EPISODE 1538

[EPISODE]

[00:00:00] AD: Gerrit Grunwald, welcome to Software Engineering Daily.

[00:00:03] GG: Thanks for having me.

[00:00:04] AD: Absolutely. So, you're a principal engineer at Azul Systems, Senior Developer Advocate. Can you tell me a little bit more about what you do, what your path is in the Java world, and in just programming generally?

[00:00:15] GG: Yes, in programming, I mean, that's interesting, because I never really studied IT or something like that. I did physics, and then found my way into software development, because I did that already. I do it already since 1982. Then, at some point, I stumbled upon Java, because I was looking for something that I can do at home on my Mac, and in the company on Windows, and so I ended up with Java and I love it still. Over the years, I fought my way into Azul, and it's a great company with great colleagues that I know from other companies. Yes, that's mainly it, in a very short form. I also run a Java user group in Windsor, where I live in Germany. The reason for that was because I was looking for people that also code Java. So, I founded the Java user group and that was part of my career. A big part.

[00:01:15] AD: Yes, awesome. Well, I agree with you on the great people at Azul. I've talked to John Ceccarelli and Simon Ritter about Java. It's been good for me, because I don't know a ton about Java. But I think it's such a fascinating ecosystem, and just like all the optimization work that goes into it, and obviously runs so much of what's happening out there.

I want to talk about performance and cold start in CRaC, especially, it is something interesting to talk about. But just to set it up, like when you talk to Java developers, the typical Java developer, how well do they sort of understand the performance knobs and things like that? Or is it mostly maybe one optimization specialist, or ops person handles that, and I mostly just focus on my code?

[00:01:54] GG: That's interesting, because when I did the session about CRaC last year, a lot of times, I figured out that lots of the developers have even no idea how the JVM itself works. That's not in general, a bad thing. Because usually, you should just use it right? And you trust the system, it will do the work for you. It's all good. But it helps if you understand how it works. To answer your question, usually, in companies, people don't really spend a lot of time on performance tuning. It depends on the on the use cases on the application that they have write. Let's say, you have some online store or something like that, then probably you run into performance problems. Sometimes let's say we have Black Friday, and then suddenly everybody's hitting your page, you need to do something because everything goes down the drain, performance wise. Then, people start thinking about, in big companies, you might have performance teams, but in small companies, it's up to the engineer. You will read a book, you try to apply some of these hints and tips from the experts. But there's no general rule to make Java just faster. This is not how it works. It's quite complex.

[00:03:08] AD: Yes, absolutely. It's fun to see all that, the cool work that's going on there, and around that. Maybe just to start, give people a baseline. I would say, you mentioned your talk, like go check out the Voxxed Days Athens talk. We'll probably link to that in the show notes, if you want to know more about that. But maybe just walk me through a job application. If I have source code on my machine, and then we talk about maybe it starts running, and at some point, it's fully optimized. What are the steps of going from source code to a fully optimized running application?

[00:03:37] GG: That's interesting, because that's exactly the stuff that most people don't know. It's fascinating once you understand how it works, so you write your code in your editor or in your IDE, and then there's just your Java source code file. So, it has a dot java in the end, and you can open it in a text editor and read it. Then you compile it using the so-called Java C compiler, and this compiler creates so called bytecode. The bytecode is like, it's like a cross platform code that can be translated into machine code from a JVM, but you need a JVM on your platform. That means if you would like to run the bytecode on your Linux machine, you need a JVM on Linux. This is how it works.

[00:04:20] AD: Only the JVM is sort of platform specific, but the complied bytecode is independent?

[00:04:26] GG: Yes. Exactly. The bytecode, you can exchange between all the platforms, as long as you can read the files, you can read them in. But the compilation down to machine code, this is done by the JVM and this is platform specific, the implementation. That also depends – so it's not only the

platform, like let's say Mac because a Mac, we have Intel CPUs, and we have ARM CPUs. So, that means there are different implementations for these different CPU architectures, right? Then, there are 64 and 32 bit, but that's way too far.

[00:04:54] AD: That's completely like abstract, I really need to think about that as a developer as long as I'm picking the right JVM at random, I think, okay.

[00:05:00] GG: No, you don't have to. You just download the JVM from whatever vendor and then you install it and just trust that it will do its work. When it then does, when you start your application, so it will take this bytecode and interpret it line by line. Every line of the bytecode will be interpreted, and it's in chunks of methods. The method is the one small part that can be optimized in the JVM. It compiles or interprets these methods down to machine code, line by line, and methods that will be called a lot will then have to be always be interpreted every time you call it. The JVM is profiling that. It's watching it. It's counting, “Oh, this method was now, let's say, interpreted a thousand times. So, maybe I should compile it.”

So, it takes this method, this hot method, and puts it in this so-called C1 compiler or the formerly known client compiler. This compiler then compiles the complete method down to machine code, which is much faster, right? It keeps the compiled version. So, the next time you call it, you can just execute and it's way faster. The problem with this is that we have this, as I told you, the C1 or formerly known client compiler, it just compiles very fast without doing a lot of optimizations. The reason is, it's the client compiler. That means when we – back in the days, when you have an application, you would like to run on your desktop machine, you would like to start it up as fast as possible, right? You don't want to wait until the compiler does really the best optimization. It should just start fast. That was the reason for the client compiler.

But then there is the server compiler. It's the C2 compiler that starts or it starts at the same speed. But the compilation takes way longer, because it does a lot of optimizations, and it really produces highly optimized code. I told you, first goes into the interpreter, read the profile. If the method was called, let’s say, a thousand times, then the JVM takes the method, compiles it with the C1 compiler, and then again, starts counting. If the method will again, be called, let's say, 10,000 times, then the JVM decides, now, it might be a good time to optimize it really well, because this method will be called a lot of times.

Then, it passes this hot method with all its profiling information from the previous runs to the C2 compiler, and then it redo the totally best optimization and compilation that's possible, and then you get the fastest code. That means interpretation, C1 compiler, C2 compiler, this is the normal flow. There are different ones, but this is the standard. This is how it works. You can imagine that it takes some time, right? There might be methods that won't be called a thousand times, so they will never be compiled. For others, it might take longer. So, that means the before they end up in the C2 compilation, might take a minute or two. This is really – and that really depends only on your code and how often specific methods are called.

[00:08:06] AD: You call them client and server computer. Where does that terminology come from? What's going on there?

[00:08:12] GG: That's what I mean. In the past, we had these two different compilers like the client and the server compiler. When you started up the JVM, you have to choose. I would like to start the JVM with the client compiler, and there was only this one compiler. So, there was interpretation, and then the compiler that you choose. Then on the server side, of course, I used the server compiler, because startup wasn't really a big problem when we have this huge application servers, okay, they start maybe in 15 minutes, but then the code is great.

On the client, when you start, for example, an IDE like NetBeans, or IntelliJ, you would really like to start it very fast. That means to use the client compiler. But then in JDK 7, they made the decision – I think it was already Oracle – they made the decision that it might be a good idea to have both compilers at the same time. First using the one that is really fast, and then the second one to make it better. And that's the so-called tiered compilation. Since JDK 8, this is the standard. Now, in all JDKs, this is how it works. We have both compilers and they work together.

[00:09:18] AD: Gotcha. How quickly those like optimizations, those compilations happen like? Is that happening during startup? Some of these methods are getting invoked a thousand times? Is it like, “Hey, pretty quickly after it's like taking requests, you just like understands when that's happening”, or is it an hour down the road? When is that happening?

[00:09:35] GG: It really depends on the application, but you can think about the interpretation of methods, right? That means that the time it takes to get into the compilation phase, it's for normal

methods, it's so fast, it's milliseconds. Then the C1 compilation starts and then it will be profiled that can be seconds, and then the C2, that could be hours. It depends how often these methods will be called and how huge the application is. If you have a small application, you don't have a lot of methods, of course, the whole optimization will be quicker. But if you have a really large application, then it can take some time to optimize the whole code. It depends when you run the code. How many methods do you touch, right? If you just start it and don't do anything, of course, it won't really optimize a lot of methods, right? The more you use it, the more optimizations are going on.

[00:10:30] AD: Gotcha. You mentioned a few times, this optimization is happening on a method level. Does that change how I, as a Java developer should write my methods? Should I write fatter methods or skinnier methods? Or is there any factor there?

[00:10:44] GG: Yes, generally, it's easier to have shorter methods. So, the shorter the method, the easier it is to optimize it. Because you can imagine, if you have a method that is 200 lines of code, then it's really hard to figure out how can I optimize this huge method. But if I have just a method, that is four lines of code, very easy to optimize, right? Of course, that leads to lots of methods. But if you split it down into smaller pieces, it's also easier to maintain. Because if you would like to support it, or let's say, you write the code, a year later your colleague comes in, and she checks the code and say, "Oh, wow, this method has a thousand lines of codes. Cool." You should keep it short.

[00:11:31] AD: Yes, yes. Okay. Talking about what D optimizations works. I remember in your talk, talking like, you could see some sort of dips in performance on D ops. What are those?

[00:11:40] GG: Yes, that's pretty interesting. I would say, this is, one on one side, it's the strength of the JVM. On the other one, it could be a performance hit, right? It's both. The way it works is, in the JVM, we have something which is called speculative optimizations. That means the JVM is speculating how the code will work. It takes a method, and then it profiles the method. Profiling means just watching the method. Then, it says, in this method, I only execute this one branch. Let's say there is an if, then clause in this method. So, it said, it's only the if one, and that will be executed every time I never touched any else. And then it just gets rid of it. So, it can just remove the whole thing and make from the whole method, just one line of code, for example.

This is great and it's much faster. But now imagine, suddenly, the else clause would have been called. So, what should the JVM do? It has to do optimize it. It takes the optimizations that it did, throws them away, and goes again, with the complete method through interpretation C1 and C2. That helps to – in case the JVM is right, the performance is awesome. If it's wrong, then it has to go again through the loop, and that's the performance. Because interpretation is slow, you first have to reach a thousand times, and then the 10,000, and then it goes back into C2. This is a little bit of the bad part and the good thing is that with this, you can really optimize an application to very tailor to the needs that you have, right? Because it depends on how you call the methods. Then, JVM can optimize this method for this specific use case, which is very powerful. It makes a lot of the performance benefits from the JVM are coming from these speculative optimizations.

[00:13:43] AD: Okay. So, the JVM is doing a bunch of work with this profiler, figuring out how to optimize all this stuff. If I'm like a pretty performance savvy Java developer, can I annotate methods or things like that to say like, “Hey, do this without even thinking about like – don't even wait for the profiler. I know this is going to be a hot path or something like that. So, just optimize that right away.” Or is it just like, “Hey, lean on the profile.”

[00:14:05] GG: You mean, giving hints to the JVM, like saying –

[00:14:08] AD: Yes, different things like that, yes.

[00:14:10] GG: Not directly on the code level. But you could, for example, you can tell the JVM at startup that it should only choose the C1 compiler and don't do the C2. Some people do that. It's like, if you have a micro service, for example, and you started up and startup time is crucial. Then, people make the decision, “Oh, I don't really need the highest optimized code. But I would like to start up as fast as possible. So, I just tell the JVM only to use the C1 compiler because this is really fast in compiling.” Yes, it might have in specific use cases.

So, I heard about people that say, “Yes, it really improved the startup time a lot.” But in general, you have these kinds of hints that you can give to the JVM. You can even stick to interpretation only if you would like to make it really slow. But it's possible. Stuff like that is possible, but you can't really tell the JVM, “Oh, this method, I don't want to optimize. Or this method, you should optimize directly.” So, this is not directly possible. You can write code that is easier to optimize for the JVM. But that needs a lot of

knowledge on how the JVM internally works with optimization. It's not something that the standard developer should do, because it doesn't make sense. You gain really, maybe nanoseconds out of that, if you don't know what to do.

[00:15:34] AD: Yes. If I'm not that concerned about startup time. Maybe I have a fleet of services, I know sort of when I need to scale up, and I have minutes even to scale up. Can I basically say, “Hey, do all the optimization work before you even start serving and are ready for requests? So, it's super fast right out of the gate?” Or does it really need sort of production traffic or regular traffic to do that?

[00:15:54] GG: No, the ways that you always have to go through this whole thing. This is how it works. In Azul, we have a JVM that's called Prime that has some specific tricks to avoid this problem. So, what we do is you run at once, and the JVM just stores all the optimizations and deoptimizations. The next time you start it, it just checks the file, “Oh, I don't have to go through this. I just take this optimization. This is the way to go.” Yes, but this is only available in the Prime JVM. So, that helps to directly, to avoid all this going through the whole trial and error thing. In the end, the JVM directly knows this is the best optimized way, and I just directly optimize it like this. Yes, we offer that in the prime JVM. But this is not part of the Open JDK. In this case, you always have to run through interpretations C1 and C2.

[00:16:53] AD: Yes, absolutely. Okay, this is potentially a dumb question. But I know Java has like the garbage collector, people talk about that a lot. Does the profiler and optimization compiler like affect garbage collection at all? Or is that just like a totally separate – does garbage collection get better over time with a running application? Or is that just a separate category?

[00:17:11] GG: That depends. That's indeed a quite complex topic, because we don't only have one garbage collector, but I think we have at least five in the JVM and they all work differently, and they have benefits in different scenarios, even. That means if you have a specific application, let's say you have a trading application, and maybe you need a concurrent garbage collector that is doing work in the background, but you have to think about like this. If you have a garbage collector that is running concurrently in the background, then it always needs CPU, right? Because when it runs at the same time as the application, then part of the CPU is needed to do the garbage collection. Then we have garbage collectors, they don't do it concurrently. That means they just do it sometimes, and then what they do, is they just stop the application completely. That's the stop the world garbage collector.

But the garbage collection process itself won't change over time a lot. It's a process that's running. It depends on how you use the application. In principle, you have to think about garbage collection, like if you remember the old days on Windows PCs, you have to do the defragmentation of the hard drive, right? This is exactly what the garbage collector is doing, just in memory. It fills it up, the application fills the memory, and then some parts will be taken away, put in use. Everything gets cluttered and at some point, the JVM has to clean it up. That's the garbage collector. For that, usually, they stop the world, clean it up, continue running, or you have something that is continuously running in the background, and then it will do the garbage collection all the time. But that needs more CPU time.

[00:18:53] AD: Gotcha. Okay. So, I want to move into options for speeding up Java, including CRaC, things like that. But one thing I love from your talk is, you split it up into different areas of performance optimization. There's the cold start, which is, if I start my application, how long until it can actually execute its first request, basically. Then, there's that warm up period, which is like how long until my code is fully optimized. Just to quantify that a little bit, how long are we talking with cold starts? Is that a second or two? It that three seconds? How long does a cold start take? I know it varies quite a bit. But what are we looking out there?

[00:19:29] GG: Yes, that's interesting. People mess that up a lot. When they talk about Java in the microservice environment, for example, in the cloud world, then they say, “Oh, Java is slow, JVM is slow. The startup sucks.” That's not really true. The JVM itself starts up in 20 milliseconds, 15 milliseconds, depends on the JVM. Because it has nothing to do, right? The thing it has to do as your application, so that means the more stuff you load on it, the more stuff it has to do, and that takes longer.

The cold start depends on the application. But usually, first, you have to do the initialization of the JVM, which is pretty quick. Then, it starts loading all the classes, and then it starts to initialize the classes, loading resources, and all these kinds of things, everything that your application needs to start. Then, at that point, it already is interpreting and compiling. At the same time, it's runs your code. That means, in the beginning of the startup, you'll see this peak in the CPU, where the CPU has to do a lot of stuff at the same time. It has to run the application, at the same time, it has to profile it, interpret it, compile it. Then, that's what we say, it's the JVM startup, is the time to first response.

So, the first time your application says, “Hey, I'm here.” That means then, at least the basic stuff of your application is loaded and is running, and that can take from milliseconds to seconds. It really depends how big the piece of code is that you try to compile and start at that point. But that doesn't really mean, like you said already. That's, for example, JVM startup. It means the first time everything's running, applications coming back with a result. It doesn't mean it's the highly optimized version of your code. Just because it came back the first time, it doesn't mean it's now the best optimized code. The more often you call it, the more optimized we'll get over time, and that's what we call the JVM warm up. So, application warm up.

This is really true story. We have customers, they have, for example, their application, they stop it in the evening at 6pm, and they have to be up and running in the morning at 8am, when the market opens. So, what they do is they start the application around, let's say, 4am in the morning, to warm it up for four hours, just to make the coat really perfectly warmed up. So, we have a solution for that. That's the stuff that I mentioned already. In the prime JVM, we have this, it's called Ready Now, where we store all these optimizations to avoid that problem. But if you don't do that, you really have to go through the whole code all over, over, and over again, to really make sure everything is touched, everything's optimized. Because when the market starts, it has to be the best code possible.

[00:22:28] AD: Is that just running on a bunch of historical data for four hours just to warm it up?

[00:22:33] GG: That's interesting. They have specific code to warm up the application. That's a very specific use case. Usually, let's say, the normal developer in the normal company, they don't have that problem. This is if you, for example, do high speed trading, right? You really need to be on point at 8am, because otherwise you lose money.

[00:22:55] AD: I was going to say, how big is the delta between sort of my first request, and then like, a fully optimized warm up? How much faster is that going to be?

[00:23:05] GG: Oh, that could be – I just can guess. But I think it could be seconds, depending on the application of it. For these guys, it's really milliseconds. It's really the point where they have to be precise, which is very specific, right? This is not really what people do, actually. Usually, people are complaining about ARM application takes five seconds to start, which is, that's a lot. I mean, just imagine you're in a in a web shop, like Amazon, and you just would like to buy something and click the

button and then five seconds, nothing happened and suddenly pink. Okay. Of course, you would like to avoid that, right?

That's the whole thing why we thought about we need to do something to improve startup time of the JVM, because these are problems that in the microservice environment, people see that because with every start of your program, you have to go through interpretation and calculation until everything's running, right? That takes some time. Depending on the technology, and on your application, it can be seconds to minutes. So, it really depends. I can be 15 minutes or 30 minutes until the first response. So, this is possible, and you can imagine you don't want to have that.

[00:24:20] AD: Exactly, yes. Okay, so then walk me through what are my options? I know CRaC is one of them. I want to talk about that. What are my options if I'm having either slow cold starts or performance? What options are there?

[00:24:32] GG: Yes, what you can do is, and at the moment, very famous is GraalVM. Probably you've heard of it. It's a different approach where they can create a native image. That means it's not really Java code that is running them. But this is like, let's say, you have a C program, which you compile down to binary and this is what you get. But they create that out of the Java code. Because it's a binary and native executable, you can start it directly. There's no compilation, no interpretation. There's no profiling. It's just the code and it will run. That has a very fast startup time. This is 20 milliseconds, 30 milliseconds, which is very attractive, of course.

But what they don't tell us that on the other hand, you really lose some performance, because that's the benefit of the JIT, right? The just in time compiler and the JVM, because you can't do this speculative optimization when you have a binary image. That means everything is compiled before it is run. So, the JVM, or let's say, the compiler has no idea what's going on in the code. It can do some static analysis and guess, okay, this is something I can optimize and they are quite good at that.

But at runtime, the JVM can, through the deoptimizations and the speculative optimizations, it can optimize the code at runtime. This is something you can't do with AOT compiled code. The ahead of time compiled code, which is that's the drawback with GraalVM. It's good from the startup, but you lose performance, and that you can measure it. It can be up to 50% slower. Then the question is, if startup is your only problem, okay, then you're fine. You can use it. But if you also need performance, then you

need to spend more money to get the same performance, right? Then, it's more expensive again. So, the question is, really, it's a tradeoff. You have to think about it. The biggest pain point in my eyes is that you leave the Java world.

So, a Java program, you can debug with the tools that we know. But once you go into the native image world, you're really in a native binary, and then you need suddenly a C debugger. You need totally different debuggers to debug that, and it's a different kind of working. You can do it. I also use it for some tools. It's nice, but it's not for everything. So, it has its use cases. But I wouldn't recommend – and it's interesting, because I always ask people in the sessions, who is using it in production, and it's really, it's not even a handful. It's maybe one at a session.

[00:27:19] AD: That's interesting.

[00:27:20] GG: It's not easy. It's for simple things, it's easy. But as it gets more complex, it is more complex. You run into strange problems, because native code reacts differently from jittered code. That's just the fact. Even Oracle is pointing out on their website, they say, just keep in mind that this is native coded. It behaves different. So, you can run into problems.

[00:27:43] AD: Yes. What about in terms of resource usage, if I'm using GraalVM, is that going to use more memory than a normal application?

[00:27:54] GG: No. That's the other benefit is you have a smaller memory footprint, because if you do ahead of time compilation, what you usually do is you just check before you run – before your compile it, you check every method that you need on every class and to compile only the stuff that you need, or the classes. Where if you have the JVM, for example, and let's say you use some library, right? This library comes with a thousand classes, then the JVM will load the thousand classes. The AOT compiler will only load the one class from the thousand that you need and compile that one. That's the benefit on the one side, because it creates a really small binary, the footprint is smaller, but to get there as harder. Because then you need to know when you compile it down to the native image, the compiler needs to know what is used. If you don't know that, then you might be able to compile it. But once you run it, it tries to call the class. Oh, it's not there, because it's not part of the binary. That fits these are the problems you have to solve. They are solvable. But it takes more steps to get there.

[00:29:01] AD: Yes. Okay. So then alternative, tell me about CRaCing and how that approach is a problem, and what's different there?

[00:29:07] GG: Yes, the CRaC approach, it's quite interesting. Everybody knows that. Everybody who has a laptop knows how it works without knowing it. Because of your work with your computer, and in the evening, if you don't really shut it down, you're just close the lid, right? Then you expect the next morning, you open it, everything's there, like you left it the day before. That's checkpoint restore. That's exactly what it does.

The operating system just detects, “Oh, the lid was closed. I save everything that's open to the disk, and this is the checkpoint. Then, when you open it again, it will just realize. “Oh, there's a checkpoint. I just restore everything back to memory and that's how it works.” We thought about this, if it would be possible to do the same for the JVM, right? Because in principle, if you think about it, the JVM is like the computer, and your application running in the JVM is like your application running on your computer.

So, there is a project on Linux called CRIU, and that's Checkpoint Restore in Userspace. That was created in 2013. So, it's already 10 years old, and already 10 years part of the kernel. That was created to stop applications on Linux to store the state, the checkpoint, and then to restore it again back to memory. In principle, exactly what we need, right?

Unfortunately, it's not so easy, and the problem is not again, the JVM, but it's the application running in the JVM. Because the JVM you can start, say, “Hello world”, then you can create a checkpoint storage and then restore, it’s all good. Let's say you have your microservice, you started up, and it has a connection to a database. Then, you create the checkpoint, that means you save everything to disk, you close the connection, and then you restore it. Then your application will start, it will try to connect to the database, and then there's no database because the connection is gone. So, that's the problem we had to solve and that's correct. That's the coordinated restore at checkpoint. We try to coordinate the whole process of creating a checkpoint and doing the restore by letting the application know, when there is the checkpoint and when it was restored.

Then, it gives the developer the opportunity to do something before the checkpoint happens and to do something before it will be restored. This to do something means for example, close the database connection. When it will be restored, just open the database first, and then restore. Right? That will

make sure that everything works after the restore. That's the main idea. The drawbacks are obvious. Because first of all, it only works on Linux. There is no way to make that work on Mac or Windows at the moment.

[00:31:52] AD: Just to be clear, this is built on top of CRIU, like the Linux, the CRIU that's in the kernel, okay?

[00:31:59] GG: Yes. We use CRIU to do that, but it would be possible to use something else. So, there are other tools on Linux, they do the same thing. But CRIU is the most favored one. Even Docker is using it and to Linux containers and Podman, they all can use CRIU and it's very mature. It's very good. Because this is not available on Windows and Mac, we can't do it. That doesn't mean we are bound to CRIU. We could also use a different project that does this checkpoint restore. That means if there would be something available on Mac or on Windows, we could use it to do the same thing. But unfortunately, there's nothing really that can really do the same stuff that CRIU can do on Linux.

[00:32:48] AD: What percentage of production job applications are running on Linux as compared to Mac or Windows?

[00:32:55] GG: I could just guess, but I would guess it's 99.9%.

[00:32:59] AD: Right. It's almost like you're worth going –

[00:33:02] GG: When I started with a CRaC talk, I always ask the people and I omitted a little bit of a joke. So, who's using Windows in production? And nobody raised their hand. But then suddenly, it was admittedly last year, two people raise their hand. Damn it, people use it in production. In the meantime, I had four people in one year in over 40 sessions that really run the Java application on Windows. So okay, they can't use it. That's a fact, which is one of the drawbacks, of course. But because everything's running on Linux, more or less, it's for the majority of people, it's usable.

The drawback is, of course, that you have to store this checkpoint somewhere. And in the JVM, it's not so easy to describe. But the JVM itself also manages its own memory on it. So, the memory that is used for your application, and the main part of it is the so called the heap. The heap is the stuff that needs to be saved on the disk. The heap can be very large. There are customers that have heaps in

the terabyte size. That means the application uses, let's say, one terabyte of RAM. And if you need to save that to a disk, that's of course a problem. But this is not the way we think about using CRaC. This is really more for microservice environments or for everything that you need to restart a lot of times over the day, for example, and that takes long.

Let's say you have an application and it takes 15 minutes to start. You need to start it every morning. Of course, it would have to start that in one second. Right. Then, it makes sense. For this kind of use cases, CRaC is really ideal. Because what it does, it takes the complete state of your application, it closes every, let's say, resource before it does the checkpoint. Then, you can restore at whatever point in time from these files. The only thing you rely on is the IO speed. So, it means if you have a really fast SSD, it just loads back to memory and that's it, and that could be milliseconds. We have startup times around 14 milliseconds. It's more or less in the same range as the native images, but you then have a complete running event.

[00:35:22] AD: Is it like the fully warmed up optimized code as well?

[00:35:25] GG: Yes. That depends on when you do the checkpoint, which is another advantage, right? Because you make the decision. You as a developer say, "Okay, I would like to do the checkpoint directly when the application starts." Okay, no optimization is done. Or you can say, "Oh, I would like to do the checkpoint after an hour, and then restore from there." The later you do the checkpoint, the more optimized the code is, obviously. Of course, also the checkpoint will be bigger, because more stuff is in the heap. Then, of course, the files will be bigger.

But you're right. What the JVM is doing, we just stop it and continue running. The JVM doesn't even know that it was stopped. So, more or less. That means for that, for the JVM, it's just everything's goes on like it was. I was fully optimized, I'll just continue running. When you start from there, it has a lot of benefits coming with it. So, you're still totally flexible. That means the JVM, if the code is changing, after that, you do the restore, you still can do the JIT compilation and optimization. And the JIT can still optimize the application to your needs, even after the checkpoint, of course, which is a big benefit. The GraalVM stuff can't do that, because it's just precompiled. We can do that, which is good.

[00:36:47] AD: I want to talk more about that point. So, if I make a checkpoint one day, and then my developers change something in the code, is that checkpoint completely invalid? Or I can still use that

checkpoint with my new deployed code that's like slightly different, and it'll get some of the benefits from the checkpoint? Or do I need to make a new checkpoint once I have like a new bundle?

[00:37:06] GG: Yes. You have to do a complete new checkpoint when you do that, because you really have to think about this is your application running in the JVM, and just stop it right here. This is the code – it's like a freeze. The restore really just started from there.

[00:37:23] AD: It needs to be the same.

[00:37:22] GG: Change the source code then it's not in your code, in the checkpoint, so you have to create a new checkpoint. First of all, it sounds like, “Oh, that's terrible. I always have to do a checkpoint.” In principle, it won't run, it will run without doing a checkpoint, right? Because then it will just go through interpretation and all of these things. That means, from your point of view, as a developer, nothing changed. You just code as before. You optimize. You do whatever you want. Once you go to production, you create the checkpoint, in your CI/CD system, in your build system, and then you deploy it with your build. It means, it's not really a big problem. Trading the checkpoint, it's principle the same, and you do, for example, the GraalVM stuff in the formerly Enterprise Edition, they changed it last week. There's something that can make the native images faster, which is called Profile Guided Optimization.

To do that, what they do is they create a profile. They run the application once they create a profile, and then they take all the optimizations and apply it when they create the native image. But also, they have to do that all the time they've changed the code. It's in principle, the same process. It's done automatically in the background. It's nothing that you have to do manually. Usually, you trigger it in your build system, you say, “Save that checkpoint, and then you usually only – you don't save it in your container. But you save it somewhere in your cluster, and then every container can just point to that volume and load it from there. So, it means you need one place with full storage for this checkpoint, and then you can start it from there.

[00:39:00] AD: Going back to your high frequency trading example for earlier, instead of doing that four-hour warmup, maybe you got the checkpoint from yesterday or from whenever they made it, and now it spins up in a second or so and they can be going right away.

[00:39:14] GG: That would be the case, but especially for them, it does make sense.

[00:39:19] AD: Okay. Is that because their heap size or something else?

[00:39:21] GG: They need to warm up the applicant in very specific ways. Okay, maybe they could do a checkpoint after they warmed it up. Then it will be – this is CS.

[00:39:32] AD: If it was at the end of trading the previous day, and they just took a checkpoint there, would that –

[00:39:36] GG: That doesn't really – yes, it might work. I'm not sure. I just heard that this is very specific.

[00:39:41] AD: It's hard because, yes, they're discounting every –

[00:39:44] GG: So, I just have the information that they do it, but how it works, it's out of my knowledge.

[00:39:52] AD: Yes. Okay, what about for things that are, maybe not my custom application, but things that are very widely used that do their job applications like Cassandra, Kafka, Elastic Search, things like that? I mean, is it possible to just basically just make one CRaC perversion of like a fully optimized thing and distribute that as part of Kafka? Do I need to do it for myself still? What does that look like?

[00:40:19] GG: That's a good question. I never really thought about it. Interesting thing is that we created this project, and we made it an open JDK project, and we'll try to make it really available for the general open JDK. But we are not really the users, right? Like the situation right now, you face me with that question, I have to think, "Hmm, that could be interesting." Because, yes, why not? I mean, in principle, everything that is running on the JVM can be checkpointed, except desktop applications. There's no support for desktop applications at the moment, and the reason for that is, that desktop code is really very specific hardwired to the operating systems and that kind of thing. It's not so easy to do that. But everything else is running on a server can be checkpointed.

I'm pretty sure you could checkpoint Kafka. I'm not sure what the benefit is, because, like I said, it only helps you starting up Kafka. If you need to start it up every day, yes, okay, maybe that helps. But if you started it once, and run it for a week, and probably if it's an hour or a second, it doesn't make it. It's interesting, because people come up with all these questions, and then you will first have to think does it make sense or not?

But we had one question, where someone said, “Yes, I have to restart this server application five times a day. That's how we use it and it always takes 15 minutes.” Of course, it makes sense to create –

[00:41:51] AD: There we go.

[00:41:54] GG: It's really, it's a per case, really, decision. You really have to think about, it doesn't make sense, you think about the process. Yes, why not? Or no, it doesn't make sense. But this is the same with the AOT stuff. So, you really have to, don't follow this, “Oh, you have to go that way.” Just think about, do you need that? Do you need to solve it and choose the right tool for the product.

[00:42:22] AD: Yes, yes. Okay, you mentioned we just want to cover with. On top of CRIU, but it has hooks into Java. So, you're doing like the before shutdown, after startup type thing. How much work is that for an average application to implement that? Or is that just, hey, your database connection, your Redis connection, and you're good to go? Or is there a lot of stuff?

[00:42:41] GG: Oh, yes. It's like you said, to make it work, we had to implement some hooks to let the JVM and CRIU trigger the checkpoint, and the JVM is checkpoint, and then we need to implement some hooks so that we can tell the application that know something is happening, right? What you have to do as a developer, let's say you have an application, then in principle, you have to close all the resources that are either open files, or socket connections. All kinds of, let's say, network connections. If you have a connection to your database, to another microservice, or to another web service, if you have a file open where you read or write stuff, these kinds of things, you have to close before you do the checkpoint, and you have to restore this or we initialize the connection in the restore mechanism.

So, there are two methods. It's before checkpoint and after restore. These two methods you have to implement in your code. Then, only in the classes where you need it. This how to tell, usually, you don't have in every microservice. Lots of different resources open. But database connection, typically, you

have, then usually, you have that in a central place where you initialize it at startup and then you use it. This initialize the class, the popular class that have to implement the resource interface, and there you have to do the closing and reopening of the of the database connection.

That's not the biggest problem. So, the biggest problem, really, is that most of the people don't use plain Java to write their code, they use frameworks, right? They use Spring Boot, or they use Micronaut, or they use Quarkus. This is the problem, because if you run for example, let's say you have Spring Boot. Spring Boot itself has lots of stuff open. Connections, maybe. But we can't just tell Spring Boot, “Hey, just close the connection, because there's no way to do it.” That's the problem, right? But we talked to this, to the people, and Spring announced it at Spring IO, a couple of weeks back that they will make create a first-class citizen in Spring 6.1, I think, which, it will come out end of this year. And that means that also Spring Boot will have the capability to use CRaC and Micro now already has support for it, and Quarkus was has some basic support, so it's not too bad.

[00:45:10] AD: That's great. Yes, seeing the community adopted it as well. I guess, the last thing is, how long does it take them to make that snapshot? Is it mostly just IO based on how big that heap is? Or is it, are there other factors there? Or is it pretty instant other than writing into disk?

[00:45:25] GG: It's more or less writing for disk. What we do is we clean the heap and compact it. So, it's like a garbage collection, because we don't need to put all the clutter in the checkpoint, right? We clean it up, and then we save everything down to disk. This again, just depends on the hard drive speed. So SSD. In principle, you could even store it in the cache. If you have a Redis, a memory cache, you can also store it there, that would be very fast. But that it's not really usable. Because somehow you have to transfer to your production environment, right? But yes, saving the checkpoint doesn't really take a lot of time. Of course, if you have a terabyte heap, it takes some time. But usually, you don't have that. So, if you have a microservice, and it's, let's say, couple of hundreds of megabytes. If it's that big, then that really – it's a second, half a second. It's really fast. That's not a problem.

[00:46:19] AD: Yes. Cool. Well, I've learned about some of this stuff. Last thing I want to ask you, just what are you excited about whether in the Java world, in the tech world, more generally, what's exciting to you that's going on right now?

[00:46:31] GG: It’s not AI, to make that clear.

[00:46:37] AD: Yes, that's like the standard answer. So, it's great to get a different one here, yes.

[00:46:43] GG: Yes, it's that hype these days. I'm not really – I'm fascinated by the technology, not the way it's used and how it's used. For me, it's hard to describe. I mean, I'm really – I love coding. It's not only Java. I also do Swift and other stuff. Also, I love hardware. I also think around with building stuff.

[00:47:03] AD: What's the latest thing you've built hardware wise?

[00:47:06] GG: Oh, it's a glucose monitor for my son because he has diabetes type one. I built a display with people and everything. Our house is full of this little tool.

[00:47:19] AD: That's so cool.

[00:47:20] GG: That's what drives me. So, this is my – I don't really have some technology, which is the best thing I can think of. So, it's everything this is –

[00:47:30] AD: Yes, real projects, helping people. That's great. That's cool.

[00:47:33] GG: Yes, it's fun. I mean, if I can help someone with that stuff, it's great. If not, it's also great. For me, it's fun to do it. If more people can benefit from it, then even better.

[00:47:46] AD: Cool. Well, Gerrit, thank you for coming on the show. I really appreciate this. I love talking with you folks at Azul, because you really know Java inside and out, and just can explain to someone like me. Or just say, everyone, go check out Azul, go check out CRaC. Gerrit, where can people find you if they want to be in touch with you?

[00:48:02] GG: The best thing is to find me on Twitter, still. This is the only social media thing that I use. And this is @hansolo_ is my Twitter handle. I'm responsive there. So, if you would like to know something, just ping me on Twitter, and I'll try to answer as soon as possible.

[00:48:20] AD: Cool. Sounds great. Gerrit, thanks for coming on the show.

[00:48:24] GG: Thanks for having me.

[END]