Why mobile web apps are slow

09 July 2013 *by* Drew Crawford Published in: iphone, rants Tags:iphone, long articles, native apps, web apps 86 comments

I've had an unusual number of interesting conversations spin out of my previous article documenting that mobile web apps are slow. This has sparked some discussion, both online and IRL. But sadly, the discussion has not been as… *fact-based* as I would like.

So what I'm going to do in this post is try to bring some actual *evidence* to bear on the problem, instead of just doing the shouting match thing. You'll see benchmarks, you'll hear from experts, you'll even read honest-to-God *journal papers* on point. **There are–and this is not a joke–over 100 citations in this blog post.** I'm not going to guarantee that this article will convince you, nor even that absolutely everything in here is totally correct–it's impossible to do in an article this size–but I can guarantee this is the most complete and comprehensive treatment of the idea that many iOS developers have–that mobile web apps are slow and will continue to be slow for the forseeable future.

Now I am going to warn you–this is a very freaking long article, weighing in at very nearly 10k words. That is by design. I have recently come out in favor of articles that are good over articles that are popular. This is my attempt at the former, and my attempt to practice what I have previously preached: that we should incentivize good, evidence-based, interesting discussion and discourage writing witty comments.

I write in part because this topic has been discussed–endlessly–in soundbyte form. This is not Yet Another Bikeshed Article, so if you are looking for that 30-second buzz of "no really, web apps suck!" vs "No they don't!" this is not the article for you. (Go read one of these oh no make it stop can't breathe not HN too I can't do this anymore please just stop so many opinions so few facts I can go on). On the other hand, as best as I can tell, *there is no comprehensive, informed, reasonable discussion of this topic happening anywhere*. It may prove to be a very stupid idea, but this article is my attempt to talk reasonably about a topic that has so far spawned 100% unreasonable flamewar-filled bikeshed discussions. In my defense, I have chosen to believe the problem has more to do with people who *can* discuss better and simply *don't*, than anything to do with the subject matter. I suppose we'll find out.

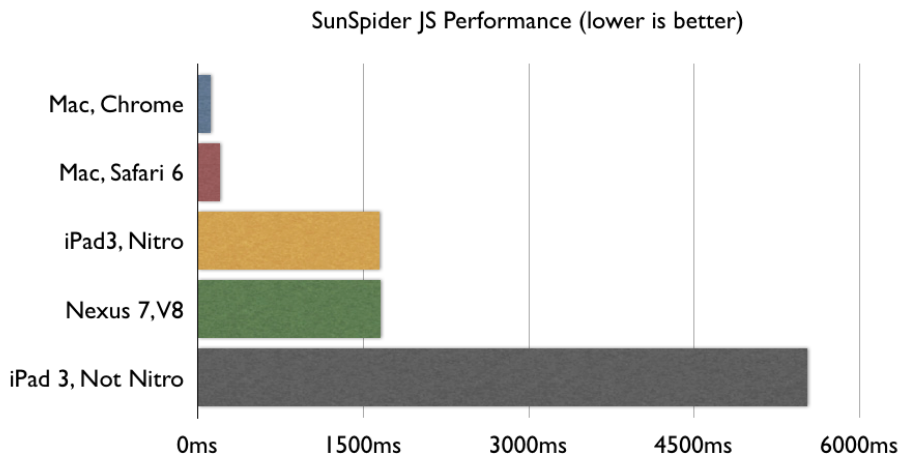
So if you are trying to figure out exactly *what brand of crazy* all your native developer friends are on for continuing to write the evil native applications on the cusp of the open web revolution, or *whatever*, then bookmark this page, make yourself a cup of coffee, clear an afternoon, find a comfy chair, and then we'll both be ready.

A quick review

My previous blog post documented, based on SunSpider benchmarks, that the state of the world, today, is that mobile web apps are slow.

> Now, if what you mean by "web app" is "website with a button or two", you can tell all the fancypants benchmarks like SunSpider to take a hike. But if you mean "light word processing, light photo editing, local storage, and animations between screens" then you don't want to be doing that in a web app on ARM unless you have a death wish.

You should really go read that article, but I will show you the benchmark anyway:



Essentially there are three categories of criticism about this benchmark:

1. The fact that JS is slower than native code is not news: everybody learned this in CS1 when they talked about compiled vs JIT vs interpreted languages. The question is whether it is appreciably slower in some way that actually matters for the kind of software you are writing, and benchmarks like these fail to address that problem one way or the other.
2. Yes JS is slower and yes it matters, but it keeps getting faster and so one day soon we will find ourselves in case #1 where it is no longer appreciably slower, so start investing in JS now.
3. I write Python/PHP/Ruby server-side code and I have no idea what you guys keep going on about. I know that my servers are faster than your mobile devices, but surely if I am pretty comfortable supporting X,000 users using an **actually** interpreted language, you guys can figure out how to support a single user in a language with a high-performance JIT? How hard can it be?

I have the rather lofty goal of refuting all three claims in this article: yes, JS is slow in a way that actually matters, no, it will not get appreciably faster in the near future, and no, your experience with server-side programming does not adequately prepare you to "think small" and correctly reason about mobile performance.

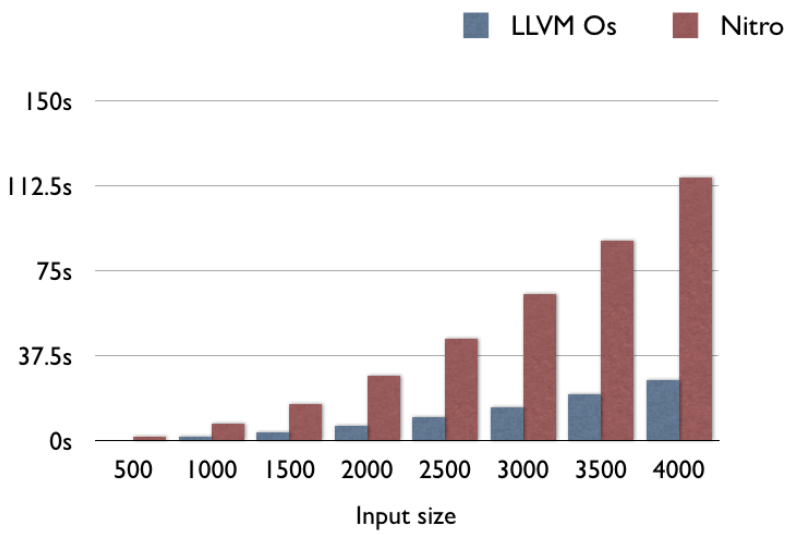
But the real elephant in the room here is that in all these articles on this subject, rarely does anyone actually quantify **how slow** JS is or provide any sort of **actually useful standard of comparison**. (You know… *slow relative to what?*) To correct this, I will develop, in this article, not just **one** useful equivalency for JavaScript performance–but **three of them**. So I'm not only going to argue the "traditional hymns" of "wa wa JS is slow for arbitrary case", but I'm going to quantify exactly how slow it is, and compare it to a wide variety of things in your real-life programming experience so that, when you are faced with your own platform decision, you can do your own back-of-the-napkin math on whether or not JavaScript is feasible for solving your own particular problem.

Okay, but how does JS performance compare to native performance exactly?

It's a good question. To answer it, I grabbed an arbitrary benchmark from The Benchmarks Game. I then found an older C program that does the same benchmark (older since the newer ones have a lot of x86-specific intrinsics). Then benchmarked Nitro against LLVM on my trusty iPhone 4S. All the code is up on GitHub.

Now this is all very arbitrary–but the code you're running in real life is equally arbitrary. If you want a better experiment, go run one. This is just the experiment I ran, because there aren't any other experiments that compare LLVM to Nitro that exist.

Anyway, in this synthetic benchmark, LLVM is consistently 4.5x faster than Nitro:



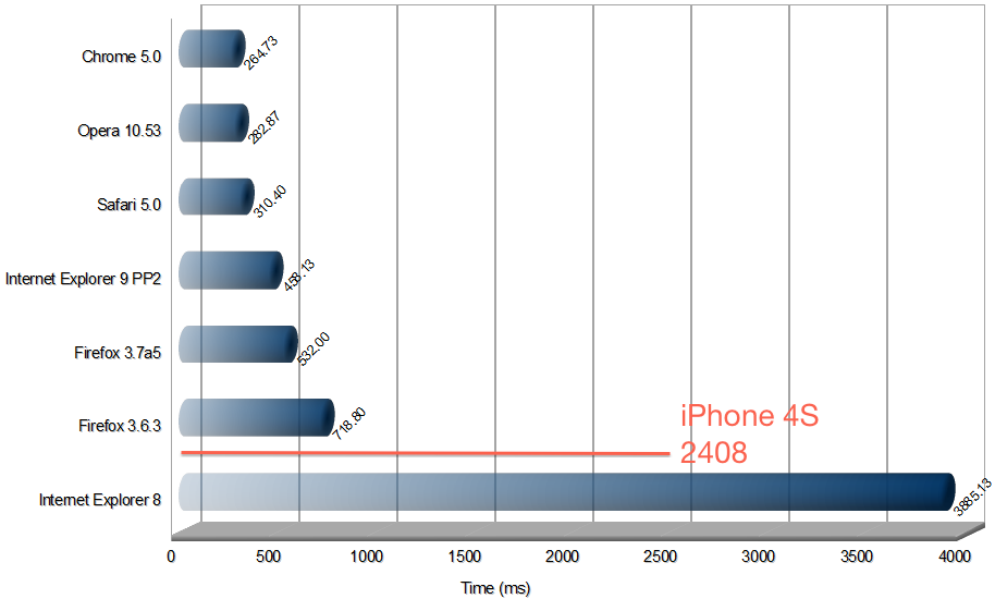
So if you are wondering "How much faster is my CPU-bound function in native code instead of Nitro JS" the answer is about 5x faster. This result is roughly consistent with the Benchmarks Game's results with x86/GCC/V8. They claim that GCC/x86 is generally between 2x and 9x faster than V8/x86. So the result seems in the right ballpark, and also seems consistent no matter if you are on ARM or x86.

But isn't 1/5 performance good enough for anyone?

It's good enough on x86. How CPU-intensive is rendering a spreadsheet, really? It's not really that hard. Problem is, ARM isn't x86.

According to GeekBench, the latest MBP against the latest iPhone is a full factor of 10 apart. So that's okay– spreadsheets really aren't that hard. We can live with 10% performance. But then you want to divide **that** by five? Woah there buddy. Now we're down to 2% of desktop performance. (I'm playing fast-and-loose with the units, but we're dealing with orders of magnitude here. Close enough.)

Okay, but how hard is word processing, *really*? Couldn't we do it on like an m68k with one coprocessor tied behind its back? Well, this is an answerable question. You may not recall, but Google Docs' realtime collaboration was not, in fact, a launch feature. They did a massive rewrite that added it in April 2010. Let's see what browser performance looked like in 2010.



What should be plainly obvious from this chart is that the iPhone 4S is not at all competitive with web browsers around the time that Google Docs did real-time collaboration. Well, it's competitive with IE8. Congratulations on that.

Let's look at another serious JavaScript application: Google Wave. Wave never supported IE8–according to Google–because it was too slow.

Notice how all these browsers bench faster than the iPhone 4S?

Notice how all the supported browsers bench below 1000, and the one that scores 3800 is excluded for being too slow? The iPhone benches 2400. It, just like IE8, isn't fast enough to run Wave.

Just to be clear: **is possible to do real-time collaboration on on a mobile device**. It just isn't possible to do it **in JavaScript**. The performance gap between native and web apps is comparable to the performance gap between FireFox and IE8, which is **too large a gap for serious work**.

But I thought V8 / modern JS had near-C performance?

It depends on **what you mean by "near"**. If your C program executes in 10ms, then a 50ms JavaScript program would be "near-C" speed. If your C program executes in 10 seconds, a 50-second JavaScript program, for most ordinary people would probably **not** be near-C speed.

The hardware angle

But a factor of 5 is okay **on x86**, because x86 is ten times faster than ARM just to start with. You have a lot of headroom. The solution is obviously just to make ARM 10x faster, so it is competitive with x86, and then we can get desktop JS performance without doing any work!

Whether or not this works out kind of hinges on your faith in Moore's Law in the face of trying to power a chip on a 3-ounce battery. I am not a hardware engineer, but I once worked for a major semiconductor company, and the people there tell me that these days performance is mostly a function of your *process* (e.g., the thing they measure in "nanometers"). The iPhone 5's impressive performance is due in no small part to a process shrink from 45nm to 32nm — a reduction of about a third. But to do it again, Apple would have to shrink to a 22nm process.

Just for reference, Intel's Bay Trail–the x86 Atom version of 22nm–*doesn't currently exist*. And Intel had to invent a whole new kind of transistor since the ordinary kind doesn't work at 22nm scale. Think they'll license it to ARM? Think again. There are only a handful of 22nm fabs that people are even *seriously thinking about building* in the world, and most of them are controlled by Intel.

In fact, ARM seems on track to do a 28nm process shrink in the next year or so (watch the A7), and meanwhile Intel is on track to do 22nm and maybe even 20nm just a little further out. On purely a hardware level, it seems much more likely to me that an x86 chip with x86-class performance will be put in a smartphone long before an ARM chip with x86-class performance can be shrunk. So Moore's Law might be right after all, but it is right in a way that would require the entire mobile ecosystem to transition to x86. It's not entirely impossible–it's been done once before. But it was done at a time when yearly sales were around a million units, and now they are selling 62 million per quarter. It was done with an off-the-shelf virtualization environment that could emulate the old architecture at about 60% speed, meanwhile the performance of today's hypothetical research virtualization systems for optimized (O3) ARM code are closer to 27%.

If you believe JavaScript performance is going to get there eventually, really the hardware path is the best path. Either Intel will have a viable iPhone chip in 5 years (likely) and Apple will switch (unlikely), or perhaps ARM will sort themselves out over the next decade. (Go talk to 10 hardware engineers to get 10 opinions on the viability of that.) But a decade is a long time, from my chair, for something that *might pan out*.

I'm afraid my knowledge of the hardware side runs out here. What I *can* tell you is this: if you want to believe that ARM will close the gap with x86 in the next 5 years, the first step is to find somebody who works on ARM or x86 (e.g., the sort of person who would actually know) to agree with you. I have consulted many such qualified engineers for this article, and they have all declined to take the position on record. This suggests to me

that the position is not any good.

The software angle

Here is where a lot of competent software engineers stumble. The thought process goes like this–JavaScript has gotten faster! It will continue to get faster!

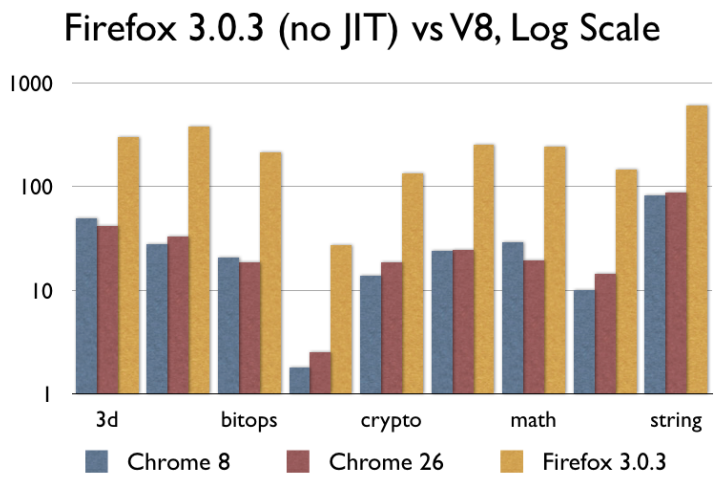
The first part is true. JavaScript has gotten a **lot** faster. But we're now at Peak JavaScript. It doesn't get much faster from here.

Why? Well the first part is that most of the improvements to JavaScript over its history have actually been of the **hardware** sort. Jeff Atwood writes:

> I found that the performance of JavaScript improved a hundredfold between 1996 and 2006. If Web 2.0 is built on a backbone of JavaScript, it's largely possible only because of those crucial Moore's Law performance improvements.

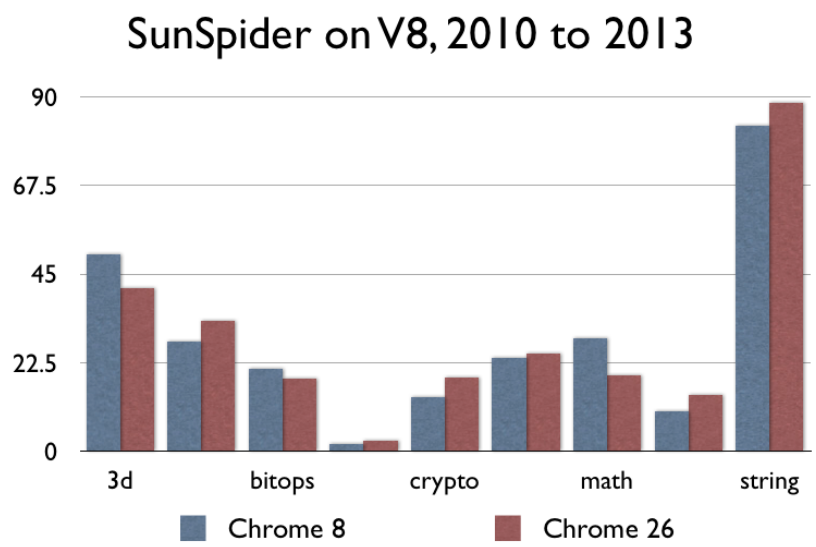
If we attribute JS's speedup to hardware generally, JS's (hardware) performance improvement **does not predict future software improvement**. This is why, if you want to believe that JS is going to get faster, by far the most likely way is by the hardware getting faster, because that is what the historical trend says.

What about JITs though? V8, Nitro/SFX, TraceMonkey/IonMonkey, Chakra, and the rest? Well, they were kind of a big deal when they came out–although not as big of a deal as you might think. V8 was released in September 2008. I dug up a copy of Firefox 3.0.3 from around the same time:



Don't get me wrong, a 9x improvement in performance is nothing to sneeze at–after all, it's nearly the difference between ARM and x86. That said, the performance between Chrome 8 and Chrome 26 is a flatline, because nothing terribly important has happened since 2008. The other browser vendors have caught up–some slower, some faster–but nobody has really improved the speed of actual CPU code since.

Is JavaScript improving?

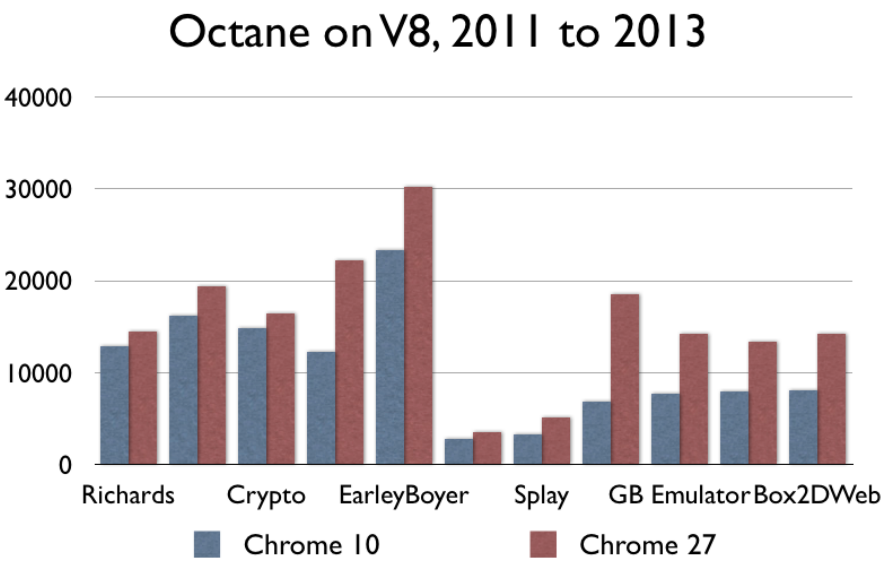


Here's Chrome v8 on my Mac (the earliest one that still ran, Dec 2010.) Now here's v26.

Can't spot the difference? That's because there isn't one. **Nothing terribly important has happened to CPU-bound JavaScript lately.**

If the web feels faster to you than it did in 2010, that is probably because you're running a faster computer, but it has nothing to do with improvements to Chrome.

Update Some smart people have pointed out that SunSpider isn't a good benchmark these days (but have declined to provide any actual numbers or anything). In the interests of having a reasonable conversation, I ran Octane (a Google benchmark) on some old versions of Chrome, and it does show some improvement:



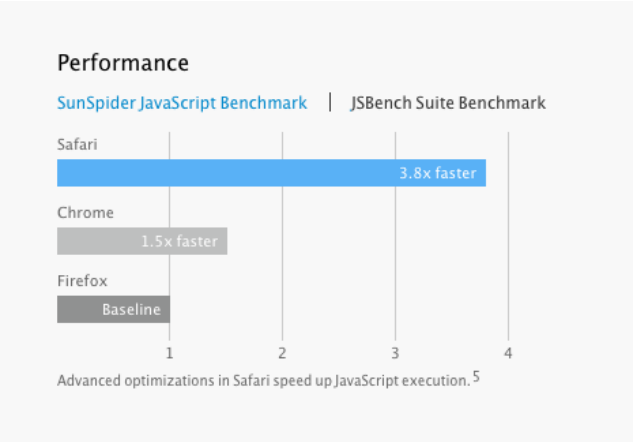
In my opinion, this magnitude of performance gain over this period is much too small to support the claim that JS will close the gap in any reasonable amount of time. However, I think it's fair to say that I overstated the case a bit–*something* is happening in CPU-bound JavaScript. But to me, these numbers confirm the larger hypothesis: these gains are not the order-of-magnitude that will close the gap with native code, in any reasonable amount of time. You need to get to 2x-9x across the board to compete with LLVM. These improvements are good, but they're not that good. **End update**

The thing is, JITing JavaScript was a 60-year old idea with 60 years of research, and literally thousands of implementations for every conceivable programming language demonstrating that it was a good idea. But now

that we've done it, we've run out of 60-year-old ideas. That's all, folks. Show's over. Maybe we can grow another good idea in the next 60 years.

But Safari is supposedly faster than before?

But if this is all true, how come we keep hearing about all the great performance improvements in JavaScript? It seems every other week, somebody is touting huge speedups in some benchmark. Here is Apple claiming a staggering 3.8x speedup on JSBench:



]56 Is Safari 7 3.8x faster than the other guys?

Perhaps conveniently for Apple, this version of Safari is currently under NDA, so nobody is able to publish independent numbers on Safari performance one way or the other. But let me make some observations on this kind of claim that's purely on the basis of publicly available information.

I find it interesting, first, that Apple's public claims on JSBench are much higher than their claims for traditional benchmarks like SunSpider. Now JSBench has some cool names behind it including Brenden Eich, the creator of JavaScript. But unlike traditional benchmarks, the way JSBench works isn't by writing a program that factors integers or something. Instead, JSBench automatically scrapes whatever Amazon, Facebook, and Twitter serve up, and builds benchmarks out of that. If you are writing a web browser that (let's be honest) most people use to browse Facebook, I can see how having a benchmark that's literally Facebook is very useful. On the other hand, if you are writing a spreadsheet program, or a game, or an image filter application, it seems to me that a traditional benchmark with e.g. integer arithmetic and md5 hashing is going to be much more predictive for you than seeing how fast Facebook's analytics code can run.

The other important fact is that an improvement on SunSpider, as Apple claims, does not necessarily mean anything else improves. In the very paper that introduces Apple's preferred benchmark, Eich et al write the following:

> The graph clearly shows that, according to SunSpider, the performance of Firefox improved over 13× between version 1.5 and version 3.6. Yet when we look at the performance improvements on amazon they are a more modest 3×. And even more interestingly, in the last two years, gains on amazon have flattened. Suggesting that some of the optimizations that work well on Sun Spider do little for amazon. [sic]

In this very paper, the creator of JavaScript and one of the top architects for Mozilla openly admits that nothing at all has happened to the performance of Amazon's JavaScript in two years, and nothing terribly exciting has *ever* happened. This is your clue that the marketing guys have oversold things just a bit over the years.

(They go on to argue, essentially, that benchmarking Amazon is a better predictor for running Amazon than benchmarking SunSpider [uh... obvious...], and is therefore good to do for web browsers which people use to visit Amazon. But none of this will help you write a photo processing application.)

But at any rate, what I can tell you, from publicly available information, is that Apple's claims of 3.8x faster whatever does not necessarily translate into anything useful to you. I can also tell you that if I had benchmarks that refuted Apple's claims of beating Chrome, I would not be allowed to publish them.

So let's just conclude this section by saying that just because somebody has a bar chart that shows their web browser is faster does not necessarily mean JS as a whole is getting any faster.

But there is a bigger problem.

Not designed for performance

This is from Herb Sutter, one of the big names in modern C++:

> This is a 199x/200x meme that's hard to kill – "just wait for the next generation of (JIT or static) compilers and then managed languages will be as efficient." Yes, I fully expect C# and Java compilers to keep improving – both JIT and NGEN-like static compilers. But no, they won't erase the efficiency difference with native code, for two reasons. First, JIT compilation isn't the main issue. The root cause is much more fundamental: **Managed languages made deliberate design tradeoffs to optimize for programmer productivity even when that was fundamentally in tension with, and at the expense of, performance efficiency**… In particular, managed languages chose to incur costs even for programs that don't need or use a given feature; the major examples are assumption/reliance on always-on or default-on garbage collection, a virtual machine runtime, and metadata. But there are other examples; for instance, managed apps are built around virtual functions as the default, whereas C++ apps are built around inlined functions as the default, and an ounce of inlining prevention is worth a pound of devirtualization optimization cure.

This quote was endorsed by Miguel de Icaza of Mono, who is on the very short list of "people who maintain a major JIT compiler". He said:

> This is a pretty accurate statement on the difference of the mainstream VMs for managed languages (.NET, Java and Javascript). Designers of managed languages have chosen the path of safety over performance for their designs.

Or, you could talk to Alex Gaynor, who maintains an optimizing JIT for Ruby and contributes to the optimizing JIT for Python:

> It's the curse of these really high-productivity dynamic languages. They make creating hash tables incredibly easy. And that's an incredibly good thing, because I think C programmers probably underuse hash tables, because they're a pain. For one you don't have one built in. For two, when you try to use one, you just hit pain left and right. By contrast, Python, Ruby, JavaScript people, we overuse hashtables because they're so easy… And as a result, people don't care…

Google seems to think that JavaScript is facing a performance wall:

> Complex web apps–the kind that Google specializes in–are struggling against the platform and working with a language that cannot be tooled and has inherent performance problems.

Lastly, hear it from the horse's mouth. One of my readers pointed me to this comment by Brendan Eich. You know, the guy who invented JavaScript.

> One thing Mike didn't highlight: **get a simpler language**. Lua is much simpler than JS. This means you can make a simple interpreter that runs fast enough to be balanced with respect to the trace-JITted code [unlike with JS].

and a little further down:

> On the differences between JS and Lua, you can say it's all a matter of proper design and engineering (what isn't?), but intrinsic complexity differences in degree still cost. You can push the hard cases off the hot paths, certainly, but they take their toll. **JS has more and harder hard cases than Lua.** One example: Lua (without explicit metatable usage) has nothing like JS's

prototype object chain.

Of the people who actually do relevant work: the view that JS in particular, or dynamic languages in general, will catch up with C, is *very much the minority view*. There are a few stragglers here and there, and there is also no real consensus what to do about it, or if anything should be done about it at all. But as to the question of whether, from a language perspective, in general, the JITs will catch up–the answer from the people working on them is "no, not without changing either the language or the APIs."

But there is an even bigger problem.

All about garbage collectors

You see, the CPU problem, and all the CPU-bound benchmarks, and all the CPU-bound design decisions–that's really only half the story. The other half is memory. And it turns out, the memory problem is so vast, that the whole CPU question is just the tip of the iceberg. In fact, arguably, that entire CPU discussion is a red herring. **What you are about to read should change the whole way you think about mobile software development**.

In 2012, Apple did a curious thing (well, unless you are John Gruber and saw it coming). They pulled garbage collection out of OSX. Seriously, go read the programming guide. It has a big fat "(Not Recommended)" right in the title. If you come from Ruby, or Python, or JavaScript, or Java, or C#, or really any language since the 1990s, this should strike you as **really odd**. But it probably doesn't affect you, because you probably don't write ObjC for Mac, so meh, click the next link on HN. But still, it seems *strange*. After all, GC has been around, it's been **proven**. Why in the world would you *deprecate* it? Here's what Apple had to say:

> We feel so strongly about ARC being the right approach to memory management that we have decided to deprecate Garbage Collection in OSX. - Session 101, Platforms Kickoff, 2012, ~01:13:50

The part that the transcript doesn't tell you is that **the audience broke out into applause upon hearing this statement**. Okay, now this is **really freaking weird**. You mean to tell me that there's a room full of developers **applauding the return** to the pre-garbage collection chaos? Just *imagine* the pin drop if Matz announced the deprecation of GC at RubyConf. And these guys are **happy** about it? Weirdos.

Rather than write off the Apple fanboys as a cult, this *very odd* reaction should clue you in **that there is more going on here than meets the eye**. And this "more going on" bit is the subject of our next line of inquiry.

So the thought process goes like this: **Pulling a working garbage collector out of a language is totally crazy, amirite?** One simple explanation is that perhaps ARC is just a special Apple marketing term for a fancypants kind of garbage collector, and so what these developers are, in fact applauding–is an *upgrade* rather than a *downgrade*. In fact, this is a belief that a lot of iOS noobs have.

ARC is not a garbage collector

So to all the people who think ARC is some kind of garbage collector, I just want to beat your face in with the following Apple slide:

This has nothing to do with the similarly-named garbage collection algorithm. It isn't GC, it isn't anything like GC, it performs nothing like GC, it does not have the power of GC, it does not break retain cycles, it does not sweep anything, it does not scan anything. Period, end of story, not garbage collection.

The myth somehow grew legs when a lot of the documentation was under NDA (but the spec was available, so that's no excuse) and as a result the blogosphere has widely reported it to be true. It's not. Just stop.

GC is not as feasible as your experience leads you to believe

So here's what Apple has to say about ARC vs GC, when pressed:

> At the top of your wishlist of things we could do for you is bringing garbage collection to iOS. And that is exactly what we are not going to do… Unfortunately garbage collection has a suboptimal impact on performance. Garbage can build up in your applications and increase the high water mark of your memory usage. And the collector tends to kick in at undeterministic times which can lead to very high CPU usage and stutters in the user experience. And that's why GC has not been acceptable to us on our mobile platforms. In comparison, manual memory management with retain/release is harder to learn, and quite frankly it's a bit of a pain in the ass. But it produces better and more predictable performance, and that's why we have chosen it as the basis of our memory management strategy. Because out there in the real world, high performance and stutter-free user experiences are what matters to our users. ~Session 300, Developer Tools Kickoff, 2011, 00:47:49

But that's totally crazy, amirite? Just for starters:

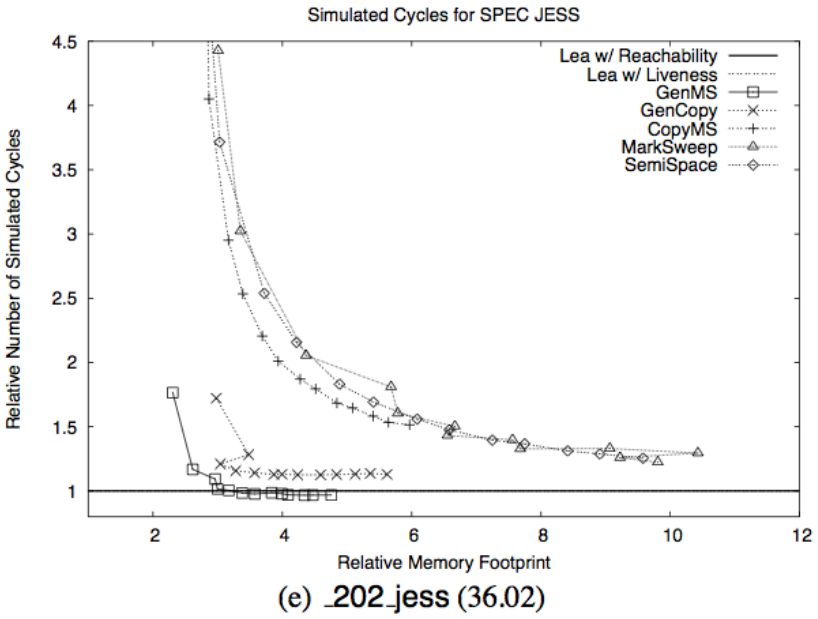
1. It probably flies in the face of your entire career of experiencing the performance impact of GCed languages on the desktop and server
2. Windows Mobile, Android, MonoTouch, and the whole rest of them seem to be getting along fine with GC

So let's take them in turn.

GC on mobile is not the same animal as GC on the desktop

I know what you're thinking. You've been a Python developer for N years. It's 2013. Garbage collection is a totally solved problem.

Here is the paper you were looking for. **Turns out it's not so solved**:



If you remember nothing else from this blog post, *remember this chart*. The Y axis is time spent collecting garbage. The X axis is "relative memory footprint". Relative to what? Relative to the **minimum amount of memory required**.

What this chart says is "As long as you have about 6 times as much memory as you really need, you're fine. **But woe betide you if you have less than 4x the required memory**." But don't take my word for it:

> In particular, when garbage collection has five times as much memory as required, its runtime performance matches or slightly exceeds that of explicit memory management. However, garbage collection's performance degrades substantially when it must use smaller heaps. With three times as much memory, it runs 17% slower on average, and with twice as much memory, it runs 70% slower. Garbage collection also is more susceptible to paging when physical memory is scarce. In such conditions, all of the garbage collectors we examine here **suffer order-of-magnitude performance penalties relative to explicit memory management**.

Now let's compare with explicit memory management strategies:

> These graphs show that, for reasonable ranges of available memory (but not enough to hold the entire application), **both explicit memory managers substantially outperform all of the garbage collectors**. For instance, pseudoJBB running with 63MB of available memory and the Lea allocator completes in 25 seconds. With the same amount of available memory and using GenMS, it takes **more than ten times longer to complete** (255 seconds). We see similar trends across the benchmark suite. The most pronounced case is 213 javac: at 36MB with the Lea allocator, total execution time is 14 seconds, while with GenMS, total execution time is 211 seconds, over a 15-fold increase.

The ground truth is that in a memory constrained environment garbage collection performance degrades exponentially. If you write Python or Ruby or JS that runs on desktop computers, it's possible that **your entire experience is in the right hand of the chart**, and you can go your whole life without ever experiencing a slow garbage collector. Spend some time on the left side of the chart and see what the rest of us deal with.

How much memory is available on iOS?

It's hard to say exactly. The physical memory on the devices vary pretty considerably–from 512MB on the iPhone 4 up to 1GB on the iPhone 5. But a lot of that is reserved for the system, and still more of it is reserved for multitasking. Really the only way to find out is to try it under various conditions. Jan Ilavsky helpfully wrote a utility to do it, but it seems that nobody publishes any statistics. That changes today.

Now it's important to do this under "normal" conditions (whatever that means), because if you do it from a fresh boot or back-to-back, you will get better results since you don't have pages open in Safari and such. So I literally grabbed devices under the "real world" condition of lying around my apartment somewhere to run this benchmark.

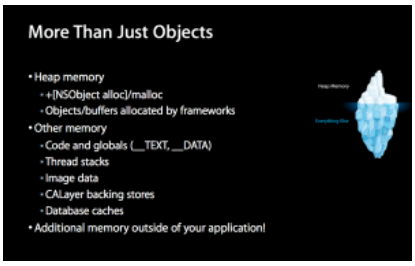
You can click through to see the detailed results but essentially on the iPhone 4S, you start getting warned around 40MB and you get killed around 213MB. On the iPad 3, you get warned around 400MB and you get killed around 550MB. Of course, these are just my numbers–if your users are listening to music or running things in the background, you may have considerably less memory than you do in my results, but this is a start. This seems like a lot (213mb should be enough for everyone, right?) but as a practical matter it isn't. For example, the iPhone 4S snaps photos at 3264×2448 resolution. That's over 30 megabytes of bitmap data per photo. That's a warning for having **just two photos in memory** and you get killed for having **7 photos in RAM**. Oh, you were going to write a for loop that iterated over an album? Killed.

It's important to emphasize too that as a practical matter you often have the same photo in memory multiple places. For example, if you are taking a photo, you have 1) The camera screen that shows you what the camera sees, 2) the photo that the camera actually took, 3) the buffer that you're trying to fill with compressed JPEG data to write to disk, 4) the version of the photo that you're preparing for display in the next screen, and 5) the version of the photo that you're uploading to some server.

At some point it will occur to you that keeping 30MB buffers open to display a photo thumbnail is a really bad idea, so you will introduce 6) the buffer that is going to hold a smaller photo suitable for display in the next screen, 7) the buffer that resizes the photo in the background because it is too slow to do it in the foreground. And then you will discover that you really need five different sizes, and thus begins the slow descent into madness. It's not uncommon to hit memory limits dealing just with a **single photograph** in a real-world application. But don't take my word for it:

> The worst thing that you can do as far as your memory footprint is to cache images in memory. When an image is drawn into a bitmap context or displayed to a screen, we actually have to decode that image into a bitmap. That bitmap is 4 bytes per pixel, no matter how big the original image was. And as soon as we've decoded it once, that bitmap is attached to the image object, and will then persist for the lifetime of the object. So if you're putting images into a cache, and they ever get displayed, you're now holding onto that entire bitmap until you release it. So never put UIImages or CGImages into a cache, unless you have a very clear (and hopefully very short-term) reason for doing so. - Session 318, iOS Performance In Depth, 2011

Don't even take his word for it! The amount of memory you allocate yourself is just the tip of the iceberg. No honest, here's the actual iceberg slide from Apple. Session 242, iOS App Performance – Memory, 2012:



And you're burning the candle from both ends. Not only is it much harder to deal with photos if you have 213MB of usable RAM than it is on a desktop. But there is also a lot more demand to write photo-processing applications, **because your desktop does not have a great camera attached to it that fits in your pocket**.

Let's take another example. On the iPad 3, you are driving a display that probably has more pixels in it than the display on your desktop (it's between 2K and 4K resolution, in the ballpark with pro cinema). Each frame that you show on that display is a 12MB bitmap. If you're going to be a good memory citizen you can store roughly 45 frames of uncompressed video or animation buffer in memory at a time, which is about 1.5 seconds at 30fps, or .75 seconds at the system's 60Hz. Accidentally buffer a second of full-screen animation? App killed. And it's worth pointing out, the latency of AirPlay is 2 seconds, so for any kind of media application, you are actually **guaranteed to not have enough memory**.

And we are in roughly the same situation here that we are in with the multiple copies of the photos. For example, Apple says that "Every UIView is backed with a CALayer and images as layer contents remain in memory as long as the CALayer stays in the hierarchy." What this means, essentially, is that there can be many intermediate renderings–essentially copies–of your view hierarchy that are stored in memory.

And there are also things like clipping rects, and backing stores. It's a remarkably efficient architecture as far as CPU time goes, but it achieves that performance essentially at the cost of gobbling as much memory as possible. iOS is *not architected to be low-memory–it's optimized to be fast*. Which just doesn't mix with garbage collection.

We are also in the same situation about burning the candle from both ends. Not only are you in an incredibly memory-constrained environment for doing animations. But there is also a huge demand to do super high-quality video and animation, because this awful, memory-contrained environment is literally the only form factor in which a consumer-class pro-cinema-resolution display can be purchased. If you want to write software that runs on a comparable display, you have to convince somebody to shell out $700 just for the monitor. Or, they could spend $500, and get an iPad, with the computer already built in.

But then how does Mono/Android/Windows Mobile do it?

There are really two answers to this question. The first answer we can see from the chart. If you find yourself with 6 times as much memory as you need, garbage collection is actually going to be pretty fast. So for example, if you are writing a text editor, you might realistically be able to do everything you want in only 35MB, which is 1/6th the amount of memory before my iPhone 4S crashes. And you might write that text editor in Mono, see reasonable performance, and conclude from this exercise that garbage collectors are perfectly fine for this task, and you'd be right.

Yeah but Xamarin has flight simulators in the showcase! So clearly, the idea that garbage collectors are infeasible for larger apps flies in the face of real-life, large, garbage-collected mobile apps. Or does it?

> What sort of problems do you have to overcome when developing/maintaining this game?
> "Performance has been a big issue and continues to be one of the biggest problems we have across platforms. The original Windows Phone devices were pretty slow and we had to spend a lot of time optimising the app to get a descent frame rate. Optimisations were done both on the flight sim code as well as the 3D engine. The biggest bottlenecks were **garbage collection** and the weaknesses of the GPU."

Totally unprompted, the developers bring up garbage collection as the biggest bottleneck. When the people in your showcase are complaining, that would be **a clue**. But maybe Xamarin is an outlier. Let's check in on the Android developers:

> Now, keep in mind these are running my Galaxy Nexus — not a slow device by any stretch of the imagination. *But check out the rendering times!* While I was able to render these images in a couple of hundred milliseconds on my desktop, they were taking almost **two orders of magnitude longer on the device**! Over 6 seconds for the "inferno"? Crazy! … That's 10-15 times the garbage collector would run to generate one image.

Another one:

> If you want to process camera images on Android phones for real-time object recognition or content based Augmented Reality **you probably heard about** the Camera Preview Callback memory Issue. Each time your Java application gets a preview image from the system a new chunk of memory is allocated. When this memory chunk gets freed again by the Garbage Collector the system freezes for 100ms-200ms. This is especially bad if the system is under heavy load (I do object recognition on a phone – hooray it eats as much CPU power as possible). If you browse through Android's 1.6 source code you realize that this is only because the wrapper (that protects us from the native stuff) allocates a new byte array each time a new frame is available. **Build-in native code can, of course, avoid this issue**.

Or, we can consult Stack Overflow:

> I'm performance tuning interactive games in Java for the Android platform. Once in a while there is a hiccup in drawing and interaction for garbage collection. Usually it's less than one tenth of a second, but sometimes it can be as large as 200ms on very slow devices… If I ever want trees or hashes in an inner loop I know that I need to be careful or even reimplement them instead of using the Java Collections framework since I can't afford the extra garbage collection.

Here's the "accepted answer", 27 votes:

> I've worked on Java mobile games… The best way to avoid GC'ing objects (which in turn shall trigger the GC at one point or another and shall kill your game's perfs) is simply to avoid creating them in your main game loop in the first place. There's no "clean" way to deal with this… **Manual tracking of objects, sadly**. This how it's done on most current well-performing Java games that are out on mobile devices.

Let's check in with Jon Perlow of Facebook:

> GC is a huge performance problem for developing smooth android applications. At Facebook, one of the biggest performance problems we deal with is GCs pausing the UI thread. When dealing with lots of Bitmap data, GCs are frequent and hard to avoid. A single GC often results in dropped frames. Even if a GC only blocks the UI thread for a few milliseconds, it can significantly eat into the 16ms budget for rendering a frame.

Okay, let's check in with a Microsoft MVP:

> Normally your code will complete just fine within the 33.33 milliseconds, thereby maintaining a nice even 30FPS… However when the GC runs, it eats into that time. If you've kept the heap nice and simple …, the GC will run nice and fast and this likely won't matter. But keeping a simple heap that the GC can run through quickly is a **difficult programming task that requires a lot of planning and/or rewriting and even then isn't fool proof** (sometimes you just have a lot of stuff on the heap in a complex game with many assets). Much simpler, assuming you can do it, is to limit or **even eliminate all allocations during gameplay**.

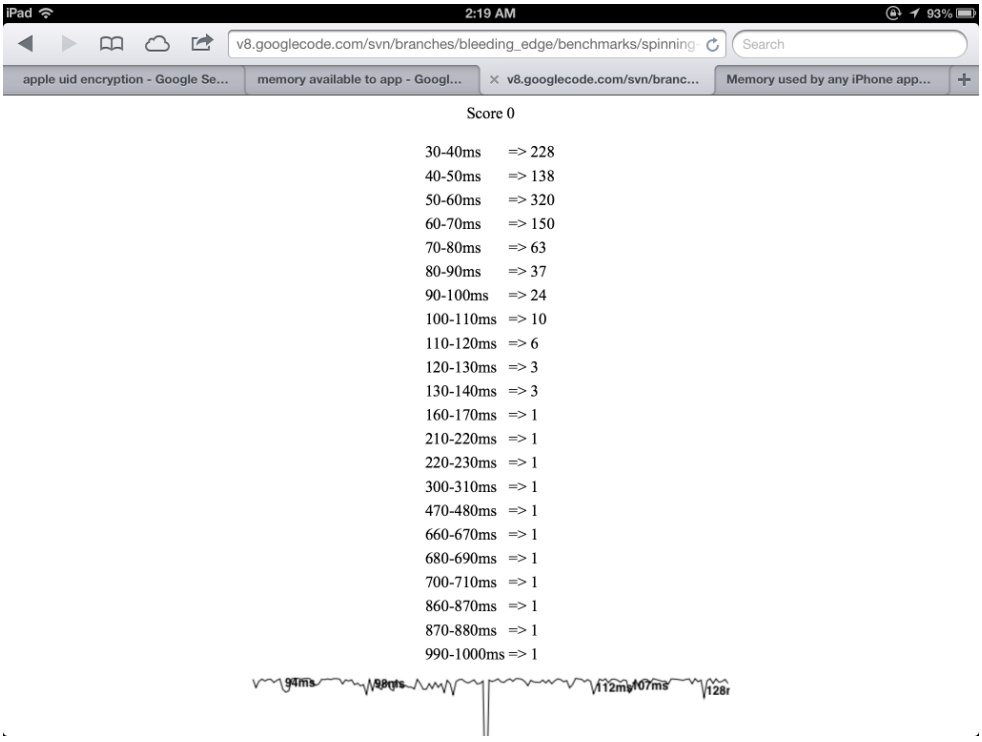
With garbage collection, **the winning move is not to play**. A weaker form of this "the winning move is not to play" philosophy is embedded in the official Android documentation:

> Object creation is never free. A generational garbage collector with per-thread allocation pools for temporary objects can make allocation cheaper, but allocating memory is always more expensive than not allocating memory. As you allocate more objects in your app, you will force a periodic garbage collection, creating little "hiccups" in the user experience. The concurrent garbage collector introduced in Android 2.3 helps, but unnecessary work should always be avoided. Thus, you should avoid creating object instances you don't need to… Generally speaking, avoid creating short-term temporary objects if you can. Fewer objects created mean less-frequent garbage collection, which has a direct impact on user experience.

Still not convinced? Let's ask an actual Garbage Collection engineer. Who writes garbage collectors. For mobile devices. For a living. You know, the person whose *job it is* to know this stuff.

> However, with WP7 the capability of the device in terms of CPU and memory drastically increased. Games and large Silverlight applications started coming up which used close to 100mb of memory. As memory increases the number of references those many objects can have also **increases exponentially**. In the scheme explained above the GC has to traverse each and every object and their reference to mark them and later remove them via sweep. So the GC time also increases drastically and becomes a function of the net workingset of the application. This results in very large pauses in case of large XNA games and SL applications which finally manifests as long startup times (as GC runs during startup) or glitches during the game play/animation.

Still not convinced? Chrome has a benchmark that measures GC performance. Let's see how it does…



That is a **lot** of GC pauses. Granted, this is a stress test–but still. You really want to wait a full second to render that frame? I think you're nuts.

Look, that's a lot of quotes, I'm not reading all that. Get to the point.

Here's the point: memory management is **hard** on mobile. iOS has formed a culture around doing most things manually and trying to make the compiler do some of the easy parts. Android has formed a culture around improving a garbage collector that they try very hard not to use in practice. But either way, everybody spends a lot of time thinking about memory management when they write mobile applications. There's just no substitute for **thinking about memory**. Like, a ***lot***.

When JavaScript people or Ruby people or Python people hear "garbage collector", they understand it to mean "*silver bullet* garbage collector." They mean "garbage collector that frees me from thinking about managing memory." But there's no silver bullet on mobile devices. **Everybody thinks about memory on mobile, whether they have a garbage collector or not**. The only way to get "silver bullet" memory management is the same way we do it on the desktop–by having 10x more memory than your program really needs.

JavaScript's whole design is based around not worrying about memory. Ask the Chromium developers:

> is there any way to force the chrome js engine to do Garbage Collection? In general, no, by design.

The EMCAScript specification does not contain the word "allocation", the only reference to "memory" essentially says that the entire subject is "host-defined".

The EMCA 6 wiki has several pages of draft proposal that boil down to, and I am not kidding,

> "the garbage collector MUST NOT collect any storage that then becomes needed to continue correct execution of the program… All objects which are not transitively strongly reachable from roots SHOULD eventually be collected, if needed to prevent the program execution from failing due to memory exhaustion."

Yes, they actually are thinking about specifying this: a garbage collector should not collect things that it should not collect, but it should collect things it needs to collect. Welcome to tautology club. But perhaps more relevant to our purpose is this quote:

> However, there is no spec of how much actual memory any individual object occupies, **nor is there likely to be**. Thus we never have any guarantee when any program may exhaust its actual raw memory allotment, so all lower bound expectations are not precisely observable.

In English: the philosophy of JavaScript (to the extent that it has any philosophy) is that **you should not be able to observe what is going on in system memory, full stop**. This is so **unbelievably** out of touch with how real people write mobile applications, I can't even find the words to express it to you. I mean, in iOS world, we don't believe in garbage collectors, and we think the Android guys are nuts. I suspect that the Android guys think the iOS guys are nuts for manual memory management. But you know what the two, cutthroat opposition camps can agree about? **The JavaScript folks are really nuts**. There is absolutely zero chance that you can write reasonable mobile code without worrying about what is going on in system memory, in some capacity. None. And so putting the whole question of SunSpider benchmarks and CPU-bound stuff fully aside, we arrive at the conclusion that **JavaScript, at least as it stands today, is fundamentally opposed to the think-about-memory-philosophy that is absolutely required for mobile software development**.

As long as people keep wanting to push mobile devices into these video and photo applications where desktops haven't even been, and as long as mobile devices have a lot less memory to work with, the problem is just intractable. You need *serious, formal memory management guarantees* on mobile. And JavaScript, **by design, refuses** to provide them.

Suppose it did

Now you might say, "Okay. The JS guys are off in Desktop-land and are out of touch with mobile developers' problems. But suppose they were convinced. Or, suppose somebody who actually *was* in touch with mobile developers' problems forked the language. Is there something that can be done about it, in theory?"

I am not sure if it is solvable, but I can put some bounds on the problem. There is another group that has tried to fork a dynamic language to meet the needs of mobile developers–and it's called RubyMotion.

So these are smart people, who know a lot about Ruby. And these *Ruby* people decided that garbage collection for their fork was A Bad Idea. (Hello GC advocates? Can you hear me?). So they have a thing that is a lot like ARC that they use instead, that they have sort of grafted on to the language. Turns out it doesn't work:

> Summary: lots of people are experiencing memory-related issues that are a result of RM-3 or possibly some other difficult-to-identify problem with RubyMotion's memory management, and they're coming forward and talking about them.

Ben Sheldon weighs in:

> It's not just you. I'm experiencing these memory-related types of crashes (like SIGSEGV and SIGBUS) with about 10-20% of users in production.

There's some skepticism about whether the problem is tractable:

> I raised the question about RM-3 on the recent Motion Meetup and Laurent/Watson both responded (Laurent on camera, Watson in IRC). Watson mentioned that RM-3 is the toughest bug to fix, and Laurent discussed how he tried a few approaches but was never happy with them. Both devs are smart and strong coders, so I take them at their word.

There's some skepticism about whether the compiler can even solve it in theory:

> For a long while, I believed blocks could simply be something handled specifically by the compiler, namely the contents of a block could be statically analyzed to determine if the block references variables outside of its scope. For all of those variables, I reasoned, the compiler could simply retain each of them upon block creation, and then release each of them upon block destruction. This would tie the lifetime of the variables to that of the block (not the 'complete' lifetime in some cases, of course). One problem: instance_eval. The contents of the block may or may not be used in a way you can expect ahead of time.

RubyMotion also has the opposite problem: it leaks memory. And maybe it has other problems. Nobody really knows if the crashes and leaks have 2 causes, or 200 causes. All we know is that people report both. A lot.

So anyway, here's where we're at: some of the best Ruby developers in the world have forked the language specifically for use on mobile devices, and they have designed a system that both crashes and leaks, which is the complete set of memory errors that you could possibly experience. So far they have not been able to do anything about it, although they have undoubtedly been trying very hard. Oh, and they are reporting that they "personally tried a few times to fix it, but wasn't able to come with a good solution that would also perserve performance."

I'm not saying forking JavaScript to get reasonable memory performance is impossible. I'm just saying there's a lot of evidence that suggests the problem is *really hard*.

Okay but what about asm.js

asm.js is kind of interesting because it provides a JavaScript model that doesn't, strictly speaking, rely on garbage collection. So in theory, with the right web browser, with the right APIs, it could be *okay*. The question is, "will we get the right browser?"

Mozilla is obviously sold on the concept, being the authors of the technology, and their implementation is landing later this year. Chrome's reaction has been more mixed. It obviously competes with Google's other proposals– Dart and PNaCl. There's a bug open about it, but one of the V8 hackers doesn't like it. With regard to the Apple camp, as best as I can tell, the WebKit folks are completely silent. IE? I wouldn't get my hopes up.

Anyway, it's not really clear why this is the One True Fixed JavaScript that will clearly beat all the competing proposals. In addition, if it did win–it really wouldn't be JavaScript. After all, the whole reason it's viable is that it potentially pries away that pesky garbage collector. Thus it could be viable with a C/C++ frontend, or some other manual-memory language. But it's definitely not the same dynamic language we know and love today.

Slow relative to WHAT

One of the problems with these "X is slow" vs "X is not slow" articles is that nobody ever really states what their frame of reference is. If you're a web developer, "slow" means something different than if you're a high-performance cluster developer, means something different if you're an embedded developer, etc. Now that we've been through the trenches and done the benchmarks, I can give you **three frames of reference** that are both **useful** and **approximately correct**.

If you are a web developer, think about the iPhone 4S Nitro as IE8, as it benchmarks in the same class. That gets you in the correct frame of mind to write code for it. JS should be used very sparingly, or you will face numerous platform-specific hacks to make it perform. Some apps will just not be cost-effective to write for it, even though it's a popular browser.

If you are an x86 C/C++ developer, think about the iPhone 4S web development as a C environment that runs at 1/50th the speed of its desktop counterpart. Per the benchmarks, you incur a 10x performance penalty for being ARM, and another 5x performance penalty for being JavaScript. Now weigh the pros and cons of working in a non-JavaScript environment that is merely 10x slower than the desktop.

If you are a Java, Ruby, Python, C# developer, think about iPhone 4S web development in the following way. It's a computer that runs 10x slower than you expect (since ARM) and performance degrades exponentially if your memory usage goes above 35MB at any point, because that is how garbage collectors behave on the platform. Also, you get killed if at any point you allocate 213MB. And nobody will give you any information about this at runtime "by design". Oh, and people keep asking you to write high-memory photo-processing and video applications in this environment.

This is a really long article

So here's what you should remember:

- Javascript is too slow for mobile app use in 2013 (e.g., for photo editing etc.).
  - It's slower than native code by about 5
  - It's comparable to IE8
  - It's slower than x86 C/C++ by about 50
  - It's slower than server-side Java/Ruby/Python/C# by a factor of about 10 if your program fits in 35MB, and it degrades exponentially from there
- The most viable path for it to get faster is by pushing the hardware to desktop-level performance. This might be viable long-term, but it's looking like a pretty long wait.
- The language itself doesn't seem to be getting faster these days, and people who are working on it are saying that with the current language and APIs, it will never be as fast as native code
- Garbage collection is exponentially bad in a memory-constrained environment. It is way, way worse than it is in desktop-class or server-class environments.
- Every competent mobile developer, whether they use a GCed environment or not, spends a great deal of time thinking about the memory performance of the target device
- JavaScript, as it currently exists, is fundamentally opposed to even allowing developers to think about the memory performance of the target device
- If they did change their minds and allowed developers to think about memory, experience suggests this is a technically hard problem.
- asm.js show some promise, but even if they win you will be using C/C++ or similar "backwards" language as a frontend, rather than something dynamic like JavaScript

Let's raise the level of discourse

I have no doubt that I am about to receive a few hundred emails that quote one of these "bullet points" and disagree with them, without either reference to any of the actual longform evidence that I've provided–or really an appeal to any evidence at all, other than "one time I wrote a word processor and it was fine" or "some people I've never met wrote a flight simulator and have never e-mailed me personally to talk about their performance headaches." I will delete those e-mails.

If we are going to make any progress on the mobile web, or on native apps, or really on anything at all–we need to have conversations that at least **appear** to have a plausible basis in *facts of some kind*–benchmarks, journals, quotes from compiler authors, whatever. There have been enough HN comments about "I wrote a web app one time and it was fine". There has been enough bikeshedding about whether Facebook was right or wrong to choose HTML5 or native apps knowing what they would have known then what they could have known now.

The task that remains for us is to **quantify specifically** how both the mobile web and the native ecosystem can get better, and then, you know, do something about it. You know–what software developers *do*.

Thanks for making it all the way to the end of this article! If you enjoyed this read, **you should follow me on Twitter (@drewcrawford)**, send me an email, subscribe via RSS, or leave a comment, and share my writing with your friends. It takes many, many hours to write and research this sort of article, and all I get in return are the kind words of my readers. I have many articles of similar depth at various stages of composition, and when I know that people enjoy them it motivates me to invest the time. Thanks for being such a great audience!

Comments

1. Pseudonym 
   Tue 09th Jul 2013 at 11:56 pm

   To answer Vlasta's question, people got into GC because RAM got cheap enough and disks got fast enough that the advantages outweighed the drawbacks for most jobs. Programs written in more modern languages with GC tended to be faster to write, *far* more robust, and not significantly slower (and sometimes *faster*) than programs written in older languages with manual memory management.

   Incidentally, the claim that programs with GC could be sometimes faster seems absurdly bold, but you have to think back to the state of mainstream programming languages in the early 90s. In C, you typically had to structure your program around the lifetimes of objects, which in turn dictated what algorithms you could and couldn't use. Also, it was hard to compare like with like because the more-modern-language implementation typically did strictly more work (e.g. took steps to actively prevent more crashes) than the C implementation. See Paul Wilson's seminal review if you want to know why GC looked like the best tradeoff back in the day.

   As Drew rightly notes, it's often *still* an acceptable tradeoff on desktop and server-grade hardware. Losing a few percent on throughput (where "throughput" can be thought of as "the amount of work which can be done just prior to the system being overloaded") is a small price to pay for faster development and maintenance, easier deployment, more flexible configurability, more robustness, more graceful degradation and so on.

   One thing that pleases me is that in the last ten years, high-performance compiled languages got ARC. Some (e.g. Objective C) got it in the compiler and some (e.g. C++) got it in the libraries, but it amounts to the same thing. This has fundamentally changed things, because while ARC is not GC, it gives you an in-practice solution for almost all

of the use cases for GC, at a fraction of the cost (both in performance cost and memory bloat cost).

You can interpret this in two ways.

One is that the GC advocates of the day were wrong, and GC was always destined to be a dead-end.

The other is that mainstream language designers of the day were wrong, and remained wrong until they actually *listened* to those who advocated GC, and did something to address their real-world concerns.

I take the latter view.

2. Erich Ocean
   Wed 10th Jul 2013 at 12:09 am

   I ran into most of that stuff developing Blossom (https://github.com/erichocean/blossom).

   I agree 100% with image or video processing, JavaScript is not a good choice.

   But even just the UI itself is very hard to do well once you add HTML and CSS into the mix!

   Blossom overcomes that by doing everything with canvases, that map 1:1 to their CALayer counterparts in WebKit, and a Core Animation-like approach in JavaScript that does hardware-accelerated CSS animations automatically.

   In other words: Blossom creates native-quality UIs by doing everything possible to use the native code paths exposed by the browser (canvas, ArrayBuffers, CSS animation, etc.).

   It does work, and being in a framework, makes it easy to use, but damn! It sucked writing the framework itself….

3. Ben Rosengart
   Wed 10th Jul 2013 at 12:24 am

   Very interesting.

   I have to ask: why doesn't Perl rate a mention?

4. Andrea Giammarchi
   Wed 10th Jul 2013 at 12:41 am

   tell JS people RAM is important too? story of my life and … NO, it's not true that JS developers don't care but it's true that most of the things you read online about JS don't consider RAM and GC usage.

   I do, and I try since ever to tell others how to consider these factors too … many of us is doing it right but I agree the road to drop the myth that "who cares, it's scripting" is hard to erase from a world that is mostly still stuck behind a philosophy that says IE8 sucks, but better than IE6 … 'cause mobile is a different beats, but many don't get it and they don't even know problems there are not JS or its features, are rather how we learned wrongly to code before the mobile browsers era. Sadly happy about this article, very real!

5. Alan
   Wed 10th Jul 2013 at 1:22 am

   Don't want to start a comment war here, so please please please, only regard my comment as a personal opinion…

   It seems odd to me that people talk about programming without GC as "chaotic"… as a C++ programmer I just can't understand where is the "chaotic" aspect of programming in C++. I think it's just one of those not-so-rare cases where people simply mark their "opinion" as "fact". Personally I don't care much about any type of managed languages, not suitable for my line of work. but I don't get people on the other side of river treating the whole managed-native thing as a "cult"… get over it please ! use whatever suits your team, your product, and your customer and keep the discussion "professional", as this article does.

6. Alan
   Wed 10th Jul 2013 at 1:25 am

   btw, great article. thanks for the effort

7. MrPoulet
   Wed 10th Jul 2013 at 1:26 am

   that why i use Flash for the web and a crosscompiled version for native app with Air

   1 source code and native performance thx to hwd acceleration.

8. mah
   Wed 10th Jul 2013 at 1:29 am

   Very good article, and reflects perfectly my personal experience. I would also like to add one point, related to the idea of holding on to garbage in your main memory. In memory constrained environments, wasting RAM is the worst thing you can do, since if you need to start dropping code pages from the page cache in order to keep some garbage around, you'll soon need to re-fetch those code pages and then you enter the unpredictable realm of disk I/O.

9. Jim
   Wed 10th Jul 2013 at 1:33 am

   Your font sucks.

10. Noam 
    Wed 10th Jul 2013 at 1:48 am

    Drew, your comment about ASM.JS, "if it did win–it really wouldn't be JavaScript", needs some qualification.

    "If it did win" – why does anyone have to win? There can be multiple subsets and different browser vendors can implement the ones that are popular or feasible.

    "It wouldn't be JavaScript" – maybe. It would still be a web app though, that runs sandboxed in all browsers, albeit faster in some browsers.

    Why I'm nitpicking on that one?
    I see a feasible future for "fast web apps with a computationally intensive component" where the JS intensive parts are written with one of those optimized subsets, and the dynamic JS we use today will be used to bind those to the UI and the network.

    We'll have to see, but I wouldn't rule out fast mobile web-apps just yet…

11. Jeff
    Wed 10th Jul 2013 at 2:07 am

    I'm a long time C/C++ programmer AND I love Javascript. In general for anything performance-intensive, I would rather be programming in a language with explicit memory management. It should be very clear to programmers what tradeoffs they are making when they opt for any language. However, my team has developed a very compelling mobile web-app in JS running in a UIWebView (meaning it has no Nitro optimization). Despite this limitation, it still performs very well on iOS devices. The only things we have problems with are crappy Android devices like the Kindle. Neither I nor any reasonably intelligent programmer would ever claim that JS performs on par with C. To frame your article that way (as well as several other of your comparisons) is a ridiculous context that unfortunately belies a very strong bias on your part and undermines your argument.

12. Lelala
    Wed 10th Jul 2013 at 2:19 am

    Instead, just build a HTML5-mobile optimized "app version" of the page – especially if the team is small, not only do you save money also you will get complexity reduced, because peoples communication is much more efficient, resulting in better software quality.

13. Drew Crawford 
    Wed 10th Jul 2013 at 2:23 am

    @mark Jumping in on two points:

    "Real-time GC" means that it doesn't use a stop-the-world design. What it *doesn't* mean is that it performs well *in a memory-constrained environment*. New GC algorithms have advantages, but they all have the property that they perform order-of-magnitude-bad *when you have memory pressure*. No free lunch, just incrementally cheaper lunch. Note how I keep qualifying my claims with "when you don't have a lot of memory"–this is the key. Available memory is *everything* with GC performance. There is basically nothing you can infer from memory-rich benchmarks about memory-poor environments, nor vice versa. Available memory changes everything.

    ObjC is dynamic, but the reason people keep "lump[ing] it in with static languages" is that it is sort of static too. e.g., you do have dynamic dispatch, but the compiler inlines a lot of things. So *in practice* you might not have *any* dynamic dispatch, even though *in theory* it's 100%. The trouble with stereotypes like "dynamic" and "static" is that every language is a particular language, and no language perfectly fits all the "dynamic language bulletpoints", because "dynamic" is just a word we made up to group particular languages, not a thing in its own right.

    Okay, I lied, three things. Here's an interesting rabbit hole to go down. NSArray isn't even an array: http://ridiculousfish.com/blog/posts/array.html

14. Marc
    Wed 10th Jul 2013 at 2:34 am

    Hi Drew,
    Great article. Thank you very much. The length of an article of this quality is immaterial, enjoyed all of it.

Your article is being noticed. I was pointed at it independently by both Herb Sutter (blog post) and Addy Osmani (retweet).

If anything is left to be desired, then some printing support would be nice.

15. sathya 
Wed 10th Jul 2013 at 2:36 am

It took me 4 hours to read this article and it's links. Really appreciate your effort on this article. Need lot more time to understand in your's depth.

Please post it's follow-ups, if anything interesting.

16. Kurt 
Wed 10th Jul 2013 at 2:43 am

JavaScript is not the problem. The problem is that a scripting language is asked to do too much.

Games also rely on scripting languages for their flexibility, and also need to fit in tight memory limits imposed by consoles. The difference is that, when a script is too slow, the game can move that functionality into the optimised engine code.

Web browsers are crippled by HTML, CSS, and developing compatible standards. Features are slowly being added, but there isn't much you can rely on. JavaScript developers just have no control over their execution environment.

17. Daniel Collins 
Wed 10th Jul 2013 at 2:54 am

I am posting this comment from a 28nm ARM compatible processor with 2GB of RAM. Get your hardware facts straight: Qualcomm's Krait (Snapdragon S4) launched in 28nm over a year ago. Samsung is planning on launching cjips at 14nm with "3d" transistors in the 2014/15 timeframe.

18. Lindsey Simon 
Wed 10th Jul 2013 at 2:59 am

Reading this inspired me to want to try to get some data on render speed in the webview in particular, but this applies to any browser.

https://github.com/elsigh/reflow-timer

Current results: http://www.browserscope.org/browse?category=usertest_agt1YS1wcm9maWxlcnINCxIEVGVzdBjBwpAVDA

19. Mattias Petter Johansson 
Wed 10th Jul 2013 at 3:06 am

Fantastically well-written article.

However, it talks a lot about JavaScript when the title is "Why web apps are slow". A more appropriate title might have been "Why JavaScript is slow on mobile". I work on the web player at Spotify and I struggle with web app performance almost daily, but the performance culprit is very, very rarely JavaScript, since the actual heavy lifting in a web app, painting and compositioning, is performed by the browser, not JavaScript. My performance optimizing is spent hunting down code that causes reflows, not optimizing the O(n) of JavaScript code.

20. Daniel Collins 
Wed 10th Jul 2013 at 3:10 am

@drew

Dynamic vs Static is not a fuzzy or arbitrary distinction. Dynamic means "at runtime", static means "at compile time." Usually, when you call a language dynamic, you are referring to the type system, specifically that types are checked at runtime.

Objective C is dynamic and/or static depending on how you write you code. However, if you're using objects with message passing, you're using a dynamic language since objects can choose to respond to messages however they want based on runtime information.

21. Henk Poley 
Wed 10th Jul 2013 at 3:19 am

I wonder why not more languages have adopted the garbage collection strategy of Erlang, where they have a heap per object (or process as they call it).

22. pfk3 
Wed 10th Jul 2013 at 3:56 am

Fascinating article with real substance. Thanks for taking the time to write it.

23. Dmitry 
Wed 10th Jul 2013 at 4:27 am

I'm very sorry, but I just can't understand how you draw an equivalence between "mobile apps" and "photo editing apps". Most of HTML-driven mobile apps today are simple CRUDs in their core, so almost all your points on performance and memory consumption are invalid for them. It's very clever to cite gamedevs to point out that GC can be a problem — but it's a preaching to a choir, games always were hard and CPU-bound.

24. Graeme 
Wed 10th Jul 2013 at 5:06 am

Thanks, I learned a bunch.

25. Zex 
Wed 10th Jul 2013 at 5:22 am

This is a long-standing problem of not having a proper language. First we need a good general purpose language, with easy syntax (that rules ObjC out), good looking (rules C/C++ out), classes (rules Google Go out), and at least the #include function so we can include a necessary library without having to use HTML (this rules JavaScript out).

My proposition would be a dialect of Google Go with the classes added. And possibly some dynamic stuff from JavaScript, but such functions should be clearly marked with a keyword so that it's clear "this function was dynamically added to the object, that's cool, but it's very slow to call it, so don't do it often".

Instead of using a GC, most objects can be allocated automatically and freed automatically immediately after they are out of scope. There can even be a function that allocates an object but tells the function to free it upon exit. This will cover most of cases. But global objects that are allocated manually in the traditional way have to be released manually as well. That's not so hard if you're a Real Programmer(tm).

Now, using this Super-Go language everything else has to be built. Including the OS. Everything. And after that there are no "desktop apps", "mobile apps" and "web apps". It's all the same code. There's no distinction between the OS and web browser.

But since ECMAScript and W3C consortium failed to think in the start (or any time later, those lazy b'stards), now we all have to think 1000 times more to cover for their crap. Still, eventually it has to converge to a single, well designed language.

Syntax is also very important. Many people refer to ObjC as "write only language". You can write it, but you don't wanna read it, coz it looks like Egyptian hieroglyphs. Could as well call it "Klingon script".

So, it's all about designing a new language, better, faster, good looking, clean, easy to learn, no GC, but still doing most of deallocations automatically (like most langs do with strings).

26. Guy Nesher 
Wed 10th Jul 2013 at 5:43 am

Interesting article.

While I completely agree with the premise, I don't think you are tackling a real problem.

I think most people do agree that Javascript is slower (and will remain slower) than native code.

Having said that, there's a big difference between a "website with a button or two" and a full spreadsheet application.

The majority of mobile apps live in that space and with the right optimisation many of the could be built using javascript without compromising user experience.

But even if we agree that user experience will be effected to some extent – you still need to consider other aspects (such as development time, costs, device support etc.) and in many cases Javascript is still good enough.

27. ksec 
Wed 10th Jul 2013 at 6:20 am

Nick Pick on a few things.
Timelines and JIT are not entirely correct but i agree with the big picture in context.

Slightly wrong will be the asm.js part. I have to say i dont like the idea that much. I think of it as an ugly hack that works really well. It will be like bytecode in JVM. And optimization against those specific Javascript patterns will land in Chrome and someday Webkit. It would only take another 12 months for all the major Javascript Engine to be on par with each other.

What is totally wrong in this article is the hardware part which i have to say nearly stopped reading and thought it was another crap article.

Intel didn't invent another transistor, just a new 22nm node design with specific for phones and Tablet. I lost the word here because Mobile would have meant Laptop for Intel, and Ultra Low Power would have meant Ultrabook. Unlike Samsung and TSMC they have been dealing with SoC Designs requirement 4 years earlier then Intel

before Intel finally woke up.

And just as another commenter has already posted, ARM is already on 28nm right now, only Apple is stuck with 32nm with Samsung. TSMC will have 20nm next year, 16nm in 2015, 10nm in 2017 and 7nm before 2020. So it is not an Intel's only game anymore. We will also have TSV stacked silicon layers as well as other graphene tech coming. I have zero reason to believe why we cant have 10x the performance of today's Mobile SoC. It is only a matter of time.

So apart from that, I think the article is great. I agree with the big picture. Unlike Mobile SoC, there is a SINGLE upcoming revolutionary battery technology that are going to improve our Battery capacity by 2 folds. And in case anyone wants to argue with this point, any breakthough now would have to take years of safety testing and 1 – 2 years before it is mass manufactured and comes on to market. In this next 5 years time frame we are going to be very battery limited. And I dont see reasons to have JS, even asm.js apps when Native Code with ARC works much better.

28. Drew Crawford 
Wed 10th Jul 2013 at 7:14 am

    @ Dmitry @Guy Nesher I think the not-so-secret-secret is that I'm just totally disinterested in the CRUD software universe. I mean, I acknowledge it exists. I just don't care. I don't want to spend my life writing CRUD apps. I don't want to write articles about how to write CRUD apps better. It isn't interesting to me. I don't enjoy it.

    So if JS is good at CRUD apps, then, *okay*. If people want to write two-button web apps then… good for them? I wish them the very best.

    But the disagreement here is happening at a higher stack frame. It's not a conversation about JS performance anymore. It's an existentialist crisis about what it really means to contribute to society as a software developer. It's a philosophical discussion about how to use our time on this planet. I sort of doubt that any conversation between strangers on this subject is useful, but I am certain that an article that consists mostly of bar charts will shed no light on the matter whatsoever.

    Probably the most useful thing that I can say is that if the problems of two-button webapps strike you as an important concern, then we have different concerns, and these differences are irreconcilable.

29. Guy Nesher 
Wed 10th Jul 2013 at 7:48 am

    @Drew Crawford

    It's funny as I've been having similar discussions with several developers I work/used to work with.

    I think that only looking at the technical challenges you face as developer misses an important point – the joy of creating something from scratch.

    This isn't really a trivial thing these days with the "services" industry becoming an ever bigger percentage of the work force.

    I haven't met many craftsman who refuse to build a product just because it's not technically challenging enough and I find it odd that developers don't share that mind set.

    That's just me really, and I will admit I've met more than a few developers that did not share my perspective.

30. Noam 
Wed 10th Jul 2013 at 7:58 am

    Drew, mobile web applications usually expose a fairly complex end-to-end system. To me those systems are more complex and interesting than trying to rewrite AutoCAD in Javascript, even if the UI aspects that are written in JS are not in themselves that interesting.

31. Dmitry 
Wed 10th Jul 2013 at 8:36 am

    @Drew Crawford
    Don't you think that it's quite pretentious to rule out *everything* that is not computationally-intensive and then declare that JS and managed languages are totally unusable on mobile? Take Github, for example. Is it CRUD-like at it's core? Yes. Is it more profitable to society than one-more-instagram-like-app-for-iphone? Definitely! Heck, just open Wikipedia in your mobile browser. It's already more valuable than 99% of AppStore and it has nothing to do with GC concerns.
    And you don't even consider a possibility of "glue" language. Take Matlab, R or Numpy — all of that languages are GCed, and all of them are used to solve numerical problems that are way harder than "drawing two draggable figures" like you've described in the article. At this level your complaints are not that more than "JS is not a silver bullet!".

« Previous comments

Add comment

Your name*
Your email address* *(will not be published)*
Your website
Your comment

[redacted]

Notify me of follow-up comments by email.

Notify me of new posts by email.

Back to top Back to top

- ## Tags

app store arduino hardware HN incentives iphone law linux long articles mips native apps notifo programmers rants steve jobs web apps wifi

- ## Subscribe via e-mail

Subscribe via e-mail

Email Address *

[redacted]