. We have customers who do this. When you're doing credit card fraud detection, you need these massive amounts of data and you need to have it all in memory at the same time. When this customer had a day and a half garbage collection pause and we put Prime on there, suddenly that one and a half day pause just disappeared and they're like, “That's amazing.” It was really good for them in terms of doing that.

[0:19:45] AD: Are there trade-offs to this pause of one? Is that going to use some resources? Yeah.

[0:19:50] SR: Well, the answer is yes, because there's no such thing as a free lunch. If you're going to do the work whilst pausing the application, that work has to go somewhere if you're doing, still running the application. Yes, it is working concurrently. Effectively, what we're doing is doing some work at the same time as the application, which then can overall, can reduce the throughput of your application, which then brings us to the second thing that we did. Because we said, okay, that works really well now. We've solved the problem of garbage collection. What we now need to do is look at how can we improve the throughput of the applications.

This is where we looked at this just-in-time compilation system. The way that works is it really happens in three different levels. You've got the first level, which is just interpreting. You take

your bytecode, you convert each bytecode as you see it into native instructions through a template, and then you execute it. That's very inefficient. The whole idea of hotspot, the name of the JVM, was to identify methods that are getting called very frequently, hotspots of code, and then compile them into native instruction so you don't have to do that interpreting every time.

That happens in two stages. You've got what's called the three originally named C1 JIT compiler, which will compile code very quickly, but doesn't apply many optimizations. You get the code compiled, you can start running it a little bit faster. Then what we do is we profile that code to see how it's actually being used in the application. Then when you get to a second threshold where it really is a hotspot, we pass it to the, again, very originally named C2 JIT compiler. That will then take the profiling information and recompile it, optimize it, and generate much more efficient code.

What we said was, well, can we do even better than that? We looked around, we found there's a really good open-source project called LLVM. It's been around for 20 years now. It's all about the back-end of a compiler, generating heavily optimized code from intermediate representation. We took that code and we integrated it into the JVM to make it work as a JIT compiler. Then we contributed all that code back to the project, because we said, well, open source, we're good open-source citizens, we'll put back what we did.

What that allows us to do is to optimize the code more efficiently in many situations. We can take advantage of some of the very low-level features that the CPU architecture has more efficiently. As an example of that, modern processors all have this idea of single instruction and multiple data vector operations. You get different levels of that, so you've got AVX, AVX2, AVX5, 12, depending on how wide the registers are. What we can do is we can make use of that more efficiently and compile code to use those operations in situations where C2 would not do that.

There's lots of things where the compiler just gets a bit confused, as soon as you start doing conditional statements and if statements, most compilers give up and go, “Oh, no. We're not going to bother with that.” Whereas, Falcon, which is our LLVM based JIT compiler will go, “Oh. Yeah, I know how to do that.” It will generate much more optimized code. The advantage of that, as you see, we got the C4 is our algorithm for garbage collection, the continuous concurrent compacting collector. That, as you said, the tradeoff of doing that is that you then reduce

throughput. By using a different JIT compiler, we can improve throughput and that takes account of the fact that we're now doing GC work at the same time.

The overall effect you get is higher throughput with lower latency. We solved both of the problems of what's happening with the JVM in the best possible way. That's the two main things that we've done internally in the JVM. Then we took the next view of things –

[0:23:56] AD: Hold on a sec. I want to talk about some of these compilers, because I think this is cool. LLVM, I've heard of this. This is open-source compiler types up. Does it work for many different languages? Is that right?

[0:24:06] SR: Oh, yes, yes. I mean, it is all about compiling code from the – They don't really focus too much on the language itself. You've got the front-end of the compiler that takes Python, or C, or C++, generates intermediate representation and then passes that to the back-end of the compiler that generates the optimized code. We've taken that back-end part and then integrated it, so it takes the same intermediate representation and just generates better code.

[0:24:33] AD: Got you. Okay. This is maybe a dumb question, because I don't know a lot about compilers, but is LLVM used for both the just-in-time compiling that Java does, but also the ahead of time compiling that other languages might need to do?

[0:24:44] SR: Oh, yes, yes. Because this project is used by lots and lots of people and there are lots of companies who contribute to this. People like Intel, NVIDIA, Apple, Microsoft, all of the people who know about these kinds of things are using it to generate the compile code. I mean, even looking at one of our competitors, Oracle have the GraalVM. GraalVM uses LLVM as part of the way that they handle other languages.

[0:25:12] AD: Very cool.

[0:25:13] SR: It's a very popular platform.

[0:25:14] AD: Okay, cool. I have a question about GraalVM, too, that I think will come up in the next part. You're talking about a couple of different ways that Python and Prime helps. You

mentioned latency with the garbage collection throughput, with the compiler third way. You were just about to go into that one before I interrupted you.

[0:25:31] SR: Yes. One of the things that you see with whole idea of JIT compilation is that you start your application and it has to go through this idea of interpreting byte codes, identifying which methods need to be compiled, using C1 to compile them, profiling them, recompiling them with Falcon or C2. If you start the same application again, it does exactly the same thing. It has no memory of what happened before. It goes through the same process of learning which methods compile doing all that work. We thought, how can we solve that problem?

What we now do is we let the application run and warm up, so that you've got all your frequently used methods compiled, and you can run it for an hour, you can run it for a couple of days or a week if you want to. Once you're happy with the level of performance, you then take a profile of the running application. That stores all the information about what classes you've got loaded, what classes are initialized, the profiling data that was collected whilst the application was running, and even the compiled code for the methods that have been compiled. When you start the application again, rather than starting from scratch, you say, “Here's the profile from when you were running before.” That way, the JVM immediately knows, “Right. I'm going to compile all these methods. I've got all the profiling information already available. I've also got copies of the compiled code, so I can sometimes use that, rather than having to recompile stuff.”

That gives you a much faster warm up cycle than you would get by just letting it run normally. You get all the work happening right up front, and then you're at that high level of performance very, very quickly compared to where you would be if you were just running without the profiling.

[0:27:12] AD: Very cool. This is the Crack CRA?

[0:27:16] SR: No.

[0:27:16] AD: That isn’t the Crack? Okay.

[0:27:18] SR: No. This is called ReadyNow!. ReadyNow! is about keeping a copy of all the information about how the JIT compilation system worked, so that you can reuse it when you

start the application up again. Crack is something that we've been working on, and we started developing probably about 18 months ago now, which is another approach that goes even further.

It's not quite productized yet, but the idea is you run your application, you let it warm up, you load all of your information, you've got everything running, and then you can take effectively a snapshot of a running application, such that you can then say, “Right, take the snapshot, store it all in a set of files.” Then when you want to start the application up again, you're literally starting from the exact point where you took the snapshot. That includes all the heap, all of the registers, the program counter and everything. You're literally starting exactly as you were before, so that you could be at the beginning of your application, but you could be running for a week before you take the snapshot and then start again.

That's very, very powerful, because it allows you, especially in a microservice environment where you're going to be starting up multiple instances of the same service, then you can actually use this technology to get very, very fast startup. We've done some benchmarking of sample applications, and we did one with Spring Boot, very commonly used framework for enterprise applications. The time to first transaction was about, I think, it was about three seconds if we just started up from cold on Spring Boot.

Using a Crack snapshot, we really got to change the name of this. Crack. Java on crack. Using a Crack snapshot, we reduced that three seconds to first transaction to 30 milliseconds. It's like two orders of magnitude faster. It has some very powerful features that we really are working on. Interestingly enough, Amazon have taken this, because we made it an OpenJDK project. They've taken it, and they've made it into their, I think they call it Snap Start, which is what they use for their AWS lambdas, so that they can get their serverless computing. I know with serverless computing, they can start it up very, very quickly with that. The problem is that it's not cross-platform yet. Only runs on Linux, so we can't really make it part of the default OpenJDK yet.

[0:29:52] AD: Yeah, very cool. Okay, so when would I use ReadyNow!? When would I use Crack? How would you distinguish those?

[0:29:59] SR: ReadyNow! is much more suited to where you've got applications that you're interested in getting fast startup in terms of the overall performance, but you're not looking to store state, because you wouldn't have any of the preloaded data and things like that. Whereas, Crack is where you want a certain state to be available right from the very beginning. There's two distinct ideas behind that.

[0:30:26] AD: Got you. With Crack, do you see people doing that more take mess snapshot right after it spins up? It's mostly that early initialization work, but not a lot of actual data type things? Or do you see it all across the spectrum there?

[0:30:43] SR: I think it's going to be all across the spectrum. I know that there are quite a lot of people look at it as, oh, this would be a really good way to be able to be ready to go right from where your static void main and then take snapshot right there, so you're ready to go. Other people are looking at it as, oh, this would be really cool, because we can populate all of our data structures. We can load everything that we need.

The downside, of course, is that the more stuff that you store, the bigger your heap is. In order to take a snapshot, we have to take a snapshot of the whole heap. If you've got an 8-gigabyte heap that's 60% full, you're going to have four and a half gigabytes of stuff that you've got to store on disk, and then you've got to load it in when you want to read stuff from it. Again, there are trade-offs between some of the things here. If you're doing more complex stuff, it can be not quite as optimal as if you're doing smaller things.

[0:31:36] AD: Yeah. Got you. What about for ReadyNow! when I take a snapshot? It's not snapshotting the heap, just more of the compile code. How big are those snapshots are?

[0:31:43] SR: Those are very small. Because we can do it in two parts. The information about the methods and the profiling thing, that's literally kilobytes. It's very, very small. It depends, to some extent, on how much data you store from the code cache. Even then, you're really only talking about a few megabytes, so it's not huge amounts of data. We can load that very, very quickly.

[0:32:10] AD: Yeah. Got you. Going back to compilation a little bit, you mentioned there's interpretive, there's C1, there's C2. What's I guess, performance impact do you get as you move from interpretive to C1 and C1 to C2? How big of a difference is that?

[0:32:25] SR: It's huge. I mean, interpretive mode is very, very slow. If you look at the performance graph, anything that's running in interpretive is really slow, because you're just not getting any optimization there. C1 is better, because now you have a method and it's compiled and it's running native instructions for that method. The real benefit you get is from being able to use Falcon, or C2 to do more heavily optimized code.

One of the massive benefits that you get from JIT compilation and in the analysis that we've done, we find over 50% of the performance improvements of using the secondary compiler, the Falcon compiler, is through what are called speculative optimizations. That's where you're saying, okay, so we've observed that the code runs in this particular way up until this point. Let's assume that it's going to run for the rest of the time in the same way. You can do clever things in terms of eliminating code. If you've got an if statement and you've only ever gone through the true branch, then you assume, okay, we're only ever going to go through the true branch in future and we'll optimize based on that.

Now, sometimes you do get situations where the assumption is wrong and you have to throw away the code and that's called a de-optimization. One of the things we do with ReadyNow! is we record all the de-optimizations that happened, so we can avoid those in future. Again, it's a trade-off, but we try to get the best possible way of doing things with that.

[0:33:53] AD: Oh, man. That's so cool, to see those optimizations and see the hard work that you and your team are doing and then just how many Java developers get a benefit from that, basically drop in. I think it's so cool. Without having to know about SIMD, or optimizations, all kinds of stuff. Yeah.

[0:34:12] SR: Exactly. It's taking that level of experience and applying it so that everybody can benefit from it without having to learn about vector operations and so on.

[0:34:20] AD: Yeah. Cool.

[0:34:21] SR: There is one other thing that we've done, which is related to that in terms of the JIT compilation, which is where we've said, okay, this idea that the JVM doesn't have any memory of what happened before. We can solve that by using ReadyNow!, we can solve it by using Crack. If you're in a cloud environment, what we're seeing is that you have the idea of the microservice architecture where you've got lots of services working together. Now they're highly cohesive in terms of the way they work together, but the JVM's underneath don't know anything about each other.

What we've said there is like, okay, what can we do to improve that? The obvious thing is to say, let's take the JIT compilation and take that out of the JVM and turn it into a centralized service in the cloud. That way, if you've got multiple instances of the same service starting up, first one starts up, it needs to compile certain methods. It sends those to the centralized service. They get compiled. They get passed back, fine. The next time you start up the same service, when you request the method to be compiled from the centralized service, it doesn't have to compile it, because it's already got it.

We're providing a memory through a cache to all of these services, and we can even tailor that in terms of the way that you can have the same method with a different set of profiling information. That provides a fingerprint, so that the centralized service, what we call the file native compiler, will then be able to return the optimized code for that, based on the profiling information, rather than just the method.

We're going to take that even one step further, because we're looking at the idea of, right, so we're running this centralized service. You pass a method to us to be compiled with profiling, we compile it, we optimize it, pass it back. Then, since we've got nothing else to do at that time, we'll start looking at the same method and exploring some other optimizations that would take longer to do. Generates me even more heavily optimized code. Then the next time somebody asks for that, we give back even better code, or we can even preempt that and say, “Oh, I've got a better version for that for you. Why don't you use that one instead?” There's lots of interesting things we can do with that.

[0:36:34] AD: That is so cool. Yeah. I talked with your co-worker, John Ceccarelli about cloud native compiler. I think it was so cool. Yeah, people want to go look at that episode. It's a few months back, but I think it's so cool what's possible there in terms of optimization work. Very cool. I love that.

You mentioned GraalVM a little bit ago. I want to talk about that. What is GraalVM? How is that a different approach to some of these problems, I think, around slow start specifically? What's that doing?

[0:37:01] SR: Yeah. GraalVM actually started as a research project way back in the Sun Microsystem days, because I remember from when I was there, what they were doing there. The idea behind GraalVM, it's really around multiple different languages. You can run Java on GraalVM. You can just as easily run things like C, or C++, or Go, all sorts of different languages. From the Java perspective, the thing that most people are looking at GraalVM from in terms of improving performance is what are called native images. This is the idea that rather than having a JVM which runs your byte codes in interpretive mode, then does the JIT compilation, whilst the application is running, you say, let's avoid all of that problem and let's compile our code before we start running it into native instructions.

You move away from the idea of write once, run anywhere platform neutrality, and you say, "Okay, I know I'm going to be running on an Intel processor. I know I'm going to be running on Linux, let's just compile it for that environment and generate a native image." That way, you get a much smaller image, because you're only compiling in the code you need, and then you also get a very, very fast start up, because there is no warm up. It's instantly at full speed for the application.

This is very good if you're doing things like serverless computing. Very short-lived defemoral services work very well with that. The problem is if you run longer running services, the performance you'll get, because you're compiling the code ahead of time is not typically going to be as good as you get with JIT compilation, because of the fact that you're looking at the running system, you can use the speculative optimizations to improve the efficiency of the optimized code. You'll see like, Graal will give you a certain level of performance, but once

you've warmed up your application, you'll see a high level of performance typically every time, but typically with using JIT compilation.

Now, GraalVM will then say, “Oh. Well, we have profile guided optimization.” The idea being that you compile your application, you run it, and then you profile it while it's running, and then you take the profile and you feed it back into the compiler and recompile the code. But it still doesn't give you the same level of optimization, because you're still doing it statically. By having the totally dynamic nature that you get with JIT compilation, it is much more flexible.

The other drawbacks, if you like, for the GraalVM native image is that you're in a closed environment. There's dynamic class loading. You can't do it runtime. Bytecode generation, you can't do it run time. Reflection needs all really cleared ahead of time. It's possible, but it's a bit more complicated when you do it with GraalVM. There are some limitations in that respect.

[0:40:00] AD: For those things, dynamic class loading, reflection, things like that, is that – I mean, does that cut out 80% of – are most Java applications doing that to the point where it'd be really hard to run on Graal? Or how they can do that?

[0:40:12] SR: I wouldn't say all, but if you look at popular frameworks, if you look at things like Spring, if you look at Micronaut, Quarkus, things like that, they all use dynamic class loading. They all use reflection. They have adapted, so that you can use Spring, you can use Quarkus with GraalVM. There is no problem with that, but it does have extra work involved, and there's a bit more overhead in terms of that.

[0:40:39] AD: Got you. Let’s say, a day down the road, once I have my Graal version running, I have my normal JVM running, Azul running, what’s the performance – how big of a performance difference in that? I know it's hard to say, but yeah.

[0:40:54] SR: Yeah, yeah. It's difficult to say. If I was to just make a blanket statement, and I'm sure that there'll be a lot of people who would go, “Oh, no. That's wrong.” I'd say, somewhere between 20% and 30% is the difference that you'll see.

[0:41:09] AD: Yeah. Okay. Okay. Sounds good. If you really need that quick startup time and you don't have some of those features, reflection, things like that, Graal can work, but you're going to be trading off that top-end performance and some of the flexibility of the Java language.

[0:41:21] SR: Yes. Yeah.

[0:41:22] AD: Okay. Cool. I like all that stuff. Azul has been on the cutting-edge with right now with Crack, all that stuff. Then you've been on the cutting-edge, I think, of Java's whole lifetime here. What do you see and that's exciting to you and new to you coming down the pipe?

[0:41:38] SR: Yeah. I mean, it's interesting to see the way that Java has evolved, especially over the last, probably about five years, I would say, because we've switched from having two, three, even four years between major releases of Java. Now, we have this time-based release cadence, and we have two releases every year. We've got March and we've got September, our new versions of Java. That has led to a much faster pace of development in terms of the platform. We've seen many more features coming through more quickly.

The other thing that's been really good from having that faster release cadence is the whole idea of being able to introduce preview features and incubator modules, so that features can be added without being set in stone. That way, people have a chance to have a look at them, provide some feedback, say of things need to be changed. Then once everybody's happy, then they actually set it in stone and make it part of the standard. That's, I think, a really good thing, because obviously, the developers of the OpenJDK are very smart people. You talked to Brian Goetz and Mark Reinhold, they're very smart people. But there's always going to be situations where people go, “Oh, yeah. But if you did it this way, or did it that way, it will be a little bit better.”

One great example of that was the introduction of switch expressions. In switch expressions, you could use a new style of syntax for them, or you could use a variation on the old style of syntax. To do the old style of syntax, you would use the break keyword with a value. People said, “Well, yeah. You can do that, and the compiler can resolve it and it all works very happily. But it's a bit more confusing for developers. So why not use a different word?” They said, “Okay. We'll take that onboard.” They switched from using break to using yield. It's a small change, but

it was possible, because it wasn't part of the standard. That, like I said, is a really good thing, being able to address minor irritations before you make it part of the standard.

[0:43:37] AD: Yeah. What does governance of Java look like? Is that a foundation? What's running the release cycle, the features, all that stuff?

[0:43:49] SR: All of that is really primarily done through the OpenJDK. This open source-project. Now clearly, in terms of the governance, Oracle are the people, because they acquired Sun, they acquired the governance of OpenJDK. They're people who make the ultimate decisions about certain aspects of that. It is an open-source project, so people are free to provide ideas. As I said, we contributed Crack as an idea. It was made a sub project within OpenJDK. There are other things like Red Hat's Shenandoah garbage collector. That was another project that was contributed there.

There's lots of ideas coming from outside of just Oracle. Ultimately, they have the final say in terms of governance. It's an open-source project and everything works nicely through that. We've seen a lot more shift away from the standardization work happening within the JCP to happening within OpenJDK, through these things called JDK Enhancement Proposals, or JEPs. A lot of the changes now that happen in Java are done through JEPs.

We still have the JCP, we still have the JSRs, because we need that standard with the testing, so that we can verify that everybody's got the same implement, or the same functional way of doing things. The governance in terms of ideas and so on, much more comes through OpenJDK and the JEPs.

[0:45:13] AD: Very cool. Very cool. Well, Simon, I appreciate you coming on. I always love talking to folks from Azul, because you're on the cutting-edge of Java and you know this stuff and I learn so much every time, even though I don't have a ton of experience with Java. Thanks for coming on and walking through all this stuff with me. If people want to find out more about you or Azul, where should they head?

[0:45:31] SR: I think, the easiest thing is our website. Azul.com is a nice, easy one to remember. I've got to say, it's always great when I give my email address, because it's

@azul.com. It's not a very long email address. The other thing you can do is on Twitter, for people who are still following Twitter. I'm @speakjava on Twitter, so you can follow me there and I post things about what's going on in the Java world, as well as a few other things.

[0:45:58] AD: Perfect. Simon Ritter, Deputy CTO at Azul Systems, thank you for coming on.

[0:46:02] SR: Thank you very much.

[END]