WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices

Sable Technical Report No. McLAB-2018-02

David Herrera, Hanfeng Chen, Erick Lavoie and Laurie Hendren

March 14, 2018

Contents

1 Introduction 3

2 Background and Related Work 4
2.1 JavaScript . . . 5
2.2 WebAssembly . . . 5
2.3 Mobile and IoT Devices . . . 5

3 Methodology 6
3.1 Representative Devices . . . 6
3.2 Browsers and Execution Engines . . . 7
3.3 Choice of Benchmarks . . . 8
3.4 Benchmark Execution and Timing . . . 9

4 RQ1: Old versus New JavaScript Engines 11

5 RQ2: JavaScript versus WebAssembly 13
5.1 WebAssembly versus C . . . 13
5.2 WebAssembly versus JavaScript . . . 14

6 RQ3: Portable versus Vendor-specific Browsers 15
6.1 Safari Browser . . . 16
6.2 Microsoft Edge Browser . . . 17
6.3 Samsung Internet Browser . . . 17
6.4 Google Chrome OS Browser . . . 17
6.5 Performance Issue - Samsung Internet Browser . . . 17

7 RQ4: Server-side Node.js versus Client-side Browsers 18
7.1 Node.js versus C . . . 19
7.2 Node.js versus Browsers . . . 20

8 RQ5: Best Performers 20

9 Conclusions and Future Work 21

List of Figures

1 Environment architecture used by Wu-Wei to record the data from each browser. . . 10
2 JavaScript Performance of the MacBook Pro 2013 laptop against native C, using the old and current versions for Chrome and Firefox 12
3 JavaScript Performance of Ubuntu Workstation against native C, using the old and current versions for Chrome and Firefox 12
4 WebAssembly performance relative to C on the MacBookPro 2013. 13
5 WebAssembly performance relative to C on the different platforms. 14
6 WebAssembly performance relative to JavaScript on the different platforms. 15
7 Performance of browsers relative to proprietary respective browsers. 16
8 The core code in the JavaScript implementation of lavamd. 18
9 Performance of Node.js in different workstations relative to C. 19

List of Tables

1 Devices with short names and basic configurations for the Ostrich experiments. . . . 7
2 The list of environments for experiments on devices. 8
3 Ostrich benchmarks 9
4 Geometric means of speedups for Firefox 57 and Chrome 63 relative to the Samsung Internet Browser on the two Samsung devices 17
5 Performance results for two versions of lavamd 18
6 Browser speedup performance relative to their respective WebAssembly and JavaScript Node.js versions for each device. 20
7 Device performance across environments using the native C raspberry pi implementation as baseline for geometric means. 21

Abstract

Recent advances in execution environments for JavaScript and WebAssembly that run on a broad range of devices, from workstations to IoT devices, provides new opportunities for portable and web-based numerical computing. The aim of this paper is to evaluate the current state of the art through a comprehensive experiment using the Ostrich benchmark suite, a collection of numerical programs representing the numerical dwarf categories. Five research questions evaluate the improvement of JavaScript-based browser engines, the relative performance of JavaScript and WebAssembly, the relative performance of portable versus vendor-specific browsers, the relative performance of server-side versus client-side JavaScript/WebAssembly, and an overall comparison to find the best performing browser/language and the best performing device.

1 Introduction

Computation via web-browsers is becoming increasingly more attractive. Web-based computations provide a simple and portable way for developers to distribute their applications. Further, the proliferation of browser-enabled devices containing sophisticated execution engines provides enormous computation capacity including devices of all sizes, from workstations and laptops to mobile phones and Internet-of-Things (IoT) devices. The research questions addressed in this paper focus on evaluating the performance of numerical computations using modern JavaScript and WebAssembly engines on a wide variety of web-enabled devices.

Previous work, circa 2014, showed that browser-based execution engines, when executing JavaScript and using the best technologies of the time, had execution times within factor of 1.5 to 2 of the performance of native C [32]. Further, at that time, two notable performance enablers were demonstrated: (1) the use of JavaScript typed arrays [6]; and (2) the use of asm.js [2]. These experiments were performed on workstations and laptops, and used the Ostrich benchmark set that is composed of 12 numerical benchmarks [20], each representing one of the computational dwarf categories [15, 16, 21]. The 2014 study demonstrated that web-based numerical computing was becoming very attractive, which subsequently inspired other projects including: (1) MatJuice: a translator from MATLAB to JavaScript [22, 23], (2) MLitB: machine learning in the browser [35], (3) Pando: a web- based volunteer computing platform [33], (4) SOCRAT: a web architecture for visual analytics [30], and (5) CHIPS: a system for cloud-based medical data [37]. These are just representative uses of web-based numerical computing, there are many other web-based applications which perform core scientific computations including those in the areas of machine learning, data visualization, big data analytics, simulation, and much more.

Since 2014 there have been many important advances in both web-based execution engines and a substantial increase in the computational power of mobile and IoT devices. On the software side, the Just-In-Time (JIT) compilers in Chrome and Firefox continue to evolve, and new browsers with JITs, such as Samsung's Internet Browser, and Microsoft's Edge have appeared. Furthermore, the invention and adoption WebAssembly has provided a new common program representation very suited to optimized numerical computing. On the hardware side, tablets and phones have become increasingly powerful computing devices, even IoT devices now have non-trivial computational power. Thus, we believe that this is a good time to examine the current state of web-based numerical computing by examining a wide variety of browser engines/technologies and a wide variety of devices. We base our experiments on the Ostrich benchmark set, and we seek to answer five major research questions.

RQ1 - ***Old versus New JavaScript engines*****:** Since the 2014 study, Chrome and Firefox, the two browsers that showed the best performance on the Ostrich benchmarks, have continued to evolve. Thus our first research question examines how much the JavaScript engines in those browsers have improved, using typical workstations and laptop architectures (similar to the types of architectures used in the 2014 study).

RQ2 - ***JavaScript versus WebAssembly*****:** Despite the best efforts of compiler researchers and developers to provide good performance for JavaScript, the dynamic nature of JavaScript makes it inherently difficult to achieve the same performance as in a statically-typed language like C. WebAssembly, a new typed intermediate representation for programs executing on web browsers, provides many more opportunities for optimizations [27]. Thus our second research question looks at the relative execution speeds of JavaScript and WebAssembly. Starting with this research question, we also broaden our experiments to include mobile and IoT devices. This reflects the reality that modern mobile and IoT devices provide substantial computational power and that browser providers are optimizing for such devices.

RQ3 - ***Portable versus Vendor-specific browsers*****:** In addition to browsers such as Chrome and Firefox, which are supported across a wide range of devices and operating systems, two vendor-specific browsers have recently been introduced: (1) the Samsung Internet browser for Samsung devices (and other android devices), and (2) the Microsoft Edge browser for Windows 10. Given that the vendor-specific browsers have specific targets, it is plausible that they might show better performance than the portable browsers. Thus, our third research question examines whether the vendor- specific browsers achieve better performance than the portable browsers when used on their targeted system.

RQ4 - ***Server-side Node.js versus client-side browsers*****:** JavaScript and WebAssembly are not only used on the client-side, within browsers, they may also be used on the server side, in the form of Node.js. Our fourth research question examines the relative performance of server-side Node.js versus client-side JavaScript and WebAssembly, for those architectures that support both.

RQ5 - ***Overall Best Performers*****:** The final research question aims to provide an overall performance summary based on all of the experiments performed for research questions RQ2 through RQ4. For our benchmark set, which are the best performing browsers, and for all browsers which are the best performing devices?

This paper is structured as follows. Sec. 2 provides the background and related work and Sec. 3 describes the methodology we used for our experiments. Sections 4 through 8 give the relevant experimental results and discussion for each of the five research questions. Finally, Sec. 9 provides the conclusion and discussion of future work.

2 Background and Related Work

The web began as a simple document exchange network mainly meant to share static files, by historical accident, JavaScript was the only natively supported language on the web [39] and from its inception, JavaScript was meant as a simple interpreted language designed for non- professional programmers. The introduction of the AJAX technology provided the web with dynamic content and caused JavaScript to be an essential part of application development.

2.1 JavaScript

Since the browser wars in 2008 [36], both main browser vendors, and the developer community took on the challenge to make JavaScript and web applications, scalable, standardized, and fast. Efficiency was brought with the introduction of the JavaScript Engines and their JIT compilers [24, 14]. Moreover the ECMAScript standard [11] along with other big projects such as the babel compiler project [17], have come together to bring standardization to the web. Over the past few years we have observed the growing dominance of web applications with an increasing number of devices supporting web technologies and ranging from smartphones and desktops, to IoT devices. As of today the peak performance of the best JavaScript engines are within 1.5 to 2 factors of native C [32]. However, despite current and past efforts to improve JavaScript, the maturation of web platforms have given rise to increasingly more intensive computations. JavaScript as the only built-in language of the web is not well-equipped to handle this increasing demand, with the language also presenting a challenge as a compilation target to other high level languages due to its dynamically typed nature.

2.2 WebAssembly

WebAssembly [27] is a new portable binary code format, that in addition to maintaining the secured, isolated, model the web provides, brings *near-native* speeds to the web and serves as a more appropriate compilation target for typed languages such as C and C++. It thus opens the doors to a variety of different languages and closes the gap in performance allowing applications that were previously hard to port to the web. Currently, the Emscripten toolkit [45] provides a framework for compiling C and C++ to WebAssembly along with an embedded execution environment for WebAssembly in JavaScript which exposes the C and C++ functions to the JavaScript run-time.

WebAssembly was built as an abstraction on top of the main hardware architectures providing a format which is language, hardware, and platform independent [8]. The low-level nature of the language should offer many opportunities for optimizations that would benefit numerical computations, at the time of writing, fixed-width SIMD feature, and parallelism via threads are in the *in-progress* stage for WebAssembly [3]. Furthermore, WebAssembly supports different integer types and single precision floating-points, which are not currently supported by JavaScript.

2.3 Mobile and IoT Devices

The increasing power of mobile devices provides a platform for sophisticated numerical computing. Indeed, this computing power can be used to ensure the privacy of personal data through efficient and effective encryption and to provide the power to support numerically-intense security check algorithms such as the 3D face recognition recently introduced on the iPhone [29]. In general, the need for the protection privacy of personal data and the rising importance of machine learning models, numerical web computations hosted at the host environment are becoming increasingly important [40].

The Internet of Things provides yet another challenge for numerical computing. As these small devices become more powerful and ubiquitous, there are many challenges for their effective and secure use [44].

Both mobile devices and IoT devices also provide internet-connected computing power that can be used for big data computations, and thus provide a potential platform for distributed computations using cloud/fog computing [18], as well as providing potential devices to be used in volunteer computing platforms [19, 33].

3 Methodology

IoT devices, Node.js and the proliferation of smartphones have brought a wide range of environments and devices that support JavaScript and WebAssembly. Browser providers are now optimizing their engines based on these different environments, each of which, contains different hardware restrictions and characteristics. This presents a challenge to browser providers whereby the optimizing objective for their engines becomes a combination of memory and throughput. In low-memory devices such as single board computers and smartphones, prioritizing throughput over memory consumption may result in out-of-memory crashes and problems such as suspended tabs. In this regard the browser providers have responded to these needs by modifying their engines to fit the hardware requirements of different devices. The V8 engine team, for instance, has tuned the garbage collection heuristics to lower the memory consumption providing low-memory mode to their engine [5].

3.1 Representative Devices

To obtain a representative measurement of JavaScript and WebAssembly performance, there is a need to consider a range of devices. In this paper we quantify the performance of both WebAssembly and JavaScript for a wide variety of devices, ranging in size and computational power, as summarized in Table 1.

To evaluate desktop and laptop performance we have executed our experiments on three machines, namely, a MacBook Pro 2013 laptop (mbp2013), an ubuntu based desktop (ubuntu-deer), and a similar windows-based desktop (windows-bison).[1] These machines allowed us to measure performance for the three major operating systems and also allow to include performance evaluations of the OS-specific Apple Safari and Microsoft Edge browsers.

For mobile devices we considered state-of-the-art tablets and smartphones. On the tablet front, or medium size devices, we have selected two of the most popular and powerful tablets currently in the market, i.e. the Samsung Tab S3 and the iPad Pro. Lastly, on the mobile front, we have chosen three popular consumer smartphones that are top of the line for three major smartphone providers, namely, the Samsung Galaxy S8, the Google Pixel 2, and the iPhone X. The inclusion of Samsung devices also allows us to evaluate the Samsung Internet Browser.

To represent IoT devices, we have selected the Raspberry Pi 3 model B (raspberry-pi), as a representative of the single-board computers and IoT devices. The Raspberry Pi allowed us to explore the performance of JavaScript, WebAssembly, and Node.js in a very low memory setting, and also provided a baseline to compare against as the lowest performing platform.

[1]The machine names mbp2013, ubuntu-deer and windows-bison, are used in our subsequent results, graphs, and tables.

Table 1: Devices with short names and basic configurations for the Ostrich experiments.

Platform	Device	CPU	RAM	OS	GCC
Laptops & Workstations	MacBook Pro 2013 (mbp-2013)	Intel Core i5 @ 2.4 GHz	8 GB	MacOS 10.13.1	LLVM-GCC 4.2.1
	Ubuntu Workstation (ubuntu-deer)	Intel Core i7-3820 @ 3.60 GHz	16 GB	Ubuntu 16.04	GCC 5.4.0
	Windows Workstation (windows-bison)	Intel Core i7-3820 @ 3.69 GHz	16 GB	Windows 10 Enterprise	Cygwin GCC 6.4.0
Single Board Computers	Raspberry Pi (raspberry-pi-3)	ARM Cortex A53 @ 1.20 GHz	1 GB	Linux Raspberry Pi 4.9.35-v7	GCC 4.9.2
Tablets	Apple iPad Pro (ipad-pro)	Apple Fusion @ 2.36 GHz	4 GB	OSX 11.0.3	-
	Samsung Tablet S3 (samsung-tab-s3)	Quad Core @ 1.6 - 2.15 GHz	4 GB	Android 8.0.0	-
Smart Phones	Samsung Galaxy S8 (samsung-s8)	Octa-core @ 1.70 - 2.30 GHz	4 GB	Android 8.0.0	-
	Google Pixel 2 (pixel2)	Qualcomm Snap-dragon 835 @ 1.90 - 2.35 GHz	4 GB	Android 8.0.0	-
	Apple iPhone X (iphone10)	Apple Fusion @ 2.36 GHz	4 GB	OSX 11.0.3	-

3.2 Browsers and Execution Engines

For each device we have experimented with many different execution engines, as summarized in Table 2.

To answer RQ1 we used versions of Firefox and Chrome that were available in 2014, as well as the most recent versions. To answer RQ2 through RQ5 we used the most recent browser-based JavaScript and WebAssembly engines, as well as server-side Node.js.[2]

Portable browsers such as Firefox and Chrome were available for all the devices, whereas OS-specific browsers such as the Samsung and Microsoft browsers were available only on some of the devices.

[2]Our test machines all had very recent versions, although slightly different. For the final paper we will request upgrades to all machines to use exactly the same versions.

Table 2: The list of environments for experiments on devices.

Environment	Version	JavaScript Engine Version	Devices
Chrome63	V63.0.3239.84	V8 6.3.292.46	iphone10, samsung-s8, pixel2, ipad-pro, samsung-tab-s3, mbp2013, ubuntu-deer, windows-bison
Chromium38	V38.0.2125.0	V8 3.28.71.2	mbp2013, ubuntu-deer, windows-bison
Firefox57	V57.0.2	-	iphone10, samsung-s8, pixel2, ipad-pro, samsung-tab-s3, mbp2013, ubuntu-deer, windows-bison
Chromium56	V56.0.2923.84	-	raspberry-pi
Firefox39	V39.0	-	mbp2013, ubuntu-deer, windows-bison
Safari11	V11.0.1	-	iphone10, ipad-pro, mbp2013
Microsoft-Edge	V41.16299.15.0	-	windows-bison
Samsung-Internet	V6.2.01.12	-	samsung-s8, samsung-tab-s3
Node.js	V8.9.1	V8 6.1.534.47	mbp2013, ubuntu-deer
	V8.9.3	V8 6.1.534.47	windows-bison
	V9.3.0	V8 6.2.414.4	raspberry-pi
Native	LLVM-GCC 4.2.1	-	mbp2013
	GCC 4.9.2	-	raspberry-pi
	GCC 5.4.0	-	ubuntu-deer
	GCC 6.4.0	-	windows-bison

3.3 Choice of Benchmarks

There are many benchmark suites available for JavaScript by both researchers [38] and browser vendors [9, 7, 10]. These benchmark suites however are not suited for the type of numerical computing typically used by scientist and engineers. To measure numerical performance of JavaScript and WebAssembly, instead, we have used the most recent version of Ostrich Benchmark Suite, which has been updated to work with the Wu-Wei benchmarking system [31]. This version of Ostrich also includes some improvements to the benchmarks which make better use of JavaScript programming idioms.

The Ostrich benchmarks were originally adapted from the Rodina [15] and Open Dwarf [13] suites, and they represent the 7 Dwarf categories in numerical computing identified by Colella [16]. Table 3 provides a summary of the benchmarks. Each benchmark contains C and JavaScript equivalent implementations. In order to experiment with WebAssembly performance we have used the C implementation and compiled from the unmodified C versions using the Emscripten [45] compiler. This allows us to quantify the performance of both JavaScript/WebAssembly against native C, with the consistency provided by having the equivalent implementations across languages. The version of the Emscripten compiler used to compile of the C benchmarks was 1.37.22.

Table 3: Ostrich benchmarks: provenance, description, and implementation details. Compared with the version of the benchmarks used in Khan et al. [32], = means the benchmark is the same, and ∼ indicates that the benchmark has been improved in the newer version used in this paper.

Benchmark	Dwarf	Provenance	Description	C	JS
back-prop	Unstructured grid	Rodinia	A machine-learning algorithm that trains the weights of connecting nodes on a layered neural network	=	∼
bfs	Graph traversal	Rodinia	A breadth-first search algorithm that assigns to each node of a randomly generated graph its distance from a source node	=	∼
crc	Combinatorial logic	OpenDwarfs	An error-detecting code which is designed to detect errors caused by network transmission or any other accidental error	=	∼
fft	Spectral methods	OpenDwarfs	The Fast Fourier Transform (FFT) function is applied to a random data set	=	∼
hmm	Graphical models	OpenDwarfs	A forward-backward algorithm to find the unknown parameters of a hidden Markov model	=	∼
lavamd	N-body methods	Rodinia	An algorithm to calculate particle potential and relocation due to mutual forces between particles within a large 3D space	=	∼
lud	Dense linear algebra	Rodinia	A LU decomposition is performed on a randomly-generated matrix	=	∼
nqueens	Branch and bound	OpenDwarfs	An algorithm to compute the number of ways to put down n queens on an $n \times n$ chess board where no queens are attacking each other	=	∼
nw	Dynamic programming	Rodinia	An algorithm to compute the optimal alignment of two protein sequences	=	∼
page-rank	Map reduce	Authors	The algorithm famously used by Google Search to measure the popularity of a web site	=	∼
spmv	Sparse linear algebra	OpenDwarfs	An algorithm to multiply a randomly-generated sparse matrix with a randomly generated vector	=	∼
srad	Structured grid	Rodinia	A diffusion method for ultrasonic and radar imaging applications based on partial differential equations	=	∼

3.4 Benchmark Execution and Timing

To run and record the experiments we used the Wu-Wei benchmarking toolkit [31]. Wu-Wei allows us to simplify the gathering, organization, verification, automation, and replication of our experiments by providing a well-defined structure to perform the data gathering along different platforms, compilers, and environments. The toolkit also provides a verification mechanism for benchmark outputs.

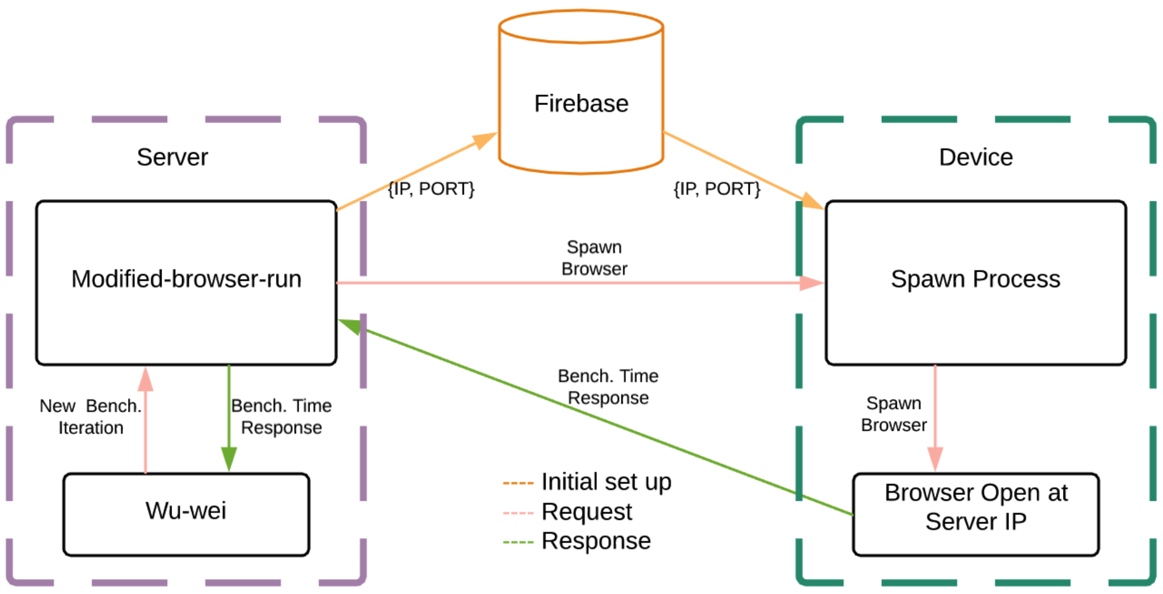

Figure 1: Environment architecture used by Wu-Wei to record the data from each browser.

Experiments on web browsers required us to extend the Wu-Wei toolkit. To run the code on different platforms/browsers, we developed a new Wu-Wei compatible environment. Fig. 1 presents the architecture we designed to record the data for a given device browser. At each benchmark iteration, Wu-Wei spawns a *locally-hosted* website and sends the request to Firebase [4], a real-time cloud based database. The request shares the browser name to be opened and the IP address of this locally-hosted site. The listening device will then spawn browser at that address which will communicate directly with the Wu-Wei host. To make this possible in mobile devices, we have also created a hybrid mobile application using web technologies and the Apache Cordova framework [1]. The hybrid nature of the app caters to both Android and iOS with a single codebase.

To record the native C and Node.js benchmark execution time, a similar architecture and environment was used. A script that runs on the target platform listens to each request, and for each request, it downloads the files, compiles (if necessary) and runs in the appropriate environment. The communication from the platform however is performed completely via Firebase. In this fashion, using one machine to control the measurements to be taken, we avoid having code replication of the benchmark suite, and we keep a centralized, structured data collection.

In terms of the actual experiment, we ran each benchmark a total of 30 iterations as suggested by Georges et al. [25], this allowed us to approximate the distribution of measurements as a Gaussian and obtain an appropriate standard deviation. Moreover, at each benchmark run, a new browser tab was spawned in order to avoid the cached JIT optimizations of the JavaScript engines. The time for each benchmark represents the main loop computation of a *medium size input* as defined by the Ostrich Benchmark Suite. The comparisons in the paper are done using speedups of a given benchmark implementation relative to other implementation/environment of the same benchmark. The error propagation for those speedups is given by Equation 1,

$$\Delta\left(\frac{a}{b}\right) = \left(\frac{a}{b}\right)\sqrt{\left(\frac{\Delta a}{a}\right)^2 + \left(\frac{\Delta b}{b}\right)^2} \tag{1}$$

Where a, b are the quantities we wish to compare and Δa, Δb are the errors of those quantities coming from measurement. Moreover, for the speedup calculations, we calculated the geometric mean for an specific environment and platform across all the benchmarks, and we used that as a comparison measure to other platforms.

4 RQ1: Old versus New JavaScript Engines

The development and improvement of JavaScript engines has continued, albeit with different objectives in mind. Due to the proliferation of smartphones and IoT devices, there have been changes in the benchmarks suites used to optimize these engines [42], the importance of optimizing throughput has been tempered by other optimizing factors such as initial page load time, memory and optimizing objectives provided by real websites. Moreover, the recent development of these engines have also seen a change of compiler infrastructure in the case of the V8 team [41] and the introduction of a brand new Mozilla Firefox Browser, Firefox Quantum [34].

With these changes in mind, in RQ1 we explore the numerical performance of Firefox and Chrome JavaScript engine versions from 2014, against the modern version of these browser engines. In the study of numerical performance in 2014 [32], the reported sequential JavaScript performance was competitive with the C version of the benchmarks to within a 1.5 to 2 factor. Therefore, an interesting question, is whether similar results can be achieved with the modern browsers, and more broadly, how have these engines evolved in the context of numerical computing since 2014.

Similar to the 2014 study, we used the Ostrich Benchmark Suite to evaluate numerical performance of these engines. We provide the speedups of JavaScript against the native C version of these benchmarks for two architectures, mbp2013 in Fig. 2 and ubuntu-deer in Fig. 3. In each case we experimented with four versions of the browsers, namely, Chromium 38[3], Firefox 39, Chrome 63 and Firefox 57. In each graph, the right-most group of bars exhibits the geometric mean and is meant to represent the overall performance for each of these browsers, the legends show wasm-c, to signify that the benchmark is implemented in C and compiled to WebAssembly.

The first observation is that the best performing browser was Firefox 57, the JavaScript performance of the browser increased significantly over the Firefox 39 version for both the MacBook Pro laptop and Ubuntu Workstation. Secondly, and perhaps surprisingly, the performance of the current Chrome browser decreased compared to the performance of the older version of the browser from 2014. Some reasons for this could be attributed to the new compiler infrastructure introduced by the V8 team in the past year and perhaps the new optimizing objectives that these engines pursue.

The overall performance of JavaScript against native C versions remained within a factor of 2. The current Firefox browser has presented an overall improvement, compared to the older Firefox version. The current Chrome browser, however, has presented a decrease in overall performance compared to the older Chrome version.

[3]Due to the unavailability of the Chrome38 browser, the comparison had to be done against the developer version of the browser.

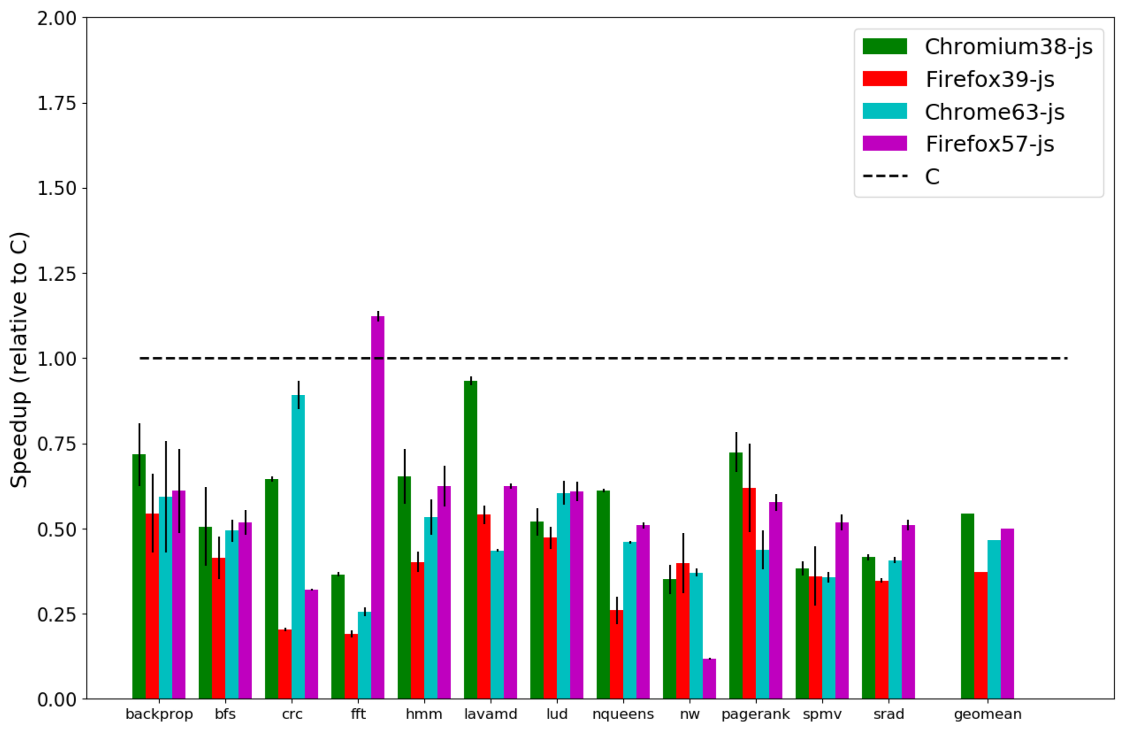

Figure 2: JavaScript Performance of the MacBook Pro 2013 laptop against native C, using the old and current versions for Chrome and Firefox

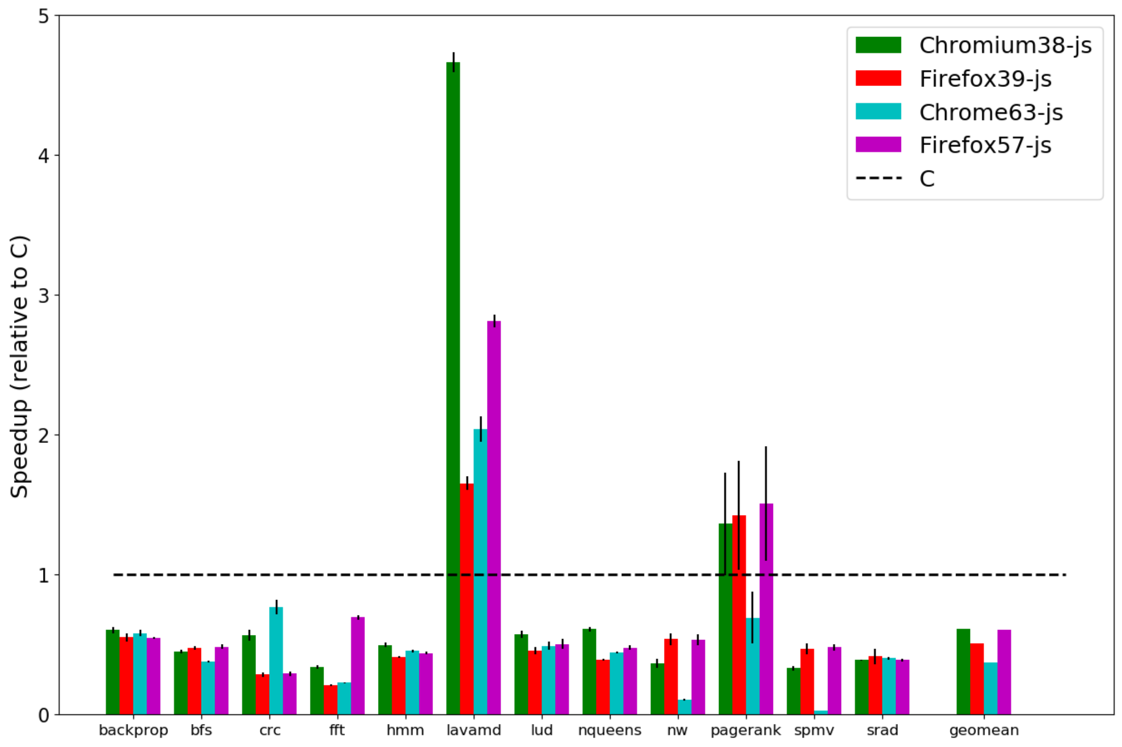

Figure 3: JavaScript Performance of Ubuntu Workstation against native C, using the old and current versions for Chrome and Firefox

5 RQ2: JavaScript versus WebAssembly

As discussed in the previous section, modern browser engines for JavaScript can perform within a factor of 2 of native C. Closing the gap further appears to be difficult due to the inherently dynamic nature and unusual semantics of JavaScript. Browsers can now also execute WebAssembly, which is a lower-level representation with clean semantics and types. Moreover, WebAssembly was build as an abstraction on top of all the major hardware architectures. This features allow for a highly optimizable language that benefits in performance. This should be particularly true for the kinds of numeric benchmarks that we are studying. Thus, RQ2 examines how WebAssembly performs as compared to native C.

5.1 WebAssembly versus C

We first conducted experiments between WebAssembly and C with three popular browsers (i.e. Chrome 63, Firefox 57, and Safari 11) on the MacBook Pro laptop. The result of the experiments are depicted in Fig. 4. There are 5 out of 12 benchmarks in which the WebAssembly code is faster than native C code, especially for `fft`, where the WebAssembly code has overwhelming speedup on all three browsers. However, considering the geometric means (given as the rightmost bars), the overall performance of WebAssembly is lower than native C. The best performance of WebAssembly is on Firefox 57 whose performance is closest to C.

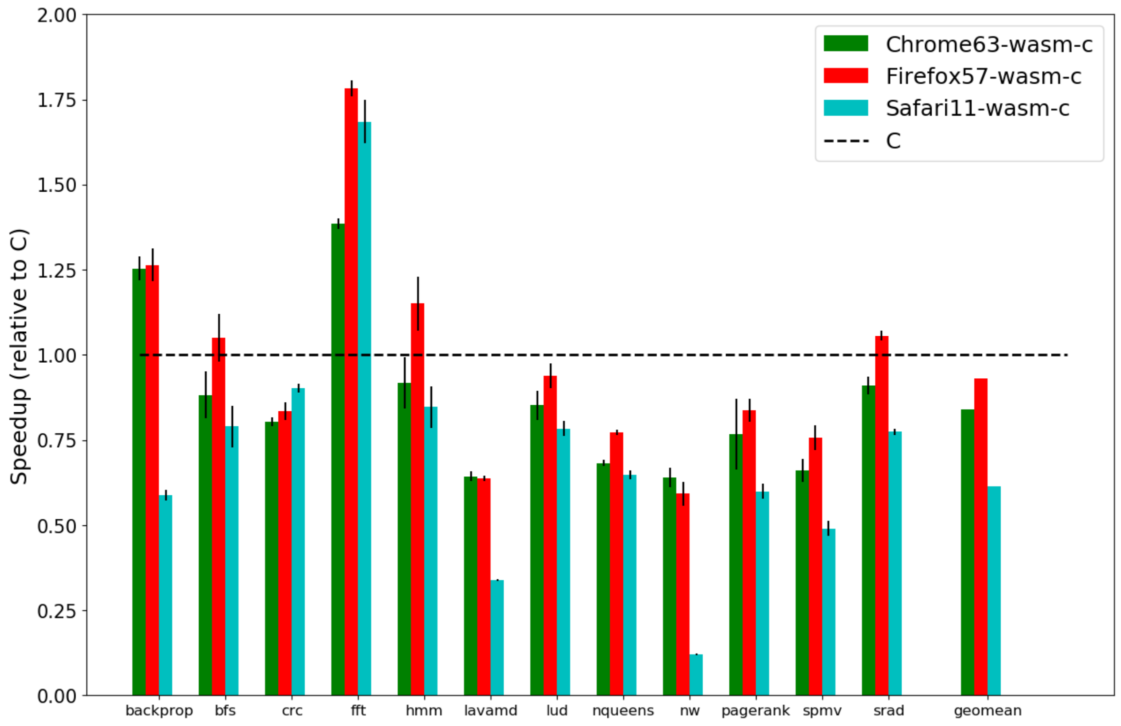

Figure 4: WebAssembly performance relative to C on the MacBookPro 2013.

Fig. 5 provides the geometric means for experiments on our laptop and two desktops. Values lower than 1 indicate a slowdown compared to the native C performance on the same laptop/desktop. We observed that:

- Most browsers execute WebAssembly at least .75x the speed of native C, except for Safari11 on the mbp2013. For the case of the mbp2013, both Chrome and Firefox give superior performance in the context of WebAssembly.
- Firefox achieved the best performance over the three platforms. Achieving even a better performance than native C on the ubuntu-deer device.

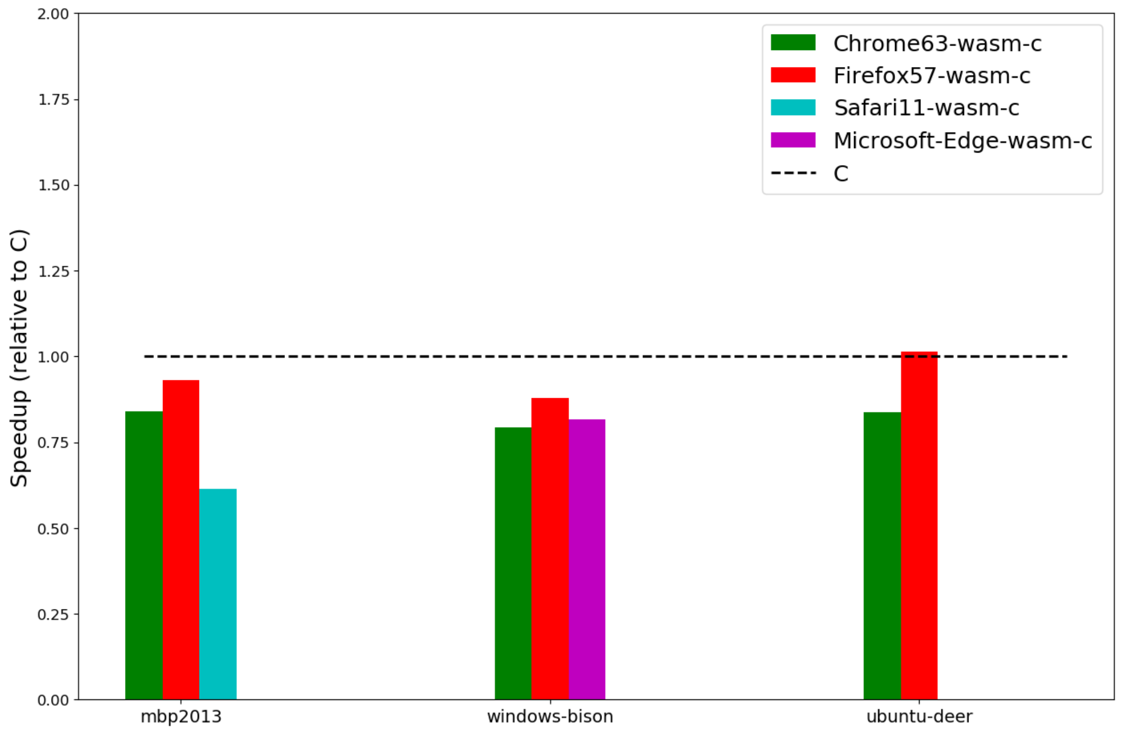

Figure 5: WebAssembly performance relative to C on the different platforms.

5.2 WebAssembly versus JavaScript

All of the major browser developers have integrated support for WebAssembly. Thus, we wanted to see what performance improvement this gives, as compared to the same browser's JavaScript. As shown in Fig. 6, this experiment considered all device/browser combinations where the browser is capable to executing both JavaScript and WebAssembly. These platforms include:

- desktops with different OS: mbp2013, ubuntu-deer, and windows-bison;
- mobile phones from different vendors: iphone10, samsung-s8 and pixel2; and
- tablets with Android and iOS: samsung-tab-s3 and ipad-pro respectively.

The baseline in this experiment is JavaScript, thus WebAssembly is faster when its speedup is greater than 1. Clearly, WebAssembly outperforms JavaScript over all aforementioned platforms, so it is clearly beneficial for numeric programs like those in our benchmark set. Two interesting observations are:

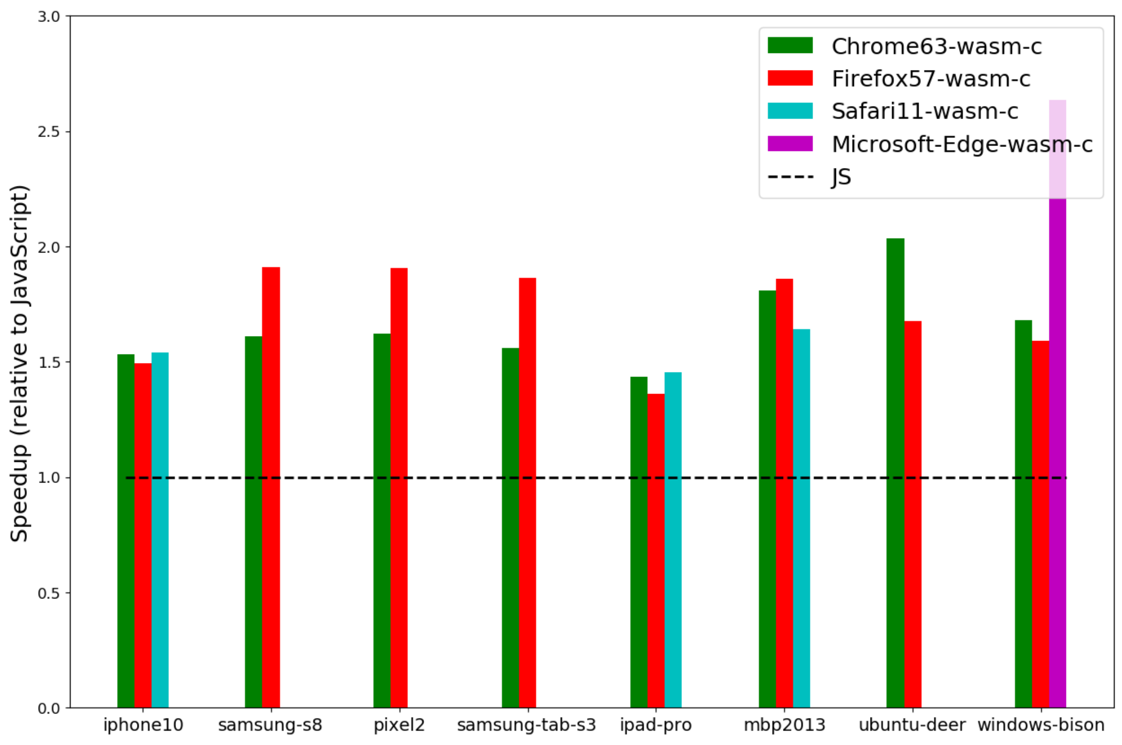

Figure 6: WebAssembly performance relative to JavaScript on the different platforms.

- the benefit of WebAssembly on Android devices (samsung-s8, pixel2, and samsung-tab-s3) is quite important, approaching a factor of 2x the speed of JavaScript.
- the benefit of Microsoft-Edge WebAssembly over Microsoft-Edge JavaScript is the greatest, with a speedup of over 2.5x. However, this is more due to the fact that the Edge JavaScript engine is slower, so the room for improvement is larger.

All browsers demonstrate significant performance improvements for WebAssembly, in the range of 2x speedups over the same browser's JavaScript engine. Furthermore, WebAssembly achieves an overall performance close to 1 against native C, achieving an even greater performance for Firefox57 in the ubuntu-deer platform.

6 RQ3: Portable versus Vendor-specific Browsers

Portable browsers, such as Firefox and Chrome, are supported across a wide variety of platforms. However, vendor-specific browsers are only available on specific platforms. The Samsung Internet Browser, for instance is available for Android devices and it is based on the developer version of the Chrome browser, Chromium [12]. The question of whether vendors take advantage of their knowledge of internal hardware architectures to bring performance to their proprietary browsers presents an interesting experiment. To study this question, we have grouped the devices by vendor and compared the performance of portable browsers against that of vendor-specific browsers, in the last subsection, we dive deep into the outliers of this measurements and determine the sources of inefficiencies for this browsers. In our experiments, the proprietary browsers and their vendors are as follows:

- the Safari browser for the iphone10, ipad-pro, and mbp2013;
- the Microsoft Edge browser for Windows 10 machine, windows-bison;
- the Samsung Internet browser for samsung-tab-s3 and samsung-s8; and
- the Chrome browser for the pixel2.

With the Firefox and Chrome browsers available on all platforms, we decided to compare their JavaScript and WebAssembly performance against the proprietary browser for each platform, except for the case of the Samsung Browser, where WebAssembly result is not available, as the browser lacks support for the language. The results are given in Fig. 7

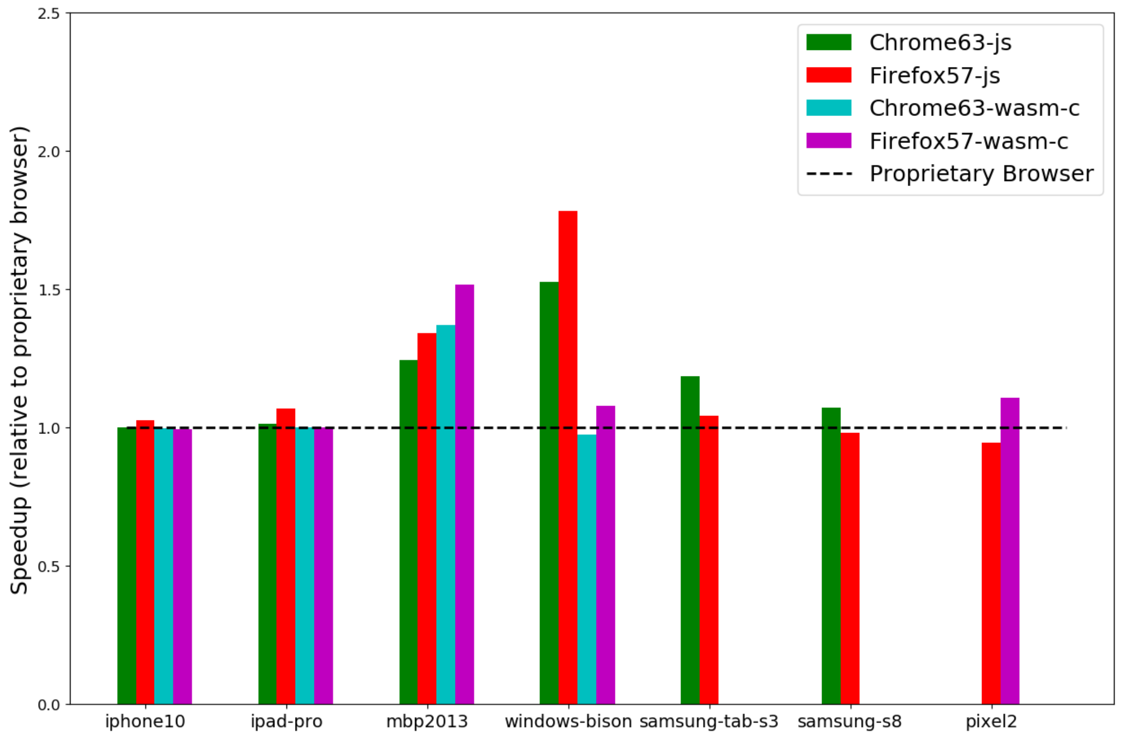

Figure 7: Performance of browsers relative to proprietary respective browsers.

6.1 Safari Browser

Starting from the left on Fig. 7, we can see that the performance of Safari11 for the iPhone X, iPad Pro, and MacBook Pro 2013, remains pretty close to that of Firefox 57 and Chrome 63, perhaps with Firefox 57 having a slightly advantage in the JavaScript front. On the other hand, for the Apple laptop, the Safari11 performance is significantly worse on both JavaScript and WebAssembly when compared to Chrome 63 and Firefox 57.

6.2 Microsoft Edge Browser

Considering the browser performance of Microsoft Edge we can see that although the JavaScript performance of the Microsoft browser is lower than that of Firefox 57 and Chrome 63, their WebAssembly performance is at par with that of the other browser, with a slight edge over the Chrome 63 browser.

6.3 Samsung Internet Browser

Both the Samsung S8 and the Samsung Tab S3, representative of the Samsung Internet browser, presented a similar behaviour for both devices, across all the benchmarks in the Ostrich Suite. This browser supports only JavaScript and it presents very similar JavaScript performance to both Firefox 57 and Chrome 63. In fact, for most of the Ostrich benchmarks, it presents an overall improvement, with the notable exception of lavamd, which was extremely slow for the Samsung browser. Table 5 exhibits the geometric means obtained by the exclusion and inclusion of the lavamd benchmark in the overall performance of the Samsung Browser.

Table 4: Geometric means of speedups for Firefox 57 and Chrome 63 relative to the Samsung Internet Browser on the two Samsung devices

Device	No lavamd		lavamd	
	Chrome63-js	Firefox57-js	Chrome63-js	Firefox57-js
samsung-tab-s3	0.92	0.76	1.19	1.04
samsung-s8	0.85	0.73	1.07	0.98

Our detailed experimental results uncovered a bug in the Samsung JavaScript engine on one benchmark, and we also found the dramatic slowdown in the lavamd benchmark. Thus, it is clear the Samsung team has modified their engine to improve performance of their browser via different optimizations. If we exclude lavamd, the performance of the Samsung Browser is better overall compared to the other browsers, with the exception of the Firefox 57 JavaScript performance. In the last subsection, we explore the lavamd benchmark further, and report the sources of inefficiency for this benchmark in the Samsung Internet Browser.

6.4 Google Chrome OS Browser

Google's own mobile phone, the Pixel 2, allows us to compare the performance between Chrome 63, as the proprietary browser, and Firefox 57. Fig. 7 displays a close but better performance for JavaScript but a slightly lower performance for WebAssembly.

6.5 Performance Issue - Samsung Internet Browser

The performance of JavaScript for different execution engines can vary significantly as they contain different optimization strategies. This can lead to significantly different results in performance even in engines that are similar. Exploring the performance of the Samsung Internet Browser, we found the lavamd benchmark performance to be wildly different as compared to the other engines.

Fig. 8 presents the key code from the body of the hot loop of the original lavamd JavaScript implementation. In the main loop computation, the JavaScript benchmark is implemented via an update of a JavaScript object with four fields `v,x,y,z`.

```
/* fv is an object list; One object has four fields v,x,y,z */
fv[first_i+i].v +=  qv[first_j+j]*vij;
fv[first_i+i].x +=  qv[first_j+j]*fxij;
fv[first_i+i].y +=  qv[first_j+j]*fyij;
fv[first_i+i].z +=  qv[first_j+j]*fzij;
```

Figure 8: The core code in the JavaScript implementation of lavamd.

We created a second version of this benchmark, replacing the array of objects with four typed JavaScript arrays. We compared the performance of the two versions on the different browsers, as shown in Table 5. By replacing this object implementation with four typed JavaScript arrays we saw an overall speedup of over 13x for the Samsung browser, but not for the other browsers, which actually gave better performance on the object version of the benchmark. Thus, the Samsung optimizer seems tuned to produce good code for flat arrays, but does not do well on arrays of objects. Despite the bad performance in this benchmark, the overall performance of the Samsung Browser was excellent, compared to both Firefox 57 and Chrome 63. This highlights how different optimization strategies give very good results on certain computations, but prove costly for other types of operations.

Table 5: Performance results for two versions of lavamd

Browsers	Objects	Arrays	Ratio
Samsung Internet	77.9s	5.80s	13.4x
Firefox 57	2.97s	6.79s	0.44x
Chrome 63	4.95s	6.47s	0.77x

In general the performance of the proprietary browsers is close for all the mobile and tablet devices, while presenting some differences in the windows-bison workstation and the mbp2013 laptop, where the proprietary browsers are performing worse overall. Therefore no clear trend of vendors taking advantage of their hardware architecture knowledge has been found.

7 RQ4: Server-side Node.js versus Client-side Browsers

Since its debut in 2010, Node.js, has grown in popularity as a server side, event-driven language. This event-driven nature, coupled with having the same language at both client and server sides, allowed Node.js to fit naturally with the client and server interaction and become a sought after language for server backends with companies such as LinkedIn, Netflix, PayPal, and Uber implementing their backend in Node.js.

Moreover, Node.js has become increasingly popular as the language for IoT and single board computers, for similar reasons [26, 28]. As has been suggested, [43] numerical computing in this sort of devices will become increasingly important as the awareness of privacy for personal information increases. As such, RQ4, dives deep into the performance of server side JavaScript and WebAssembly comparing firstly against the native performance and secondly, against the corresponding browsers for the workstations, laptop and the raspberry-pi.

7.1 Node.js versus C

Server code must be inherently fast, in order to respond to the demand of incoming requests that could potentially be serving thousands of users at a time. Moreover, in the context of numerical computing, many numerical libraries are implemented in C and C++ to provide performance, making this comparison an interesting question. In this subsection, we explored the relative performance of Node.js against their native C counterpart.

The Node.js versions, platforms, and GCC versions used for each platform are stated in Table 2. Fig. 9 displays the geometric means for speedup of server WebAssembly and JavaScript relative to C for each corresponding platform. As usual, a bar below 1 means the JavaScript/WebAssembly code is slower than the C version.

In all cases the WebAssembly version is much better than the JavaScript version, and for most cases the WebAssembly version is approaching the performance of native C. The best WebAssembly Node.js version achieved a speedup of 0.81 performance against native C. On the other hand, for JavaScript the best overall geometric mean obtained was of 0.45 versus native C. We note that the Node.js performance of ubuntu-deer seems poorer than expected, and as indicated in our future work discussion, we are continuing to examine this issue.

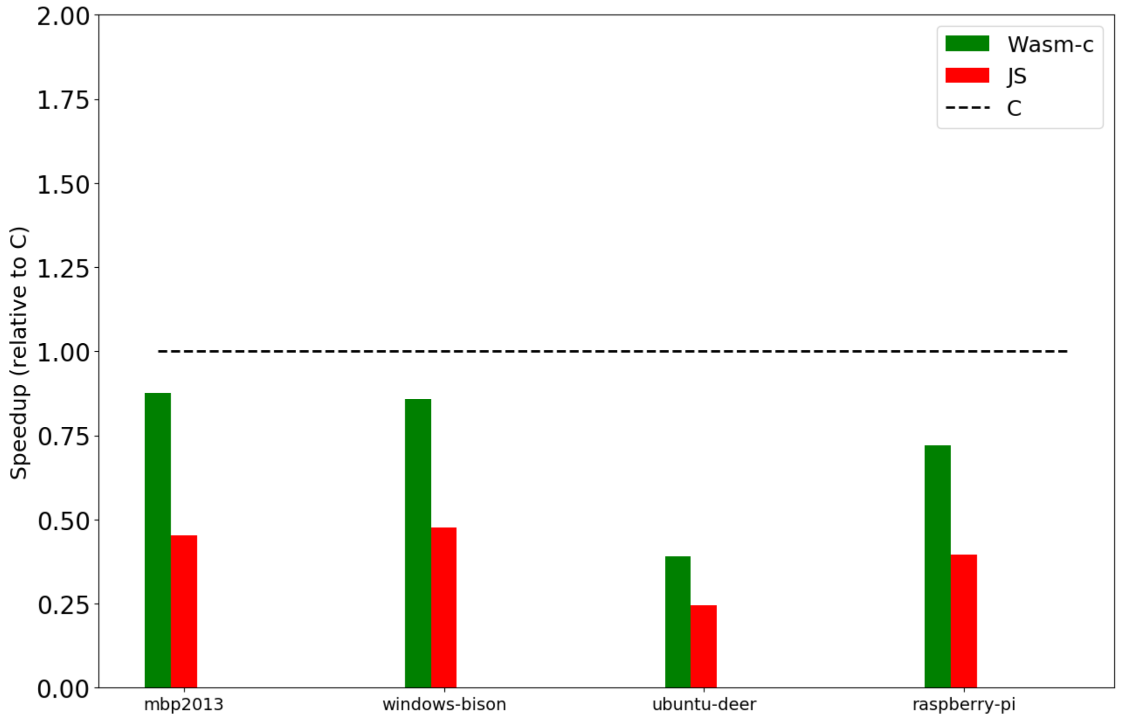

Figure 9: Performance of Node.js in different workstations relative to C.

7.2 Node.js versus Browsers

The second part of this question, studies the numerical performance of Node.js against the browsers for each platform. Table 6 contains the results of Node.js speedups relative to the corresponding browser.

Table 6: Browser speedup performance relative to their respective WebAssembly and JavaScript Node.js versions for each device.

Device	*Chrome63-js*	*Chrome63-wasm-c*	*Firefox57-js*	*Firefox57-wasm-c*	*Safari11-js*	*Safari11-wasm-c*	*Microsoft-Edge-js*	*Microsoft-Edge-wasm-c*	*Chromium56-js*
mbp2013	1	1	0.9	0.9	1.2	1.4	-	-	-
windows-bison	1	1.1	0.9	1	-	-	1.5	1.1	-
ubuntu-deer	0.6	0.5	0.4	0.4	-	-	-	-	-
raspberry-pi	-	-	-	-	-	-	-	-	1.1

Given the best performing browsers, Firefox 57, and Chrome 63, we can see that the results of these speedups are fairly close for both WebAssembly and JavaScript, indicating the performance of server-side Node.js as being really close to the best performing client-side browsers. Server-side Node.js performance was slightly better than the Safari 11 and Microsoft Edge browsers.

The exception is the ubuntu-deer machine, where the performance of the browsers is better by more than a factor of 2. Further exploration of this result is required to fully understand the slow performance of Node.js in the ubuntu-deer machine and will be part of our future work.

Node.js was overall slower than native C code for both JavaScript and WebAssembly, although the best WebAssembly gave reasonable performance of 0.8 the speed of native C. Server-side Node.js matched the best browser performance for both JavaScript and WebAssembly for all the devices, except for the ubuntu-deer device.

8 RQ5: Best Performers

In this last section, we finalize the results by making a comparison of all devices, against a common baseline. As the baseline we have chosen the Raspberry Pi native C performance, as it is the lowest performing device, and it allows to look at the factor speedups more easily. Table 7 shows the results of this experiment. The last row of the table, shows the overall arithmetic mean of all devices for both the Firefox57 and Chrome63 browsers. The last two columns on the table, present the mean WebAssembly and JavaScript performance of each device, and the overall web performance for the device along all browsers explored for it.

The best performing browser for numerical computations based on the Ostrich Suite is the new, upgraded, Firefox Quantum version 57 for both JavaScript and WebAssembly. In terms of the best

Table 7: Device performance across environments using the native C raspberry pi implementation as baseline for geometric means.

Device	*chrome63-wasm-c*	*chrome63-js*	*chromium56-js*	*firefox57-wasm-c*	*firefox57-js*	*safari11-wasm-c*	*safari11-js*	*microsoft-edge-wasm-c*	*microsoft-edge-js*	*samsung-internet-js*	***mean-wasm***	***mean-js***
iphone10	8.1	5.3	-	8.1	5.4	8.1	5.3	-	-	-	8.1	5.3
samsung-s8	3.3	2	-	3.6	1.9	-	-	-	-	1.9	3.4	1.9
pixel2	3.4	2.1	-	3.7	2	-	-	-	-	-	3.6	2
ipad-pro	5.1	3.5	-	5.1	3.7	5.1	3.5	-	-	-	5.1	3.6
samsung-tab-s3	2.1	1.4	-	2.3	1.3	-	-	-	-	1.2	2.2	1.3
mbp2013	7.9	4.3	-	8.7	4.7	5.7	3.5	-	-	-	7.4	4.2
ubuntu-deer	8.9	4.4	-	10.8	6.4	-	-	-	-	-	9.9	5.4
windows-bison	8.5	5	-	9.4	5.9	-	-	8.7	3.3	-	8.9	4.7
raspberry-pi	-	-	0.4	-	-	-	-	-	-	-	-	0.4
mean	5.9	3.5	-	6.5	3.9	-	-	-	-	-	-	-

devices, the two workstations are first, as expected, followed by the new iPhone 10, which surprisingly comes before the MacBook Pro laptop from 2013. This illustrates the power of modern mobile devices and the excellent performance of the Apple device powered by their A11 chip. The order of the next performers is mostly as expected, with the iPad Pro coming next, for both JavaScript and WebAssembly, followed by the Samsung S8 and Pixel 2 devices, where the performance was rather close. Surprisingly, the Samsung Tab S3 tablet came after the smartphones. Last but not least, as expected, the low powered Raspberry Pi model 3.

The best overall browser performance was by the Firefox 57 browser, the order of device performance has the two workstations first, surprisingly followed by the iPhone 10, the rest of the order is: MacBook Pro 2013 laptop, iPad Pro, Samsung S8, Pixel 2, Samsung Tab S3 and lastly, Raspberry Pi Model B.

9 Conclusions and Future Work

In this paper we addressed five major research questions which examined the current performance of JavaScript and WebAssembly on both the client-side web browsers and the server-side Node.js. These research questions seek to determine how well these technologies perform for numerical computations.

We have studied a wide range of devices including workstations and laptops, mobile devices including state-of-the-art smartphones and tablets, and the Raspberry Pi as representative of an IoT device. In order to perform standardized experiments, we designed and implemented a platform for performing the experiments using a central benchmark repository based on the Wu-Wei version of the Ostrich benchmark suite, and an architecture for deploying on our wide range of target architectures, including mobile devices.

Our first research question examined the performance of new versus old JavaScript browser technologies on our numerical benchmark set. Given that both browsers have undergone major changes in recent years, and that they now also consider other factors for optimization, such as start-up speed and responsiveness, this research question is quite relevant. For our benchmarks, compared to the technology from 2014, the performance of current Chrome browser appears slightly slower, whereas the current Firefox browser is slightly faster. In both cases the results are still within a 2x slow down over native C, which remains quite impressive.

Our second research question addressed the impact of the new WebAssembly technology, that has been incorporated into most of the major browsers. For our numerical benchmark set we found that WebAssembly provides excellent performance, with WebAssembly in Firefox nearing the performance of native C, for our numerical benchmark set. All browsers which incorporated WebAssembly technology showed impressive gains over their JavaScript engines. Thus, it would appear that WebAssembly is very important for the future of efficient numeric computations on the web.

Our third research question looked at the relative performance of portable (Firefox and Chrome) browsers versus OS-specific browsers (Samsung Internet Browser, Microsoft Edge and Safari). One might expect that OS-specific browsers would outperform portable browsers when applied to their target OS. However, this was not the case. Both Chrome and Firefox outperformed Microsoft Edge, and performed slightly better or similarly to Safari and Samsung Internet Browser.

Our fourth research question investigated the relative performance of server-side Node.js versus client side browsers, for both JavaScript and WebAssembly. We expected to find that server-side Node.js would perform similarly or better than the browser. This was the case when comparing against most browsers, but we found that the Firefox browser was significantly faster than the equivalent Node.js. Since this was so surprising, we revisited this experiment several times, and will continue to explore why this is the case.

Our final research question compared all the devices and browser and Node.js technologies. We normalized everything to the execution speed of native code C of on our slowest device, the Raspberry Pi. A value greater than 1 indicates a better score. As expected, the laptop and workstations scored highly with mean scores of 4 to 5.5 for JavaScript and scores of 7.4 to 8.9 for WebAssembly. Also, as expected, the lowest score was 0.4 for JavaScript on the Raspberry Pi. The big surprise to us was that the iPhone X performed in the same range as the workstations and laptop. The other mobile devices performed somewhat as expected, in the midrange between the Raspberry Pi and the workstations. From the browser perspective, over all the devices, Firefox and Chrome with WebAssembly were both excellent performers, with average scores of 6.5 and 5.9 respectively.

We will make our raw data and experimental setup available (see additional material), and we hope that our results, and the experimental setup will be useful for both compiler developers and for those who would like to implement numeric computations on the web. Our results show that good performance can be achieved for numeric benchmarks, and excellent performance can be achieved with WebAssembly.

We intend to continue our work by extending our benchmark set to include a variety of machine learning benchmarks. We believe that exploiting all of the computing power available on mobile devices could be very interesting in this field. We will also develop some core computations that can be used to expose the major differences in optimizations in different browsers, so that all browser engines can use this information to provide improved performance.

References

[1] Apache Cordova. URL: `https://cordova.apache.org/`.

[2] asm.js. URL: `http://asmjs.org/spec/latest/`.

[3] Features to add after the MVP. URL: `http://webassembly.org/docs/future-features/`.

[4] Firebase. URL: `https://firebase.google.com/`.

[5] JavaScript Heap Size Reduction. URL: `https://v8project.blogspot.ca/2016/10/fall-cleaning-optimizing-v8-memory.html`.

[6] JavaScript Typed Arrays. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays`.

[7] Kraken Benchmark Suite. URL: `http://krakenbenchmark.mozilla.org/`.

[8] Non-Web Embeddings. URL: `http://webassembly.org/docs/non-web/`.

[9] Octane Benchmark Suite. URL: `https://developers.google.com/octane/`.

[10] Speedometer. URL: `http://browserbench.org/Speedometer/`.

[11] Standard ECMA-262 ECMAScript Language Specification 8th Edition (June 2017). URL: `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

[12] The Chromium Projects. URL: `https://www.chromium.org/Home`.

[13] Asanovic, Krste and Bodik, Ras and Catanzaro, Bryan C. and Gebis, Joseph J. and Husbands, Parry and otherss,sanovi, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[14] Lars Bak. Google Chrome's Need for Speed. URL: `http://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html`.

[15] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J. W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, October 2009.

[16] Phillip Colella. Defining software requirements for scientific computing, 2004.

[17] Henry Zhu Daniel Tschinder, Logan Smyth. Babel is a JavaScript compiler. URL: `https://babeljs.io/`.

[18] Laouratou Diallo, Aisha Hashim, Momoh Jimoh Eyiomika Salami, Sara Babiker Omer Elagib, and Abdullah Ahmad Zarir. The Rise of Internet of Things and Big Data on the Cloud: Challenges and Future Trends. 10:49–56, 03 2017.

[19] Tomasz Fabisiak and Arkadiusz Danilecki. Browser-based harnessing of voluntary computational power. *Foundations of Computing and Decision Sciences*, 42(1):3–42, 2017.

[20] Faiz Khan and Vincent Foley-Bourgon and Sujay Kathrotia and Erick Lavoie . Ostrich Suite – DLS'14 version, 2018. URL: `https://github.com/Sable/Ostrich`.

[21] Wu C. Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. OpenCL and the 13 Dwarfs: A Work in Progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 291–294, New York, NY, USA, 2012. ACM.

[22] Vincent Foley-Bourgon. MatJuice: A Matlab to JavaScript static compiler, 2016.

[23] Vincent Foley-Bourgon and Laurie Hendren. Efficiently Implementing the Copy Semantics of MATLAB's Arrays in JavaScript. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 72–83, 2016.

[24] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based Just-in-time Type Specialization for Dynamic Languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

[25] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[26] Dominique Guinard and Vlad Trifa. *Building the Web of Things: With examples in Node.js and Raspberry Pi.* Manning Publications Co., 2016.

[27] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200. ACM, 2017.

[28] Steve Hodges, Stuart Taylor, Nicolas Villar, James Scott, Dominik Bial, and Patrick Tobias Fischer. Prototyping Connected Devices for the Internet of Things. *Computer*, 46(2):26–34, 2013.

[29] Apple Inc. About Face ID advanced technology. URL: `https://support.apple.com/en-ca/HT208108`.

[30] Alexandr A. Kalinin, Selvam Palanimalai, and Ivo D. Dinov. SOCRAT Platform Design: A Web Architecture for Interactive Visual Analytics Applications. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, HILDA'17, pages 8:1–8:6, 2017.

[31] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, Erick Lavoie, Hanfeng Chen, Prabhjot Sandhu, and David Herrera. Ostrich Suite – Wu-Wei Compatible Version, 2018. URL: `https://github.com/Sable/ostrich-suite`.

[32] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, Erick Lavoie, and Laurie J. Hendren. Using JavaScript and WebCL for numerical computations: a comparative study of native and web technologies. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 91–102, 2014.

[33] Erick Lavoie, Laurie Hendren, and Frédéric Desprez. Pando: A Volunteer Computing Platform for the Web. In *Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on*, pages 387–388. IEEE, 2017.

[34] Mark Mayo. Introducing the New Firefox: Firefox Quantum. URL: `https://blog.mozilla.org/blog/2017/11/14/introducing-firefox-quantum/`.

[35] Edward Meeds, Remco Hendriks, Said Al Faraby, Magiel Bruntink, and Max Welling. MLitB: machine learning in the browser. *PeerJ Computer Science*, 1:e11, 2015.

[36] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Browser Wars: Metaphor, Web Browser, Microsoft, Internet Explorer, Netscape, Netscape Navigator, Mozilla Firefox, Google Chrome, Safari (Web Browser), Opera (Web Browser), Usage Share of Web Browsers*. Alpha Press, 2009.

[37] Rudolph Pienaar, Ata Turk, Jorge Bernal-Rusiel, Nicolas Rannou, Daniel Haehn, P. Ellen Grant, and Orran Krieger. *CHIPS – A Service for Collecting, Organizing, Processing, and Sharing Medical Image Data in the Cloud*, pages 29–35. Springer International Publishing, 2017.

[38] Gregor Richards. JSBench Benchmark Suite. URL: `http://jsbench.cs.purdue.edu/`.

[39] Charles Severance. JavaScript: Designing a Language in 10 Days. *Computer*, 45(2):7–8, February 2012.

[40] Katie Shilton. Four Billion Little Brothers?: Privacy, mobile phones, and ubiquitous data collection. *Communications of the ACM*, 52(11):48–53, 2009.

[41] V8 team. Launching Ignition and TurboFan. URL: `https://v8project.blogspot.ca/2017/05/launching-ignition-and-turbofan.html`.

[42] V8 team. Retiring Octane. URL: `https://v8project.blogspot.ca/2017/04/retiring-octane.html`.

[43] Rolf H Weber. Internet of Things–New security and privacy challenges. *Computer law & security review*, 26(1):23–30, 2010.

[44] I. Yaqoob, E. Ahmed, I. A. T. Hashem, A. I. A. Ahmed, A. Gani, M. Imran, and M. Guizani. Internet of Things Architecture: Recent Advances, Taxonomy, Requirements, and Open Challenges. *IEEE Wireless Communications*, 24(3):10–16, June 2017.

[45] Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.