Numerical Computing on the Web: Benchmarking for the Future

David Herrera
McGill University
Canada
david.herrera@mail.mcgill.ca

Hanfeng Chen
McGill University
Canada
hanfeng.chen@mail.mcgill.ca

Erick Lavoie
McGill University
Canada
erick.lavoie@mail.mcgill.ca

Laurie Hendren
McGill University
Canada
hendren@cs.mcgill.ca

Abstract

Recent advances in execution environments for JavaScript and WebAssembly that run on a broad range of devices, from workstations and mobile phones to IoT devices, provide new opportunities for portable and web-based numerical computing. Indeed, numerous numerical libraries and applications are emerging on the web, including Tensorflow.js, JSMapReduce, and the NLG Protein Viewer. This paper evaluates the current performance of numerical computing on the web, including both JavaScript and WebAssembly, over a wide range of devices from workstations to IoT devices. We developed a new benchmarking approach, which allowed us to perform centralized benchmarking, including benchmarking on mobile and IoT devices. Using this approach we performed four performance studies using the Ostrich benchmark suite, a collection of numerical programs representing the numerical dwarf categories identified by Colella. We studied the performance evolution of JavaScript, the relative performance of WebAssembly, the performance of server-side Node.js, and a comprehensive performance showdown for a wide range of devices.

CCS Concepts • **General and reference** → *Empirical studies*; *Performance*;

Keywords performance, web-based scientific computation, scientific web benchmarking, JavaScript, WebAssembly, Ostrich benchmarks

DLS '18, November 6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6030-2/18/11…$15.00
https://doi.org/10.1145/3276945.3276968

ACM Reference Format:
David Herrera, Hanfeng Chen, Erick Lavoie, and Laurie Hendren. 2018. Numerical Computing on the Web: Benchmarking for the Future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '18), November 6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3276945.3276968

1 Introduction

Computation via web-browsers is becoming increasingly more attractive, and computation-heavy numerical web-based applications and libraries are becoming more prevalent. Recently, the introduction of WebAssembly to all major browsers has provided even more potential for efficient web-based numerical computing [22]. Further, the proliferation of browser-enabled devices provides enormous computation capacity including devices of all sizes, from workstations and laptops to mobile phones and Internet-of-Things (IoT) devices. This paper presents a benchmarking approach and framework that can evaluate the performance of both JavaScript and WebAssembly over a wide variety of browsers and devices. We use this benchmarking approach to evaluate the performance of numerical programs.

Our previous 2014 study, focused only on laptops/workstations, showed that browser-based execution engines, when executing JavaScript and using the best technologies of the time, had execution times within a factor of 1.5 to 2 of the performance of native C [26]. Further, at that time, two notable performance enablers were demonstrated: (1) the use of JavaScript typed arrays [32]; and (2) the use of asm.js [10].

The 2014 study demonstrated that web-based numerical computing was becoming very attractive, which subsequently inspired web-based numerical projects including: (1) MatJuice: a translator from MATLAB to JavaScript [18], (2) MLitB: machine learning in the browser [30], (3) Pando: a web-based volunteer computing platform [29], (4) SOCRAT: a web architecture for visual analytics [25], and (5) CHIPS: a system for cloud-based medical data [33]. Recently, there are also 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. An important example is the emerging trend of moving expensive computations from cloud-based computing platforms to the edge of the network [13, 38], especially those involving machine learning tasks with newer toolkits such as TensorFlow.js [8]. Applications such as these require more performance for numerical computation tasks than was typical of general web applications as studied in 2010 [36].

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, providing even more opportunities for efficient web-based numerical computing. On the software side, the Just-In-Time (JIT) compilers in Chrome and Firefox continue to evolve, with major changes and redesign of the compilation engines; and new browsers with JITs, such as Samsung's Internet Browser, and Microsoft's Edge have appeared. Furthermore, the invention and adoption of WebAssembly have provided a new common program representation well 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. Given the increasing use of the web for numerical computing, the major changes in browser technologies, and the increasing power of mobile devices, we feel that this is an ideal time to perform a broad study of numerical computing on the web. We make the following contributions:

Benchmarking approach: We designed and implemented a generalized and centralized benchmarking approach to simplify the tasks required to track the relative evolution of multiple language technologies on a growing number of software and hardware configurations. Our approach contributes: (1) the Wu-Wei Benchmarking Toolkit [28] which comprises both reusable file system conventions to organize artifacts as well as tools to automate common tasks of benchmarking, which can run from a laptop or workstation; and (2) a new execution environment compatible with the toolkit that enables the remote execution of benchmarks on a wide range of devices, including mobile and IoT devices. We use the toolkit and environment for the four studies done for this paper.

Old versus new JavaScript engines: Our first study examines how the numerical performance of JavaScript engines in Chrome and Firefox has changed (for better or worse) since the 2014 study, using workstations and laptop architectures similar to those used in 2014. In addition to showing the performance evolution, it also provides a new baseline to which we can compare new browsers, new technologies such as WebAssembly, and more devices, including mobile and IoT devices.

WebAssembly versus C and JavaScript: 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 such as C. WebAssembly, a new typed intermediate representation for programs executing on web browsers, provides many more opportunities for optimizations [22]. Thus, in this study we measure the relative performance of JavaScript and WebAssembly, and examine whether WebAssembly can come close to the performance of native C. We perform this study across the complete spectrum of devices from laptops/workstations to an IoT device.

Server-side Node.js performance: Node.js provides a convenient mechanism to execute JavaScript and WebAssembly directly on a server, rather than via a web browser. This study determines if numerical server-side Node.js performance comes close to native C performance, and if there are any significant performance differences between sequential server-side Node.js and the equivalent browser-based JavaScript/WebAssembly engines.

JavaScript/WebAssembly/Device showdown: Our final study is a showdown between all devices for both JavaScript and WebAssembly. The intent is to measure the performance of each language/device combination, so that we can compare all of them easily. In this way, we can see the performance gap between workstations/laptops and other devices such as tablets, mobile phones and IoT devices. This allows us to answer questions such as: "Is the numerical performance of modern mobile devices close to that of workstations and laptops?".

Overall, our benchmarking approach should be of interest to those who wish to evaluate performance of JavaScript and/or WebAssembly on a variety of devices. We believe that many different languages may start to target WebAssembly, and so this can be very useful in their performance evaluation. Further, our performance studies of numerical computation on the web should be of interest to: (1) end-users who wish to execute numerically-intensive web-based applications on personal devices; (2) scientific programmers who wish to distribute and execute their code on the web; (3) VM developers who wish to evaluate their execution engines for numerical computing; and (4) to those who have a general interest in the current state of numerical computing on the web. Our complete results, benchmarks and experimental platform are publicly available. The experimental set-up and approach are available at https://github.com/Sable/ostrich-suite. The data and main results are available at https://github.com/Sable/dls18-ostrich.

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 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 increasing the demand for improved JavaScript performance.

2.1 JavaScript

Since the standardization of JavaScript and the introduction of dynamic loading to the web, the developer community took on the challenge to make JavaScript and web applications both scalable and efficient. Efficiency was achieved through the introduction of the JavaScript Engines and their JIT compilers [11, 19]. Moreover, the ECMAScript standard [6], along with the W3C standards [42], 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, ranging from smartphones and desktops to IoT devices. In 2014, the sequential numerical performance of the best JavaScript engines was recorded to be within a factor of 1.5 to 2 of native C [26].

The maturation of web platforms has given rise to increasingly more intensive computations. JavaScript as the only built-in language of the web has presented challenges in providing support to some of these applications. Further efforts to improve JavaScript performance [12, 24, 46] have led to, WebAssembly, a new binary code format.

2.2 WebAssembly

WebAssembly [22] 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.

The competitive performance of WebAssembly compared with native code over micro-benchmarks has been addressed by Haas et al. [22]. In our study we compare WebAssembly versions of programs to both C and JavaScript versions, to measure the relative performance of WebAssembly versus both native C code and JavaScript. Further, we explore a range of numerical benchmarks from the Ostrich Benchmark Suite [26] compiled from C to WebAssembly for a variety of environments representative of the web ecosystem. Currently, the Emscripten toolkit [46], based on the c-lang-llvm pipeline to generate WebAssembly code, 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 [44]. The low-level nature of the language should offer many opportunities for optimizations that would benefit numerical computations. At the time of writing, a fixed-width SIMD feature, and parallelism via threads are in the *in-progress* stage for WebAssembly [43]. Furthermore, WebAssembly supports different integer types and single precision floating-points, which are not currently supported by JavaScript.

2.3 JavaScript Performance

There have been many benchmark suites proposed for JavaScript by both researchers [35] and browser vendors [2–4]. Some are not maintained anymore [3], while others are not representative of numerical computing typically used by scientists and engineers [35].

In terms of other performance studies of JavaScript, in 2016, Selekovic et al. [37] explored 16 popular JavaScript repositories. Their goal was to find the main inefficiencies in code that were common in the repositories, and they identified the incorrect use of APIs as the most prominent cause. JSBench [34] explored the distinction between JavaScript performance of real websites compared to a benchmark suite, pointing out that using a benchmark suite as an optimizing objective may lead to a false sense of performance in terms of real-world use of JavaScript.

For our purposes, we are specifically looking at the web technologies potential for numerical performance. All the benchmarks suites mentioned fail to fill our target goals in two aspects. First, they are not benchmarks representative of the typical kernels seen in numerical computations, and secondly, they have no equivalent WebAssembly implementation. Instead, to measure numerical performance of JavaScript and WebAssembly, we have used the most recent version of Ostrich Benchmark Suite [26]. These benchmarks have both JavaScript and C implementations that can be used to: (1) compare C and and JavaScript performance; and (2) compare both to WebAssembly by translating the C versions with a suitable compiler.

As a last point, there does not exist an open source benchmarking toolkit to cohesively, and centrally measure performance across a variety of devices on the Web.

2.4 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 [23]. In general, the need for the privacy protection of personal data and the rising importance of machine learning models, numerical web computations hosted at the host environment are becoming increasingly important [39].

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 [45].

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 [15], as well as providing potential devices to be used in volunteer computing platforms [16, 29].

3 Benchmarking with the Wu-Wei Toolkit

Over the last years, our research and this study in particular highlighted the need to develop reusable solutions to assess the performance of language implementations. We successfully addressed: (1) the combinatorial explosion of hardware and software configurations that need to be compared, which was manifest in the significant redundancy of the Makefile infrastructure of our previous study [26]; (2) the tediousness of performing experiments and organizing results; and (3) the difficulty in running benchmarks remotely on mobile and embedded devices.

Our solution was to develop: (1) *file system conventions* to organize the various artifacts, the experiment parameters, the measurements and reports, and the expected results from a valid execution; (2) *command-line tools* to automate the tedious experimental parts; and (3) a strategy for *remote execution* of benchmarks. We provide a high-level sketch of the solutions here, more detail can be found in the *wu-wei-handbook* [28].

3.1 File System Conventions

Our file system conventions are organized around a *repository*, that is a directory that contains many *artifacts* required for the study. Each artifact is itself contained in its own directory that minimally contains a file, e.g. `compiler.json`, that describes the artifact and how to perform actions with it, if applicable. It may also contain other files, such as a licence.

Artifacts are organized in categories, stored in corresponding directories, that follow the main dimensions of the comparative studies we made in the past. *Benchmarks* represent general algorithms or applications, which are further divided into *Implementations* in possibly different programming languages. The benchmark description file contains pre-defined inputs as well as their expected output, which must be consistent for all implementations. *Compilers* take benchmark implementations as inputs and generate a new *Build* artifact. They can be used to cross-compile benchmarks to other languages or simply package them for execution. *Environments* execute the *Builds* to produce metrics, stored in a *Run*. *Runs* may then be combined and analyzed to produce different reports. In addition, *Platforms* are used to store the different hardware/-operating system combinations analyzed. And finally, an *Experiment* contains the parameters that were used for a specific study and may refer to other artifacts as dependencies.

3.2 Command-line Tools

The description files of the previous artifacts are automatically found and processed by tools to automate various tasks. The tools are organized around a *configuration*, which represents a combination of various artifacts to measure, e.g. *crc* benchmark, *js* implementation, *chromium38* environment, *browserify* compiler, and *mbp2018* platform. By automating the generation of configurations from generic description files, we removed much of the boiler-plate code that was previously necessary.

Common tasks have dedicated tools, all invoked on the commandline with the prefix `wu`. The command `wu init` creates a repository for performing a study by creating the sub-directories to store the different artifact categories. `wu platform` produces a report on the hardware capabilities of the current platform. `wu install` adds an artifact in the repository, either copying or downloading the necessary files and adding its dependencies. `wu build` compiles a benchmark's implementation sources and creates a Build. `wu run` executes a Build on an Environment and saves the measured metrics in a Run. The output of the execution is compared to its expected result specified in the Benchmark description and returns an error if they differ. `wu report` combines the results from multiple Runs into a table and automatically computes the average and standard deviation. It can also produce a table of relative speeds, as speedups or slowdowns.

3.3 New Environment for Remote Execution

Our new Environment is illustrated in Fig. 1 and enables the execution of benchmarks in a remote environment, such as mobile phones, tablets, and Raspberry Pi boards. It works in combination either with a portable mobile application that we developed for both iOS and Android using web technologies and the Apache Cordova framework [9] or a command-line script that can be run on the device. Both sides communicate through Firebase [17], a cloud database that provides real-time updates.

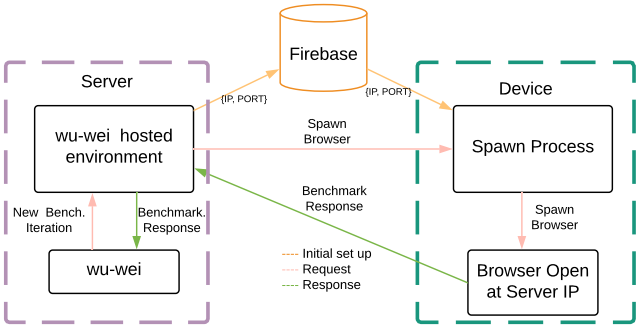


Figure 1. Environment architecture used by Wu-Wei to record the data from each browser.

For mobile devices, when executed by *wu run*, the Environment spawns a local server, which serves the benchmark files and listens for results, and sends its address as well as the targeted browser to Firebase, in the form of a request intended for the remote platform. On the other side, the mobile application receives the request and opens the requested browser with the address, which downloads the files, executes the benchmarks, and answers with the result and metrics. For other kinds of remote devices, such as the Raspberry Pi board, the strategy is similar but a listening script is used instead and the files are sent through Firebase rather than from a local server.

Our new benchmarking infrastructure enabled us to reorganize the artifacts of our previous study [26] as independent components, removing much of the redundancy present in the original infrastructure and making it easier to extend our study to new compilers and environments. Further, our tools made the execution of benchmarks and exploration of results easier. The addition of a new strategy for remote execution significantly extended the number of devices that could be tested, avoided code replication of the benchmark suite, and enabled a centralized, and structured data collection. To the best of our knowledge, we were the first to create such tools for comparative performance studies on a comprehensive set of devices, execution environments, and benchmark implementations.

4 Methodology

In this section, we explain our choice of benchmarks, improvements made to them, as well as details about execution and timing. Lastly, we give the specifications and justifications for all the different environments and devices we used during the performance studies. [1]

4.1 Choice of Benchmarks

To measure the numerical performance of JavaScript and WebAssembly, we have used the most recent version of the Ostrich Benchmark Suite [26], representing the 7 Dwarf categories in numerical computing identified by Colella [14]. Each benchmark contains C and JavaScript equivalent implementations. This made it convenient to add WebAssembly, since for the WebAssembly performance we simply used the C implementation of each benchmark compiled using the 1.37.22 version of the Emscripten [46] compiler. This allowed us to quantify the performance of both JavaScript/WebAssembly against native C, with the consistency provided by having the equivalent implementations across languages.

4.2 Improvements to Ostrich

The Ostrich benchmarks were improved in three different ways. (1) we reorganized the benchmarks to abide to the conventions of Wu-Wei in terms of folder structure and configuration files; (2) we updated the Ostrich benchmarks to use current JavaScript idioms and libraries for arrays; [2] (3) we added an output verification section to each benchmark, which verifies the correct output is given at the end of each run. [3]

4.3 Execution and Timing

Each benchmark was executed a total of 30 times for each environment as suggested by Georges et al. [20], this allowed us to approximate the distribution of measurements as a Gaussian and obtain an appropriate standard deviation. Moreover, for each benchmark run, a new browser tab was spawned in order to avoid the cached JIT optimizations of the JavaScript engines. We include the compilation time in the measurements since this is representative of how numerical computations are actually performed in practice: a user is interested in obtaining results with minimum latency the first time it is run rather than maximizing throughput over long-lived computations. The time for each benchmark represents the main loop computation of a *medium size input* as defined by the Ostrich benchmarks. 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 a specific environment and platform across all the benchmarks, and we used that as a comparison measure against other platforms.

4.4 Representative Devices

Table 1 contains the specifications of the devices used for our experiments. We considered a wide range of devices covering four major categories: Desktops & Workstations, Tablets, Smart Phones, and IoT. We have chosen the initial categories based on the perspective of users. However, each category corresponds to different form factors, which determines cooling and battery, and thus throttling and turbo behaviours of the system and is thus interesting from that perspective as well. For each of these categories, we aimed to have at least one representative device. To evaluate desktop and laptop performance we have executed our experiments on three machines,

[1] The data and plots for the study are available at: https://github.com/Sable/dls18-ostrich

[2] The benchmarks now use the `ndarray` [5] library and array literals instead of the `new Array()` construct

[3] A Wu-Wei compatible experiment is available under: https://github.com/Sable/ostrich-suite

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

Platform	Device	CPU	RAM	OS
Laptops & Workstations	MacBook Pro 2018 (mbp2018)	Intel Core i5 @ 2.3 GHz	8 GB	MacOS 10.13.4
	Ubuntu Workstation (ubuntu-deer)	Intel Core i7-3820 @ 3.60 GHz	16 GB	Ubuntu 16.04
	Windows Workstation (windows-bison)	Intel Core i7-3820 @ 3.69 GHz	16 GB	Windows 10 Enterprise
Single Board Computers	Raspberry Pi (raspberry-pi-3)	ARM Cortex A53 @ 1.20 GHz	1 GB	Linux Raspberry Pi 4.9.35-v7
Tablets	Apple iPad Pro (ipad-pro)	Hexa-core Apple Fusion @ 2.39 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 Snapdragon 835 @ 1.90 - 2.35 GHz	4 GB	Android 8.0.0
	Apple iPhone X (iphoneX)	Apple Fusion hexa-core @ 2.39 GHz	4 GB	OSX 11.0.3

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

Environment	Version	Devices
Chrome63	63.0.3239.84	samsung-s8, pixel2, ipad-pro, samsung-tab-s3, mbp2018, ubuntu-deer, windows-bison, iphoneX
Chromium38	38.0.2125.0	mbp2018, ubuntu-deer
Firefox57	57.0.2	samsung-s8, pixel2, ipad-pro, samsung-tab-s3, mbp2018, ubuntu-deer, windows-bison, iphoneX
Chromium56	56.0.2923.84	raspberry-pi
Firefox39	39.0	mbp2018, ubuntu-deer
Safari11	11.0.1	iphoneX, ipad-pro, mbp2018
Microsoft-Edge	41.16299.15.0	windows-bison
Node.js	8.9.1	mbp2018, raspberry-pi,ubuntu-deer,windows-bison
Native	GCC 5.4.0	mbp2018, raspberry-pi,ubuntu-deer,windows-bison

namely, a MacBook Pro 2018 laptop (mbp2018), an Ubuntu-based desktop (ubuntu-deer), and a similar Windows-based desktop (windows-bison).[4] These machines allowed us to measure performance for the three major operating systems and also 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. 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. Finally, to represent IoT devices, we have selected the Raspberry Pi 3 model B (raspberry-pi) due to its popularity, as a representative of the single-board computers.

4.5 Browsers and Execution Engines

For each device, we have experimented with many different execution engines, as summarized in Table 2. For our first study we used versions of Firefox and Chrome that were available in 2014, as well as the most recent versions (at the time of the experiment). For the remaining three studies we used the most

[4]The machine names mbp2018, ubuntu-deer and windows-bison, are used in our subsequent results, graphs, and tables.

recent browser-based JavaScript and WebAssembly engines. Note that 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. Note also that for all of the desktops, laptops and android devices, Chrome 63.0.3239.84, and Firefox 57.0.2 use their respective JavaScript engines while for the ipad-pro and iphoneX devices, Firefox 57.0.2 and Chrome 63.0.3239.84 use the JavaScriptCore Apple engine based on WebKit [7].

5 Old versus New JavaScript Engines

In our first study, we compare the numerical performance of the Firefox and Chrome JavaScript engine versions from 2014, against the modern version of the same engines. In the study of numerical performance in 2014 [26], the reported sequential JavaScript performance was within a factor of 1.5 to 2 of the native C version. 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.

5.1 Experimental Set Up

We did not have access to the same machines used in the 2014 study so we repeated the same experiments on different machines with similar configurations. Microsoft and Apple are not making available the specific releases of Internet Explorer 11 and Safari that were used in the previous study. We therefore focus on Chrome and Firefox only.

We provide the speedups of JavaScript against the native C version of the Ostrich benchmarks for two architectures, mbp2018 and ubuntu-deer in Fig. 2. In each case we experimented with four versions of the browsers, namely, Chromium 38 [5] and Firefox 39 from 2014; and Chrome 63 and Firefox 57 for the modern versions. 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.

5.2 Results

The first observation is that the best performing browser was Firefox 57; the JavaScript performance of the browser significantly improved over the Firefox 39 version for both the MacBook Pro laptop and Ubuntu Workstation. Secondly, and perhaps surprisingly, the numerical performance of the current Chrome browser decreased compared to the performance of the older version of the browser from 2014. We hypothesize two potential reasons for this. First, V8's introduction of a new compiler infrastructure, including a brand new interpreter, named Ignition, and a new optimizing compiler named Turbo Fan [40]. Second, the change of optimizing objectives for JavaScript engines. Since 2015, browser vendors have changed their optimizing objectives from a set of benchmarks to real-websites, resulting in the retirement of the Octane Benchmark Suite [41] from the Google team, and the development of a new methodology for measuring performance. This methodology bases optimizations of the engines on real-world examples, using real websites such as the Facebook or Wikipedia [31].[6]

To explore this, rather than taking only two end-points in time, we decided to study the evolution of numeric performance including several intermediate time points. For this we ran the benchmarks on different major versions of the engines, focusing on the introduction of the new engine for V8, and taking evenly spread versions for Firefox as shown in Fig. 3. For each browser version on the plot we show the final geometric mean across the whole benchmark suite using C as a baseline and showing speedup against C. On the left we have the Chromium versions starting from Chromium38, where Chromium57 is the browser version right before the switch of engines, and Chromium59 is the first Chromium version using the new compiler architecture. We also verified that the versions not shown in the plots follow either monotonically increasing or decreasing behavior from the neighbour versions shown.

The first observation is that even before the change of browsers, the performance of Chrome had already decreased, we conjecture this might be due to the change of methodology and optimizing objectives for the V8 engine, which started around Chromium 49. Secondly, looking at Chromium59, we can see from the plot that the change of compiler infrastructure came at a cost in performance for numeric benchmarks. However, since that time they have slowly improved their engine so now numerical performance is close to the point where it was before the change. In terms of Firefox, the JavaScript engine seems to have increased its performance of the numerical benchmarks steadily since the measurements in 2014.

5.3 Key Insights

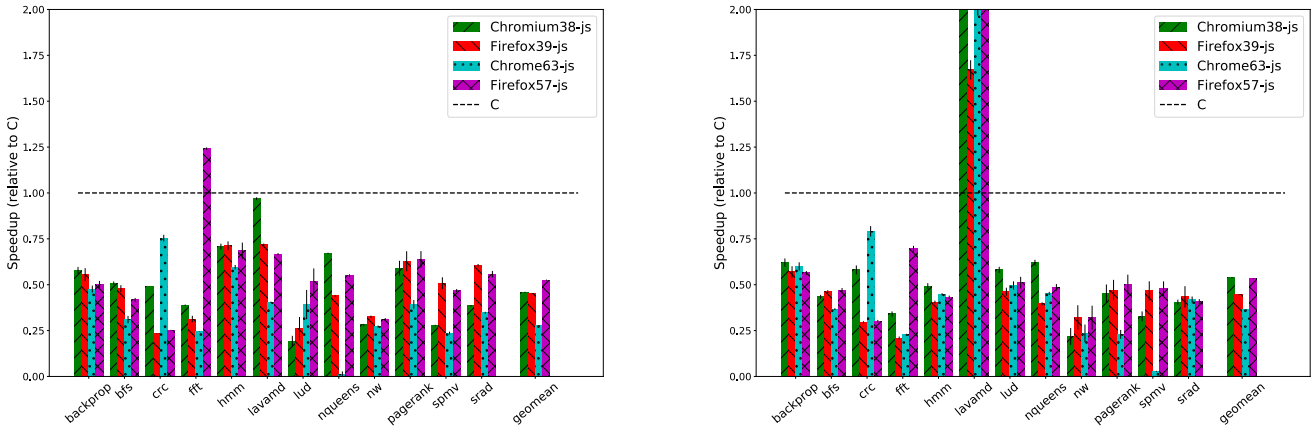
The overall performance of JavaScript against native C versions remained within a factor of 2. The current Firefox browser has presented an overall improvement, compared to the older Firefox version. The current Chrome browser, however, has presented a decrease in overall performance compared to the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

6 WebAssembly versus C and JavaScript

Although 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 unusual semantics of JavaScript. However, browsers can now also execute WebAssembly that has been built as an abstraction on top of all the major hardware architectures, with a fairly direct mapping to the architectures.

[5] Due to the unavailability of the Chrome38 browser, the comparison was done against the developer version of the browser.

[6] We believe that numeric performance is another important metric, especially with the increase in web-based numeric libraries and applications.



(a) mbp2018 **(b)** ubuntu-deer

Figure 2. JavaScript performance against native C using the old and new versions for Chrome and Firefox. The figures use the same y-axis scale to enable easy comparison. The value of the bars for the lavamd benchmark on the right graph are out of range, the values are 4.6(0.4), 2.0(0.5), and 2.8(0.3) for Chromium38-js, Chrome63-js and Firefox57-js respectively.

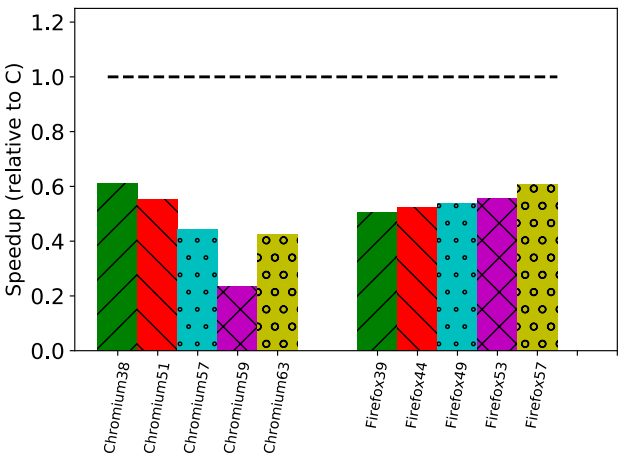


Figure 3. Browser progression since 2014 in terms of the Ostrich Benchmark suite using the ubuntu-deer machine, the values of the bars correspond to the geometric means measured across the whole benchmark suite for each browser.

This provides new opportunities for performance optimizations, and the potential of matching the performance of native C. Our next study examines the performance of WebAssembly across a wide range of devices. We first compare WebAssembly against native C, and then against JavaScript. To generate the WebAssembly code we used the C version of the benchmarks and compiled the C code using the Emscripten [46] toolchain, based on the c-lang/llvm pipeline [27]. To generate the native executable for the C benchmarks, we used the GCC toolchain [1].

6.1 WebAssembly versus C

Scientists and engineers interested in running their numerical computations using C typically use desktops for executing the resulting programs. We investigate how WebAssembly fares against this historical choice. To do this, we measured the relative performance between WebAssembly and C. For this particular experiment, we use three devices supporting three operating systems, mbp2018, ubuntu-deer and windows-bison.

Fig. 4 depicts the performance of WebAssembly using the C version of the benchmarks as a baseline for Chrome 63, Firefox 57, and Safari 11 (for the MacBook Pro laptop only). For the best performing browser, Firefox 57, 5 out of 12 benchmarks (backprop, fft, hmm, lud, srad) are better or within the performance of native C, in particular `fft`, where the WebAssembly code has overwhelming speedup over C for the three browsers. In general, however, considering the geometric means (given as the rightmost bars), the overall performance of WebAssembly is lower than that of native C. The values of the geometric mean bars are 0.70, 0.84, 0.58 for Chrome 63, Firefox 57, and Safari 11 respectively. Corresponding to slowdowns of x1.41, x1.20, x1.76 over native C.

Fig. 5 provides the geometric means for experiments on our laptop and two desktops. Again, 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 0.70x the speed of native C (equivalent to a slowdown of 1.42), except for Safari 11 on the mbp2018. Thus, the performance is getting close to C.

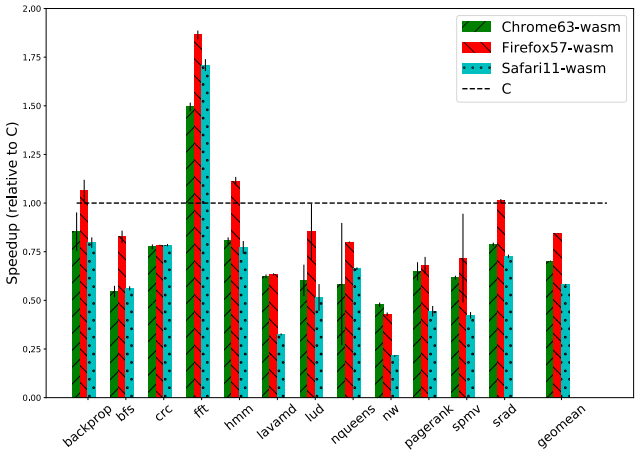


Figure 4. WebAssembly performance relative to C on the MacBookPro 2018.

- Firefox57 wasm achieved the best performance over the three platforms, however, no platform was able to outperform the native C code.

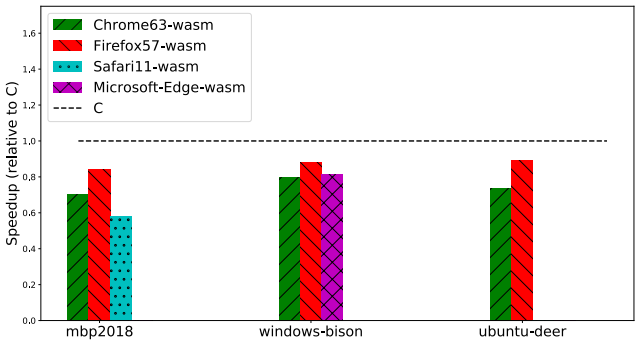


Figure 5. Geometric mean speedups over all benchmarks for WebAssembly relative to C on the different platforms.

6.2 WebAssembly versus JavaScript

It was shown in 2014 [26] that JavaScript was becoming fast enough for numerical computations and that the asm.js subset had a measurable advantage in performance. As an evolution of asm.js, WebAssembly offers the promise of better performance with the same portability as JavaScript. We therefore quantify the realization of that promise on major browsers that currently support it.

To do so, we report on the relative performance of WebAssembly and JavaScript within the same browser. 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: [7]

- desktops with different OS: mbp2018, 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.

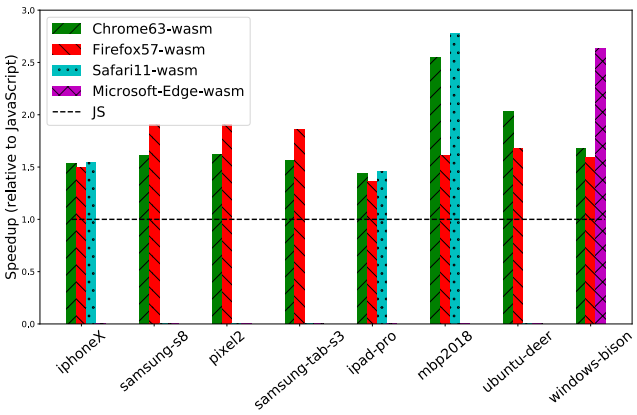


Figure 6. WebAssembly performance relative to JavaScript on the different platforms.

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

- 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 over our set of benchmarks.

6.3 Key Insights

All browsers demonstrate significant performance improvements for WebAssembly, in the range of 1.5-2x speedups over the same browser's JavaScript engine. Furthermore, WebAssembly achieves an overall performance close to 1 against native C.

7 Server-side Node.js Performance

Since its debut in 2010, Node.js, has grown in popularity as a server side, event-driven language. This event-driven nature,

[7] Note that all the browsers available to the ipad-pro and the iphoneX devices use the JavaScriptCore Apple engine based on the WebKit restriction imposed.

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 [21]. As a server-side language, is expected that the language will be able to handle a large number of requests from clients, and moreover, different workloads from the ones experienced in a web browser. Thus, it is interesting to see how fast Node.js is, versus C.

A second interesting question is if the server side Node.js engines have the same performance as their equivalent client-side browser engines. The Google V8 team announced first-class support for Node.js in 2016. As V8 is optimized for real websites, it remains as an interesting question, whether the V8 team exploits specific optimizations that would benefit the Node.js loads.

To answer these two questions, we evaluate the performance of server-side Node.js standard V8 version, first against native C, and lastly against client-side browsers of the same platform.

7.1 Node.js versus C

The Node.js versions, platforms, and GCC versions used for each platform are stated in Table 2. Fig. 7 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 Node.js(WebAssembly) version is more performant than the Node.js(JavaScript) version. Furthermore, Node.js(WebAssembly) is approaching the performance of native C for all the platforms except for Ubuntu-deer. Exploration of this issue is given at the end of the section. The best Node.js(WebAssembly) version achieved a speedup of 0.9 performance against native C. On the other hand, for Node.js(JavaScript) the best overall geometric mean obtained was of 0.5 versus native C.

We performed further profiling of the ubuntu-deer issue, using machines with different operating systems but the same hardware specifications. We found that this issue was specific to the Ubuntu operating system on the Sandy Bridge E processor. The issue arises in the Node.js implementation of Float32 arrays after the introduction of the new V8 engine infrastructure in v8.0.0 of Node.js, and is still present in >v10.0.0. [8]

7.2 Node.js versus Browsers

The second part of this performance study studies the numerical performance of server-side Node.js against the client-side

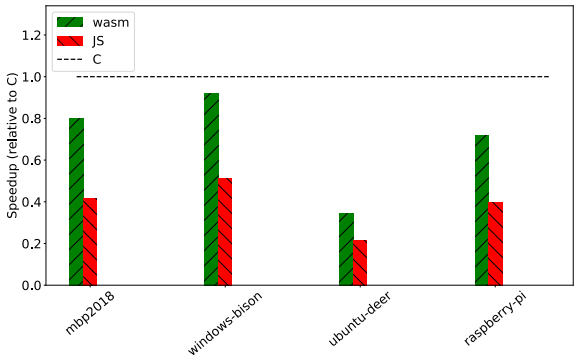


Figure 7. Geometric mean speedup of Node.js in different workstations relative to C.

browsers for each platform. Specifically, we take the performance of Node.js V8 version, and take a ratio against the performance of the browsers for the same platform. Table 3 contains the results of Node.js speedups relative to the corresponding device browser's. e.g. The ratio of Safari11-JS performance, is taken against the Node.js standard V8 JavaScript performance. Note that this comparison is using the Node.js v8 version, against all the browsers supported by that particular platform.

Table 3. WebAssembly and JavaScript Node.js speedup performance relative to the browsers for each device.

Device	*mbp2018*	*windows-bison*	*ubuntu-deer*	*raspberry-pi*
Chrome63-JS	1.5	1	0.6	-
Chrome63-wasm	1.1	1.1	0.5	-
Firefox57-JS	0.8	0.9	0.4	-
Firefox57-wasm	0.9	1	0.4	-
Safari11-JS	2	-	-	-
Safari11-wasm	1.4	-	-	-
Microsoft-Edge-JS	-	1.5	-	-
Microsoft-Edge-wasm	-	1.1	-	-
Chromium56-JS	-	-	-	1.1

We first observe that Node.js performance was better than the Safari 11 browser, in terms of both the wasm and JS performance, while the Microsoft Edge browser had a similar performance for wasm and a much lower performance for JavaScript. Given the best performing browsers, Firefox 57, and Chrome 63, we can see that the results of the speedups are fairly close for both WebAssembly and JavaScript, with the exception of Chrome63-JS where we get a speed-up of 1.5, and the ubuntu-deer machine, where the performance of

[8] For more information visit: https://github.com/nodejs/node/issues/22467

the browsers is better by more than a factor of 2, due to the performance bug found with ubuntu-deer.

7.3 Key Insights

Node.js was overall slower than native C code for both JavaScript and WebAssembly, although the best WebAssembly result gave a reasonable performance of 0.9 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 workstation. This demonstrates two important points, first, that extensive performance studies of this magnitude are required to make general conclusions about engine performance and secondly, numeric benchmarks can help identify performance issues in the new versions of engines.

8 JavaScript/WebAssembly/Device Showdown

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 speedup factors over 1. A number greater than 1 is the speedup relative to Raspberry Pi native C. Table 4 displays the results of this study. 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.

8.1 Key Insights

The best overall browser performance was by the Firefox 57 browser. WebAssembly is significantly faster than JavaScript, over all devices. Finally, the order of device performance has the MacBook Pro, followed by the two workstations, surprisingly followed by the iPhone X, the rest of the order is: iPad Pro, Samsung S8, Pixel 2, Samsung Tab S3 and lastly, Raspberry Pi Model B.

9 Conclusions and Future Work

In this paper, we have studied numerical computing performance on the web, for a wide range of devices and for both JavaScript (and Node.js) and WebAssembly. Due to a large number of experimental dimensions, benchmarking in this context is very challenging, particularly if one wants to have a centralized and automated benchmarking system that can benchmark across a wide range of devices including tablets, mobile phones, and IoT devices.

Our first contribution was to design the Wu-Wei benchmarking toolkit. Wu-Wei allowed us to explore performance consistently across benchmarks, implementations, compilers, environments, and platforms. It also allowed to have reproducible experiments and a consistent way to benchmark for C, JavaScript, and WebAssembly across a variety of environments and devices. For the performance studies in this paper, we developed a Wu-Wei enabled version of the Ostrich benchmark set to which we added a Wu-Wei compatible compiler to transform C code to WebAssembly. Moreover, to handle the mobile devices, we extended Wu-Wei with a hybrid mobile app and a Firebase listener, which allowed us to perform centralized benchmarking across all of our devices. We expect that this benchmarking framework will be of general use to researchers/developers interested in performing additional numerical experiments, as well as to researchers who would like to use the approach for other benchmarking. For example, programming language researchers may develop new compilers targeting WebAssembly, and this approach could help them evaluate performance, including performance impacts of different code generation and optimization strategies.

Using our benchmarking approach we performed four performance studies on the Wu-Wei version of the Ostrich benchmark suite. We first showed the historical evolution of Chrome and Firefox, starting with the versions available in the previous 2014 study. This showed regular performance improvements for Firefox, but a dip in numerical performance for Chrome, followed by some improvement, corresponding to major changes in the architecture of the Chrome JavaScript engines. In general, the performance of numerical JavaScript remains within a factor of two of native C code. We believe that it would be interesting for browser developers to use our approach to evaluate numerical performance at each release, as one measure of browser performance.

Our second study showed that WebAssembly outperforms JavaScript, and is approaching the performance of native C, for our numerical benchmark set. This is very encouraging, and these results should be of interest to scientific programmers and developers of numeric/computation-intensive libraries, since it shows that WebAssembly is a very attractive target. The performance of WebAssembly should only improve as it matures, and as it includes further features such as support for data and task parallelism [43].

Our third study showed that the performance of server-side Node.js is similar to the equivalent browser-based versions, for both JavaScript and WebAssembly. There was a notable exception, where the ubuntu machine had significantly worse Node.js performance. By analyzing previous versions we identified when this performance issue had first been introduced. This shows another example of where using numeric benchmarking at each release could identify performance issues, as they arise.

Our final study showed the overall numeric performance of all devices and WebAssembly/JavaScript. The results showed that: (1) the performance of the mobile phones was very impressive, (2) WebAssembly performed significantly better than JavaScript across all devices, and (3) that Firefox was slightly faster than Chrome for both JavaScript and WebAssembly.

Table 4. Device performance across environments using the native C raspberry pi implementation as baseline.

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	**mean-wasm**	**mean-js**
iphoneX	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	-	-	-	-	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	-	-	-	-	2.2	1.3
mbp2018	10.2	4	-	12.2	7.6	8.4	3	-	-	10.3	4.9
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	6.2	3.5	-	6.9	4.3	-	-	-	-	-	-

We hope that others will adopt our benchmarking approach, and that numeric benchmarking will become one of the standard performance metrics to track for web-based languages. We are planning to further extend our experiments in two ways. First, we would like to experiment with other types of parameters such as: (a) comparing cold start versus warm start, and (b) comparing speedups for different input sizes, including the small and large inputs available in the Ostrich benchmark set. All of the experiments in this paper are available, and further experiments should be possible using our experimental framework. Second, we plan to further extend our benchmarks to include more machine learning benchmarks, and we welcome other contributions which can help to form the next versions of the Wu-Wei Benchmarking Toolkit and the Ostrich benchmark set.

References

[1] 1988. Gnu compiler collection (gcc) internals. Retrieved September 4th, 2018 from http://gcc.gnu.org/onlinedocs/gccint/

[2] 2010. Kraken JavaScript Benchmark (Version 1.1). Retrieved April 14, 2018 from http://krakenbenchmark.mozilla.org/

[3] 2012. Octane Benchmark Suite. Retrieved April 14, 2018 from https://developers.google.com/octane/

[4] 2014. Speedometer. Retrieved April 14, 2018 from http://browserbench.org/Speedometer/

[5] 2015. Ndarray - Multidimensional arrays for JavaScript. Retrieved June 29, 2018 from https://github.com/scijs/ndarray

[6] 2017. Standard ECMA-262 ECMAScript Language Specification 8th Edition (June 2017). Retrieved April 14, 2018 from http://www.ecma-international.org/publications/standards/Ecma-262.htm

[7] 2018. Framework - JavaScriptCore. Retrieved April 14, 2018 from https://developer.apple.com/documentation/javascriptcore

[8] 2018. TensorFlow.js. Retrieved April 14, 2018 from https://js.tensorflow.org/

[9] Apache Cordova. 2011. A hybrid mobile app. Retrieved April 14, 2018 from https://cordova.apache.org/

[10] asm.js. 2014. The specification of asm.js. Retrieved April 14, 2018 from http://asmjs.org/spec/latest/

[11] Lars Bak. 2018. WebAssembly Core Specification, W3C First Public Working Draft, 15 February 2018. Retrieved June 30, 2018 from https://www.w3.org/TR/wasm-core-1/l

[12] Benjamin Bouvier. 2013. Efficient float32 Arithmetic in JavaScript. Retrieved January 19, 2018 from https://blog.mozilla.org/javascript/2013/11/07/efficient-float32-arithmetic-in-javascript/

[13] Antoine Boutet, Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Rhicheek Patra. 2014. HyRec: Leveraging Browsers for Scalable Recommenders. In *International Middleware Conference*. ACM, 85–96.

[14] Phillip Colella. 2004. Defining software requirements for scientific computing. Retrieved April 14, 2018 from https://www.krellinst.org/doecsgf/conf/2013/pres/pcolella.pdf

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

[16] Tomasz Fabisiak and Arkadiusz Danilecki. 2017. Browser-based Harnessing of Voluntary Computational Power. *Foundations of Computing and Decision Sciences* 42, 1 (2017), 3–42.

[17] Firebase. 2011. A Real-time Database on Cloud. Retrieved April 14, 2018 from https://firebase.google.com/

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

[19] 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. 2009. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Conference on Programming Language Design and Implementation*. ACM, 465–478.

[20] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Conference on Object-oriented Programming Systems and Applications*. 57–76.

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

[22] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Conference on Programming Language Design and Implementation*. 185–200.

[23] Apple Inc. 2017. About Face ID Advanced Technology. Retrieved April 14, 2018 from https://support.apple.com/en-ca/HT208108

[24] Peter Jensen, Ivan Jibaja, Ningxin Hu, Dan Gohman, and John McCutchan. 2015. SIMD in Javascript via C++ and Emscripten. In *Workshop on Programming Models for SIMD/Vector Processing*.

[25] Alexandr A. Kalinin, Selvam Palanimalai, and Ivo D. Dinov. 2017. SOCRAT Platform Design: A Web Architecture for Interactive Visual Analytics Applications. In *Workshop on Human-In-the-Loop Data Analytics*. Article 8, 6 pages.

[26] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, Erick Lavoie, and Laurie J. Hendren. 2014. Using JavaScript and WebCL for Numerical Computations: A Comparative Study of Native and Web Technologies. In *Symposium on Dynamic Languages*. 91–102.

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE, 75.

[28] Erick Lavoie. 2016. Wu-Wei Handbook. Retrieved August 24, 2018 from https://github.com/Sable/wu-wei-handbook

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

[30] Edward Meeds, Remco Hendriks, Said Al Faraby, Magiel Bruntink, and Max Welling. 2015. MLitB: Machine Learning in the Browser. *PeerJ Computer Science* 1 (2015), e11.

[31] Franziska Hinkelmann Michael Hablich. 2016. How V8 Measures Real-world Performance. Retrieved April 14, 2018 from https://v8project.blogspot.ca/2016/

[32] Mozilla JavaScript. 2018. JavaScript Typed Arrays. Retrieved April 14, 2018 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

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

[34] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. *USENIX Conference on Web Application Development* 10 (2010), 3–3.

[35] Gregor Richards. 2013. JSBench Benchmark Suite. Retrieved April 14, 2018 from https://plg.uwaterloo.ca/~dynjs/jsbench/

[36] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Conference on Programming Language Design and Implementation*. ACM, 1–12.

[37] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in Javascript: An Empirical Study. In *International Conference on Software Engineering*. ACM, 61–72.

[38] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *Internet of Things Journal* 3, 5 (2016), 637–646.

[39] Katie Shilton. 2009. Four Billion Little Brothers?: Privacy, Mobile Phones, and Ubiquitous Data Collection. *Commun. ACM* 52, 11 (2009), 48–53.

[40] V8 Team. 2017. Launching Ignition and TurboFan. Retrieved April 14, 2018 from https://v8project.blogspot.ca/2017/05/launching-ignition-and-turbofan.html

[41] V8 Team. 2017. Retiring Octane. Retrieved April 14, 2018 from https://v8project.blogspot.ca/2017/04/retiring-octane.html

[42] W3C. 2018. Google Chrome's Need for Speed. Retrieved April 14, 2018 from http://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html

[43] WebAssembly. 2015. Features to Add after the MVP. Retrieved April 14, 2018 from http://webassembly.org/docs/future-features/

[44] WebAssembly. 2016. Non-Web Embeddings. Retrieved April 14, 2018 from http://webassembly.org/docs/non-web/

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

[46] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *ACM International Conference Companion on Object-oriented Programming Systems Languages and Applications Companion*. 301–312.