EPISODE 601

[INTRODUCTION]

[0:00:00.3] JM: When software is performing sub-optimally the programmer can use a variety of tools to diagnose problems and improve the quality of the code. A profiler is a tool for examining where a program is spending time. Every program consists of a set of different functions. These functions call each other. The total amount of time that your program runs is the sum of the time that your program spends in all of its different functions. When you run a program, you can execute a profiler on that program and the profiler will give you a breakdown of which different functions time is being spent in.

If you have function A, B and C, your profiler might say that your program is spending 30% of time in function A, 20% of its time in function B and 50% of its time in function C. This is pretty important if you have a program that's running really slow, seemingly. You need to be able to see where it's spending all of its time because a slow running program is not really running slow. It's running perfectly fast. It's just spending a lot of time in one particular area of the program in a way that may not be so productive to you as the user.

Julia Evans is a software engineer at Stripe and the creator of a Ruby profiler called Rbspy. Rbspy can execute on a running Ruby program and report back with a profile. As Julia explains, a profiler turns out to be a nontrivial piece of software to build. To introspect a Ruby program, you need to understand how the Ruby interpreter is translating Ruby code into C structs for execution.

This episode is about profilers, but in order to talk about profilers, we also have to talk about Ruby, the Ruby interpreter and the way that executing programs are laid out in memory. It was a really enjoyable and, in some ways, low level conversation, but none of it is too hard to understand. So I think it was a really good explainer, especially for people who know that programs execute, and in order to execute, those programs are sitting somewhere in your machine while they execute. This episode digests a little bit of that content. How do programs actually execute, especially interpreted programs, because we know that these interpreted programs are getting executed in some way, but they're not getting all compiled into byte code

or they might get compiled into something that's like byte code or they might get compiled into some other intermediate representation, which is what happens in the Ruby interpreter, for example. But it's not just going directly into ones and zeros, and you may not know how that works. We explained it in this episode.

I also want to announce that we're looking for writers for Software Engineering Daily. We want to bring in some new voices. We're focused on high quality content about software that will stand the test of time. Today's topic is a great example. Profilers is not something I've read much about relative to how important they are and how interesting their creation turns out to be. If you want to write, go to softwareengineeringdaily.com/write to find out more. We're looking for part-time and full-time software journalist. We're also looking for volunteer contributors who just want to write about software engineering. We want to explain technical concepts. We want to tell the untold stories of the software world and we'd love to hear from you. So go to softwareengineeringdaily.com/write send me an email jeff@softwareengineeringdaily.com. I'd love to hear from you.

[SPONSOR MESSAGE]

[0:03:45.7] JM: At Software Engineering Daily, we have user data coming in from so many sources; mobile apps, podcast players, our website and it's all to provide you, our listener, with the best possible experience, and to do that we need to answer key questions like what content our listeners enjoy? What causes listeners to log out, or unsubscribe, or to share a podcast episode with their friends if they liked it?

To answer these questions, we want to be able to use a variety of analytics tools such as mixed panel, Google Analytics and Optimizely. If you have ever built a software product that has gone for any length of time, eventually you have to start answering questions around analytics and you start to realize there are a lot of analytics tools.

Segment allows us to gather customer data from anywhere and send that data to any analytics tool. It's the ultimate in analytics middleware. Segment is the customer data infrastructure that has saved us from writing duplicate code across all of the different platforms that we want to

analyze. Software Engineering Daily listeners can try Segment free for 90 days by entering sedaily into the how did you hear about us box at sign up.

If you don't have much customer data to analyze, Segment also has a free developer edition. But if you're looking to fully track and utilize all the customer data across your properties to make important customer-first decisions, definitely take advantage of this 90-day free trial exclusively for Software Engineering Daily listeners, and if you're using cloud apps such as MailChimp, Marketo, Intercom, AppNexus, Zendesk. You can integrate with all of these different tools and centralize your customer data in one place with Segment.

To get that free 90-day trial, sign up for Segment at segment.com and enter sedaily in the how did you hear about us box during sign-up. Thanks again to Segment for sponsoring Software Engineering Daily and for producing a product that we needed.

[INTERVIEW]

[0:06:15.2] JM: Julia Evans, you are a software engineer at Stripe and you're the developer of a Ruby profiler called rbspy. Welcome to Software Engineering Daily.

[0:06:25.1] JE: Thanks so much having me.

[0:06:26.6] JM: So we're talking about profilers today, and I want to start by describing what a profiler does. What does it mean to profile a program?

[0:06:36.8] JE: Right. There are actually a few different kinds of profilers, right? You have CPU profilers, memory profilers. I'm most just interested in talking about CPU profilers right now. So what it means to take a CPU profiler of a program is basically you want to know what functions it's spending its time in.

There's one way to do this. The main way I'm interested in is what's called a sampling profiler. So what you do is you have your program and then let's say a thousand times a second or a hundred times a second you're like, "What is the stack of my program right now? What functions is it running?" Then if you have that, then you can get a really good picture of like where your

program is spending its time, right? Maybe it's spending its time downloading a file from the internet and you're like, "Oh! I spend like 97% of my time in the download file function," and then you can go there and make your program faster. So that's what profiling is about.

[0:07:24.2] JM: And why is that important? How would a developer use that ability to be able to breakdown where in a program their application is spending a lot of time?

[0:07:36.2] JE: Well, I guess the way to think about it is if your program is slow, I think it's basically a waste of time to try to guess why it's slow. I've tried to do that before, or maybe I'll ask the smartest person, "Hey, do you know why this is slow? This is what I saw." I think this is a really bad idea, right? Because if your program is slow, the best thing to do is to just like objectively figure out why it's slow. So if you know like 80% of my time has been spent in this function, then that's it, and you know where to start, right?

[0:08:06.4] JM: Is there a time when I should profile a program? Should I do it regularly as a cleaning operation or is it only when you really sensed that something is going wrong with the program?

[0:08:18.0] JE: I usually do it if I realize there's a problem. I don't think that programs to be just like inherently that important a lot of the time, right? It's like if your program is fast enough, it's fast enough. So I think it's only worth making it faster if you really feel like you have a problem.

[0:08:31.3] JM: Let's say I want to profile a program that is running on my computer. If I want to profile a program, what information would I like to learn about that program?

[0:08:42.2] JE: Which functions it spends most of its time in?

[0:08:45.5] JM: And in order to improve a program, I need to know which of those aspects of the program are performing poorly. So in order to introspect where it's spending its time and which of those expenditures of time are performing poorly, how am I going to find that out? How am I asking the program where are you spending your time?

[0:09:06.4] JE: Right. So the tricky thing about this is that every programming language has different profiling tools. So I think maybe the hardest thing about profiling – So if you have a Python program, you have to use a Python profiler, and if you have a C program, you use a C profiler. Java has really, really good profiling tools, and I think Java probably has the best profiling tools, Java and C and other languages, like almost all other languages. I'm probably leaving something out, but almost all other languages have worst profiling tools, and I think like this is the most challenging thing about profiling, is that it's different for every language.

[0:09:39.6] JM: As you said, this profile of reports, the different areas of time that your program is spending time in. So these reports of self-time and total time. Explain what the difference between self-time and total time is for a profile.

[0:09:57.6] JE: Sure. So self-time is how much time is being spent in the function. Okay. If you have your function, and your function might call other functions. So when time is being spent in the function, there's a part of the time which is being spent just in that function, like maybe doing whatever, like doing operations inside that function. Then there's time which is being spent in other functions which that function calls. The total time is the total amount of time being spent in the function and every other function it calls, and self-time is time that's being spent just in that function.

So I think like kind of the ideal thing or like the easiest thing to debug is if you have a function with extremely high self-time where all the time is really being spent just in that function. Because if it's spending 80% of the time just in that function, you can be like, "Oh! This function is the problem." Right? Either it's being called too much or like it's doing something to slowly.

[0:10:48.0] JM: Right. If you have a function and then there are nested functions within that, the time that it spends in the nested functions are going to be attributed to the total time measurement and the time that it's actually spending within this function itself is self-time. As you've said, obviously, if you looked at your program and it's spending a ton of time in self-time and you look at this program and you say, "This is clearly spending too much time. I'm going to need to improve this program." A high measurement of self-time would mean that you should be improving this function itself as supposed to other functions that it's calling.

Now, that is not to say that –

[0:11:24.6] JE: Exactly.

[0:11:25.4] JM: That's not to say though that if you have a function with a high total time, that there's not a problem with this function because you could have a function that is calling external functions egregiously and therefore maybe you need to iron out some of the external calls that this function is making.

So with this report of self-time and total time, what would you do to evaluate what improvements to make to a program.

[0:11:51.5] JE: So I think that's where you – I think this really depends on your knowledge of your own program, if that makes sense? For example, for my profiler actually are based by – It will frequently tell me like it's spending all this time in this one Linux system call, which we're not going to get into it right now what it is, but it'll tell me, "Oh, you're spending all of your time in that function." But I know that that function is really core to my program and I know that I'm calling it the right amount of time, if that makes sense? Even if I'm spending 50% of my time in that function, I know that there's not much I can do about it unless I add caching, which is maybe something I don't want to do right now. So I feel like you really just didn't need to lead on your understanding if you're on code and what's appropriate and what's not.

[0:12:34.5] JM: You pointed out that there's a difference between profiling and benchmarking. A benchmark is a test that I can use to measure how fast my code is. So I could use a benchmark to test my code and then profile it at the same time to get a sample and then use that as – So I'm using profile and benchmarking together. Could you clarify that difference between profiling and benchmarking and explain how you use those two strategies together.

[0:13:02.0] JE: Definitely. So I think of benchmarking kind of like science, right? If somebody tells you your program is slow. Maybe it's kind of hard to get from that to like a concrete statement of a problem with your program, right? I think the ideal thing, like if you're trying to make your program faster, is like you discover a problem where it's slow and then you figure out a way to consistently reproduce that and say like, "Okay. When I run my program in this way,

like maybe with this input or maybe like I call the specific functions, then it's slower than I think it should be." Right?

So like that thing where you run your program in one specific way is called a benchmark, right? So maybe like you're running your program on some specific input and it takes like 25 seconds to run and you're like, "Well, that's too slow." Then once you have that measurement where it takes like 25 seconds – And I think it's important to realize that's there's often a lot of uncertainty in benchmarks. Typically, if I run a benchmark, it might take like 25 seconds one time and 23 seconds another time, and like you can end up with like 10% or 20% of difference just depending on what's going on in your computer at the time, right? It's important to be aware of that.

But once you have this benchmark where you're like, "Okay. This thing takes 25 seconds." Then you can start to say like, "Okay. Where's that 25 seconds have been spent?" right? Like 10 seconds have been spent talking to the network. What's going on? So you benchmark and then you profile and then you figure out why it's slow and then you make changes to try to improve it and then you run the benchmark again after, right?

I think the stuff of running the benchmarking again after is really important because I think it's very easy to profile and then like try to make something better, but then it might not actually make a difference in practice with your real program. So you need to go back to the benchmark to make sure that you actually made a difference.

[0:14:45.2] JM: We're talking here about benchmarking locally. So you're probably going to do a benchmark on your local machine, but you also talk about the importance of monitoring production performance and profiling in production. We'll eventually get into discussing how you design a profile, but just a profiler, but talking a little bit more about strategies for people who are using profilers, how is the act of monitoring production performance, profiling in production, how does that fit into the workflow of somebody who's trying to improve the performance of code?

[0:15:20.8] JE: Right. We're talking about benchmarking, and it's kind of hard to benchmark web applications because in practice you have a lot of users with a lot of inputs, like who are making a lot of request to your web application, and I think it can be really tricky to isolate what

exactly is going on that is making your program slow. So I think benchmarking is a lot easier when you have something like a command line program or a desktop program, which you can really reproduce easily locally what's happening. But with something that's like a web application that's running on a server somewhere, it's often much better to monitor – one thing you wonder is you can say like, "Okay, how long does every HTTP request to my web application take?" Then you can send that to like your monitoring service that certainly we have, that I have at work, but hopefully you also have. Then it will say, "Okay. The median request time was like maybe a hundred milliseconds," but then maybe like the P95, like the 95 percentile request time was like 3 seconds and then you'll be like, "Oh, no. My P95 request time is at 3 seconds. That's no good. I can't have requesting 3 seconds all the time. I need to make that faster."

Then when you make your performance improvements, you can go back to your monitoring and like your statistics about like what's actually happening in production and make sure that that's actually improved. It's really my favorite thing to look at a graph and see like this line of P99 request time drop when you make an improvement. It's so rewarding.

[SPONSOR MESSAGE]

[0:17:01.0] JM: Dash is a conference by Datadog coming to NYC this July. This event will bring together dev and ops engineers who are scaling up and speeding up their apps, infrastructure and teams. Dash will feature hands-on workshops and trainings, plus speaking sessions from engineers who are taking their systems and organizations to the next level of performance, reliability and scale.

Learn more at softwareengineering.com/dash. Dash is in New York City this July and is a conference brought to you by Datadog. Get more details at softwareengineeringdaily.com/dash and use code D-A-S-H S-E-D to get a discount on a Dash pass. That's DASHSED and you go to softwareengineering daily.com/dash to find it.

Thanks to Datadog.

[INTERVIEW CONTINUED]

[0:18:05.4] JM: If I want to profile a program, I could get that program to emit information periodically and then I could just use that information to get a picture of what the program is doing. This would be kind of like putting in logging statements and then just reading the logging statements and aggregating them. What's wrong with that approach of profiling?

[0:18:27.6] JE: That's a great. So I think there's nothing – I don't think that there's anything wrong with any approach that works for you in some sense. If you've got a log statement that's like, "I started making this network last," and then you put it in other log statement after which says like, "Oh, I stopped," and you can see that there's like two seconds between the start and the stop, then you know that they take two seconds and that's too long and that's fine.

There are systems that help you do this even better. So there's these like distributed tracing, like request tracing systems, like Zipkin or Light Staff or – There are a bunch of others that will make it even easier for you to gather this kind of data of, like this block of code took this amount of time. Yeah, I don't think that there's anything wrong with that, and this can be an extremely useful way of debugging.

I do think that like the challenge with that is having to edit the code to add new statements, to add new things that you're wondering especially if you're putting in print statements in production. I have foot imprint statements in production to debug things and it works. I have gotten some good results with that, but it takes a long time. You need to have the statements. You need to get it code reviewed. You need to play it, and then maybe you'd put the wrong ones and that wasn't really the problems. You should take them out and add different ones. I think it can be unpleasant to have to resort to that and it's easier if you can have something which you put in your program, like maybe a tracing system or like a profiler where you don't have to change the program, much less you hope potentially get the same information.

[0:19:52.7] JM: I guess what I should have said is there are tradeoffs to doing that. To looking at profiling as something where you are putting the code inside of your program and your program now has profiler code within it, the approach you took for designing your profiler was to move the profiling program to a separate process that would introspect the state of the running program. Why is that an advantageous approach? Maybe what are some advantages to looking at the program from the outside in?

[0:20:31.1] JE: I think the main advantage is really that usability and like the ability to get started really easily. If you have a program which is slow right now and it's like using all your CPU and you're like, "What's going on?" It's time-consuming to have to say like, "Okay. I'm going to go now include a gem," like the stackprof gem, which is a great Ruby profiling gem. So you need to like include a new profiling library and you need to start profiling, then you need to gather the data, then you need to analyze the data. I think if you've never done that before, it can be a lot of steps. What I see is a lot of the times people just won't profile their programs, because they're like, "Oh, this is too many steps. I don't have time for this." It's like maybe not that big of a performance issue, but it's not worth solving. So people will end up just not fixing performance issues because there are too many steps involved to profiling their programs. I wanted to make something which you could use immediately right away and start getting information about what was happening with your program.

[0:21:27.0] JM: To take a step back, why did you decide to build a profiler for Ruby in the first place?

[0:21:32.3] JE: So I'm kind of obsessed with like debugging and profiling tools, I think. I learned about 5 years ago, I learned about strace, which is this tool that you can use to see what system calls your program is using and I was like, "Wow!" I can see every file my program is opening. I can know what my programs are doing. I didn't know that I had access to all these information, and I think I developed this like aggressive sense of entitlement to information on what my programs are doing. So I kind of noticed that when I had a Ruby program and it was slow, I couldn't just find out why and I felt like kind of grumpy about that. I was like, "I should be able to know why," right? I have a right to know what my program is slow. It's my computer. This is my program. I'm like rude, right? I should be able to know. Why can't I know? So I wanted to write something to fix that problem.

[0:22:22.0] JM: And when you look at this design problem from a high level, do you want to figure out what is going on in that program? Tell me about some of the high level design decisions that you were going to have to make in designing a profiler.

[0:22:34.8] JE: Yeah. So I can talk about maybe how this is possible and how I got started building this, I think. The main thing I needed to do, I wanted to do, is I wanted it to be a separate program. Like you have a Ruby program that's running and I want to be able to start a different program that tells me what it's doing. But then like how do you do that, right?

The way I got started with this was this guy who works at Shopify called Scott Francis, and he had this script that used GDB, which is a C debugger and it would tell him what his Ruby programs were doing. Because the Ruby interpreter is C program and GDB is a C program debugger. So you can use a C debugger to figure out what the Ruby interpreter is doing by like looking at its internals. So basically I found out that that like he [inaudible 0:23:20.3] was explaining how to do it with GDB and I was like, "Oh! Cool. This is possible," and that's the approach I started with.

[0:22:53.7] JM: Yeah. I mean, it's worth pointing out here. Ruby is an interpreted language. If I understand correctly, you have a C program running that is interpreting your Ruby code on-the-fly and turning it into in-memory structures that the C program itself can interface with. Is that an accurate description?

[0:23:46.5] JE: Yeah. That's exactly right.

[0:23:48.5] JM: Okay. So describe in a little more detail. How does the Ruby interpreter, which again, is a C program. How does the C program, the Ruby interpreter interact with the Ruby code that is executing?

[0:23:59.2] JE: Right. So I think from a profiling perspective, I think the most useful thing to understand is that you have two stacks inside of the Ruby interpreter. You have the C stack, which is the stack of C functions that the Ruby interpreter is currently running. If you use like C profilers on the Ruby interpreter, that's what you'll see. It will be like, "Oh!" You'll be like, "What is Ruby doing?" and it's like it's Ruby code. It will be like in the VM execute function or whatever. You're like, "It's not that useful," right? Of course, it's running Ruby code. There's that stack.

Then there's the Ruby stack, which is in a different place in the Ruby interpreter's memory, which has like one Ruby stack frame, which is just like a C data structure inside the Ruby interpreter for every function on your Ruby call stack.

[0:24:43.8] JM: This is the information that you wanted to get at. You wanted to get the specific information for what was going on in your program, which is in C code or C structures that your Ruby interpreter had broken Ruby code down into.

[0:25:02.8] JE: Right. Yeah. This is an interesting thing, because it kind of feels like it might be impossible.

[0:25:08.5] JM: Or unsafe.

[0:25:09.4] JE: Or unsafe. Certainly unsafe. Right. It feels like impossible and unsafe. The Ruby interpreter doesn't make it that easy for you to get this information. It doesn't provide like a public API for you to just like reach into its memory and figure out what it's doing. So it might seem like it's impossible to do this. I think like people have traditionally thought of like dynamic language profiling as kind of a hard problem. It's easy to profile static languages and it's hard to profile dynamic languages, but it turns out that it's actually possible to do this.

[0:25:39.2] JM: Well, why is that? Why is that an assumption that people make that static languages are easier to introspect than dynamic languages?

[0:25:48.5] JE: I guess because – It seems like you have to cooperation from the interpreter, and I think a lot of interpreters aren't built with a view towards making them easy to profile. I think that's why. Yeah, because the interpreter kind of has to tell you in some way what it's doing, right? If the interpreter isn't designed to make it easy to find out what it's doing, then how are you going to find out?

[0:26:11.1] JM: Well, through your strategy apparently.

[0:26:14.5] JE: Yeah, through a lot of hacks.

[0:26:16.8] JM: Through a lot of hacks, and you talked about this in this presentation that I'll put in the show notes, but you've got a Ruby program that's running and you decide to profile it. So that means you have to find this program in memory and find what it is doing. How do you do that?

[0:26:32.0] JE: Yeah. So I have this code for the Ruby interpreter on my laptop, like I did git clone, github.com/ruby/ruby. So what that means is that like I know in principle everything about how the Ruby interpreter works, and I know everything about how the Ruby interpreter works in principle for every Ruby version, right? If I have like a Ruby 2.2 program running on my laptop, I can see all of its code and I can see in particular – So you have – You talk about how you have the Ruby stack in memory, right? So you have these stack frames, and for Ruby this stack is like this continous region of memory. I want to say like – I'm making this up, but like something like maybe 500 bytes per stack frame.

So if you can find the location of the stack, which you can relatively easily. An important thing to know is that it is possible to read memory out of a process. Like you can read arbitrary memory from another process on Linux and on Mac as long as your root, which is I think surprising. I didn't know that you can do this before I started this project. I mean, you can.

So as long as you can find out where in the process is memory, the stack is, you can read the stack out of the process in memory, right? Then you have some bytes which represent the current Ruby stack. The "only" thing you have to do now is figure out what all of those bytes mean and how they correspond to actual Ruby functions.

So the way you do that – We have these bytes which represent a stack frame entry, and those bytes correspond to a C struct, which is in the Ruby interpreter. We know the definition of that C struct because it's in the Ruby source code. So it's called – What is it? Like Ruby control frame T is like the type of one of these structs on the stack. Basically, what you can do is you can be like, "Well, I know –" And if you have the source code for a C struct, it almost gets represented in memory the same way. That's how C compilers work. It's important that this is true, because otherwise dynamic linking won't work, which we can maybe go back to later. But anyway, C structs almost like are represented in memory in the same way. If we have the source code, we

can just say like, “Okay. I know how to find the 4th field of the struct and I know like what it’s going to do because I have its source code.”

[0:28:39.8] JM: Okay. Let me see if I’ve got this correct. So the Ruby interpreter is a C program. It’s running and it’s looking at your Ruby code and turning that into C structs. The Ruby interpreter has two stacks. One stack is the execution of the Ruby interpreter itself. So that is like a stack of just C code that’s looking at your Ruby code and then it has another stack that is the representation of your Ruby code as it’s executing. Although it’s been turned into C structs so that the C program can perform actions on those C structs that represent your program. As you’ve said, on Linux, you can read any processes memory, which is important, because this is what you’re going to do. You’re looking at this Ruby interpreter’s memory and you’re looking specifically at the stack that is the Ruby interpreter side of things. Your Ruby code executing as time goes on in C code, in the C structs, and do I have it right so far? Is that right?

[0:29:40.3] JE: Yeah.

[0:29:40.5] JM: Okay. So you pointed out, this is really important that in Linux you can read any processes memory, because otherwise you wouldn’t be able to do any of these that we just mentioned. Although I wonder, so if I’m looking at this memory as my – And this is an in memory section where my program is executing, isn’t this just like changing all the time? Because programs like change pretty fast because they’re executing.

[0:30:05.2] JE: Yeah. So this I a great point. I think I still haven’t completely resolved this. There are basically two ways to approach the fact that programs are changing all the time. One thing that you can do is when you profile a program, you can stop it and be like, “Okay. Stop.” Going to figure out what you’re doing and then I’m going to start it again. This is the way – There’s a profiler that works very similarly to have my profiler works with Python. It’s called Pyflame. It’s really great. If you’re interested in Python profilers, you should definitely check it out. That’s what it does. It will like stop your Python program, figure out what it’s doing and then start it again.

My [inaudible 0:30:35.0] currently does not do this. It basically just like gets into a race and it’s like, “Okay. I bet I can figure out what your stack is faster than like contingent.” Basically, the bet

it's making is that Ruby is not that fast and that it's faster. This seems to work out most of the time. I think I'm going to add an option to stop the program, because I think it doesn't totally workout all the time. But in general, I think Ruby and Python programs don't change the stack that often compared to like how fast you can run a profiler that's written in like a much faster language.

[0:31:05.4] JM: We should also point out the application that we're talking about here is a profiler. So you don't need to be 100% accurate. You're just trying to get enough of a sample to understand where your program is spending time, right?

[0:31:19.6] JE: Yeah. That's a really important point. If you sometimes get it a little bit wrong or if you like drop a few samples, you're trying to get just like a statistical idea of what your program is doing. So you don't need to be perfect, which is very important, because it's not perfect.

[0:31:33.3] JM: So my program has a thread in memory that is executing right now and my C interpreter has translated that code into C structs and we can introspect it for the reasons that you just mentioned. What can we do with that information? Now that we've got these C structs that represent the threads of our Ruby code, what can we do with that information?

[0:31:58.3] JE: Right. So basically what we end up doing is we end up – We actually need to follow a lot of pointers. The struct that you get represents like one stack frame and then there are pointers from that stack frame which eventually get you to like the current function name and line number, which are like somewhere else in the program summary. The details of that aren't super interesting. So once you followed all of these and collected all the information end up with a stack trace, which has like all of the functions that are currently running. What I do with that is generate basically really good visualizations that help you figure out what your program is doing.

The main visualization that my profiler generate is a flame graph which is this visualization – Is this answering your question also?

[0:32:41.8] JM: It is. Just to point something out, what was really educational about looking at this profiler that you built, I'm not particular expert in profiling programs. I don't have a strong affinity for debugging such as you do. But it made me think more deeply. I didn't know what is it mean and how does an interpreter work before I started thinking about this, and then as I watched your presentation and read some of your material, like, "Oh! I understood. Okay. This is how a program actually executes. It's sitting in memory. It's actually a stack trace in memory."

Like I think of a stack trace, my experience with stack trace is you hit a bug and then your program emits a stack trace and it's like the post mortem on what went wrong, but now what I realize is a stack trace is something that exists in memory, because your program is executing and it needs to keep that entire stack trace in memory as it's executing.

[0:33:34.9] JE: That's right, yeah, and it's a really important point about like program execution and how it works, because yeah, like you said, it's a real thing, and every time you call a function, you add a new stack frame on to your stack. Then when you return, that stack frame like leaves the stack and then you go back to the previous stack frame.

Okay. This is sort of a tangent, but I think it's cool. If you think about recursive functions and like why recursive functions work – For example, if you have a tree and you're running a recursive function to traverse a tree. I think the reason – If you write a non-recursive version of that function, then you need to basically like have a stack where you keep track of where you are. The reason recursive functions work is that you can use like basically your program stack as a data structure to track your progress with the program, right? It's like not like magic. It's actually just like you have a data structure that you're using, but your data structure is your program stack.

[0:34:25.4] JM: Yeah. No, absolutely. Okay, to return to profilers. You've got these in memory C data structures. You've got stack traces sitting in memory. You've got a stack trace sitting in memory that is your program. As it's executing, it's changing all the time because your program is executing, but the data structure that describes the state of your program at any given time is an in memory C data structure. Unfortunately, it's not really in a format that's easy to read. So you want to look at this in memory C data structure and what do you need to do with it to actually extract useful information?

[0:35:00.8] JE: Right. The main thing that you need to do is you need to be able to like take a C struct definition and like figure out what the bytes. Kind of like interpret the bytes according to that C struct definition, right? I think this is an interesting thing, because it's like the easiest programming language to do since – And sometimes it's like C or C++, right? I don't know C or C++, so I didn't use either of those languages. It's also easy to do this in Rust, which is what I did use. But it's like pretty hard to do this kind of thing in Ruby, right? Which is why my profiler is not written in Ruby, because extracting memory from a program and like interpreting it as like a specific C data structure and then like figuring out what pointer that's pointing to and then like extracting more memory from a program and doing all that very quickly is like an extremely – You could probably technically do this in review, but it would be extremely difficult and it's not like the kind of thing Ruby is well-suited for.

[0:35:50.1] JM: What was Rust a better fit for that?

[0:35:51.3] JE: Basically, because Rust has really good support for reading C data structures then for like representing them. There's this program for Rust called bindgen where you can take any C header file and create basically like Rust findings, which basically just like makes those C data structures into like Rust data structures, which will be represented in the same way.

So I could just say like, "Hey, read this thing." I could make bindgen automatically generate findings for like every struct that I was interested in in Ruby, and I could do it. I needed to work with 30 different review versions, so I like ran bindgen on like Ruby's internals for 30 different versions and then save like 30 Rust files for every single version, because of course Ruby changes constantly and I was dealing with like unstable internal Ruby data structures. Then once I had all of that, I could just say like, "Okay, I need the Ruby RB control frame T struct and I need it for Ruby 2.3," and like I know what it is and then just like interpret these bytes as if they were from that data structure and it would all just work really easily and transparently, and it wasn't a circle. It was just actually pretty simple, which is I think surprising.

[SPONSOR MESSAGE]

[0:37:01.6] JM: Users have come to expect real-time. They crave alerts that their payment is received. They crave little cars zooming around on the map. They crave locking their doors at home when they're not at home. There's no need to reinvent the wheel when it comes to making your app real-time. PubNub makes it simple, enabling you to build immersive and interactive experiences on the web, on mobile phones, embedded in the hardware and any other device connected to the internet.

With powerful APIs and a robust global infrastructure, you can stream geo-location data, you can send chat messages, you can turn on sprinklers, or you can rock your baby's crib when they start crying. PubNub literally powers IoT cribs.

70 SDKs for web, mobile, IoT, and more means that you can start streaming data in real-time without a ton of compatibility headaches, and no need to build your own SDKs from scratch. Lastly, PubNub includes a ton of other real-time features beyond real-time messaging, like presence for online or offline detection, and Access manager to thwart trolls and hackers.

Go to pubnub.com/sedaily to get started. They offer a generous sandbox tier that’s free forever until your app takes off, that is. Pubnub.com/sedaily. That's pubnub.com/sedaily. Thank you, PubNub for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:38:45.1] JM: And none of these is very performance-intensive to just understand what’s going on in the program, right? This is not causing a performance hit to the system itself while you’re doing the profiling.

[0:38:56.7] JE: No. It uses a little bit of CPU. I mean, it uses more CPU depending on basically how deep your stack is, right? If you have a stack, which is like a thousand things, it takes actually non-trivial amount of time to read all the memory out of your target program. But if your program doesn’t have a very deep stack, it’s not that big of a deal.

I guess what did I see? If you have a lot of things in your stack, like maybe a hundred or something, then I could easily sample a hundred times a second, and then if there are not too

many things in your stack, then it's easy to sample like a thousand times a second, which is definitely often enough for most Ruby programs.

[0:39:28.4] JM: I think we've done a pretty good job of breaking down what your profiler consists of. I think we've talked about the tool chain in some detail. So you said, I think, GDB, which is a C debugger. You use GDB in order to get the dump of the Ruby interpreter at any given time. Was that correct?

[0:39:47.0] JE: No, actually. This is an interesting point. I use GDB to learn how to do it. GDB is able to do this and I kind of just like figured out what GDB does, and then I didn't use GDB after that.

[0:39:59.6] JM: Oh, okay.

[0:40:00.3] JE: I just didn't use GDB as a learning tool.

[0:40:02.5] JM: I got it.

[0:40:03.3] JE: Actually, I don't use – GDB uses a system call called Ptrace, and I don't use the Ptrace system call. I use the process VM read V system call, but they do similar things, but Ptrace will stop your program and process VM read v will not by itself.

[0:40:16.8] JM: Okay.

[0:40:17.8] JE: And Ptrace is much more complicated.

[0:40:19.4] JM: You mentioned that you used a library called bindgen within Rust to turn the C data structures into Rust data structures. Are there any other tools that you used in building your profiler that are worth pointing out?

[0:40:34.7] JE: I don't think so.

[0:40:35.5] JM: Okay.

[0:40:36.5] JE: No. Those are the main ones.

[0:40:37.4] JM: Okay. The profiler generates flame graphs. Explain what a flame graph is.

[0:40:42.3] JE: Okay. I'll try to do with my voice, but I'll try. Yeah, it's hard to do that looking at one. Basically it's a visualization invented by Brendan Gregg who's this like profiler wizard, and it's a way to visualize easily where your program is spending its time. Basically, what you'll end up seeing is like you'll have like something like really large white areas on your flame graph and like the large areas are where your program is spending its time. I think it's my favorite way to visualize profiling information.

We're talking about this like self-time, total time representation from before, but that doesn't let you see the whole stack and I think it can be hard to figure out what's going, and I found a flame graph to be a more useful tool than just like the self-time and total time representation. The nice thing is that you can use flame graphs from profiling tools from lots of different programming languages. It's just like a pearl script, which takes a really simple input format and then outputs an SVG. So it's easy to incorporate in like any of your profiling workflows.

[0:41:37.2] JM: Yeah, I think of it like a histogram that tells you the different amounts of time that your program is spending in nested calls within that program so that you can really break down, like you said, the self-time versus the total time, except you get a finer granularity of what that total time consists of.

[0:41:59.5] JE: Yeah. It's just like a much richer representation. Something really interesting happened when I use flame graphs in Rbspy, which is that one person commented on my like testimonials page that I have in GitHub and they were like, "Oh, I don't know. I've never heard of flame graphs before and I didn't know you could use them." But because Rbspy by default, I learned about. Other Ruby profilers also support flame graphs, but I think they just didn't realize that it was an option. So I think there's something really powerful about like generating a rich or powerful visualization by default without making people go through extra steps to do it.

[0:42:35.9] JM: That's a good point. I mean, I think of this as a design decision in your "product". I mean, you could have designed your profiler to not generate visualizations, but you just built in visualizations by default, which is I think a great design decision.

[0:42:51.9] JE: Yeah. So the pearl script I used – Normally what you'll say is that like, "Oh, I can clone this git repository and then you can put it in your path and you can make it work." Right? Which of course you can do. But I check the license and I realize that I could just take the pearl script and compile it into my binary so that it just like comes with rbspy and you don't need to get an extra git repository of like add an integration. It's just in the binary and it just works.

[0:43:17.9] JM: Another great developer ergonomic decision. So working with profilers, are there instances where my profiler can lie to me or give me some kind of misleading information?

[0:43:30.5] JE: Can your profiler lie to you? Well, okay. One really important thing to know about CPU profilers is there are kind of two ways to measure your program's time. You can measure like how much time it's spending on the CPU and you can measure how much time it's spending in total. This is like a different notion of total than like the self-time versus total time, right?

For example, if your program is waiting for [inaudible 0:43:52.5] request. It's not using the CPU at all, but it is still waiting for something. Rbspy actually right now only supports total time and it doesn't support just like figuring out how much CPU time your program is using, which is like something I'm working on. But I think it's really important to understand the distinction, because otherwise you can end up being misled.

[0:44:11.9] JM: Would you've done anything different if you were building the profiler from scratch today?

[0:44:15.7] JE: I can't think of anything yet. I built it pretty recently, so I think maybe I haven't learned that much. I don't know if I've learned enough yet.

[0:44:22.6] JM: What are you working on now within the scope of rbspy? Are you still adding features to it or improving performance in any way?

[0:44:31.6] JE: Adding features to it extremely slowly, because I built it on sabbatical. So I took three months off work to work on it, and now I'm back at work where I am not working on profilers. So it's much harder to find the time. But one thing I want to add, I have something like prototype code to do and I just need to get it working, get it merged, is figuring out when the program is doing garbage collection. So it's like you're spending like 12% of your time in GC, because I know that that can be an issue in a lot of Ruby programs.

[0:44:57.0] JM: Oh, yeah. Have you worked on optimizing code that is spending too much time in GC? This is kind of adjunct to the discussion of profilers. I've heard discussions of profilers in the Java world where they're often times they are talking about this very issue. How do you make your program more efficient in terms of garbage collection?

[0:45:17.6] JE: Yeah, I haven't worked on that a lot. I know Aaron Patterson has worked on it a lot and he's actually I think working on making the Ruby GC faster.

[0:45:25.3] JM: Okay. I feel like we touched on this a little bit, but do you have any insights for why or how dynamic language is compared to statically typed languages in this realm of performance and introspection?

[0:45:38.4] JE: Yeah. So this is a really interesting question, because if you look at Java's profiling tools, you have these profilers, like YourKit, which is this commercial profiler for Java, which is like outrageously powerful, right? You can attach to a Java program and you can find out like anything about what it's doing. You can see what – Doing in GC. You can see where it's spending its time and you can start like tracing every function call. You can do like kind of anything, right? If you look at C profilers, you see kind of the same thing where you have like perf, which is this extremely powerful C profiler that's part of Linux kernel, and you have like a lot more profiling and debugging tools. For dynamic languages, you kind of don't have this.

I hear people say a lot like, "Oh, it's because people don't care about performance," right? I think the joke people think is like if you cared about performance, you wouldn't be using Ruby. I guess it's true to some extent, but of course people do care about performance just a little bit at

least, because if your program – If your HTTP request are taking 30 seconds. Of course, you still need to know why, right?

[0:46:32.6] JM: Or they didn't care about programming performance when they were writing their MVP, and then the program took off and they're like, "Okay. Now, we have a lot of users and performance actually matters." It's not like let's support the entire program to Java to solve our performance issues.

[0:46:46.2] JE: Yeah. Often, that's not realistic, and often it doesn't makes sense, right? It's often enough just to make Ruby program a little bit faster. So I'm not really sure why there's like such a difference in how much investment there is. I feel like maybe one reason for it – I'm really speculating here. Is that to write a profiler, I think you need to maybe think more about like systems concerns, maybe you need to know C. You need to understand a little bit about how like the Ruby interpreter is implemented.

Of course, like C programmers are more likely to know C than Ruby programmers, because they're C programmers. So it's like if you're working in one of these like faster languages, maybe you're like more likely to have the skills that you need to write a profiler than if you're working in a dynamic language. I mean, I'm not sure.

[0:47:27.7] JM: All right. Well, not a problem. I know you're not – You haven't spent your time delving into the differences between static and dynamic languages. So that's certainly an opportunity for another show.

[0:47:39.0] JE: Yeah. I do think that is like a cultural question rather than a technical question. Like the question is like why are people like not spending their time and money on developing these profilers and not like why is this not possible.

[0:47:49.9] JM: Right. So it's not a reason like; well, dynamic languages, you don't know the type until the execution, and therefore it's harder to introspect the profiling characteristics, a program that's running in that language. No. That's not why this doesn't exist. It's probably because people are writing Java, high-frequency trading systems and machine learning training algorithms and these are really high throughput, big data, map reduce and low latency sensitive

operations. So it's really important to profile these things. Whereas in at least the reputation is Ruby is like for Ruby on Rails applications and it's like Airbnb is making a request to schedule a room, and it's like, "That's not a performance-sensitive issue."

[0:48:38.5] JE: Right. Though of course it is a performance-sensitive because web latency is like very important, and if your program is – If your website is slow, then you lose money. Of course, it actually matters, and there are businesses that are [inaudible 0:48:50.1], etc., that like do sell this kind of thing, right? But I think the tools are just a lot less powerful.

I feel like there's maybe a question of like maybe what people's expectations are. I think maybe in Java, people have higher expectations about like what they expect, like what they think you should be able to get out of a profiler than in Ruby. I don't know how many of Ruby programmers are also Java programmers and have had experience optimizing Java programs.

[0:49:16.4] JM: Yeah, also, isn't like when you get this performance sensitivity, you just start using JRuby, and so you can use Java profilers perhaps?

[0:49:24.5] JE: That's possible. I don't have any experience with JRuby. This is a very interesting question.

[0:49:29.0] JM: Okay. Well, Julia, it's been really great talking to you. I've enjoyed this conversation a lot.

[0:49:32.4] JE: Thank you so much for having me. This has been the best.

[0:49:34.6] JM: Okay. Awesome.

[END OF INTERVIEW]

[0:49:39.1] JM: If you are building a product for software engineers or you are hiring software engineers, Software Engineering Daily is accepting sponsorships for 2018. Send me an email, jeff@softwareengineeringdaily.com if you're interested. With 23,000 people listening Monday

through Friday and the content being fairly selective for a technical listener, Software Engineering Daily is a great way to reach top engineers.

I know that the listeners of Software Engineering Daily are great engineers because I talk to them all the time. I hear from CTOs, CEOs, directors of engineering who listen to the show regularly. I also hear about many, newer, hungry software engineers who are looking to level up quickly and prove themselves. To find out more about sponsoring the show, you can send me an email or tell your marketing director to send me an email, jeff@softwareengineeringdaily.com.

If you're a listener to the show, thank you so much for supporting it through your audienceship. That is quite enough, but if you're interested in taking your support of the show to the next level, then look at sponsoring the show through your company. Send me an email at jeff@softwareengineeringdaily.com. Thank you.

[END]

