EPISODE 1040

[INTRODUCTION]

[00:00:00] JM: V8 is the JavaScript engine that runs Chrome. Every popular website makes heavy use of JavaScript, and V8 manages the execution environment of that JavaScript code. The code that processes in your browser can run faster or slower depending on how hot the code path is. What hot means is if a certain line of code is executed frequently, that code might be optimized to run faster.

V8 is behind-the-scenes doing all these work in your browser all the time, evaluating the code in your different tabs and determining how to manage that runtime in-memory. As V8 is observing your code and analyzing it, V8 needs to allocate resources in order to determine what code to optimize. This process can be quite memory intensive and can add significant overhead to the memory of Chrome.

Ross McIlroy is an engineer at Google where he worked on a project called V8 Lite. The goal of V8 Lite is to significantly reduce the execution overhead of V8. Ross joins the show to talk about JavaScript memory consumption and his work on V8 Lite, and we've done some great shows on JavaScript in the past. If you want to find those old episodes, you can go to softwaredaily.com and look for JavaScript episodes. If you're interested in writing about JavaScript, we have a new writing feature. You can check it out by going to softwaredaily.com/write.

[SPONSOR MESSAGE]

[00:01:38] JM: You might be aware of the Techmeme Ride Home Podcast, which covers broad tech industry news every day in 15 minutes. In early March, the host of that show, Brian McCullough also launched a daily coronavirus podcast called the Coronavirus Morning Report. In just 15 minutes every morning, it gives you the latest about the coronavirus. So you don't need to be nervously refreshing your newsfeeds all day.

I've been at home quarantined, and when I'm cooking, I can quickly listen to an episode about vaccine development techniques. I go for my daily walk outside and I can learn about how long

COVID-19 stays on surfaces and whether I should be concerned when I'm picking up my Amazon packages outside. It is a strange world that we're currently living in.

According to the New Yorker Magazine, Coronavirus Morning Report is one of the top coronavirus podcasts to listen to saying that it stays informed and non-hysterical and focused. Brian McCullough knows how to podcast. He's been doing it for years, and you can search your podcast app now for coronavirus morning and subscribe to the coronavirus morning report.

[INTERVIEW]

[00:03:05] JM: Ross, welcome to the show.

[00:03:06] RM: Thanks for having me, Jeffrey.

[00:03:08] JM: V8 is a JavaScript engine that runs Chrome. Explain what the purpose of a JavaScript engine is.

[00:03:15] RM: The JavaScript engine is there to run all the JavaScript, and JavaScript is the coding language of the web. It what allows developers to do programmatic things on the web, so moving buttons about, or responding when those button clicks happen, or maybe animating. So the V8 engine is what powers that all underneath.

[00:03:37] JM: You've spent some time researching operating systems, and my sense is that a JavaScript engine is something that is so complicated. It borders on the complexity of an operating system. How does operating system design compare to browser design?

[00:03:54] RM: Yeah, that's right. Certainly, I would compare the Chrome browser to an operating system and has so many components that are similar. In a previous life, I worked on the Blink scheduler, which is part of Chrome. So it's got a scheduling engine just like the operating system. In V8 itself, we're doing memory management. We're doing garbage collection. We're doing compilation of jittered code. There are lots of components in there and defi nitely feels like operating system work. It's really interesting.

[00:04:25] JM: We're going to talk about V8 Lite today, but before we get into V8 Lite, I'd like to know a little bit about your prior work. What parts of the browser and V8 did you work on prior to V8 Lite?

[00:04:39] RM: Sure. I'm part of the Chrome mobile team here in London. We've been concentrating for a longtime on memory usage. When I started on that team, we were optimizing Chrome's memory usage to get it to fit on low-end mobile Android devices and where Chrome had previously been on big desktop, beefy machines, trying to fit in mobile. Involved lots of optimizations to shave memory off it.

I then worked on the scheduler in Chrome where we tried to reorder tasks. So taking more important tasks and moving them to the front of the task queue and taking idle tasks and waiting until the Chrome browsers overall idle before executing those tasks.

I then moved on to the V8 team, and on V8 we started a big project to build a new interpreter. Previous to that point, V8 at all was just-in-time, all JavaScript to machine code and we realized that on mobile devices, that machine code was a lot of memory. Often, these functions weren't getting executed very often. We could save a lot of memory if we just compiled them to a compressed bytecode and interpreted them instead of executing them directly. We built this interpreter. Initially, it might have only been for mobile devices, but it synergized really well with a new optimizing compiler we're building in the same time and simplified the whole pipeline.

We ended up having a big project to – Which we called Ignition and TurboFan, which was replacing completely the old compilation pipeline, which was [inaudible 00:06:10] and Crankshaft replacing it with Ignition and TurboFan, where Ignition is the bytecode interpreter and TurboFan is the new optimizing compiler. After that, we've been working on various compilation optimizations here in London and the V8 Lite project

[00:06:26] JM: All right. Let's talk a bit about memory management, because that directly corresponds to the work you did on V8 Lite. How much memory does Chrome use on a machine?

[00:06:41] RM: I think it depends very much on, A, what your machine is, and B, what type of user you are. There are a lot of particularly type users who have maybe hundreds of tabs open all the time. I'm one of them. In these situations, Chrome is going to take a lot of your system memory. In fact, most of the time, all my applications are sitting in Chrome. They're running JavaScript applications in a tab. So it's not really surprising that Chrome's taking all that memory on desktop.

Where it becomes a problem is on these low-end phones where you really have much less memory. Things we're thinking about here, maybe phones that have 512 megabytes of RAM, which might sound reasonable, but then you have to remember the hardware takes 100 megabytes. The operating system takes 100 megabytes. Other applications takes 100 megabytes. Really, at the end of the day, Chrome ends up having about 100 megabytes of memory left to do all of its operations. So that's like rendering pictures, leading out the page, and doing JavaScript execution at the same time.

It depends where you're sitting. But on a mobile device, a typical webpage, really expect maybe 30 to 40 megabytes in the render process, which is the process that's displaying the tab. All of that maybe 40% to 60% would be the V8 JavaScript heap. V8 is a quite sizable proportion of that memory usage generally.

[00:08:15] JM: How does the runtime for Chrome on desktop compare to Chrome on mobile in terms of memory footprint?

[00:08:22] RM: We have done a lot more optimization work on mobile to fit the general cases, to just compress the general cases. The memory usage is generally lower. There are other aspects as well. On mobile, you generally are viewing one tab and the operating system behind-the-scenes – In Chrome, each tab is a separate operating system process. On Android, the operating system can kill processes when it's running low on memory. I would do that without interacting with the main Chrome process particularly.

[00:08:56] JM: That sounds like a problem.

[00:08:57] RM: It can be a problem, yeah. But in some sense, it's kind of useful for us because we can create those tabs and then leave it to the operating system. We give it some hints as to which tabs are important and might be switched back to. We can leave it to the operating system to kind of garbage collect those tabs and as they go.

That typically scales much more widely depending on the phone. So if you have a high-end 8 gigabyte of RAM device phone, then you'll probably find a few switch between your last 4, 5, maybe 10 tabs. They're all still there, whereas if you have a lower-end 1 gigabyte phone. You switch to a couple of tabs back and you see the whole tab reloading, and that's because the process has been killed in the background and Chrome needs to reload the webpage.

[00:09:41] JM: The memory usage we learn in software engineering school or computer science school consists of the stack and the heap. Is V8 mostly – Or the Chrome browser, is it mostly consuming stack or heap?

[00:09:58] RM: It's mostly heap. When we talk about heap, we have probably more categorizations than just stack and heap. There's defi nitely some stack memory, and that's generally kept relatively small and it depends. It could spin up a lot of threads, then each thread needs to have a stack. So that might take some memory.

Then there's heap, and in V8 we kind of partitioned that between what we'd call maybe native heap, which is your kind of typical C++ heap which you would allocate with malloc or new. Manage that memory yourself as you're coding. Then what we would call the managed heap, which is the garbage collected JavaScript heap and where all JavaScript objects live. That has a very different feel because the garbage collector is in charge of collecting that memory and freeing up the memory as it goes. Typically, that's the largest part of memory for us in V8 and it's one of the more substantial parts of memory in Chrome overall as well.

[00:11:00] JM: Tell me about some of the objects that might make up the heap. If I'm browsing a typical website or I've got a website open. I'm on hackernews.com and I've got 3 or 4 other Chrome tabs open. What is in my heap on Chrome?

[00:11:21] RM: Lots and lots of things. I'll start with the things that are in V8 and then talk about some other things that are in Chrome overall. On V8's managed heap, the garbage collected heap, you'll have all your JavaScript objects. For example, any functions, JavaScript functions that have been compiled to bytecode, that will live in the managed heap. We also have metadata, which is where V8 Lite actually came in where we keep track of what those functions look like, what type of objects those functions have seen and those types of things and you use that optimize the code later. Then you have your actual objects. If you create arrays or strings or things like that in JavaScript, then that also lives in the managed heap.

You then have what's called the DOM, which lives outside of V8. This is in the render engine in Blink. That's called the document object model. If you know your HTML, then that's your kind of element. So maybe your div elements and your MH elements and things like that. They're represented in a DOM, which is also garbage collected by a separate garbage collector in Chrome called oil pan.

Then you have lots and lots of other things. You'll have your images on your site and they'll be in both the kind of compressed form, the JPEG form, but we also need to decompress it to show it on the screen. So they'll be in the decompressed forms as well. There'll be the HTML source for the actual webpage that came in, the CSS, all the objects that are needed to process that. Then of course Chrome needs to display this stuff on the screen. I think a large part of the memory ends up being the GPU command buffer. So the commands that Chrome is sending to the GPU to draw things on to the screen.

[SPONSOR MESSAGE]

[00:13:15] JM: Looking for a job is painful, and if you are in software and you have the skillset needed to get a job in technology, it can sometimes seem very strange that it takes so long to find a job that's a good fit for you.

Vettery is an online hiring marketplace to connect highly-qualified workers with top companies. Vettery keeps the quality of workers and companies on the platform high, because Vettery vets both workers and companies access is exclusive and you can apply to find a job through Vetter by going to vetter.com/sedaily. That's V-E-T-T-E-R-Y.com/sedaily.

Once you’re accepted to Vettery, you have access to a modern hiring process. You can set preferences for location, experience level, salary requirements and other parameters so that you only get job opportunities that appeal to you.

No more of those recruiters sending you blind messages that say they are looking for a Java rockstar with 35 years of experience who's willing to relocate to Antarctica. We all know that there is a better way to find a job. So check out vettery.com/sedaily and get a $300 signup bonus if you accept a job through Vettery.

Vettery is changing the way people get hired and the way that people hire. So check outvettery.com/sedaily and get a $300 at bonus if you accept a job through Vettery. That's V-E-T-T-E-R-Y.com/sedaily.

Thank you to Vettery for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:15:04] JM: Now, to take a different approach to understanding the V8 runtime, I’d like to spend a little time talking about what happens when code is compiled in V8. I know there’s interpretation and compilation. I believe that the JavaScript code is turned into JavaScript bytecode, and then the bytecode eventually execute on the underlying machine or translated from bytecode to the underlying machine code. Can you give me a brief overview of the compilation and runtime of JavaScript code that’s executing in my browser?

[00:15:44] RM: Sure. What happened is, in simplified form, Blink, the rendering engine and Chrome will come across a script tag, which is in JavaScript to be run. Since that URL, well, just the data the JavaScript source to V8. So now we have a string of JavaScript source. We then run that through what we call the parser, which will take the JavaScript and turn it into a kind of machine representation which we call an abstract syntax tree, which is just a way of us understanding what the actual JavaScript semantically means and a tree form that makes it easy to do some compilation on it. That’s the outcome of the parser and that feeds into the bytecode generator, which is the first part of the ignition interpreter.

The bytecode generator will walk that abstract syntax tree and generate bytecodes for each of those operations. An example might be you have a function which has A+B and returns that. The bytecode interpreter will generate bytecodes for loading the variable A, loading the variable B and then doing a JavaScript edition on those variables and then returning the result. So there'd be a couple of bytecodes there, and that will be the bytecode for that function.

Then once we have the functions available, we can start executing the scripts. We do that through ignition, the interpreter. So we'll execute a function and the ignition interpreter will start reading that bytecode and kind of bytecode by bytecode. Read the first bytecode, and it has a bunch of little snippets of machine code which performs the operations for each of the bytecodes.

For example, we have a snippet code for the add operation, the add bytecode. When we see that add bytecode, we jump to that snippet of code and we perform the operation. These are typically – Because they're JavaScript operations, they're not as simple as you might think. It's not just an add machine code, because of course JavaScript, the plus operator, you can add strings. You can add integers. You can doubles. You can add all kinds of things you can and objects together and you need to [inaudible 00:18:14] those objects into some kind of type that can be added together, like an integer or a string.

The interpreter typically has inline code for the typical operations, adding integers or adding strings, and then it will call out into the runtime to do the more complicated things. That's your kind of interpretation of the script. As you're executing that, we build up some knowledge of what those operations are. For example, in this app bytecode, we have a site table which is called a type feedback vector. In there, we store some metadata about what we've seen in those adding operations. Is it only been integers that we've added in this function or are we adding strings or that combination of everything?

Once a function gets hot enough, we'll decide to optimize it with our optimizing compiler, which is TurboFan. TurboFan will take the bytecode and this feedback vector and it kind of glues all together and tries to make a kind of optimized form of code. If you'd only seen integer addition and that operation there, then TurboFan will kind of speculatively assume that you're only ever going to see integer operations in the future and generate some machine code that's kind of targeted specifically for that and can be a lot clustered because of that. That's kind of a quick flow through the whole pipeline.

[00:19:42] JM: That's a great description. One concept I want to focus on a little bit more is the idea of a hot code path, and this is the idea that you have different pieces of code that are executing in your website. Some of those pieces of code are going to be executed more often than others. You want those pieces of code to be optimized for. You want them to be identifi ed as a hot code path so that you can optimize for them. You touched on a few things there. I think one is that the inlining of code is one way to speed up code. Then another is just putting it through an optimizing compiler. Can you just talk a little bit more about if you've identifi ed in your JavaScript engine that a piece of code is hot. It's going to be executed frequently, what are the things you can do to speed up that code?

[00:20:49] RM: I think this would be a whole talk in itself. But there're various things. One of the biggest things that typically speeds up code on real websites is identifying what we would call the shape of the objects. Say you have an object that you've created in JavaScript and it has different fi elds in that object, A and B, and if you know what the object looks like, then if you're optimizing a function that reads values off of that object, then you can do a much more optimal job.

The real problem here is that in JavaScript, objects are dictionaries in a sense. There's no strong class hierarchy like you would have in Java. So we have to do very kind of – If we're doing the naïve approach, we'd have to do very slow operations to look up properties in each of these objects. If you're doing the naïve approach, you would maybe need to treat it like a dictionary, like a hash table and looking up exactly where each property sits in that object and then reading out the actual one that you want.

But if you know the structure of the objects, if you know that all of the objects of this particular type were created with A in the first and field of the objects lay out, then you can just read that directly. Then if you know even more what type of objects A generally is, if it's a number or something else, then you can propagate that knowledge down and optimize code even better based on that so you can then later optimize a way of various checks that you would need to do in JavaScript and kind of turn the code into much more simple looking code from the machine point of view.

[00:22:38] JM: The reason we're having this conversation is I wanted to talk to you about a project called Lite Mode for V8. When we're talking about making a lighter version of V8, are we talking about it in terms of making the code run faster, more memory efficiently? What are the goals of the V8 Lite project?

[00:23:04] RM: Sure. The goal of Lite Mode was to make V8 more memory-efficient. We started the project looking at really low-end Android phones and realizing that the user experience of Chrome there would be a lot better if we used less memory even that meant we couldn't do all the optimizations we might normally do in V8 to speed up JavaScript execution.

For example, generating this optimized code I was just talking about. We started on a road of like seeing all the – We looked up how memory was being used by V8 in the heap and we identified maybe 30% to 40% of the heap was being used to store metadata about functions to allow V8 to later speed up the execution of those functions and we thought, "Well, if we cut that all out and then just see how fast V8 is. Is it fast enough still for these low-end devices?" and then see where we are.

The results were very promising. We could cut these memory usage out and websites still performed reasonably well, and JavaScript-heavy benchmarks and things like that suffered quite a bit, but it was still seem promising. Then we realized that we could actually rather than taking all these memory out, we could more optimally only allocate that memory when it was really needed. We talked about it being more lazy. This is what the project evolved into a project for all devices whereas it started a project just for these low-end devices. It evolved into, "Well, can we take these optimizations and apply them everywhere by only lazily allocating metadata when it's needed?"

We ended up getting almost all the savings that we could get with the Lite Mode. We could apply it to all use cases, so desktops and high-end mobile devices are now getting the memory savings that we're doing with Lite Mode and with none of the performance regressions.

[00:25:11] JM: How many people were working on V8 Lite?

[00:25:14] RM: There were three of us working on V8 Lite.

[00:25:17] JM: What's the process for managing such a project to reduce the memory consumption? How do you set goals and deadlines?

[00:25:31] RM: We started with a week-long hackathon just to see what we thought we could do where we just carve bits out of V8 and did experiments where we measured particular parts of memory usage and saw how impactful it would be to remove those parts. After that hackathon, we had a good sense of – We could probably reduce V8's memory usage by about 20% if we're aggressive like this. Then we spent a couple of months actually making it possible to do this Lite Mode specifi cally for low-end devices. There was a number of things we knew we needed to just do there. We kind of carved out three main projects as part of this Lite Mode, and one was bytecode fl ushing. One was lazy feedback and one was lazy source positions.

With the lazy feedback, we knew that was going to be one of the bigger projects. So we started with the approach where it's actually no feedback vectors at all, and then we built on to a project where we lazily allocated them as we go. So the no feedback case allowed us to kind of figure out all the places that V8 was depending on this feedback vector metadata that we were trying to remove or make lazy. Then once we had done that, then we could work on the process of making it lazily allocatable or optimizing the performance of it and optimizing the performance of V8 overall.

The process is basically like identifying those memory impactful projects, carving out steps to allow us to make independent progress on those projects and identify where we feel like it's doing well. Then typically there's a period of optimization and kind of stabilization and things like that happens after that.

[00:27:15] JM: So as you were fi guring out that you could reduce the memory footprint by 20%, what kinds of tools were you using to get an understanding of the memory utilization of V8? How did you examine the runtime to understand that there was the room for this 20% gains in memory footprint?

[00:27:45] RM: It’s a good question. We have recently good tooling to trace. Because heap is garbage collector, the garbage collector needs to trace through all the objects and heap to do that garbage collection. You can tweak that slightly and do a similar type of trace, but rather than doing garbage collection, you just collect data of what type of objects are there in the heap and how large each of them is. We have that tooling in V8. That’s kind of heap statistics tooling. From that, you can work out different types of objects in the heap and you can aggregate them in different ways, and that was what we spent our time doing, was kind of tracing heaps for typical websites and then aggregating the data.

The other important part of this is having realistic benchmarks. So we have a set of – We call them system health benchmarks in Chrome, which are pre-recordings of websites, typical websites, and kind of scripted interaction with those websites. So you might have a news site and the scripted interaction is clicking an article, clicking back, [inaudible 00:28:51], clicking another article, etc. So that gets us reasonable confidence to this as a kind of typical usage of Chrome. Then we can run our memory tooling on that and get some analysis of where memory is going when you’re doing a typical interaction on a website.

[00:29:06] JM: As we talk about the areas for improvement, one area of extraneous memory was used during exception handling. You wrote about this in the article about V8 Lite. You talked about it in a talk that you gave. In improving the extraneous memory around exception handling, you changed how you generate source positions. Can you explain what a source position is and how it’s used?

[00:29:37] RM: Sure. Yeah. Source positions are basically a mapping between the bytecode, which we are executing when we’re executing the JavaScript and the actual source code, the JavaScript source code that went into generating that bytecode. Of course, when an exception is thrown, we aren’t interpreting the actual JavaScript source code itself. We only have the position in the bytecode that the exception was thrown at. We used these source position tables to kind of map back from that bytecode offset to a particular source code offset, and that’s what allows us to show the line number and column number of the JavaScript that caused the exception produce the fill symbolize stack trace and the console when you see an exception being thrown.

What we were doing originally was – Well, to generate these source position tables, you need to have both the JavaScript and the bytecode available to you at the same time. The easiest time to do that is when you're generating the bytecode in the first case because you can then say, "Okay, this is the JavaScript I'm generating bytecode for." So that's the source position for that particular bytecode.

But that means every function needs to have the source position table, and we realized that was taking a reasonably large chunk of memory and it's maybe about 30% the size of the bytecode itself. Most functions don't throw exceptions. Even if they do throw exceptions, they are probably caught by other code in a tri-catch. If the exception is caught in a tri-catch, you don't actually need to map the positions back to JavaScript source code. It doesn't need to be symbolized.

With the lazy source position and project, what we did was we turned off collecting these source positions when we're compiling the code, and instead only did it later when we're symbolizing the code. It's a bit of a tradeoff there. You effectively need to recompile the function when you're symbolizing the stack trace. So that can cause some slowdown, but it only causes slowdown in the cases where exceptions are being thrown. It shouldn't cause problems on real webpages and may verify that it didn't cause any problems in typical webpages, but it saved a fair amount of memory because we're only doing it lazily and most functions don't need these sort of position tables.

[00:32:04] JM: There is a data structure in V8 memory manager. I think you mentioned this a little bit earlier called a feedback vector. Can you explain what purpose a feedback vector serves?

[00:32:18] RM: Sure. The feedback vector is another table that goes along with the bytecode, and for certain bytecode operations, it's useful for V8 to know something about the types in that particular bytecode operation is expected to operate on. I gave the example before of the addition operator, and that's a typical one that's used where you might be adding integers or strings or other types of numbers. If we can work out the typical operations that are being done at that particular function site, particular call site, then when we later go to optimize that code, we can optimize that operation to be very specifi c to the types that are expected and doing a string concatenation if that operation always see strings or doing an integer addition operation if it's integer additions.

This type feedback is very important for generating optimized code. We talked about before, we don't optimize all the code that we see. There are a lot of JavaScript code that's only run once or twice or during initialization or uncommon code paths. So we were allocating these type feedback vectors for all the functions even though most of the time we didn't use that data.

With lazy feedback vector allocation, we deferred the allocation with these type feedback vectors until a later point where we thought it might be useful and we thought – When we started collecting type feedback to then do the later optimization of those functions.

[00:33:55] JM: Can you clarify how storing data about the type of a variable is useful for making a more effi cient JavaScript engine?

[00:34:08] RM: Sure. I'll give an example of something which is actually useful for both optimized code and is useful for the interpreter, and this is an example of why we wanted to lazily allocate feedback vectors even for functions that might not end up being optimized, because it speeds up execution of kind of warm functions. That is when you're loading properties off of objects.

Objects in JavaScript are very dynamic. The class hierarchy is very dynamic itself. It's not like JavaScript or C++ where you have an object and you know statically ahead of time generally that a particular field is a particular offset into an object, a particular. You know that the object is 8 bytes and the first 4 bytes are pointer for field A and the second 4 bytes are an integer value that's in the object.

For JavaScript, you can create objects that have any number of properties. You can add and delete properties on those objects dynamically. Subtyping is done with a kind of a prototype form where you have a kind of prototype object that's attached to the object, and if you try to load a property from that object that's not on the object, then you need to look at the prototype object.

Actually, loading one of these properties off of an object in the generic case when you know nothing about the object itself is very complicated. You need to look at, "Okay, how did the V8 decide to represent this particular object? Okay, for this particular property, where is it stored? Is it in this object or is it somewhere out the property chain?"

Just doing that generically is very slow. What we do is we have what we call maps and are typically called shapes in other VMs or hidden classes, and they're kind of a descriptor of the layer of a particular object. When we want to load a property off of one of these objects, we do all that work to work out where that property is for this particular type of object. Then in the feedback vector, we store – All right. we've seen this particular object map before or this particular hidden class before, and the work we did to work out which property to load off that object told us this particular offset on the object that you get in.

The next time you do that operation, you can very quickly just check, "Okay, does this object match the same type of hidden class that we've seen the last time that we did this operation? In typical webpages, this will be the case. You have a function that's probably operating on the same object types. It's called monomorphism. If that's the case, then we can just very quickly look up the data that we've already stored beforehand and load that property very quickly. That allows the interpreter to load it very quickly. Then it allows the optimizing compiler to generate just-in-time code, machine code, that does that check and then very optimally does the load of that property from the object without having to do all the other checks and asking the runtime and how the structure of the object looks.

[00:37:34] JM: These feedback vectors are memory intensive, right? That was one of the recognitions of the V8 Lite Project, was that if you don't allocate the feedback vectors so aggressively, you can improve the performance of the browser. Am I understanding that correctly?

[00:37:56] RM: Yeah, that's right. They typically took about 8% of V8's heap. When we make them lazily allocated, I think we now take about 2% of the V8 heap and we could reduce 6% of the V8 heap by just lazily allocating them when we need them.

[00:38:17] JM: So what did that mean in practice, the lazy allocation versus the eager allocation?

[00:38:22] RM: The eager allocation would be allocating in a type feedback vector for every function that gets compiled. The lazy allocation, what it means is – So we use the same kind of profiling infrastructure as we would use for working out if a function is hot enough to be optimized. We use that same mechanism for working out whether the function should have a feedback vector allocated for it. But we just do it much sooner. So maybe it's roughly about 10 times sooner than we would optimize the function for optimized code.

For example, maybe after 10 executions of a function, we'll allocate a feedback vector. Then after 100 executions of that same function, we'll run it through TurboFan and generate optimized code for that function depending on the size of the function.

[00:39:11] JM: If you do not allocate these feedback vectors eagerly, what's the result on the overall performance of the browser?

[00:39:21] RM: It's interesting. We thought that we would see quite a regression, and certainly when we didn't allocate the feedback vectors at all and we saw a very, very significant regression, but that was expected. But just doing this lazy allocation, we managed to get rid of all the regressions that we saw in the benchmarks by optimizing and tuning some other parts of our runtime system. For our telemetry, the system helped benchmarks. I mentioned before, where we run real webpages. We actually saw improvements in performance in some cases typically due to the fact that the garbage collector needs to run less often. But also due to the fact that just doing the generation, like collecting this type feedback information for these infrequently executed functions was actually costing us work if we weren't going to use that type feedback later. So just not doing that for those seldom execution functions was saving us that effort.

[SPONSOR MESSAGE]

[00:40:29] JM: As a company grows, the software infrastructure becomes a large complex distributed system. Without standardized applications or security policies, it can become difficult to oversee all the vulnerabilities that might exist across all of your physical machines, virtual machines, containers and cloud services. ExtraHop is a cloud-native security company that detects threats across your hybrid infrastructure. ExtraHop has vulnerability detection running up and down your networking stock from L2 to L7 and it helps you spot, investigate and respond to anomalous behavior using more than 100 machine learning models.

At extrahop.com/cloud, you can learn about how ExtraHop delivers cloud-native network detection and response. ExtraHop will help you find misconfigurations and blind spots in your infrastructure and stay in compliance. Understand your identity and access management payloads to look for credential harvesting and brute force attacks and automate the security settings of your cloud provider integrations. Visit extrahop.com/cloud to find out how ExtraHop can help you secure your enterprise.

Thank you to ExtraHop for being a sponsor of Software Engineering Daily, if you want to check out ExtraHop and support the show, go to extrahop.com/cloud.

[INTERVIEW CONTINUED]

[00:42:03] JM: I'd like to change the topic to talking about garbage collection and how garbage collection may or may not have created opportunities to improve memory management in your V8 Lite Project. Can you give me an overview for how garbage collection works in JavaScript and what opportunities you saw in improving it when you're working on V8 Lite?

[00:42:31] RM: Sure. There's another team that works in garbage collection in the Munich office. They know a lot about this. The main things they've been looking at in the garbage collection team are trying to reduce junk caused by garbage collection. Typically, if you're executing some code and then you need to do a garbage collection, the kind of typical approach you might use is what's called a stop the world garbage collector, where you stop everything executing so that you can trace through all the pointers in the heap. Workout what objects are still alive and then flush out all the dead ones. That's your garbage collection. But while you're doing that garbage collection, JavaScript execution is paused. That can cause junk on your webpage. If the garbage collection takes a couple of tens or hundreds of milliseconds, then that might be visible to the user.

For quite a few years now, the garbage collection team have been working on reducing that junk and they've done that in various ways. One of the first things they did was make it incremental so the garbage collector could do some work and then go back to executing JavaScript and then do some more work and continue where it left of. Then more recently, they've been making it more and more concurrent.

Doing all the garbage collection work on another thread than the one that's executing JavaScript and allow the JavaScript to execute at the same time as the garbage collector is working. That's one of the main projects that garbage collection team have been working on.

We also have worked with them on tuning the garbage collection heuristics to reduce the amount of memory used by these low-end devices. An interesting tradeoff there, because the more garbage collection you do, the more memory you save, but then the more time you're spending doing garbage collection. We have these kind of heuristics where if you have a higher-end device, we try and do a bit less garbage collection and the memory might grow a bit larger, but then your experience is probably going to be better because it's not doing that garbage collection all the time, whereas in lower-end devices, we tune it more aggressively to do more aggressive garbage collections more often and keep your memory within budget.

[00:44:42] JM: Moving from a top the world garbage collector to, I guess, a gradual dynamic garbage collector sounds like a really complicated refactoring, but I guess I would have to talk to somebody on the garbage collection team to learn more about that. If we zoom out, what are the results of the Project Lite, the V8 Lite efforts?

[00:45:06] RM: Sure. The result we got in the end was that we over the course of a couple of Chrome releases, we reduced V8's heap size by about 20%. That's a couple of megabytes on typical mobile pages and probably tens of megabytes on desktop pages, and we did it without regressing performance. Your Chrome browser running right now will be running our V8 Lite optimizations.

[00:45:33] JM: Is this particularly important for low-end devices or am I going to feel this on a top-end pixel device?

[00:45:45] RM: Yeah. So it's defi nitely more important on the low-end devices, and that's why we started there. I think you would possibly see it, feel it slightly on the higher-end devices where in the examples of V8, we managed to keep tabs open in the background for a bit longer. If you're switching between your tabs, then they're still available.

Particularly on desktop, if you have hundreds of tabs, then the savings kind of multiply up and you might be able to now have like 20 more tabs without running out of memory or without swapping in the background and slowing down your whole system. Yeah, if you're using all that memory, this can definitely help. Even if you're not, we saw those reductions in garbage collections times. So that could lead to you having a slightly less junky experience on the web. You see less stutters happening if garbage collection ends up running on the main thread.

[00:46:43] JM: What are the areas for signifi cant improvements that you still see in memory footprint for V8?

[00:46:51] RM: One of the other more recent projects that have just landed is a project called Pointer Compression, which is a really interesting project than itself. What we did today was compressing the points in the V8 heap from 64-bit values on your 64-bit machine, down to 32-bit values. Since V8's heap typically has a lot of these pointers and about 70% of the heap is these pointer types, this has a dramatic reduction and memory usage as well. We just landed that in M80, which should be Chrome stable right now. That reduced V8's heap by another 40% over and above the V8 Lite savings. This year, we've had really signifi cant reductions and memory usage.

Going forward, I think there are always things we can do to optimize the data that we store for metadata and things like that and we'll probably start looking a bit more other things, like responsiveness and the execution time for a while and then come back and have another look at memory usage. But there's always new things to reduce and always new things that popup as the top memory users and V8's heap.

[00:48:05] JM: We've been talking about V8 in the context of the Chrome browser, but V8 also runs Node.JS. Does V8 Lite improve the performance of Node.JS?

[00:48:20] RM: It's certainly enabled on Node.JS and the optimizations that I was mentioning. I think the behaviors are sometimes slightly different on Node.JS, but I think the benefits can probably be seen in similar ways for functions that are not very often executed, and that's where we'll get the more savings. It depends really on the application itself and how that application is structured. I think the times we would see this most often is in the more long-running Node.JS systems where you might end up being able to save a lot memory. Yeah, I think depending on the applications, this could definitely save – Certainly save memory usage on these Node.JS applications which could save users money if they're using cloud services.

[00:49:08] JM: What are you personally working on today? Are you still on the V8 team working on memory improvements or have you switched to something else?

[00:49:17] RM: Yeah, still in the V8 team. There's a team of us here in London that are looking at – We've switched to looking at the responsiveness and execution time of JavaScript code, and what we're looking at is actually a new middle tier compiler. When I mentioned the compilation pipeline earlier, I talked about ignition being the bytecode interpreter and TurboFan being the optimizing compiler, and we think there's a space in the middle there for functions that are not quite as hot, hot enough to be executed by TurboFan but could do some optimization. We're building a mid-tier compiler. We're calling it TurboProp at the moment, which takes some of the components of the TurboFan optimizing compiler and strips them down to make a compiler that can generate code faster. But the code that it generates might not be quite as optimal as the optimizing compiler is. By using that, we should be able to optimize more code with this kind of leaner optimizing compiler and do it earlier, and then if that code gets even hotter, then we can titter up again to the fully-optimizing TurboFan. We're hoping this can reduce JavaScript execution time by maybe up to 10% on typical websites.

[00:50:31] JM: I have a few questions about the future. How will WebAssembly change the design of V8?

[00:50:39] RM: WebAssembly is, I guess, already changed the design of V8 in some ways. It operates on slightly different heap. The memory that WebAssembly functions were able to operate on typically are not in this V8 memory managed heap, this garbage collected heap.

They're separate kind of linear arrays of memory. But one interesting thing that might be coming in the future is garbage collected objects in WebAssembly, and this is really to allow WebAssembly to support more dynamic languages. It's kind of focused very much on C++ at the moment, but it would be interesting to have other dynamic languages compiled to WebAssembly. To do that, it would be good if you didn't need to compile the whole runtime for that other language on to WebAssembly, but you could use some parts of the V8 runtime system itself. I think the garbage collected objects would be an interesting project that could bring that, where you have these objects that are managed by V8's garbage collector that can be operated on by WebAssembly code and which could be running a different dynamic language.

[00:51:47] JM: How do you think browser will work differently in 5 years?

[00:51:52] RM: Good question. I think – I mean, we've seen the move to mobile. I think it's going to continue being more of that type of approach where we might even see websites being more dynamic, being able to kind of view something and throw it up on to a desktop machine. This boundary between mobile and desktop, being maybe less, less of a thing and having these websites able to scale more dynamically between kind of mobile experience and a desktop experience.

It'd also be interesting to see if websites kind of become more componentized, maybe different websites working together in an app-like manner. I think that's one of the great strengths of the web, is having this kind of open ecosystem that different companies can all interact and then maybe work together or bring the best parts of their services together, which is something that the web can provide other platforms maybe can't.

[00:52:53] JM: All right. Well, Ross, it's been a real pleasure talking to you and thanks for coming on the show to discuss V8 Lite.

[00:52:58] RM: Thanks for having me on. It's been great to chat too.

[END OF INTERVIEW]

[00:53:10] JM: Apache Cassandra is an open source distributed database that was first created to meet the scalability and availability needs of Facebook, Amazon and Google. In previous episodes of Software Engineering Daily we have covered Cassandra's architecture and its benefits, and we're happy to have Datastax, the largest contributed to the Cassandra project since day one as a sponsor of Software Engineering Daily.

Datastax provides Datastax enterprise, a powerful distribution of Cassandra created by the team that has contributed the most to Cassandra. Datastax enterprise enables teams to develop faster, scale further, achieve operational simplicity, ensure enterprise security and run mixed workloads that work with the latest graph, search and analytics technology all running across hybrid and multi-cloud infrastructure.

More than 400 companies including Cisco, Capital One, and eBay run Datastax to modernize their database infrastructure, improve scalability and security, and deliver on projects such as customer analytics, IoT and e-commerce.

To learn more about Apache Cassandra and Datastax's enterprise, go to datastax.com/sedaily. That's Datastax with an X, D-A-T-A-S-T-A-X, @datastax.com/sedaily. Thank you to Datastax for being a sponsor of Software Engineering Daily. It's a great honor to have Datastax as a sponsor, and you can go to datastax.com/sedaily to learn more.

[END]