EPISODE 680

[INTRODUCTION]

[0:00:00] JM: JavaScript performance has improved overtime due to advances in JavaScript engines, such as Google V8. A JavaScript engine performs compiler optimization, garbage collection, hot code management, caching and other runtime aspects that keep a JavaScript program running efficiently.

JavaScript runs in browsers and servers. The resources that are available to a JavaScript engine can vary widely across different machines. JavaScript code is parsed into an abstract syntax tree before being handed off to the compiler toolchain, in which one or more optimizing compilers produce efficient low-level code.

In recent episodes about WebAssembly, we have covered compiler pipelines. In an episode about GraalVM, we explored the impact that code shape has on the efficiency of JavaScript execution. Mathias Bynens is a developer advocate at Google working on the V8 JavaScript engine team. In today‘s show, we explore how a JavaScript engine works and how compiler toolchains can adapt the hot code paths depending on what code needs to be optimized for. I really enjoyed talking to Mathias and I look forward to doing more shows on JavaScript infrastructure.

Before we get started, I want to mention that we are looking for writers. We are making a big push towards written content on the site. So you can apply at softwareengineeringdaily.com/jobs. We’re also looking for podcasters potentially. We have really high standards for podcasters, but we also have a job posting there. So please do apply.

[SPONSOR MESSAGE]

[00:01:51] JM: Your audience is most likely global. Your customers are everywhere. They’re in different countries speaking different languages. For your product or service to reach these new markets, you'll need a reliable solution to localize your digital content quickly. Transifex is a

SaaS based localization and translation platform that easily integrates with your Agile development process.

Your software, your websites, your games, apps, video subtitles and more can all be translated with Transifex. You can use Transifex with in-house translation teams, language service providers. You can even crowd source your translations. If you're a developer who is ready to reach a global audience, check out Transifex. You can visit transifex.com/sedaily and sign up for a free 15-day trial.

With Transifex, source content and translations are automatically synced to a global content repository that's accessible at any time. Translators work on live content within the development cycle, eliminating the need for freezes or batched translations. Whether you are translating a website, a game, a mobile app or even video subtitles, Transifex gives developers the powerful tools needed to manage the software localization process.

Sign up for a free 15 day trial and support Software Engineering Daily by going to transifex.com/sedaily. That's transifex.com/sedaily.

[INTERVIEW]

[00:03:39] JM: Mathias Bynens, you are a software engineer at Google working on developer advocacy on the V8 team. Welcome to Software Engineering Daily.

[00:03:48] MB: Yeah, thanks for having me. It’s great to be here.

[00:03:50] JM: I’m looking forward to talking about some aspects of modern JavaScript, and JavaScript has a lot of applications, a lot of facets to it, though the one that people are probably most familiar with is they go to a webpage, they open the webpage and that webpage has some JavaScript on it, and it might also have some tags that import JavaScript, and all of these ultimately turns into code that executes on my machine.

Give me a high-level perspective for the steps that are happening as this webpage with some various forms of JavaScript opens up.

[00:04:26] MB: Okay. Do you have a day or two? Because we can go over everything. No. You know what? I'm going to skip over the part where the browser parses the HTML and actually loads the webpage from the server and loads the scripts as well. I'm going to skip over that, because it's not my area of focus. We're going to go straight into what happens as soon as the browser has the JavaScript and then tries to run it.

So the first thing that happens is the JavaScript source code goes into the JavaScript engine sparser, which then takes that code and turns into an object-based representation, which we call an abstract syntax tree or an AST. Armed with AST, we can do some transformations on the code, do some [inaudible 00:05:06] here and there, but mainly the reason to have this AST is so that we can feed it into the first phase of execution, which is the interpreter.

So our interpreter in V8, it's called Ignition. This interpreter generates bytecode, which then runs in the interpreter. At that point we're actually running the JavaScript already, which is great. It's like all we have to do is sparse it and then generate some bytecode in the interpreter, which can happen pretty quickly. So that's great for a code that only needs to run once. For example, a lot of code on the web is code that initializes the webpage and you don't run that in a loop very open, right? You only run it once. You load the page. So all that magic can happen in the interpreter with the bytecode that we generate.

But that's not the end of the story, because sometimes code gets repeated quite often or we see that the same code paths get hit over and over and over again, and that point it might make sense to start optimizing those specific functions, right?

That's why while we're running the bytecode in the interpreter, JavaScript engines – Well, V8 does this, but all JavaScript engines basically do this. They actually collect feedback while they're running. This means that when we see a specific function getting called many times with the same kinds of arguments, like always with numbers, for example, the JavaScript engine will remember that and then we can try to create optimized machine codes in the optimized compiler that is specifically optimized for those cases that we've seen. In this example, it would be for the number case. We have an interpreter, and then the next level is the optimizing

compiler, which produces this highly optimized machine code that can run directly on the CPU. That's it.

[00:06:46] JM: The steps are, first, you have JavaScript gets – Some of it is on the webpage, some of it gets importer. But you have JavaScript, and it gets parsed into an abstract syntax tree. Then the abstract syntax tree is interpreter into bytecode. Then the bytecode gets turned into, gets compiled into machine code. Then depending on how frequently that code is getting called, for example, if it's in a loop that's getting called every 100 milliseconds and you're just sitting there on the webpage, then your compiler is going to realize, "Okay. We're taking this bytecode and we're turning it into machine code over and over and over again. We should probably just leave it in machine code and re-execute it?" Am I understanding it correctly?

[00:07:33] MB: Something like that, yeah. The important part is that the optimizing compiler only really kicks in when the JavaScript engine detects that particular function is what we call hot. Hot code is code that becomes a bubble, like for performance. Code that gets repeated very often or code that is ran in a tight loop, for example. It really depends on what kind of code you're running.

On an average webpage, for example, you will have lots of code that is used to initialize the page, and you only really want to run that once. In that case, it makes sense to run it in the interpreter. We produce bytecode, which can happen pretty quickly. We can produce bytecode quite fast, and then we only have to run it once. So we just run it in the interpreter, and that's it. We're done with it. That's the main benefit of the interpreter. It can actually get you running quite quickly. It is very quick at producing some runnable code.

[00:08:23] JM: To be clear, that JavaScript bytecode that comes out of the interpreter, that still has to be compiled at some point into a machine code, right?

[00:08:30] MB: Yeah. Yeah, but it's being interpreted. He bytecode runs in the interpreter. The virtual machine, the JavaScript engine doesn't directly turn it into machine code. It just runs in the interpreter itself. As a result – The bytecode that we can produce is pretty small. It doesn't consume a lot of memory, because every JavaScript engine just makes up its own bytecode basically. There is no standardized bytecode format for JavaScript. So JavaScript engines parse

it into their own abstract syntax tree format, and then they feed it to the interpreter. Generate some kind of bytecode, and then at that point they can start running this.

For some code, that's where it ends. It never leaves the interpreter. It only runs once and that's fine. For those specific functions that we detect are hot, we take that bytecode that we have from the interpreter, and together with this profiling data that we collected while we were running the code, like we saw, "Oh! This function keeps getting called with two number arguments." So we're going to optimize specifically for that case. Then the optimized compiler produces a highly optimized machine code.

The difference between the bytecode and the optimized machine code, is that this optimized machine code can run directly on the CPU. These are direct CPU instructions that can run. Whereas the bytecode still runs in the JavaScript in the interpreter itself. The difference is that the interpreter code, the runtime performance is a little bit slower. It is kind of slow, but the benefit of the interpreter is that it is very quick to get something running. It's quick to generate code, but executing the bytecode is not the most efficient. The optimizing compiler is the other way around. It takes a little bit longer to produce this optimized machine code and to do all the optimizations that we want and to produce a machine code.

But once we have it, we can run that optimized machine code very efficiently. You can see there's a big of a tradeoff between you have to be very careful at selecting what code and what functions you optimize, because if you optimize run functions or if you try to optimize everything, then nothing will be fast, because it takes a long time to produce that code.

[00:10:33] JM: This is ahead of time compilation versus just in time compilation.

[00:10:38] MB: Yeah, pretty much. Yeah. Yeah, with the interpreter, you produce a bytecode right away as soon as you get to the abstract syntax tree. Then the optimizing compiler, that's what it's called, just in time compilation, so JIT.

[00:10:50] JM: Can you zoom in – Because something that's been confusing me for a wile is can you zoom in on the – When the interpreter is going to run bytecode, what's going on there? So I understand that in the path where the code is hot and it goes through the optimizing

compiler, and the optimizing compiler takes the bytecode and goes in and turns it into a machine code. The interpreter, doesn't that also have to turn bytecode into machine code, but I guess it's just less optimized?

[00:11:19] MB: Right. Yeah. So eventually everything becomes machine code, right? But the thing is, with the interpreter and the bytecode that runs in the interpreter, we do not control the machine code that it gets produced directly. You could say that the bytecode is running in the JavaScript's engines interpreter. Whereas with the optimizing compiler, we actually look at your architecture and the platform that you're using and we produce only those exact CPU instructions that you need to get this thing to run. So it's much more low-level, but that means we can also squeeze up all of the performance that we can. But it takes longer for us to get there.

[00:11:57] JM: Yeah.

[00:11:58] MB: Does that make sense?

[00:12:00] JM: It does. I guess the thing that I have had – And maybe I'm just not well-educated on this, and maybe you don't know in detail what goes on here, but I'm really curious about the transition from that bytecode to machine code. What kind of application is responsible for doing that? For making that execution happen? What does that kind of look like? What does that process look like?

[00:12:23] MB: Okay. The interpreter in V8 is called Ignition, and that's what produces the bytecode, and the bytecode then runs in this interpreter. The optimizing compiler, I haven't mentioned. The name is turbofan, for what we have in V8. That's the name for optimizing compiler.

So the transition goes – Wow! This is difficult to explain without being able to visualize it, I guess.

[00:12:45] JM: We can move beyond it. I'm probably nit-picking details. I guess if people want to know the process of bytecode turning into a machine code, they can look it up on the internet.

But talking more abstractly, I think there are a lot of people out there who have heard this term JavaScript engine, and JavaScript engine is a large abstraction. There're a lot of things that a JavaScript engine does. Give me a high-level definition for what a JavaScript engine is and what the responsibilities of a JavaScript engine are.

[00:13:12] MB: Okay. A JavaScript engine takes your JavaScript code and executes it. But more than that, it tries to optimize the hot functions and identify the parts of your code that might become performance bottlenecks and then make those run extra efficiently. Sometimes JavaScript engines are referred to as virtual machines. It's basically all of the same thing. JavaScript engines are processed virtual machines, because they let you execute computer programs in a platform independent environment. Basically, JavaScript is the language that you write your code in and you don't write different JavaScript all of a sudden, because your user that is browsing your website is using a Linux computer, for example. All of the code is being dynamically generates as part of this JavaScript engine. That's why it's called a virtual machine.

[00:14:00] JM: There are different virtual machines for different browsers. So Google has a JavaScript engine, the V8 for Chrome, and Apple has their own, Microsoft has their own, Chakra. Mozilla has their own, SpiderMonkey. Why are there different JavaScript engines?

[00:14:18] MB: I guess if you're maintaining your own browser engine, then it just makes sense to create your own JavaScript engine as well, because you want to a fine-grained control over all the different bottlenecks and all the different tradeoffs. Some of the tradeoffs that we mentioned earlier – For example, just when you look at the JavaScript engine specifically, there's a tradeoff between using an interpreter, because it is pretty quick at generating codes. But running the codes takes a little bit longer. It's not super-efficient at running the code.

But on the other side, if you use the optimized compiler, it takes a lot longer to produce the code and to get something running. But by the time it runs, it runs super-efficiently. So that's one tradeoff. Another tradeoff when it comes to JavaScript engines, which also goes in between interpreters and optimized compilers, is that for interpreted codes, the bytecode that you produce is usually pretty small. It doesn't require a lot of memory. It's just a few instructions in the bytecode basically.

But for optimizing compilers, they produce this highly optimized machine code. But if you could the number of instructions, you'll see that it's a lot larger, like a factor of, let's say, 8, compared to the bytecode. So memory-wise, the memory footprint of this optimized machine code is a lot larger than it is for bytecode that you can just run in the interpreter. That's another tradeoff. Running code in the interpreter is more memory efficient, but producing this optimized machine code requires more memory.

That's why there are actually a lot of differences between all those JavaScript engines that you mentioned, like SpiderMonkey from Mozilla, JavaScriptCore from Apple. Google has V8, and Microsoft has Chakra, and there're other JavaScript engines that are specifically optimized for IoT devices and small, low capable hardware devices as well.

For each of those different use cases, you can imagine that people have different preferences and different kind of scenarios that they want to optimize for. That is why a lot of differences actually take place in this interpreter and optimized compiler pipeline if you start comparing the JavaScript engines.

[00:16:23] JM: What are some of those examples? What would be something that maybe Google would subjectively say, "Oh, we want to optimize for this kind of use case –" Maybe, I don't know, ads or something, or something related to search. We want to be compiling these kinds of hot code paths more aggressively than – Apple might say, "Oh, actually, we don't care about those as much, and maybe we want to compile something related to privacy or something more aggressive like that," or is it not that abstract? What level are these subjective decisions taking place? Does it have a user experience component?

[00:17:03] MB: It depends on what kind of use cases your JavaScript engine has as well, right? V8 is used in Google Chrome as you mentioned, but it's also being used in a lot of other products. It's being embedded in Node.js, for example, which is a project that Google does not control. So Node.js is a very important use case and we try to support it to the best of our abilities.

In fact, last year, we made Node.js a first class citizen in our testing infrastructure alongside Chrome. Node is very important for us, and we do everything we can to support the Node

project when it comes to what we change in V8. For example, we count [00:17:38] new changes in V8. If doing so, would break a Node.js test. We run the entire Node.js test suite for every commit that we land. If it breaks anything, then we can't even land it.

Of course, that also determines kind of the architectural decisions that we make when we design things or we change components in V8, because we don't just want to support the web. We also want to support all these other embedders, including Node.js. As you can imagine, the code that one can write for a website or for a long-running Node.js server can be quite different and can have very different characteristics.

Even those different platforms and those different embedders for JavaScript engines for the site, there are still a lot of tradeoffs to make, like the ones we discussed earlier, and there is no clear one true solution to this, right? In V8, we decided to have a single interpreter, which is Ignition, and we went with a single optimizing compiler, which is TurboFan. But if you look at other JavaScript engines, you'll find that some of them have two or even three optimization tiers with multiple different optimizing compilers. Each of those have different characteristics when it comes to how long it takes for them to produce some machine code and how optimized that machine code can be.

Of course, the more optimization tiers you add to your codebase, that's a tradeoff you have to make in terms of code complexity, and maintenance, and the maintenance cost goes up, of course, the more code that you produce and the more code that you add. There's also some benefits, because you have more find-grained control over how much time you want to spend generating optimized code and how optimized should this code be.

Deploy ML models the smart way and hardware, and head to algorithmia.com to get started and upload your pre-trained models. If you use the code SWEDaily, they will give you 50,000 credits free, which is pretty sweet. That's code SWEDaily.

The expert engineers at Algorithmia are always available to help your team successfully deploy your models to production with the AI layer tooling, and you can also listen to a couple of episodes I've done with the CEO of Algorithmia. If you want to hear more about their pretty sweet set of software platforms, go to algorithmia.com and use the code SWEDaily to try it out or, of course, listen to those episodes.

[INTERVIEW CONTINUED]

[00:21:02] JM: With the example of Node.js, if the applications are so different for a long-running server versus a browser, why wouldn't you fork V8? Why is the JavaScript V8 engine the same one that's used in the browser and in Node.js?

[00:21:18] MB: Yeah. So if you look at V8 from 10 years ago, when the project was first open-sourced, it had a completely different pipeline than what we're looking at today. In fact, we only just launched Ignition and TurboFan, our new interpreter, our new optimizer compiler just last year in Chrome 59 I believe it was. Of course, when we were working on this new pipeline, Node was already a thing. We already knew we wanted to support Node more actively. Yeah, that went into the design of Ignition and TurboFan.

[00:21:47] JM: Okay. Well, let's a little bit more about the specific optimizations that a JavaScript can make. So different JavaScript objects can take on different shapes, and we did a show recently about the GraalVM where we talked about the subject of different code shapes and how that impacts the interpretation and the compilation pipeline of the Java virtual machine. Let's review some of that materials. So what is a code shape? Why is the idea of a code shape important?

[00:22:23] MB: Okay. A shape is something that JavaScript engines use and they attach it to properties in objects. So to simplify an example here, let's say you have a bunch of objects in

your codebase. It's very common to use objects in JavaScript, right? If you look at real-world codes, you'll notice that it's actually fairly common to have objects that have different values in them, but at least some of your objects in your codebase will have the same property keys.

So let's say, a simple example, you have an object. It starts out as an empty object then you add a property X to it, and then you add a property Y to it. Maybe you have not one object like that, but you have hundreds, or thousands of these objects. So the thing is, if two or more objects have the same set of properties attached to them, the values can be different. The values don't matter. But if the same properties are attached to them, you can say that these objects have the same shape. This is something that JavaScript engines use for lots of different optimizations.

Well, the simplest one to explain would be memory optimization. If you are indeed going to assume that you have, let's say, a thousand different objects all with the same shape, all with a property X and some property Y, then you have a choice to make as a JavaScript engine, right? You have to store all this information in this object somewhere. So the naive way to store this would be you create some kind of JavaScript object data structure, you store the property names in there. You store the values of the object in there and you also have to store some other things called the property attributes, which are things like is the property writable? Is it endurable? Does it show up in a for in loop? Is it deletable? Which means is it configurable? So you'll need to store all those things.

If you're going to assume that you're going to see multiple objects with the same shape in a real-world codebases, then it would actually be quite wasteful to duplicate all that information for all those objects, right? All you really need to store for each individual object is the values that are unique to that object. So that's what JavaScript engines do. They separate the shape, which has the information about the property names and whether the object property is endurable, writable and configurable. They store that separately in the shape, and then each object only has to hold the specific values that are unique to that object.

This means that even if you have a thousand or a million of these objects with the same X and Y shape, we don't have to store that shape a million times. We can only store it once, and all these objects then points to that same shape. Hopefully you can see that this actually saves us

a lot of memory, because we only have to store this once. That's the first optimization that shapes enable.

[00:24:59] JM: So how might those shapes vary in different situations and how might that lead to some savings? Could you rephrase, emphasize why that would lead to savings?

[00:25:10] MB: Okay. One thing that JavaScript engines all like to do is use something called inline caches. Inline caches is another optimization that would be impossible without the use of shapes. It's a little hard to explain this without being able to visualize it. I actually recently did a presentation about this with my colleague, Benedikt Meurer as JSConf. I'll make sure I'll get you the link in case people are interested. They can look it up there.

[00:25:32] JM: Sure. Yeah, please.

[00:25:33] MB: There are some visualizations that might make this easier to understand. Basically, let's say you have a function that loads a property off of an object. Maybe the function is called get X, and all it does is it takes an object as its argument and then it returns object .x, right?

The first time we run this code, it's not optimized. We run it the very first time. I told you before that the JavaScript engine will be collecting profiling data and feedback as it's running the code for the first time and for the first couple of times in the interpreter. The JavaScript engine at some point will say, "Okay. I see that this function is being called with an object, and this object has this particular shape."

Now, the JavaScript engine can then remember the shape that is has seen, and to load the actual X property value, it needs to do several lookups. It needs to look at the shape that is associated with the object. It then needs to look at the object and read out the offset, which is also stored in the shape. Then it needs to know where to find the actual value that corresponds to this particular particularly in this object with this particular shape. It needs to perform a bunch of lookups.

The first time – Well, there's no choice but to do these lookups. The JavaScript engine cannot avoid them. But if we keep repeating the same function, if the function becomes hot, then we can actually cache some of those lookups and avoid repeated lookups for the same property in the future. If you can imagine this function, get X, that we talked about before, if this is run a couple of times and it always receives an object with the same shape, then the JavaScript engine can optimize for that. The optimized compiler can generate code specifically optimized for that. It can just return the value that you're looking for directly, bypassing lots of those lookups that would be otherwise expensive. That's the main benefit of having shapes in the first place, it's so you can do this thing, which is called inline cache.

[00:27:29] JM: Are there are some development or improvements underway right now that are being worked on around the area of code shape and inline caching?

[00:27:39] MB: This is one of the fundamentals, I would say, of a lot of JavaScript engines. In fact, this is one of the few things that almost all JavaScript engines have in common. They all implement shapes and ICs, or inline caches in one way or another. Of course, the implementations might vary. Yeah, I don't expect this to change anytime soon.

But there are other things and other optimizations that are a little bit more different in between JavaScript engines. If you want, I can give an example of some of those. Yeah. So for arrays, for example, we spend a lot of time talking about objects, but arrays are also common in JavaScript, and they are treated a little bit different.

For arrays, they behave kind of similarly to other objects, right? Except that it's most common to access index properties. You access the properties on an array by referring to the index, and then you get the value out of them. Conceptually, you can think of them as objects, but JavaScript engines decide to store these special index properties separately for optimization purposes.

In V8, we have some special mechanisms in place, which have also been in place for a long time. We saw some of the fundamentals in the V8 runtime. We call them elements kinds. It's kind of a weird name. Basically, whenever you create an array in your codebase, V8 keeps track of the kinds of values that the array contains and it tries to give the array a label, or a tag if you

will, that says, “Okay, this array contains only integers,” for example, “or this array contains doubles,” which is a more generic collection.

Another thing would be if the array contains not just numbers, but other values as well, like objects or maybe undefined, or strings, then we would call those just regular elements. We have all these different elements kinds available, and whenever people perform any kind of operation on an array, it could be looping on the array with four each, or it could be using map, or reduce, or filter, or any of that stuff. Now, V8 can look at the elements kinds of that particular array and it can use that to optimize specifically for those elements kinds.

That’s why if you have an array that only consists of numbers that behind scenes we know are doubles, or even better, if behind the scenes we know that they’re all small integers, or SMI's as we call them, then we can produce optimized machine code specifically for that case, specifically knowing that we only have to deal with small integers in that case.

[00:30:08] JM: I’d like to take a step back and get an understanding of how a browser, or perhaps a server, in the case of Node.js, is managing memory in JavaScript? I think we’ve talked about a number of different things that could be managed. You’ve got inline caching. There’s probably other kinds of caching to talk about. There’s the code itself, like the hot code. I think of hot code as a kind of cache. Maybe you’re caching code that is going to be executed again in the near future, or you’re hoping it will be.

So there’s these different areas of management of memory that needs to take place in the JavaScript runtime. How are these different areas of memory managed and how does the amount of memory available to a JavaScript process, or a browser, how does that get allocated or governed?

[00:31:06] MB: I can tell you that when it comes to – You want to talk about garbage collection?

[00:31:09] JM: We could talk about garbage collection. I think I’m talking about garbage collection. I’m also talking about just how the quantity of memory, whatever memory is – Because I feel like my browser – So my browser, I feel like if I open tabs, if I open enough tabs, a Gmail, and Google Docs, and Zencaster, these things that consume a lot of memory. At a

certain point, my browser reaches a point where it starts to slowdown and my overall system usage, maybe like 40% or 50%, which is high. If I look at the memory pressure, then I start to see like 40%, 50% and things starts to feel slow on my computer.

But it never gets to the point where my browser is eating up so much memory that my computer slows to a crawl. I guess it was a too big of a question. But I'm really trying to understand how memory is managed both within a browser and between a browser and an operating system.

I guess we could start with the topic of JavaScript, since that's – With the topic of garbage collection, since that's a little more well-scoped. But I'm just trying to give you a picture for, I guess, the expanse of what I'm curious about. So we could talk about garbage collection first.

[00:32:21] MB: Right. Okay. Yeah, one of the problems with JavaScript is that it is single-threaded. Specifically, on the web, if you visit a website, then there is only one main thread, right? That's a thread JavaScript runs on. Also, all these other things that the browser does also need to happen in the same thread, like paint, layout, rendering of your page. That is constantly competing with your JavaScript execution. Ideally, of course, your users want a 60 fps, silky-smooth experience and without any jank. So how do you make that happen?

It's pretty hard already. The thing is garbage collection – I keep thinking about garbage collection, because that really used to make things a lot harder, because garbage collection is something that happens in the JavaScript engine for JavaScript specifically, but also in the browser. As a developer, you don't have direct control over when and how that happens.

Even if you build some kind of web app experience that has good performance, every now and then you could get unlucky when the garbage collector kicks in at the wrong time, and that would cause your frame rate to drop and your user to have a not so smooth experience anymore.

One big thing that happened there over the last couple of years, at least in V8, is that the memory team, who can answer these questions in a lot more detail than I could, they have been working on making our garbage collector almost entirely concurrent. Most of the garbage collection that is happening not does not block the main thread anymore in V8 and in Chrome,

which means that there's less jank. Even when garbage collection kicks in, you won't even notice, because it's happening on a separate draft and it doesn't lock up your browser. It doesn't lock up your web application. It doesn't decrease your frame rate.

[00:34:04] JM: Is that the same for Node and for the browser?

[00:34:08] MB: Yeah, we were talking about the differences between the web and Node.js and writing JavaScript for the web, versus writing JavaScript for Node. There's a big difference depending on what you're doing, right? Of course, even just the Node ecosystem is huge and there're all these different use cases. You could write a command line utility in JavaScript that runs in Node, but you can also write a server that might have an uptime of multiple months. That is a very long-running JavaScript application, which if you compare it to a command line utility, or a website on the web, that's a completely different experience, right? That's a completely different kind of code.

In terms of how JavaScript engine should optimize all these different use cases and support all these scenarios, that's a really intriguing question to me. Because on the web, for a website, mostly you have some JavaScript that you want to run once at the beginning. It can run in the interpreter, and then you're done with it.

Maybe there are some hot functions. Maybe there are some hot loops or things that get repeated over and over again, and then the JavaScript engine can optimize those. That's it for the web. But then then you look at the example of a Node.js server, and that's a completely different use case. You have a little bit of code that runs on startup, but at that point your server is running and you have this long-running process that is constantly running new and new JavaScript. Most of that become hot overtime. In that case, we want to optimize that code and run it in TurboFan, or as long as we can, and these are completely different use cases, and you can imagine there are some interesting tradeoffs to make if you want to support these all in a generic. Ideally, your JavaScript engine is not forked or you don't have different code paths supporting, "Oh, okay. This piece of our JavaScript engine is for supporting long-running Node.js applications, and this part is for supporting the web, and this part is for supporting this other use case." That is not very maintainable and it wouldn't scale very well. It's tricky, but there's lots of

heuristics that go into V8 and other JavaScript engines to try and make these tradeoffs and try and tune the knobs to make sure that we have the best possible experience overall.

[SPONSOR MESSAGE]

[00:36:24] JM: DigitalOcean is a reliable, easy to use cloud provider. I've used DigitalOcean for years whenever I want to get an application off the ground quickly, and I've always loved the focus on user experience, the great documentation and the simple user interface. More and more people are finding out about DigitalOcean and realizing that DigitalOcean is perfect for their application workloads.

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

Get your free $100 credit at do.co/sedaily, and thanks to DigitalOcean for being a sponsor. The cofounder of DigitalOcean, Moisey Uretsky, was one of the first people I interviewed, and his interview was really inspirational for me. So I've always thought of DigitalOcean as a pretty inspirational company. So thank you, DigitalOcean.

[INTERVIEW CONTINUED]

[00:38:31] JM: There's also the concern of the de-optimization process. If you have hot code that you no longer need, I guess this is one form of garbage collection, so I'm not sure if you're super familiar with it. But if something changes, like if a type of argument changes to the

function and you need to de-optimize a piece of hot code, then you might need to throw a piece of code out and replace it with a new version. Tell me a little bit more about the de-optimization process? How is code evaluated for whether or not it is still hot.

[00:39:07] MB: Okay. I can talk a little bit about that. It's actually separate from garbage collection. Although, conceptually, I can see how you can think of it that way. That's an interesting topic. Okay, to go back, we have some code. It runs in the interpreter and we collect some profiling data while it's running in the bytecode and the interpreter, right?

To go back to our earlier example, I thought we had a function that adds two numbers together, X and Y, and it just pluses them basically. Okay. The feedback that the JavaScript engine would get from that, if you keep calling that function with two numbers, we would remember that, we would say, "Okay, this keeps getting called with numbers. This code is becoming hot." So we optimize for the cases that we've seen, which in this case is the arguments are both numbers. So we produce optimized code for those cases for only that case. It's only one case, basically, and we can run that code very quickly once it's done being generated.

Then imagine, after the code gets hot, you call the same function, but now with different argument types. Like you said, maybe you're not calling them with numbers, but now suddenly by accident you pass in a string, or two strings. In that case, the optimized machine code includes a check when we generate it. It includes a check to see if our assumptions that we made, if they're still correct. In this case, it would include a check, "Are the two arguments still numbers?" In that case, "Okay, run this highly optimized code that we put together." The else branch for that check says, "Okay, if not, if it's not two numbers, then I don't know what to do with it. I don't have optimized code for this yet." In that case, you have to de-optimize.

De-optimizing basically means that we're going back to the interpreter, we're going back to the bytecode that we had before that supports all these different cases, but it runs more slowly, because it runs in the interpreter. Once that happens, we de-optimize. We're back in the interpreter. We're back at executing our code, and I know de-optimizing sounds like a bad thing, but really it's an important feature of the JavaScript engine, because it allows your JavaScript code to run correctly and it allows us to implement the spec correctly and support it even when things go wrong.

Now, when we de-optimize, we get back to the interpreter. Let's say after that, you keep calling the same function, now with two string arguments over and over and over again. So we already had optimized codes for the case where we have two numbers. But if the function becomes hot again, if we keep getting same types as arguments, then eventually the optimizer compiler will kick in again and say, "Okay, not I know about these two cases. So I will just optimize and create some code for a case where we have two strings," and it will add that to the optimized machine code that was there before.

Instead of an if, it basically becomes a switch statement, where it checks, "Okay, are the two arguments numbers like I saw in the first case? In that case, run this optimized code. If not, then check if the two arguments are two strings, in which case I have brand new optimized machine code. Just use that instead. If it's none of these things, then we have to de-optimize and start to whole thing over again."

One of the worst things that can happen for your application is de-optimization loop. If you keep going back and forth between optimized code and bytecode, and as a JavaScript developer, you can do a lot of work to avoid that by trying to make sure that you keep passing the same types of arguments to the same functions.

[00:42:22] JM: We've talked about JavaScript in the context of the browser, in the context of the server. What about JavaScript in other contexts, like for internet of things, for example? If we're running Node on an internet of things, would we need domain-specific compilers, or domain-specific runtimes for that kind of application?

[00:42:42] MB: Ooh, that's an interesting question. I'm not sure if I have a good answer to that. I know there is a specific JavaScript engine that is made for a hardware with low capabilities. It's called XS, and it's made by Moddable. It's a super cool project, because it's incredible how small of a footprint it has. It runs with very constrained memory available. It runs on very constrained devices. I recommend you look into it if you're interested in this kind of stuff.

When it comes to your broader question, I guess are you asking can we use JavaScript to compile to all these different platforms and targets in a way?

[00:43:19] JM: [inaudible 00:43:19]. We can do that. Is JavaScript in its current form and the current version of the V8 well-suited to run, I guess, internet of things applications, or are there other domains where people want to run JavaScript where perhaps the engine would be to be forked or changed?

[00:43:38] MB: I guess, right now, to be honest, the V8 engine might not be the best fit for constraint devices. One thing that we could potentially do there is work on an interpreter only mode. Because if you're on one of these super limited devices, you basically care about running some JavaScript. You don't necessarily care if it can run with an optimal performance, right? So it would be better for those kind of use cases if V8 could have a certain mode where the optimized compiler just wasn't part of the binary anymore. That would make the whole V8 process a lot smaller. It would make sure everybody don't need as much memory, and we could still run JavaScript. It would just run in the interpreter.

[00:44:20] JM: Okay. Separate question. Is there security concerns around a JavaScript engine? Does a JavaScript engine have to worry about anything related to security?

[00:44:32] MB: Oh, definitely. Yes. Yes, very much. So it's kind of crazy to think about it, but every time you visit a website, you're basically loading untrusted code on to your machine, and then the JavaScript engine is executing that, right? Running JavaScript without even knowing what it is. So the JavaScript engine has a very big responsibility at making sure that happens as securely as possible.

One particular example of that would be typed arrays. This is a relatively new feature in the history of the JavaScript language I could say. There have been lots of vulnerability reports related to typed arrays, where people could write some JavaScript and, basically, in just a few lines of code, trigger an out of bounds memory read and write. Giving an attacker full access to a random chunk of memory in the computer, which could potentially contain sensitive information.

There were a lot of bugs with typed arrays in every single JavaScript engine. We have a team at Google called Project Zero, and they found a bunch of these vulnerabilities, and they actually

fixed the spec as well. There were some spec issues that they've found to help avoid – To reduce the chances of this happening again in the future. They made the spec more robust.

Yeah, it's very easy to make a small mistake that would give a malicious actor the ability to write some JavaScript code that then access this random memory on your machine. That's, of course, something we're working to prevent.

[00:45:54] JM: That term spec, is that referring to the JavaScript spec that all of the different browsers are going to adhere to?

[00:46:02] MB: Right. Yes. Yeah, I was referring to ECMA Script spec. That's the official name of the documents. But, yeah, everyone just calls it JavaScript. There's this committee called TC39, which is the technical committee number 39 from the ECMA organization, and they get together every two months to talk about the evolution of the language. They discuss new proposals to add new functionality to the language, and they discuss all kinds of other changes to the JavaScript specification.

Indeed, ideally, all JavaScript engines try to implement the spec as closely as possible, but even with a spec, sometimes there are some case that are underspecified, and that's where it becomes dangerous, not just for security, but also for interoperability. That's why testing is also really important, and there is this shared test repository for ECMA Script test, or JavaScript test, that all browser vendors and all JavaScript engines use, and it's called test262.

For every new feature that goes into the language, there is this long process of five stages. It starts with stage zero, because, of course, we're at zero, and it ends at stage four. At which point the proposal makes its way into the official spec text, the tests are added to this test262 repository. But none of these stages really matter to JavaScript developers. I think the most important one is stage three, because at that point, the committee considers the proposal and the proposed spec changes to be more or less complete, which means that the JavaScript engines can start to implement this proposal behind the flag and maybe ship it once I think it's mature enough. So stage three is where it's at.

[00:47:35] JM: You mentioned that bugs can be found in the spec. For example, a security issue was found in the spec, how does a spec get evaluated for flaws? Because you can't actually run the spec code, right? How do you evaluate a spec for security issues?

[00:47:54] MB: Right. That's the thing. It is a very dry technical document, but in the end it's still humans that are interpreting it and they read between the lines sometimes. Sometimes people have different interpretations of the same piece of text.

When I said before, and they fixed the spec, there was not really a security book in the spec, but the spec was missing some things that made it very easy to make certain mistakes that in some cases multiple JavaScript engines [inaudible 00:48:21]. Some of them actually made exactly the same mistake and it caused security issues in multiple browsers.

So by making the spec more robust by adding things like, "Oh, yeah. Assert that this value is smaller than this value." Small things like that can actually make a big difference when you're writing to spec texts.

[00:48:38] JM: Mathias, we're nearing the end of our time. Is there anything else you want to add about JavaScript, or browsers, particularly speculation on the future? I try to task different guests about where they think things are going. Do you have any closing thoughts?

[00:48:54] MB: I'm really excited about the way JavaScript has been growing, especially the last couple of years. It seems like more and more exciting new proposals are coming in. I can't wait to see what happens to the language next. I am also looking for feedback from the JavaScript developer community, because I think that's a view that's still a little bit underrepresented in these TC39 meetings. So if people have any ideas or any feedback on existing proposals, or even if they have proposals on their own, they can always reach out to me and I can work with them to make sure these get presented in front of the committee.

Finally, I have a little bit of advice for JavaScript developers, because I know we've been talking about JavaScript engine internals quite a bit, but at the end of the day, as a JavaScript developer, I think the best advice that anyone can give you today is just to write modern and idiomatic JavaScript code. Just write codes that is optimized for readability, code that makes

sense for you and let the JavaScript engine worry about making it fast, because that's kind of its job. That gives you the best chances of producing code that runs quickly and performantly on a multitude of different JavaScript engines and web browsers.

[00:49:58] JM: Yeah. It's interesting you say that. That last conversation with the JVM guy about GraalVM. One of the things he touched on in that show was how fast Java has gotten. I remember even when I was in college, people talked about Java as if this language is so much slower than C, or C++. I don't know about the speed differences today between C++ and Java. Perhaps, it's still miles ahead. But it sounded like there had really been a lot of gains in Java. So I wouldn't be surprised if the same was true of JavaScript. I'm sure you're born witness to that, where you've got a bytecode language, you've got a higher language and then people just make improvements overtime to it and it just gets more and more efficient and things keep getting faster. Eventually you can really just let the compiler architecture do a lot of the work.

[00:50:52] MB: Mm-hmm.

[00:50:53] JM: Cool. Well, Mathias, thank you for coming on Software Engineering Daily. It's been really great talking to you.

[00:50:57] MB: Yeah, thanks for having me. I had a great time.

[END OF INTERVIEW]

[00:51:02] JM: Nobody becomes a developer to solve bugs. We like to develop software because we like to be creative. We like to build new things, but debugging is an unavoidable part of most developers' lives. So you might as well do it as best as you can. You might as well debug as efficiently as you can. Now you can drastically cut the time that it takes you to debug.

Rookout rapid production debugging allows developers to track down issues in production without any additional coding. Any redeployment, you don't have to restart your app. Classic debuggers can be difficult to set up, and with the debugger, you often aren't testing the code in a production environment. You're testing it on your own machine or in a staging server.

Rookout lets you debug issues as they are occurring in production. Rookout is modern debugging. You can insert Rookout non-breaking breakpoints to immediately collect any piece of data from your live code and pipeline it anywhere. Even if you never thought about it before or you didn't create instrumentation to collect it, you can insert these nonbreaking breakpoints on the fly.

Go to rook out.com/sedaily to start a free trial and see how Rookout works. See how much debugging time you can save with this futuristic debugging tool. Rookout integrates with modern tools like Slack, Datadog, Sentry and New Relic.

Try the debugger of the future, try Rookout at @rookout.com/sedaily. That’s R-O-O-K-O-U-T.com/sedaily. Thanks to Rookout for being a new sponsor of Software Engineering Daily.

[END]