ISSN:0975-9646

Universal High-Performance Applications with WebAssembly

Jebreel Alamari
Computer Science Department
University of Colorado at Colorado Springs
Jalamari@uccs.edu

C.Edward Chow
Computer Science Department
University of Colorado at Colorado Springs
cchow@uccs.edu

Abstract— Building and maintaining applications is an expensive operation. According to Kinvey report, building an enterprise level mobile application costs about $270k. Supporting multiple platforms and operating systems is the driving cost of the development process. In this paper we investigate the possibility of using web browsers as computation containers that allow code to run independently from the underlying operating system. Progressive Web Applications became popular in the recent years, however their performance still worries developers. Therefore, in this paper we concentrate on the performance aspect of web applications.

Keywords— (web applications, universal applications, web browsers, web assembly, JavaScript, WebAssembly)

I. INTRODUCTION

Looking at technology from the outside might not show the whole picture about it. Unfortunately, it is the case with web browsers. People tend to view web browsers as applications that allow them to fetch and view documents stored on a remote server somewhere on the Internet. Some technical people are aware of some browsers' features such as making a secure connection with the server and storing credentials for their online services. Even if they are using the browser as an interface to interact with more advanced online services, such as Google documents [3] or Office 365 [4], they may still underestimate its role in making those services possible.

In fact, web browsers are much more than that. They perform numerous jobs that seem to be taken for granted. Such jobs are taking a foreign code from unknown origin, not all the times, and execute it safely on our machines. This code is written by unknown developer, and we as a consumer, we may have no idea about what that code is doing.

If we compare that to native codes that we run on our machines; it is a completely different process. For example, we download executables, mostly from known sources, and install them locally. If we need to use the code, we search for it on our system and run it manually. On the other hand, the browser downloads the code, runs it with a little intervention form our side.

Ensuring the safety of our system when running foreign code is a challenging task. However, the browser handles that kind of threat by using the concept of isolation. The foreign code is sandboxed [5]. Meaning, it runs in a designated memory space and all its activities are contained in that space. So, the code is not allowed to access other processes' address space, even the address space of the browsers engine itself.

Browsers do not stop at that point. They try to run that code efficiently and at a high speed. Since JavaScript is the main programming language for computation in the browsers, after the failure of Java Applets [6], it became feasible to optimize browser engines to run JavaScript code faster.

One of inaccurate ideas about JavaScript is that it is slow, and it is only good to do form processing with some fun animation in the web page. Sure, JavaScript can do all that, but nowadays it can do much more.

Even though it is still considered a loosely typed language with an interpreter, its performance has dramatically improved [7]. Thanks to the optimizations done to JavaScript engines, now we can run CPU intensive tasks easily in the browser.

II. RESEARCH MOTIVATION

The browser is almost available on every machine. All major Operating systems, ship with a web browser preinstalled. Even, low end devices such as IoT and smart phones, come with their own browsers. That makes the web browser the most ubiquitous application in the history of computing.

The ubiquity of web browsers makes them attractive to developers who want their code to reach out to users no matter what platform they use. The browser literally became a *container* to run that code.

Even though, web browsers seem to have a lot of potential to become the standard platform to run code, they are still far from achieving that. In this paper we evaluate the performance of existing web browsers to see how suitable they are to run applications that require native implementation, due to performance requirements.

III. CROSS PLATFORM FRAMEWORKS AND RELATED WORK

In the recent years, we noticed a shift on how software is being built in terms of making cross platform applications. Companies such as Microsoft came up with universal applications that run on Windows 10 and Windows Phone [8]. The goal was to shorten development time and const. Other frameworks emerged to build applications that run on completely different Operating systems. To name a few, Apache Cardova [9], Fuse Open [10], React Native [11], NativeScript [12], and recently Futtler[13]. They work differently, but they all promise the ability to build applications and compile them to work on virtually all available platforms.

A. *Cross platform frameworks disadvantages*

However, despite the high performance of some of code generated by these frameworks, such as Flutter they still suffer from major problems that can be summarized as follows.

- The learning curve of these frameworks is steep in some situations. Meaning, that for a company that decides to go with one of them, their developers need training. Training costs money and time. In addition, sticking with one framework might not last for long. Switching from a framework to another requires more training.
- Like adapting a new programming language, switching to another framework is in some situations like learning a new programming language. Sure, most of them rely on JavaScript for processing and communicating with backend services; however, they require more than just a vanilla JavaScript code. The layout of the code, packages, and modules make the framework more intimidating for new commers.
- It is always better to rely on well standardized technology instead of nonstandard one. When investing in one of the frameworks, businesses stay under the mercy of the framework maintainers. Yes, they are all open source, but you may not have the expertise required to implement new features that you might want. It is even worse when there is a major update on the target platforms. That update could break your code. In this situation, you need to wait for framework developers to update their code base, so you can rebuild and redistribute your code.
- Most of these frameworks support iOS and Android platforms. That brings us back to the first square in term of supporting all devices. In case of new platforms such as Ubuntu Phone [14] of Kai OS [15] or Tizen [16], we need to wait for developers to include support for those Operating Systems. One of the obstacles that face companies who want to build their own OSs is the lack of applications. Relying of such frameworks is not a full solution for this problem.

B. *Web Applications advantages*

Web applications introduce their own challenges that we are going to discuss later in the paper, however they solve almost all problems with native executables.

- *Portability*
  As discussed in the previous sections, web apps only need a browser to run, and the browser is almost everywhere. Regardless of the operating system, underline architecture the code with still run.
- *Ease of development*
  Developing for the web front end, we use HTML5, JavaScript, and CSS. That is all a developer needs to master. Straight forward syntax that does not need to change from time to time, because these three technologies are well standardized in virtually all browsers.
- *Ease of distributions*
  With the world wide web, the only thing we need to distribute a web app is to share a link. No App Store license or fees required. No app store ban on your application for whatever reason the platform stakeholders decide to take your application down of their platform.
  Updating your application, is even more convenient. Update it once in one place, and it takes effect on every user's device.

C. *Web Applications disadvantages*

Switching to building web applications is not a magic bullet that solves all the problems we mentioned. Here are some existing obstacles that makes companies and developers go with building a native code or even use one of the frameworks we mentioned before.

- *Performance*
  As we strive to make a single code that is platform, architecture independent, the performance is still the main problem here. Like Java that came to solve this problem three decades ago with having a Virtual Machine to facilitate the executing on different computers, it has not achieved the same performance as code written for a specific machine [17]. That is due to the overhead of the virtual machine.
  JavaScript, the main language on the browser, runs like Java. There is a Virtual Machine or an interpreter, that allows JavaScript to run on different architectures. Unlike Java that is well typed, JavaScript is untyped programming language. That imposes a new problem with JavaScript execution performance because there is a type check on the variable whenever we need to manipulate its value.
  JavaScript engines adapted the technique used in Java VM, by introducing JIT (Just in Time) compilers. JIT compilers bring a substantial improvement to the performance of JavaScript, however, unlike Jave JIT, they need to do deoptimization if the type of an object has changed during the run time. These deoptimizations are expensive especially if types keep changing during run time. It is common in JavaScript Object to change shapes; simply by adding or removing properties during runtime. JavaScript implementers need to comply with its standards that allow this kind of behavior [18].
- *Connectivity*
  In some sense, web applications, are websites. Accessing a website requires having an internet connection. On the other hand, unless it requires connecting to a backend service, a native app is stored locally on user’s device and available when needed.
  This issue persisted with web applications for a long time. However, recently, this is its way to go.
  Nowadays, web browsers ship with tremendous capabilities such as storing data in a persistent storage in user’s hard drive. These storages are efficient to store data and retrieve them. They are implemented natively in the browser and there is a JavaScript APIs to access and use them.

The browser has different types of persistent storages. For example, LocalStorage, a quick way store data as a Key Value pairs, and Indexed DB is a persistent high-performance transactional database [19] that is W3C recommendation [20].

We can also store code files locally with appCache API [21]. The user only needs to connect the application once, then the app will be installed in the browser. There will be no need to connect to the internet unless local data has cleared which is a problem that needs to be considered.

Keeping the data save in the local storage is handled using WebCryptography API that is also natively implemented in the browser [22].

- *Native feeling*

  This is still a challenging part however with using clever CSS styling we can make it look close to native. Browsers use their proprietary rendering engines. The application may look different than native counterpart because of the way HTML5 elements are rendered on the page. However, with clever CSS rules manipulations, and third-party libraries, it is possible to mimic the native UI of major platforms.

IV. METHODOLOGY

There is a lot of research done to test the performance of code written in JavaScript [7][23]. Our goal here is different. It is to test whether web browsers are ready to become the containers that take care of computation on all platforms.

We were dedicated to cover almost all aspects of computation that a native code does and compare it to the web version of the same code.

First, we started by running an image manipulation program using OpenCV [24] library. We tested the same algorithm on the same data on different operating systems and browsers on those operating systems.

We compiled the code targeting all operating systems in our experiment. The rationale behind this experiment is to see the performance difference between the code when is run and managed by the browser, and when the same code it is run and managed by the operating system.

We used Emscripten compiler to compile [25] C/C++ codes and export them to the browser. We utilized Ostrich benchmark suite [26] for this purpose. Ostrich was a good candidate for our experiment because it supports C/C++ and JavaScript. However, its support for WebAssembly is limited. We had to modify *the build system* for the benchmark to make it suitable to be compiled by Emscripten compiler. Our updated version of Ostrich benchmark kernels are fully compiled to standalone WebAssembly modules.

Another challenging task we faced, in order to make a fair performance comparison, was compiling code to native iOS and Android codes.

A. *Experiment Environment*

We tried to cover as many as possible devices and operating systems. Table1 of devices that we use in our experiments.

Operating System	*Device specs*
iMac, running macOS Catalina	3.2 GHz Quad-Core intel Core i5 with 16 Gig of RAM.
Dell inspiron, running Windows 10	2.4 GHz Quad-Core Intel Core i7 with 12 Gig of RAM.
Dell OptiPlex 990, running Ubuntu 18.4	3.7 GHz Intel Core i7 VPro with 8 Gig of RAM
Galaxy S10 Plus, running Android 10	2.73 Octa-Core Samsung Exynos 9820 with 8 Gig of RAM.
iPhone X S MAX, running iOS 13	2.6 Hex-Core A12 Bionic with 4 Gig of RAM.

TABLE 1: EXPERIMENT ENVIRONMENT

Machine	Specs
Local Server	2.20 12-Core intel Xeon with 64 Gig of RAM
AWS EC2 instance	8 vCPU and 16 Gig of RAM.

TABLE 2: BACK-END

1) *The back end*

Just like a typical web application, having a high performance back-end with high bandwidth is crucial. Our experiment is mainly concerned about how fast the code can run in the browser compared to the equivalent native code; however, it is important to load the code logic as fast as possible to the web page.

2) *Web Browsers*

For the sake of platform independence, we only test browsers that are not proprietary. In other words, browsers that are available for all devices being tested. So, we excluded Safari, Samsung browser. We also excluded Microsoft Edge and Opera because they are both running V8 JavaScript engine like Google Chrome. Therefore, the browsers that we tested were Mozilla Firefox, and Google Chrome. Mozilla SpiderMonkey JavaScript engine seems to be the right JavaScript engine to compare with V8 because they are both open source and available on different platforms.

V. RESULTS AND DISCUSSIONS

In this section we show the results of our experiments. Figures 1,2,3 shows the performance of Native code, Chrome, and Firefox, respectively. On the *x-axis* we have *OpenCV* algorithms that we implemented, and on the *y-axis*, we see the time taken by the process to finish manipulating the image in Milliseconds. Native code is the code that we complied and natively on every platform.

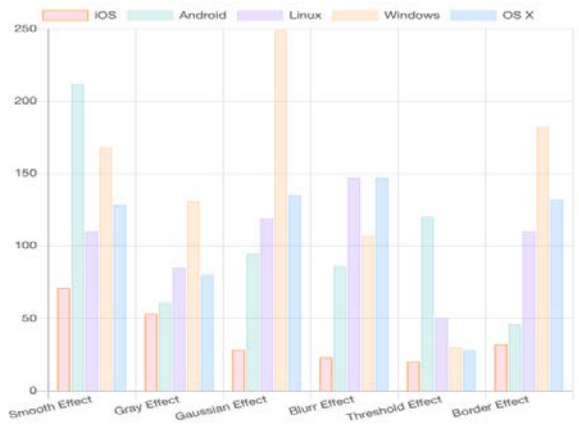

FIGURE 1: Performance Of Native Code. Time in ms

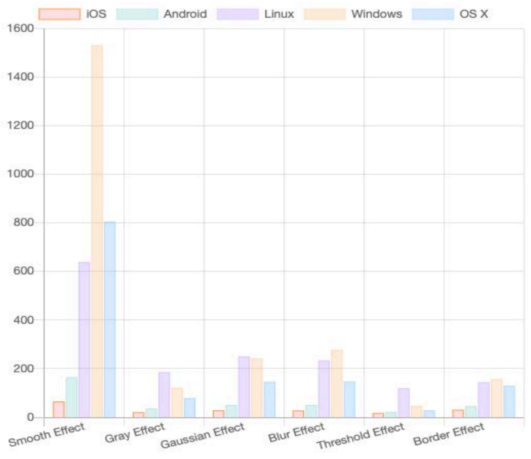

Figure 2: The performance in Google Chrome. Time in ms.

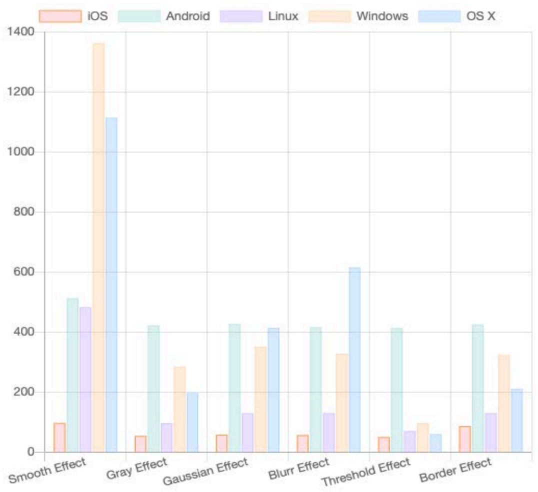

Figure 3: The performance of Firefox. Time in ms.

Regardless of what machine perform the best, our goal is to compare every machine with itself. In other words, when the code is running on the same machine, but one is in the browser and the second on the machine itself.

Overall results don't show a significant difference in the performance. It means a developer decides to develop such application using web technologies, the app is not going to suffer much of a performance issue.

With the introduction WebAssembly (high performance compact bytecode), We decided to test its performance compared to native C/C++ using OpenCV capability. In figure 4 we see how much time(ms) it takes WebAsseembly (WASM) compared to C++ code to perform face/eyes detection. It is worth mentioning that we used Haar.js face detector.

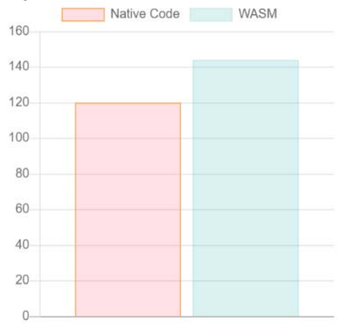

Figure 4: OpenCV face/eye detector performance. Time in ms.

Using Haar.js could be the reason why WASM code was slower than C/C++. We may end up with better results if our face detector was compiled to WebAssembly.

Figure 5 depicts the average time, in seconds, taken by WASM compared to the average time taken by native code in all devices in the experiment. Both codes do the same work on the exact input and generate the exact output.

The source code here was taken from Ostrich benchmark that we compiled to WASM using Emscripten compiler.

It is worth mentioning that these kernels were also compiled to native Android/iOS codes. Even though it was a time-consuming task, but we thought it was the right way to get a clear picture about WASM performance.

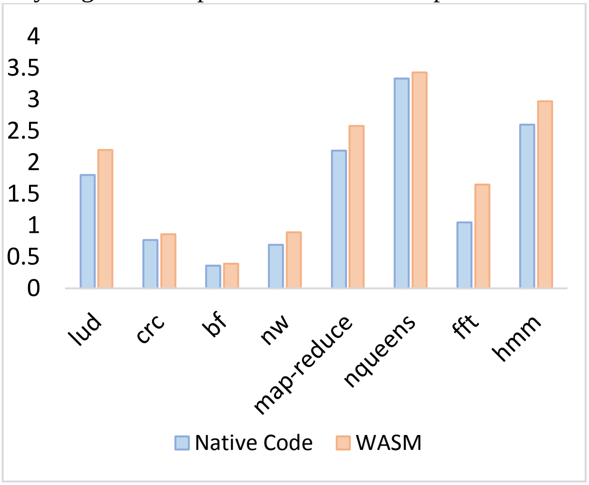

Figure 5: Performance of WASM vs Native. Time in Seconds

VI. CONCLUSION

In this paper we demonstrated the ability for web browsers to handle CPU-intensive tasks. Since it is the most ubiquitous application, the browser as a computation container will improve the portability of apps. Encouraging, developers and searchers to target web browsers, would free software industry from single platform domination and app stores restrictions. The browser, as VM to run web applications, will continue to fulfill Java goal of building code once and run it everywhere. Developers will be relieved from maintaining different codebases in order to support multiple operating systems. In addition, supporting computation in the browser will open the door for new operating systems, because all applications will be available for them as long as there is a web browser installed on the system.

REFERENCES

[1] “Figuring the costs of mobile app development | Formotus.” https://www.formotus.com/blog/figuring-the-costs-of-custom-mobile-business-app-development (accessed Nov. 19, 2019).

[2] “web.dev,” web.dev. https://web.dev/progressive-web-apps/ (accessed Sep. 21, 2019).

[3] “Google Docs: Free Online Documents for Personal Use.” https://www.google.com/docs/about/ (accessed Mar. 13, 2020).

[4] “Microsoft Office Home.” https://www.office.com (accessed Aug. 10, 2020).

[5] S. Van Acker and A. Sabelfeld, “JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript,” in Foundations of Security Analysis and Design VIII: FOSAD 2014/2015/2016 Tutorial Lectures, A. Aldini, J. Lopez, and F. Martinelli, Eds. Cham: Springer International Publishing, 2016, pp. 32–86.

[6] S. Gritzalis and J. Iliadis, “Addressing security issues in programming languages for mobile code,” in Proceedings Ninth International Workshop on Database and Expert Systems Applications (Cat. No.98EX130), Aug. 1998, pp. 288–293, doi: 10.1109/DEXA.1998.707415.

[7] C. Radoi, S. Herhut, J. Sreeram, and D. Dig, “Are Web Applications Ready for Parallelism?,” p. 2.

[8] QuinnRadich, “What’s a Universal Windows Platform (UWP) app? - UWP applications.” https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide (accessed Aug. 11, 2020).

.

[9] “Apache Cordova.” https://cordova.apache.org/ (accessed Aug. 11, 2020).

[10] “Fuse Open,” Fuse Open. https://fuse-open.github.io/ (accessed Aug. 11, 2020).

[11] “React Native · A framework for building native apps using React.” https://reactnative.dev/ (accessed Aug. 11, 2020).

[12] “Native mobile apps with Angular, Vue.js, TypeScript, JavaScript - NativeScript,” NativeScript.org. https://www.nativescript.org/ (accessed Aug. 11, 2020).

[13] “Flutter - Beautiful native apps in record time.” https://flutter.dev/ (accessed Aug. 31, 2020)

[14] “Devices | Ubuntu Phone documentation.” https://phone.docs.ubuntu.com/en/devices/ (accessed Jul. 3, 2020).

[15] “Home,” KaiOS. https://www.kaiostech.com/ (accessed Jul. 21, 2020).

[16] “Tizen | An open source, standards-based software platform for multiple device categories.” https://www.tizen.org/ (accessed Aug. 21, 2020).

[17] “Java | Definition & Facts,” Encyclopedia Britannica. https://www.britannica.com/technology/Java-computer-programming-language (accessed Mar. 28, 2020).

[18] “ECMAScript® 2019 Language Specification.” https://www.ecma-international.org/ecma-262/10.0/index.html (accessed Aug. 11, 2020).

[19] A. Al-Shaikh and A. Sleit, “Evaluating IndexedDB performance on web browsers,” in 2017 8th International Conference on Information Technology (ICIT), May 2017, pp. 488–494, doi: 10.1109/ICITECH.2017.8080047.

[20] “Indexed Database API 2.0.” https://www.w3.org/TR/IndexedDB-2/ (accessed Jan. 20, 2020).

[21] “5.6 Offline Web applications — HTML5.” https://www.w3.org/TR/2011/WD-html5-20110525/offline.html (accessed Aug. 31, 2020).

[22] “Web Cryptography API.” https://www.w3.org/TR/WebCryptoAPI/ (accessed . Aug. 3, 2020).

[23] D. Herrera, H. Chen, E. Lavoie, and L. Hendren, “WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices,” p. 26.

[24] “OpenCV.” https://opencv.org/ (accessed Mar. 14, 2020).

[25] A. Zakai, “Emscripten: an LLVM-to-JavaScript compiler,” in Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH ’11, Portland, Oregon, USA, 2011, p. 301, doi: 10.1145/2048147.2048224.