EPISODE 1534

[INTRODUCTION]

[0:00:00] JC: Ahead of time compilation does just what it sounds like, right? It says, rather than try and compile this stuff while you're running the code, I'm just going to scan all your source code and I'm just going to compile everything before you run it.

[INTERVIEW]

[0:00:15] A: John Ceccarelli, welcome to Software Engineering Daily.

[0:00:19] JC: All right. Thanks for having me.

[0:00:19] A: Awesome. Yeah. It's good to have you back, because you were here just a few weeks ago, or a month or two ago, talking about Azul and Platform Prime, this new JVM that you all have that's really efficient on things. Today, you're going to be talking about cloud native compilers. Maybe just give us a little introduction about you and what you do and what cloud native compiler is.

[0:00:37] JC: Sure. Yeah. I run product management at Azul Systems. We make high-performance runtimes. We make the best the best JVMs in the planet. I am Head of PM for Azul Platform Prime, which is our high-performance, optimized build of OpenJDK. Along with that, I've been able to – I've had the pleasure of bringing to market Cloud Native Compiler, which is an exciting new add on tool to Prime, that allows you to get to full power on any size machine. Yeah.

[0:01:13] A: Awesome. Awesome. For those that maybe don't know as much, let's go a little bit into the background on how compilation works in Java. Because I think of that initial compilation step, using Java C, getting from my actual code to bytecode, but there's more than that. Can you walk people that maybe aren't as familiar, like what's going on when we talk about compilation in Java?

[0:01:33] JC: Sure, yeah. When you talk about compilation, most people think Java C, they think Jenkins and so forth. Java is based on write once, run anywhere. The way that you do that is that you compile through Java C, you compile down to your JAR files, and that's got the compiled classes in it. Those classes are able to run on any system in an interpreted mode. You can run those as is anywhere.

That interpreted mode, if you just run those class files as they are, that's not machine code specific. There's still an interpreter in the middle that is taking that as generalized, able to be run anywhere binary code, and then translating that down to machine code. To get really fast, what you want to do is actually turn that code into optimized machine code for this specific machine that you're running on. That's what the JIT compiler does. JIT stands for Just In Time. That means that it doesn't try to pre-compile these things ahead of time and it doesn't try to compile everything in your application. It just goes through and it runs things in the interpreter for a while, and then it has little counters, and it just counts how many times you call each method. We call that hotness, right? How many times something's been recalled, we call that hotness.

Based on hotness, when a method gets hot enough, it's been called enough times, it will say, “Okay, let's kick that over to JIT compiler. Let's make a highly optimized version of that.” Even that is interesting, because it doesn't just say, “Okay, just compile this as it is.” It looks at the usage patterns of how your code is actually being used to make super, super optimized fast versions of that method.

There's lots of things that the JIT compiler does. One of the things that it does is if you've got like, let's say, you've got an if-else loop. You're to optimize that into machine code and include in the instructions the function for checking a value and then checking which side of that branch you want to go down, makes the code slower. If the profile shows that in the entire life of your JVM, you've only gone down one path, then the JIT compiler will take that whole structure out and put it in as if only you can only go down that one path. This is called speculative optimization. It means based on your usage patterns, we are speculating that this is how your code is going to behave in the future, and so we're going to optimize it specifically for that.

The OpenJDK includes its JIT compiler, which is called hotspot, because it's looking for the hotspots and just compiling those. Yeah, that is what JIT compilation means and how it contributes to the eventual speed of your application.

[0:04:36] A: Awesome. It's actually doing this as your program is running, it's optimizing it. Is this continually tweaking it over and over over time, even a particular method? Or is it more like, once a method's optimized and compiled, generally that stays, I guess, especially that speculation, speculative stuff, if it goes down the wrong branch, does it have to recompile that, or what does that look like?

[0:04:57] JC: Yeah. The profiling part happens, there's three parts. There's the interpreter stage. That's just counting how many times you're actually doing the method. Then once it says, "Hey, we should optimize this thing," then it goes into the profile, the tier one profiling. It makes a very quick compilation of it. It's faster than interpreter, but still not very fast. Nowhere near the speed that you're fully optimized the method will be at. In that period, it counts the usage and so forth. It makes it speculation. It puts its optimized method into a call stack. From that point forward, it's no longer profiling.

It's not looking at, "Hey, 10 minutes ago, I thought this was the best way to do this method. Now I see that actually, there's a better way to do it, so I'm going to recompile it and put it back in." None of the JIT compilers, do they do that. You can get into – get into a situation where it turns out, one of your speculations was wrong. In our if-else example, up until this time, we've always gone down the left path, and then all of a sudden, people start doing the right path. When that happens, we get what's called a de-optimization. That method gets kicked out, gets sent all the way back to the interpreter and we do the whole thing again.

This can be an individual method. But oftentimes, a major change in traffic patterns. Usually, that's what causes these de-optimizations, can cause what we call a de-optimization storm. That sounds scary and it can be scary, because if you get a large number of methods, very important methods that are getting called very actively getting kicked back to the interpreter, now all of a sudden, that code is running at 1/10th the speed, all kinds of implications for CPU use, your JVM is going to now struggle to keep up with CPU with a load, which would cause load balancers to

start spinning up new JVMs. Could cause you to start missing SLAs, as far as response time, so all kinds of things can happen when you get a de-op store. Yup.

[0:07:00] A: Awesome. You mentioned 1/10th the speed. Is that about the speed up you see, as it goes from interpreted down into the more optimization code?

[0:07:09] JC: Oh, yeah. The interpreted is orders of magnitude slower than fully optimized code, or the magnitude. 1/10th is probably a very conservative.

[0:07:17] A: Really. Okay. Wow, that's pretty wild. I know when I start up a JVM, it takes a little bit of time to do stuff. Is it doing any compilation at that point? Or is it purely waiting until runtime to actually get into more of that compilation stuff?

[0:07:30] JC: No. It's waiting until after main. It has to actually start running your code after main start executing those methods and so forth. Well, most things do and we'll talk a little bit later about some of the cool things we can do to actually front load some of those optimizations. Yeah, once you get into main, once you're actually exercising the code, that's when you see that happening. Yeah, you'll often see what we call the warmup curve, which is when you start a JVM, it takes a certain time to get from when you first start it to when you're running at full strength, at full maximum throughput.

Those things are due to sometimes, it's stuff that code is doing. A lot of frameworks do a lot of resource loading and so forth. They got to get all that stuff set up before they can actually start executing all their code. Most of the time, that's just because you got your application threads and your JIT threads battling for the same resources.

[0:08:26] A: Awesome. Awesome.

[0:08:26] JC: Big deal in the warm up.

[0:08:28] A: Yeah. Awesome. I want to talk about cloud native. One last question. I've also heard of a AOT. Can you describe the difference between the JIT, just-in-time and the AOT, ahead-of-time compilation?

[0:08:40] JC: Right, right. Ahead of time compilation does just what it sounds like. It says, rather than try and compile this stuff, while you're running the code, I'm just going to scan all your source code and I'm just going to compile everything before you run. A couple of things with AOT. First is that it's not profile-guided optimization. That means, it's giving you the lowest common denominator. It's not able to take any of that knowledge of what you did in the past, what kind of optimization, what kind of usage patterns you were seeing to provide a really optimized code. It's just given you the lowest level of optimizations, compiling everything, so that any path and your code will work. That's one of the things. It will always be lower eventual code than a profile guided JIT compilation.

The other thing about ahead-of-time compilation though is that you're limited in which programs you can actually use it for. Because you have to be able to compile everything from the source code. Many, many programs use dynamic class loading. That's when it's actually, those classes aren't defined in your source code, but are actually being defined by the program as you're running and going forward. Of course, there's no way in ahead of time compiler would be able to do that. The main benefits of ahead-of-time compilation is that you can get a lower memory footprint, because when you do ahead-of-time compilation, you know that these are the only things you're ever going to run, because by definition, you can't run anything else. You can actually turn off everything else. That is one reason why a lot of people do that.

They will pay the price for in eventual code speed, just to be able to run these things on smaller images. That is one very valid advantage of AOT. Yeah, there's a lot of projects out there right now, who are trying to play in this space and solve this particular problem. I mean, another one that's very hot right now is CRaC, coordinated restore at checkpoint, which is a project that Azul has been leading in OpenJDK. It's an experimental project in OpenJDK.

What CRaC does is it does something different. It goes ahead and runs your Java, Java environment, your Java program with the entire, all the bells and whistles, dynamic class loading, everything else. Then you get it to a point where you like where you're at. All of your initialization is over, all your classes are warmed up, you're ready to receive traffic at 100%. Then what you do is you take a snapshot of the program at that point, and you can save that snapshot. Then you can launch as many JVMs as you want from that snapshot.

CRaC is great. We've got a lot of enthusiasm about it here internally. Very exciting to see the community really picking it up. Micronaut, **[inaudible 0:11:45]** and Spring Boot, all three of the main microservices architectures all have support for CRaC right now. Even more exciting, AWS lambda has recently introduced SnapStart, which is a feature where they're using the CRaC API to snapshot lambda services. Then they use their internal snapshotting mechanism that runs on AWS, which I believe is called Firecracker. They're using the CRaC APIs to do the checkpointing. Then to restore from those checkpointing. That's available in production for lambda users today. A lot of excitement around CRaC.

Thing about CRaC is CRaC doesn't work for all programs out of the box, because obviously, you want to restore things, but there's certain things that you can't just bake into that, into that snapshot and restore it. If you've got things like resources, database connections and file handles and so forth, right? Those things need to be reconstituted. It's really good for stateless apps. A stateful app would be very difficult to run on CRaC. It's very good for stateless apps. Even with stateless apps, any resources that can't be saved in the snapshot, you have to program your way around.

We have APIs that you can call to restore those things, post the snapshot restore. It doesn't mean that your programs won't. You can't just take any program. Some programs will run out of the box, especially small, micro-stateless, microservices don't have a lot of resources that need to be dynamically loaded. May run out of the box on this. If you just take some massive monolith program that you had running somewhere, high chances it won't run out of the box on there.

What we're looking for is a solution, and different solutions will fit different people and give different advantages, right? The CRaC solution is great, because you get the first warmed up transaction in milliseconds. It's very, very quick to start. We're really looking for something that's going to really work on everything. It's going to work on stateful apps, stateless apps, huge machines, resource constrained microservices, things that need to start up all the time, things that a startup once and run for eight months before restarting. We want something that works for all of those things.

Especially, we want something that is going to give you more power, more throughput, lower infrastructure costs. When you're doing Java deployments, it's always the game of tradeoffs. You're always doing tradeoffs. Like I said, when you're doing – you can tradeoff, do I want smaller machines? Then I have to do less on those machines, or do I want big monolith machines? Maybe I'll have some unused capacity there. You're doing tradeoffs around, do I want horizontal scaling, horizontal elastic scaling? Or do I want aren't one big machine that's going to always be ready to handle all the traffic and so forth?

Deploying java is a game of tradeoffs. The tradeoff that affects your code the most is the one that you don't even know you're doing it, because the JDK is doing it for you. That is just how aggressively are we going to JIT this code? Like I said before, when you're starting up, you have resource contention. You're trying to run JIT and your application at the same time. You're splitting your threads between the JIT compiler and the application code. If you're running on some big 32 V-core bare metal machine, maybe you don't care. You got plenty of cores. Let's do it. If you're running on 4 V-core, 2 V-core machine, you are really tight on what you've got CPU for.

Compilations take time. They take CPU time. There's no such thing as a free lunch. There's no such thing as an optimized method. What OpenJDK and hotspot has done is they found this one level that they find is acceptable, and it's acceptable over the entire wide range of systems that people run on, right? They say, “Okay, this is how much JIT we're going to do and it's going to get you this faster code.” It represents a good tradeoff between how fast I want my code to be and how much CPU I'm willing to devote to JIT, in the meantime.

Most people think that that's just as fast as Java runs, because they don't have a lot of experience with other JIT compilers. Now, there's more and more JIT compilers. Growl has its own JIT compiler that produces JIT code as well. Prime has its JIT compiler, which is called the Falcon JIT Compiler. Falcon JIT Compiler is just a lot more intensive. It can give you up to 45%, 50% faster code. What we're looking for is a solution, where I can get that goodness, get that full speed, that highest possible speed of eventual code, without sacrificing the performance because we're spending so much percentage of our available CPU power on JIT compilation.

[0:17:22] A: Awesome. I assume that's where cloud native compilation comes in. Can you tell us, what the cloud native compiler's doing? How that interacts with my running code?

[0:17:33] JC: Yeah, you betcha. Yeah, cloud native compiler, what we said is, let's use the cloud. This is what we're doing. We're splitting our big apps that we thought were one big thing into multiple services, where we're using SaaS services to perform various functions that we used to have to code ourselves and run ourselves on that JVM. We thought, why don't we do this towards the actual internals of the JVM?

We decided to take the JIT function and offload it to a dedicated, scalable server. What we have is this Kubernetes server that you install on your system, and that can be on-prem, in the cloud, or whatever. It's not a hosted SaaS right now. It's not like you just – you send your code to our servers and we return it. Right now, it's something that you maintain and you put into your data centers.

It's elastic, which means you can bring – and it it's best when run in the cloud. It's best when you run it in sort of some elastic capacity cloud, where you're not paying for the resources that you're using all the time. You only pay for them when you're actually using them. This means that practically, you can get practically unlimited compilation capacity for your apps at practically zero client side CPU spend.

Remember, JIT compilation, we're not talking about three hours, four hours. JIT compilation lasts minutes. For really heavy programs, we've seen it last one hour, an hour and a half, or something. But if you look at the lifetime of your entire run, it's a very small fraction. As long as you're using elastic resources, you can really afford to bring a lot of computing power to doing your JIT compilation when you need it. Then you just put it to sleep when you don't and you don't pay for it.

[0:19:32] A: Yup. How does that work? I guess, is my application code then sending the program all the way over to CNC, the cloud native compiler with some profiling data? It's got it runtime? Does CNC already have my application code, and it's just having the profiler? What does that interaction look like?

[0:19:48] JC: Right. The way JIT compilers work is the application says, talks to the JIT compiler and the JIT compiler sends stuff back to the application. The application says, “Here's a method. Please compile it.” Then the JIT compiler says, “Okay. Well, give me your state info. I need your VM state info. What is the state of your JVM? What are the states of the various variables and so forth?” Then the JIT compiler says, okay, to optimize this thing, I need to get a bunch of information. It sends a bunch of questions back to the JVM, the JVM provides all that information, then the JIT compiler compiles the method, and then it returns the method.

It is sending code, but it's not sending your code over. It's sending each individual method, plus the information about that method. That happens on your JVM when you're doing local JIT. All we did is we took that JIT and we put it on a server, and then that communication is happening over the network. Yes.

[0:20:42] A: That's super cool.

[0:20:43] JC: For those people who are like, “Oh, my God. Single point of failure. What happens if the network drops?” If the network drops, you just switch back to local compilation. Nothing bad is going to happen to your JVM. It just means that your JVM is not going to be spending local resources to compile that, until the connection is reestablished.

[0:21:00] A: Yeah, very cool. If I have, let's say, a 100 instances of my application running that need this JIT, can that compiled stuff on my CNC, can that be shared across those application being like, “Hey, we've already compiled that”? Or is each one doing it individually? What does that look like?

[0:21:15] JC: Yeah, exactly. I mean, some people when you tell them this, they go, “Well, you're just moving jobs from one place to the other and you're adding complexity just so that you can move things from one place to the other. What's the point?” The first one is the one that I've already talked about. Sometimes you need to move it somewhere else, just because you have no capacity to do it locally. Once you start moving things to a centralized service, things get very interesting. The Java JIT compilation model was made, was built in the 90s. In the 90s, we had bare metal and that's about it.

We didn't have anything called elastic capacity. It made sense that each JVM would have to be completely self-contained and not know anything about not even what other JVMs in the cloud were doing both. What it did the last one, it's like that guy from the mental rehab and no short-term memory. You wake up and you're like, “Oh, what am I doing? I guess, I'm running this thing. Okay, let's run it.” When you say, there is a cloud, when you say, we are using virtualization, so we are creating hundreds and hundreds of copies of this one application and running it exactly the same way, but in a 100 replicas, then really interesting things happen.

The first and the most obvious one is, can the compiler have a cache? If I'm compiling the same – if I'm providing the same compilation a 100 times, do I really need to do that compilation a 100 times? The answer is obviously no. Yeah, we built a cache into cloud native compiler. When it gets compilation requests, first thing it does is go and be, “I already have this up. Yup, here I have it. There you go.” It serves it up. If it doesn't, then it will compile it and put it in the cache for next time.

[0:23:11] A: Wow. Going back to even the JIT versus ahead-of-time stuff, for applications that our, I guess, changing frequently, I understand why that JIT matters, but for, let's say, Kafka or something like that, where if they cut a version and it's running across hundreds of thousands of machines, probably, and probably running fairly similarly, could CNC make a JIT version of this, that could then be shared across all these different Kafka instances? Or is there for things, for applications like that, is there some different compilation method? Or what's anything going on there?

[0:23:47] JC: Right, right. It's important to note that we're not creating images, right? It's not like we're creating an image of a warmed-up application that then you can reboot. That's more like CRaC and AOT does that stuff. What you're doing, if you look at the whole process that I described earlier, where like, okay, I start up, I get past main, I run everything in the interpreter until I can learn what the heck is this thing that I'm running and which methods are hot? How are they being called? I go through that whole profiling thing before I can get enough information. That lasts a long time.

Often, it changes over time. Which is very important, because often, what happens is in the initialization phase, certain things will be hot during the initialization phase, because you're

warming up all this stuff and getting it ready to go. Then when you start running it normally, Different things. You got different usage patterns, so different things. You'll get a lot of de-ops. Then later, there could be more load, or whatever and you'll have even more de-ops.

What you're really looking for is over the life of the entire program, what were the optimizations that I needed exactly? What is exactly was the information that they needed? Takes a lot of time to do that and it takes a lot of CPU power. It takes a lot of time when you're running the code in interpreted mode and it's running really slowly. You ask the question, if I've got a 100 Kafka instances that are running exactly the same code and handling the exact same load, basically behaving exactly the same, I already know the answers to all these questions. Why do I got to go through the profiling phase at all?

That's where another piece of technology we have, which is called ReadyNow! comes in. ReadyNow! just that, it just records all of that, makes a list of basically, now that I've watched your app running for an hour, I know that these are the things that you need to have compiled. Then when you start again and you feed that profile in, it will try to front load as many of those things before main. Even before you reach your main method, we're going to try and clock, compile as much as we can.

Now, again, there are certain things, like dynamic class loading and so forth that have to happen after main. But yeah, and with that, you can jump into, if not fully warmed up code, in many cases, fully warmed up code, especially in the financial industry, for people who really care about every single trade being exactly as fast as it can be, they code their apps in certain ways, such that they can front load everything to pre-made, and they jump right into a fully warmed up machine. Yeah, but even in other things that do post-main optimizations, the fact that we've done so much before main means that you're getting to full speed a lot, lot quicker.

[0:26:42] A: Got you. Just so I understand the difference between CRaC and ReadyNow!, CRaC is more about just that initial JVM startup making it a lot quicker and loading that stuff. Whereas, ReadyNow! is, hey, once you've run for a while, we've done some optimizations, we'll create that ReadyNow! profile and that can be used on startup next time with some of these compilation optimizations?

[0:26:59] JC: Yeah. CRaC is just about getting you to a certain point, usually at the beginning of your – at the beginning of your run. Whereas, ReadyNow! is saying, what are the things I know I'm going to see 20, 30 minutes from now, an hour from now, a day from now? Because I recorded that long of a stretch. I know what's going to happen after I get through this in that phase, and so forth. One of the things we're really interested in is combining those two. What if you use CRaC to front load all of that, right? What if you use ReadyNow! to front load all of that? Then you got it all front loaded, warmed up, now you take the snapshot and now you launch from that snapshot, we think that's going to prefer, provide the best experience, the best overall experience. Yeah.

[0:27:54] A: Yup. Wow, that's super cool. I've been talking to some my Java friends. I know they're excited about some of these optimizations that are coming down. What are some of the best programs for cloud native compiling? Where are you seeing adoption?

[0:28:05] JC: Right, right, right. First, to talk about Prime. The best programs for Prime are any program you want to run really fast. Anything that you care about, it's going to run really fast, it's going to have really low latencies. It's not going to have lots of jitters and pauses, and so forth. It's going to give you the lowest cost of carrying that load, you want to run in Prime.

Then you should always run first just regular Prime without anything and just see how it behaves. Then you look at it and you say, “Okay, what's my warmup behavior, right? Am I pegging the CPU too high, because the JIT is running too much, and so forth?” Then when you find that yeah, there is contention here, that this warmup process is either taking too long, or because it does take long – We do a lot more optimization. Our optimization phase, our warmup phase is longer than hotspots, right? When you find that, hey, this doesn't fit my deployment model, or it doesn't fit on this machine, or whatever, that's when you want to take a look at – take a look at CNC and offloading those things.

[0:29:12] A: Awesome. Very cool.

[0:29:14] JC: Like I said before, on a 32 V-core machine, you're probably not going to need CNC.

[0:29:24] A: Okay. Tell me a little bit more about just the deployment model of CNC. You mentioned a little bit around Kubernetes. How does that look like if I want to get started with it?

[0:29:33] JC: Yeah. It is a Kubernetes cluster that you run. We have helm charts for installing it. You run it on your infrastructure, and we recommend that being cloud elastic, cloud infrastructure. If you're on AWS, you can use their EKS managed Kubernetes, or you can just set up some JVMs and set up how to configure a Kubernetes cluster on them. You set up this system. It's very easy. We've got a built-in metrics that can show you whether the system had enough resources or not, for any given time to handle the compilation request that they were running. That's how you would run in production, right?

Now, if you just want to start small, you want to just check it out, you want to kick the tires or whatever, then you can also run it just as mini-cube instance. Let's say, you're working in a company, you want to take a look at it, but your company has a centralized Kubernetes team that you have to go and get their permission to even start up a cluster, and they got to set up a bunch of networking rules and so forth and you're like, “Oh, how am I going to try this out?” Don't worry about it. Just spin up one beefy JVM, one beefy instance, then you just run it as a single node mini-cube instance on there, and then just try running one or two JVMs against that. Warming up one or two JVMs against that. You obviously can't warm up a whole fleet on that. If you just want to see the power and see what a difference it makes, you can easily evaluate it that way.

[0:31:04] A: Yeah. One of the one of the cool things about Kubernetes that I've seen is just more projects, where it's just like, “Hey, go run it yourself. Here's a Helm chart.” You can just throw it into your existing cluster, assuming you can get through the Kubernetes team and all that stuff. Allowing you to run more sophisticated services, which is pretty cool. This is called cloud native compiler. Is this mostly cloud native people that are running this? Are people running it on premises as well? Or what do you see in there?

[0:31:31] JC: People who have highly advanced IT departments that have built some hybrid, or private cloud, where all of the servers that they're running on are theirs, but they can split that capacity between various places based on cloud-like load balancers and hypervisors are

running an on-prem. Mostly, we found people running this on public clouds, as your Google Cloud platform, AWS.

[0:32:03] A: Do you see a big difference in, I guess, how your on-premises customers and more cloud native customers run not just in this CNC instance, but just in terms of the either types of apps are running, or how they're configuring and running things like that? Are they pretty similar across those two?

[0:32:19] JC: It's all very similar. It's all very similar. There's not a lot of differences to it. I think, some of the things are around what your strategies are for bringing up the – what your strategies are for bringing up the resources that are needed on the cloud native compiler. You can let the auto-scaling work for you. Some people prefer to just pre-warm things by manually going in and spinning up the resources before their big fleet redeploys go. There's some different differences there.

[0:32:55] A: It's pretty predictable in that sense, like when you're doing a new deploy, you know you're going to need more CNC resources and you can speed that up.

[0:33:01] JC: Yeah. Then again, you've got your plan compilation stuff, and you've got your unplanned compilation stuff. Especially with Kubernetes, Kubernetes load balancing, Kubernetes rebalancing of pods, if it spins up a bunch of nodes, and then there's a bunch of pods on those nodes, and then traffic goes down and it spins down a bunch of pods, it winds up with 40 nodes that are only at half capacity. What does it do? It starts, okay, I'm going to take these nodes down here over here. I'm going to shut them down over here, and I'm going to start them up over there.

You can have lots of restarting of your apps that was not triggered at all by your build script, by your intentional action. That can happen a lot. Then, of course, you've got elastic scaling, where people are scaling out to meet new demand. Nodes go down all the time in the cloud, so you've always got things rebooting there. It's really a difference about dealing with planned restarts, versus unplanned restarts.

[0:34:03] A: Yup. Very cool. What about, will we see a hosted version of cloud native compiler, where maybe I don't even have to run my own Kubernetes cluster, I can hook into something from this old directory?

[0:34:14] JC: Well, I mean, technically, it's very doable, right? We're starting with the self-hosted approach. I feel like, latency, security, all these things are real. Also, another thing that's real is network traffic costs. If it was in our virtual product in our VPC, then then somebody, either you or us, or probably both of us are going to have to pay for sending all that data back and forth. If you set it up within your own cloud network internetwork, internetwork network traffic is free, so you don't pay for that. Right now, we're sticking with this, but our ears are open and we may come out sometime in the future with that hosted service and for some use cases.

[0:35:04] A: Yeah. Yeah. That network traffic cost, it just blows me out on how much innovation it limits, just because like, now we can't do this as a service, because it's just going to cost so much money.

[0:35:14] JC: Oh, man. It adds up. It adds up. You look at it and you're like, getting all your commits down for what your CPU cost is going to be and you think, that's going to be the main thing. Depending on what you're doing and how much you're pushing network in and out of the VPN, boy that network cost can be just as painful.

[0:35:35] A: Right, right. Yeah, it can be killer, especially, yeah, if you have your own multi-region data, anything like that. Yeah, it can really sneak up on you. Cool. This is really cool. I mean, this is exciting. What are you excited about for the future? I know you mentioned CRaC and ReadyNow!. What kind of things y'all working on at Azul?

[0:35:51] JC: Right, right. A couple of things we're working on. One of the things that we're working on, like we said, ReadyNow!, we want CNC to just take care of all of ReadyNow!. Today, when you're running ReadyNow! on prem, the JVM itself puts out a profile and then you have to figure out how to get that profile and put it somewhere that will survive in case that JVM, if it's in a container, goes away. Then how to feed it back into new JVMs that are running. Also, making sure that you've got the best possible profile. Usually, that's a manual action. You'll manually run some code through something and then you'll look at it and you'll go, okay, you

know, what we'd like to do is I've got a 100 JVMs running the same code. Let's pull information from some subset of them, all 100 would be silly. But some subset of them. Then from those, be able to have some smarts about like, yeah, this is the one that you really want.

Having ReadyNow! be a part of CNC is something that we're very excited about and moving to the market very soon. We're also looking at CRaC. We're going to be doing a lot of work in CRaC and getting that into our products right now. It's just an experimental thing. We're going to be getting that into our products going forward. The other thing are what I call super-duper optimizations. Because really, Falcon right now, they're already super optimizations when you compare them to what you get out of hotspot. Are there optimizations that we just never even considered? Because geez, that thing would take 12 minutes to optimize one method.

What if it's a really important method, all right? What if you only had to spend that 12 minutes once? Then you cache it, and then everybody else would get it for free. All of a sudden, like I said, once you move this out to a system that is shared, that has a memory, has a long-term memory, that isn't bound by the compute power that's available on your JVM, lots of really interesting things open up. We're going to do a lot of tinkering over the next couple of years. There's going to be some exciting stuff coming out.

[0:38:10] A: Oh, man. It's so cool just seeing all the new stuff that's opening up, because of the elasticity of cloud, or just the different – the way things change that way. Man, I love it. It's so fun. Yeah, I'll be –

[0:38:21] JC: It's exciting time. It's an exciting time. A lot of people doing things with the JVM that we never even thought was possible before. Yeah, here it is. That's our bread and butter. That's what we do. Nobody does Java like we do. We're the largest company that just does Java and that's all we do is Java.

[0:38:44] A: I love it. I love the work that y'all are doing in the space. If people want to find out more about cloud native compiler and Prime as well, where should they find that?

[0:38:51] JC: Just go to azul.com. Both cloud native compiler and Prime are free for development and for evaluation. You can run some tests and so forth. Performance test,

performance optimization in Java is a bit of a dark art. We have a lot of packaged benchmarks that you can run for Solar, Cassandra, and so forth, to see the power of it. We also have a team of some of the world's best Java experts standing by in case, if you're looking at it and want help getting even better results. Just reach out to us. Hit the Contact Us button anywhere on there and reach out to us and then we'll get in there and give you some free Java performance consulting.

[0:39:33] A: Awesome. Yeah, be sure to check that out, everyone. John Ceccarelli, Senior Director for Product Management at Azul. Thanks for coming on Software Engineering Daily.

[0:39:41] JC: Alex, always a pleasure. Thanks for having me.

[0:39:43] A: Great.

[END]