EXAMINING PERFORMANCE BENEFITS OF REAL-WORLD WEBASSEMBLY APPLICATIONS: A QUANTITATIVE MULTIPLE-CASE STUDY

Lappeenranta–Lahti University of Technology LUT

Bachelor's Programme in Software Engineering

2022

Teemu Ketonen

Examiner: Erno Vanhala, D.Sc. (Tech.)

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Science

Software Engineering

Teemu Ketonen

Examining performance benefits of real-world WebAssembly applications: a quantitative multiple-case study

Bachelor's thesis

2022

39 pages, 11 figures, and 2 tables

Examiner: Erno Vanhala, D.Sc. (Tech.)

Keywords: WebAssembly, performance, benchmark, web applications, optimization

The advancement of the Web platform has pushed developers to create more interactive and sophisticated applications. Traditionally, these apps have been created in JavaScript, as it is the only high-level language that is natively supported on web browsers. Due to the language's design, which promotes ease of use over performance, JavaScript is not well-equipped to handle the growing number of compute-intensive applications. WebAssembly is a relatively new Web standard for running low-level code at near-native speed to particularly address this limitation. This study aims to bridge a gap in earlier research, which covers only theoretical benchmarks that do not necessarily present real applications, to demonstrate that WebAssembly outperforms JavaScript.

A quantitative multiple-case study was undertaken to support the hypothesis that WebAssembly performs better in performance-demanding applications found in the real world. Data was collected from prepared measurements discovered in primary sources highlighting the transition from JavaScript to WebAssembly. The results showed a significant improvement in performance, with execution times ranging from 2x to 39x faster. In addition, the complexity of an application seemed to have a factor in how much it could benefit from the shift.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT Teknis-luonnontieteellinen

Tietotekniikka

Teemu Ketonen

Tarkastelu reaalimaailman WebAssembly-sovellusten suorituskykyhyötyihin: kvantitatiivinen monitapaustutkimus

Tietotekniikan kandidaatintyö

39 sivua, 11 kuvaa ja 2 taulukkoa

Tarkastaja: TkT Erno Vanhala

Avainsanat: WebAssembly, suorituskyky, mittaaminen, verkkosovellukset, optimointi

Web-alustan teknologinen kehittyminen on antanut mahdollisuuden rakentaa yhä interaktiivisempia ja monimutkaisempia sovelluksia. Perinteisesti nämä sovellukset on tehty JavaScriptillä, koska se on ainoa verkkoselaimilla tuettu korkean tason ohjelmointikieli. JavaScript ei kuitenkaan ole tarpeeksi kykenevä vastaamaan näiden sovellusten kasvavaan suorituskykytarpeeseen, koska se keskittyy suorituskyvyn sijasta helppokäyttöisyyteen. Tämän puutteen korjaamiseksi kehitettiin WebAssembly, joka on suhteellisen uusi Web-standardi ohjelmakoodin suorittamiseen hyvällä suorituskyvyllä. Tässä työssä pyritään täyttämään aikaisemman tutkimuksen jättämä aukko käsittelemällä teoreettisten suorituskykymittausten sijasta reaalimaailman käyttökohteita. Tarkoitus on varmistua, että WebAssembly suoriutuu määritellyssä tehtävässään paremmin kuin JavaScript.

Aiheen tutkimista varten toteutettiin kvantitatiivinen monitapaustutkimus, jotta tulokset olisivat paremmin yleistettävissä. Tavoitteena oli etsiä ulkoisista lähteistä reaalimaailman käyttötapauksia, jotka korostivat siirtymistä JavaScriptistä WebAssemblyyn sekä näiden suorituskyvyn vertailua. Tulokset osoittivat merkittävää parantumista; joissakin tapauksissa suoritusaika oli jopa 39 kertaa nopeampi. Lisäksi sovellusten kompleksisuuden havaittiin vaikuttavan siihen, kuinka paljon ne voivat hyötyä WebAssemblyn käyttöönotosta.

ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
CPU	Central Processing Unit
GPU	Graphics Processing Unit
JIT	Just-In-Time
SIMD	Single Instruction Multiple Data
Wasm	WebAssembly

Table of contents

Abstract

Abbreviations

1. Introduction 6
2. Background and related research 8
   2.1 WebAssembly 8
   2.2 JavaScript 11
   2.3 Performance comparison 12
   2.4 WebAssembly utilization 15
3. Research method 17
   3.1 Research approach 17
   3.2 Multiple-case study method 17
   3.3 Data collection 18
4. Results 19
   4.1 1Password 19
   4.2 Figma 21
   4.3 Micrio 23
   4.4 TensorFlow 26
   4.5 Summary and Implications 29
5. Discussion 30
6. Conclusions 32
References 33

1. Introduction

The Web has come a long way since its inception; back then, it was just a simple document exchange network that allowed people to publish information through text. Now it has evolved into a widely accessible application platform and a popular target for a growing number of applications — many of which we use daily. Traditionally, these apps were written in JavaScript because it was the only programming language that was natively supported by all major web browsers until recently. Despite significant and ongoing advances in JavaScript engines, the performance of JavaScript is still inadequate for compute-intensive applications, such as 3D-visualizations, games, and cryptographic algorithms. It also has plenty of other problems, especially as a compilation target [9, 33, 70, 76] which is problematic given that many applications on the Web are compiled from languages like C and C++ to JavaScript [41, 47, 113].

WebAssembly is a relatively new (2015) standard supported across all major browsers for running portable low-level code at near-native speed [41, 46]. It addresses the shortcomings of previous attempts and puts a focus on performance to allow new and existing use cases to fully utilize the Web platform. WebAssembly programs are generated binaries that can be downloaded and processed far quicker than JavaScript applications, which are hand-written by developers and require a few extra steps since JavaScript favors convenience over performance. In contrast, WebAssembly applications are constructed by compiling existing programs into bytecode. As a result of these features, WebAssembly has become an emerging technology in the Web community for performance-demanding tasks [45].

Previous research has strongly demonstrated that WebAssembly outperforms JavaScript in complex tasks [41, 44, 46, 59, 68]. Many of these findings are based on compute-intensive benchmarks [41, 44, 46], such as scientific applications and image processing [46], which fit the description of WebAssembly’s intended use case [101]. However, they are specifically designed to highlight differences in performance, and as a result, they may not necessarily present real applications found on the internet. To address this issue, this thesis

reviews the performance of real-world WebAssembly applications to support the findings of the previous research. The main research question is: What are the benefits of WebAssembly over JavaScript in terms of performance in complex Web-based applications? To provide a greater generalization of the results, the topic is approached utilizing a multiple-case study methodology. Data is gathered from web applications that have made the switch from JavaScript to WebAssembly and released benchmarks comparing the performance of the two versions.

In the following chapter, a review of related literature will be examined to provide insight into what WebAssembly is, how it differs from JavaScript, and how it can be used to replace JavaScript in performance-demanding parts of an application. In addition, the state of adoption of WebAssembly is overviewed and various use cases are identified. The third chapter goes into greater detail about the used research methodology and how the study will be carried out to support the hypothesis. The fourth chapter presents the results by comparing the performance of different JavaScript and WebAssembly applications. This is followed by a discussion in the fifth chapter about the meaning of the findings and to address certain limitations of the research. The sixth chapter summarizes the thesis and proposes conclusions based on the results to back up previous research.

2. Background and related research

This chapter presents a general overview on WebAssembly and JavaScript, followed by a side-by-side comparison of these two languages to outline their advantages over the other and their distinct use cases. The understanding of WebAssembly's position and purpose in the Web environment is important to properly utilize it. Finally, the findings of previous studies about the prevalence of WebAssembly on the Web are presented to provide insight into its state of adoption.

2.1 WebAssembly

WebAssembly (abbreviated Wasm) is a standard for running low-level bytecode on the Web [41, 44]. It is the outcome of unremarkable cooperation across all major browser vendors with the common goal of replacing prior insufficient technologies with a better alternative [41, 99]. And unlike its predecessors, WebAssembly offers a universal solution to high-performance applications without sacrificing safety, compactness, or portability [41]. It enables applications written in high-level languages, such as C/C++ and Go [46, 50] to run in the browser at near-native speed and with predictable performance [41, 44, 46].

The browser giants (i.e., Google, Mozilla, Microsoft, and Apple) first unveiled the technology in 2015 while it was still in the works [6], and eventually implemented it in their browsers in 2017 [102]. At the time of writing (October 2021), over 94% of all browser installations support WebAssembly [21]. Due to the wide browser coverage, WebAssembly has been growing in popularity [45] and has given web developers a new tool to work with in situations where performance is crucial. Despite its name, WebAssembly is not limited to browsers but was designed to be portable, and therefore it does not depend on JavaScript or the Web [41]. This has led to the expansion of use cases to other domains like Internet of Things (IoT) and embedded devices, serverless cloud computing, and standalone runtimes [50, 74].

Portability is not the sole reason why WebAssembly has been widely used outside browsers. Other factors include well-structured design, built-in security, not to mention the competitive performance with native code. Specifically, the formal semantics of WebAssembly has led to a particularly clean design that can be verified and formally analyzed for soundness [41, 99], thus leaving little room for bugs and inconsistencies. Also, since code on the Web is retrieved from untrustworthy locations, the code must be first validated before it can be safely executed [41]. Therefore, significant effort has been put into ensuring the safety of WebAssembly. For example, applications cannot corrupt their execution environment or access arbitrary memory [41, 99, 103] unlike in C/C++, where memory safety concerns are ever so prevalent [10, 77, 109]. These measures protect users from buggy or malicious code and provide developers a safe environment to work with [103].

Before the introduction of WebAssembly, JavaScript was the only supported programming language by all browsers [41, 46] for over 20 years [107]. It was designed to be a high-level language, yet many applications on the Web are compiler-generated through Emscripten [27] and Cheerp [49] from C/C++ programs to a low-level variant of JavaScript called asm.js [41, 99]. Asm.js is an extremely restricted subset of JavaScript that provides only numbers, arithmetic, function calls, and heap accesses. Objects and strings, for example, are not supported. These changes make it highly optimized, give compilers a clear definition of what kind of code should be produced, and thus performs better than ordinary JavaScript. [54, 114]

Despite the fact that JavaScript has seen significant performance gains in modern virtual machines, it still suffers from inconsistency and a number of other issues, particularly when used as a compilation target [9, 70, 76]. Other prior developments like Native Client and Microsoft's ActiveX have also attempted to run low-level code on the Web but have come short in doing so [41]. Consequently, WebAssembly was developed to replace these obsolete technologies, especially asm.js [41]. Nowadays, many compilers provide a drop-in replacement for asm.js and prefer compilation to WebAssembly [28]. However, the aim of WebAssembly is not to substitute JavaScript but to complement it in performance-demanding applications [51, 63] to provide a smooth experience. A JavaScript application

programming interface (API) is used to load WebAssembly modules and invoke methods inside the modules [56, 57]. Likewise, WebAssembly can also call JavaScript functions (e.g., to perform some actions and return values). This is important because WebAssembly cannot access the web page contents (DOM) directly nor use Web APIs [57], which means that the two must communicate together. Traffic of this sort can cause significant overhead due to context switching, though it has been improved upon recently [17].

To enable humans to read and edit WebAssembly, there is also a textual representation of the binary code called WebAssembly text format [41, 59, 99]. Listing 1 shows a function in C language for multiplying two numbers together, followed by listing 2, which shows the result as if the function was compiled to WebAssembly in the human-readable form. This intermediate form allows code to be written, viewed, and debugged manually [104] (i.e., to review and understand how the instructions were compiled, and to help in performance optimization).

Listing 1. C code for multiplying two numbers

```c
int multiply(int x, int y) {
  return x * y;
}
```

Listing 2. WebAssembly text format for multiplying two numbers

```
(module
  (func (export "multiply") (param $x i32) (param $y i32) (result i32)
    local.get $x
    local.get $y
    i32.mul
  )
)
```

However, the intention of WebAssembly is not to be written by hand but to act as a compilation target for other programming languages [68, 108]; thus used to compile existing programs, libraries, and projects to the Web. A wide variety of backends are supported for compiling, like C, C++, C#, Go, and Rust [46, 50], which gives developers the much-needed freedom to select a language of their liking, and ultimately makes WebAssembly easier to

adopt. Although various existing languages are supported, there are also new ones specifically designed to target WebAssembly, such as AssemblyScript, which is a TypeScript-like language that is easily accessible as it integrates with the existing Web ecosystem [92]. This makes it straightforward for web developers to start utilizing WebAssembly since they do not have to learn a new language but can still enjoy the same benefits.

2.2 JavaScript

JavaScript is one of today's most widely used programming languages [38, 76]. It is best recognized for its use in web applications, but it is also becoming increasingly popular in other areas like server-side applications, mobile apps, and desktop apps [76]. Due to the wide extent of the language, it is critical that the performance of JavaScript is up to par even with complex applications. This demand has been facilitated by substantial advancements in JavaScript performance in recent years, such as the development of greatly optimized just-in-time (JIT) compilers [37, 76].

JavaScript is an interpreted programming language, which means that the code is interpreted at runtime rather than being compiled into bytecode in advance. In addition, one does not have to explicitly define variables before using them or specify types for variables, making it a dynamic language. [112] While these properties have multiple inherent benefits (e.g., faster development cycle, portability, and the convenience of use), it also harms the performance. To address this issue, JIT compilers have been introduced to improve the performance of JavaScript in browsers [5]. Although the implementations differ from browser to browser, it fundamentally works the same: by optimizing code that is called often to reduce compilation time [40]. This addition has had tremendous effects on performance, with some reporting up to 20x faster execution time [108].

2.3 Performance comparison

The first point of inspection into a web application's performance starts from fetching the necessary source code. Code that is sent over the internet should be concise to reduce response time [41], allowing users to interact with the site contents faster. WebAssembly is better in this regard because it is smaller than JavaScript due to its compact binary format, even when minified and compressed [15, 41]. Once the resources are downloaded, JavaScript code needs to be parsed into an Abstract Syntax Tree (AST), which is then converted to an intermediate representation [15, 67]. In contrast, WebAssembly does not require this step since it is already an intermediate representation. However, it still needs to be decoded and validated until it is ready for compilation, but this takes far less time than parsing JavaScript. [15, 16, 105] In addition, due to its binary format, WebAssembly can be used for streaming compilation, where the code is compiled while it is still being downloaded [16, 41]. Another distinct feature is that compiled WebAssembly modules can be cached on the client's machine. This means that the modules do not have to be downloaded and compiled every time resulting in notably faster page reloads. [13, 55] Together these factors make the load times of WebAssembly significantly better than those of JavaScript.

After the initialization phase is complete, the code needs to be prepared through optimization and compilation until it is finally ready for execution. In the case of JavaScript, the JIT compiler can start optimizing only after it has observed what types are being used. In addition, depending on the used types, the same code may need to be compiled multiple times and can be subject to additional re-optimization. WebAssembly on the other hand is precompiled and already optimized by a compiler like LLVM, and thus less work is needed for both optimization and compilation. [15] Although browsers have different implementations for compiling WebAssembly, it is much closer to machine code than JavaScript (e.g., explicit types) [15], and therefore WebAssembly has far less benefit from JIT compilation [108]. The entire execution flow from start to finish is summarized in Figure 1, which demonstrates that WebAssembly is presumptively and generally faster.

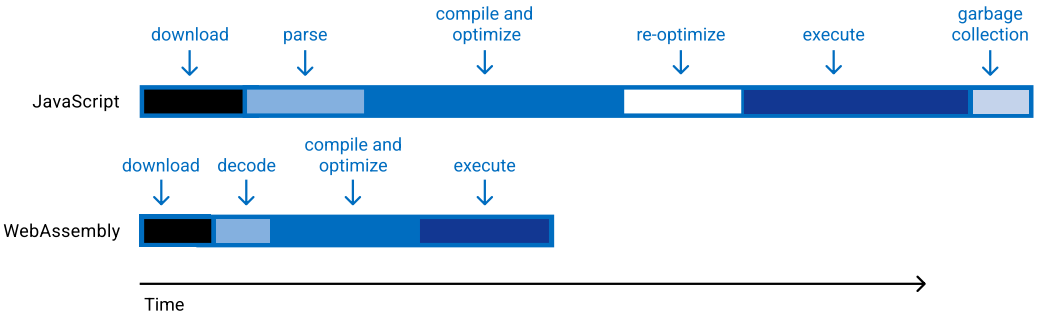

Figure 1. Start-up performance of JavaScript versus WebAssembly, adapted from [15]

While JavaScript can run as fast as WebAssembly, it is notoriously difficult to compile efficiently [46]. Browsers have different JIT compilers with their own optimization methods [15], which may not always work in one's favor, and thus it is hard to get predictable performance across browsers. WebAssembly is different in this regard, as one of its major design goals was to provide an edge over JavaScript in terms of execution speed with consistent performance [41, 46]. And it has succeeded in just that — many previous studies have demonstrated that WebAssembly performs 1.5-2x faster than JavaScript [41, 44, 115].

In the paper that introduced WebAssembly, the researchers show that a C program compiled to Wasm rather than asm.js runs 34% faster in Google Chrome [41]. In addition, multiple benchmarks indicate that WebAssembly runs near to that of native code (see Figure 2) [41, 46]. There are however mixed results regarding the performance over JavaScript since the findings are much dependent on the used benchmarking tools, environments, and input sizes. JavaScript may very well perform better than WebAssembly in some scenarios, but it is generally slower. [108] Figure 3 demonstrates the outcome of SPEC CPU benchmarks, which include scientific applications and image/video processing, to compare the performance of asm.js to WebAssembly. These results show that WebAssembly performs 1.54x faster on Chrome and 1.39x faster in Firefox, which support the findings of Haas et al. [46]

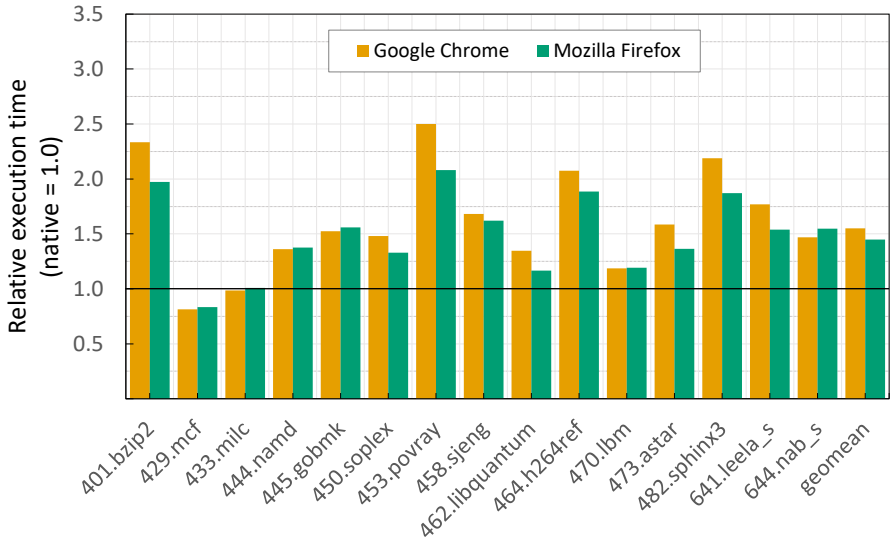

Figure 2. WebAssembly SPEC CPU benchmarks, adapted from [46]

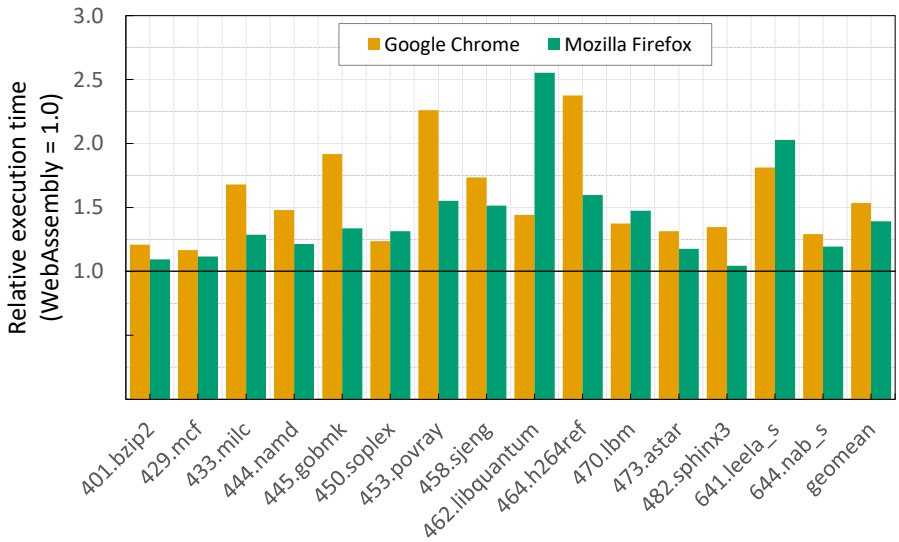

Figure 3. Relative execution time of asm.js to WebAssembly, adapted from [46]

It is important to note that WebAssembly is still relatively new and being actively developed. One of the planned features is garbage collection to handle memory allocation and deallocation automatically [41, 46]. In other words, memory for new variables is handled on behalf of the developer and discarded when it is no longer needed. The lack of this feature can be seen as one downfall of WebAssembly because the code is prone to memory leaks

and requires more effort from developers. However, this also means the performance is better and more consistent [15]. In contrast, JavaScript does have garbage collection, making it an attractive language for developers to use due to its convenience [48]. There are also other features that can boost the performance of WebAssembly applications: threads for concurrent execution [100] and Single Instruction Multiple Data (SIMD) to run the same operation on multiple data entities simultaneously [106]. In addition, WebAssembly has native support for 64-bit integers [26], which may benefit arithmetic calculations since JavaScript only supports 53-bit integers [78].

2.4 WebAssembly utilization

A study published by Musch et al. in 2019 [64] reveals that in the Alexa Top 1 million websites ranking, 1 out of 600 sites utilize WebAssembly. In comparison, JavaScript is used by almost every popular site [70]. The utilization rate is understandable since WebAssembly is an emerging technology and was only four years old at the time of the research. Among the other most noteworthy of the results is that over 50% of these sites use it for malicious purposes like unconsented cryptocurrency mining or obfuscation, to avoid detection by analysis tools such as adblockers. However, a more recent study by Hilbig et al. [45] highlights that the number of WebAssembly-based cryptominers has decreased significantly. In addition, testing for WebAssembly support within browsers was found to be a common use case by JavaScript libraries [45, 64].

Musch et al. [64] manually classified 150 Wasm modules and proposed six categories for their purpose as depicted in Table 1. As previously mentioned, half of these modules were identified to be used for malicious purposes, but other fields of use were recognized as well. For example, games accounted only for 3.5% of the analyzed sites while libraries nearly 40% [64]. A further study then analyzed the prevalence and the nature of these libraries [45]. They discovered that almost 35% of all domains that use WebAssembly are using Hyphenopoly.js, which is a polyfill for text hyphenation if the browser does not support it natively [65]. Another discovery was a library called long.js for 64-bit integer computation [20], which was found in 25% of the sites [45]. And lastly, they found Draco, though less

widespread (1.8%), it is a library used for “compressing and decompressing 3D geometric meshes and point clouds” [39, 45].

Table 1: Categorization and prevalence of WebAssembly applications, adapted from [64]

Category	# of unique samples		# of unique websites	
Custom	17	(11.3%)	14	(0.9%)
Game	44	(29.3%)	58	(3.5%)
Library	25	(16.7%)	636	(38.8%)
Mining	48	(32%)	931	(55.7%)
Obfuscation	10	(6.7%)	4	(0.2%)
Test	2	(1.3%)	244	(14.9%)
Unknown	4	(2.7%)	5	(0.3%)
Total	150	(100.0%)	1,639	(100%)

There are also other studies that have attempted to analyze the use cases of WebAssembly applications. Namely, Romano and Wang have created a tool called WASim [73] to help classify the purposes of these applications by analyzing code features using machine learning. Through manual inspection, they divided the purposes into eleven categories, which further expands on the classification made by Musch et al [64]. In addition, Hilbig et al. [45] also did something similar recently (2021) by categorizing 100 randomly sampled Wasm binaries, which produced the most comprehensive classification amongst the similar studies (see Table 2). According to the study, games and text processing applications are one of the most common uses. Though they did find a variety of other less frequent use cases as well, like chat and visualization applications, media players, and blogging sites. [45]

Table 2: Use cases of WebAssembly and their prevalence, adapted from [45]

Application domain	# Binaries	Application domain	# Binaries
Games	25	Online gambling	2
Text processing	11	Barcodes and QR codes	2
Visualization / Animation	11	Room planning / Furniture	2
Media processing / Player	9	Blogging	2
Demo e.g., of a programming language	7	Cryptocurrency wallet	2
		Regular expressions	1
Wasm tutorial or test	5	Hashing	1
Chat	3	PDF viewer	1

3. Research method

This chapter describes the chosen research method as well as the reasoning behind its selection. Subsequently, an overview is presented on how the data was gathered and how it will be analyzed to support this study.

3.1 Research approach

The objective of this study was to understand the performance benefits of using WebAssembly in real-world applications. Creswell & Creswell [19] suggest that a quantitative approach is the best-suited for testing a theory, and it is often used with numerical data for describing a phenomenon [81]. In contrast, a qualitative approach is narrative [110] and focuses more on words [75]. Consequently, since computational performance is related to numbers in terms of execution speed, a quantitative approach is seen as more suitable for this study.

3.2 Multiple-case study method

Case study research is often conducted as a qualitative study but it can also include or be limited to quantitative data [110]. It is described by Yin [110] as an inquiry aimed at gaining a comprehensive understanding of a contemporary issue within its real-life context. Case study research can also contain multiple studies, and it can be wise since it allows to explore the phenomenon more effectively according to Stake [80]. Similarly, Baxter & Jack [7] conclude that a multiple-case study is the most advantageous method for researchers to study a phenomenon. Therefore, a quantitative multiple-case study approach was selected because it allows for better generalization of performance improvements across different applications.

Stake [80] recommends selecting 4-10 studies when conducting a multi-case study since two or three cases do not present enough generalization. However, Yin [110] argues that a multiple-case study is both costly and time-consuming to conduct making it beyond the reach of a single student. Yin [110] adds that a multiple-case study may be preferred over a single case study even if there are only two cases. This is because single case studies are more vulnerable to misinterpretation and multiple cases have more analytical benefits and are more compelling due to greater evidence [110].

3.3 Data collection

Primary sources were collected from a website named “Made with WebAssembly” [52], which showcases production applications, libraries, and sites that utilize WebAssembly. The collection is open source and is updated by multiple contributors [93], and provides a good catalog of potential cases that would be hard to find otherwise manually. Upon further inspection, the contents were analyzed to select cases that specifically provide performance benchmarking details over their prior JavaScript implementation. Nine different cases were identified to be suitable for this study, which were further narrowed down to four cases based on the quality and extent of the available documentation.

Although many other applications are using WebAssembly, such as the Zoom web conferencing software [69, 116], Adobe Photoshop for the Web [66], and uBlock Origin for content-filtering [53, 94], there is insufficient documentation or no documentation at all about the potential benefits they got from switching to WebAssembly. Thus, they would provide no value to the research since it relies on publicly available data.

4. Results

This chapter encompasses the results of the conducted multiple-case study. Each case is reviewed separately by first introducing the service or product in question and then presenting the potential benefits and the process of switching to WebAssembly.

4.1 1Password

1Password is a popular password manager with a focus on security, ease of use, and a convenient autofill system (see Figure 4). Their product is trusted by over 15 million users and more than 75 000 businesses for managing passwords and securing sensitive information. [1, 42] 1Password is available on all major browsers, desktop, and mobile, making it easy to authenticate on different devices [1]. The need for password managers arises from the way we authenticate to various online services, for example, to read through emails, watch movies and TV shows, or shop clothes online.

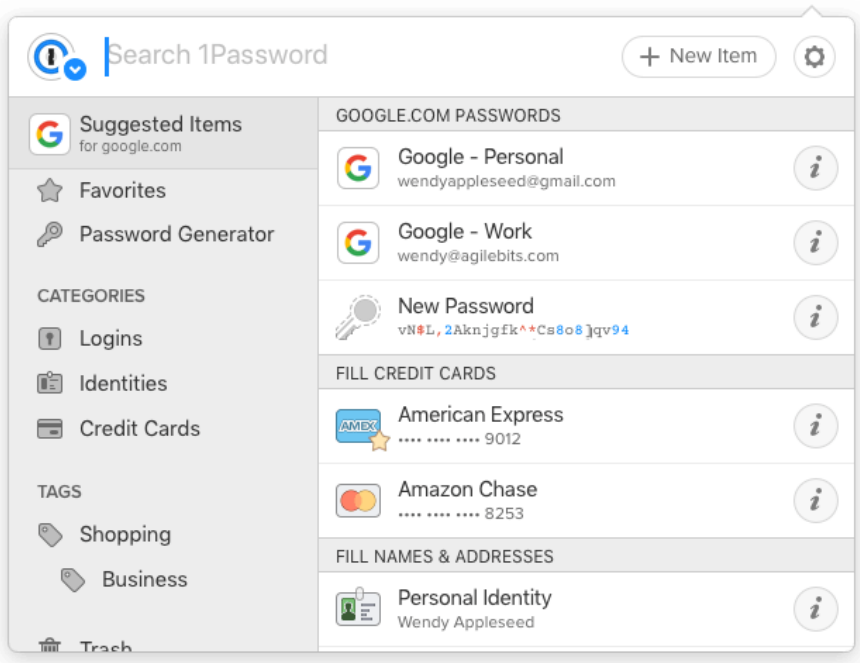

Figure 4. 1Password browser plugin [2]

Alphanumeric passwords continue to be the most dominant approach in user authentication [11, 43]. They protect accounts with sensitive data and valuable assets, which engages the interest of attackers. Therefore, creating strong passwords is an important step to protect accounts from exploitation. However, according to Adams et al. [4] users may find it challenging to remember secure passwords that are long and random. And this combined with the fact that people have more accounts than ever before and are increasingly resorting to insecure practices, like reusing passwords across various websites, has raised the concerns of security researchers [31]. Password managers, such as 1Password, offer a potential solution to the current password conundrum. A password manager is a tool for storing all passwords for different services in one place and for filling in the authentication details later automatically. This eliminates the need for users to remember all of their passwords, making authentication more secure by default.

In May 2019, the 1Password team reported that they had transitioned to using WebAssembly in their browser plugin. It is unknown whether they were using plain JavaScript or asm.js before the change. And although not much is reported about the subject, they highlight that their page filling and analysis system operates 2x faster using WebAssembly. Furthermore, websites with many input fields are reportedly 13x faster in Google Chrome and 39x faster in Mozilla Firefox. [82] They seemed to be pleased with the result because later in the same year they rewrote the core of their product in Rust and WebAssembly to aim for better portability and improved performance on the Web [12, 23]. This transform required considerable effort since the core was previously written in Go [23], which can be also compiled to Wasm [36]. 1Password was however more interested in Rust because some of its language features, such as memory safety and the gained performance benefit from not having a built-in garbage collector [23]. In addition, Rust can be easily compiled to Wasm [58] and provides a developer-friendly environment for doing so.

On top of rewriting the core, they also touched on other parts of the product base, including Markdown parsing and time-based one-time password (TOTP) generation [12]. By rewriting parts of their product in Rust, 1Password was able to share code between multiple platforms, such as desktop and browsers environments. This is a significant benefit in addition to the

improved performance, as it saves both time and money while ensuring consistent behavior across different implementations.

4.2 Figma

Figma is a web-based tool for editing graphics, wireframing, creating user interfaces, brainstorming, and rapid prototyping (see Figure 5) [29]. It is widely popular among graphic designers [25, 32], and many large enterprises like Microsoft, Spotify, Zoom, and Uber have picked up the tool to streamline their design workflow and create great user experiences [30]. Figma allows for real-time collaboration and since it is strictly web-based it can be accessed anywhere with Internet access [29].

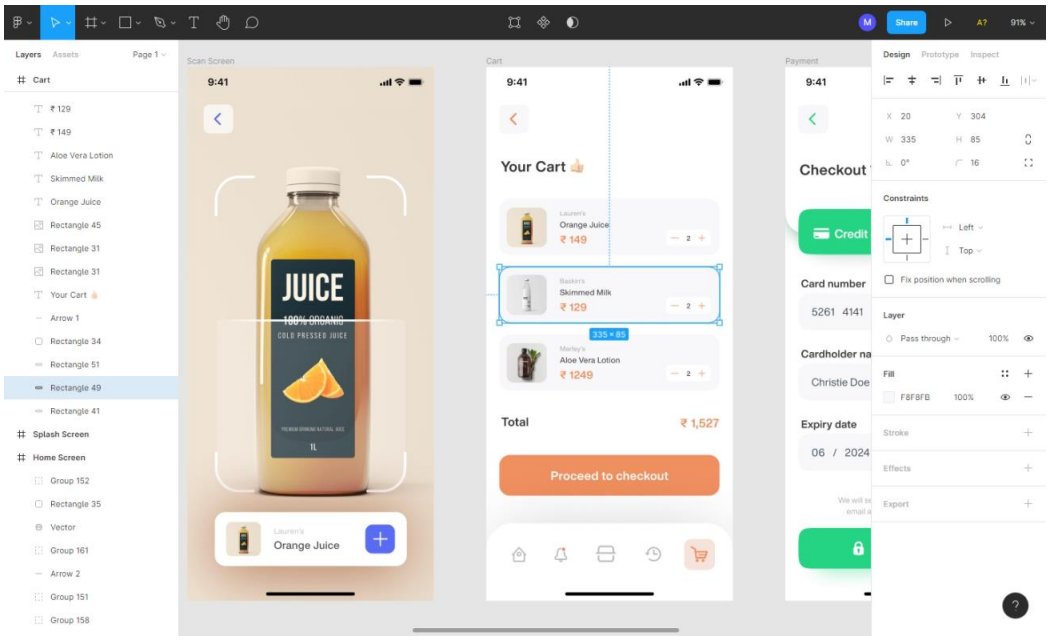

Figure 5. Figma editor view (screenshot from [29])

Figma's interactive editor is the center and the most performance-intensive part of their product. This means that the performance of the editor is pivotal and that it must be up to par for optimal user experience. Interestingly, the developers behind Figma had become aware of WebAssembly's development and had observed other applications benefiting from

it, which led to them giving it a try themselves [96]. The transition turned out to be relatively painless since Figma is written in C++ [96], meaning it can be easily compiled to WebAssembly. Furthermore, since their editor was previously cross-compiled to asm.js using Emscripten [95], which provides a drop-in replacement in favor of Wasm [28], no additional work was needed on their part.

The most prominent result they found from switching to WebAssembly was faster load time for Figma design documents. Figure 6 shows that regardless of the document size, the load time improved by more than threefold. This makes the editor more user-friendly when opening and switching between large design documents. They also discovered that the compressed Wasm code is only slightly smaller than the asm.js equivalent, which did not quite meet the anticipation they had. [96] However, the uncompressed code resulted in nearly 2x smaller size [96], which supports the findings of Haas et al. [41].

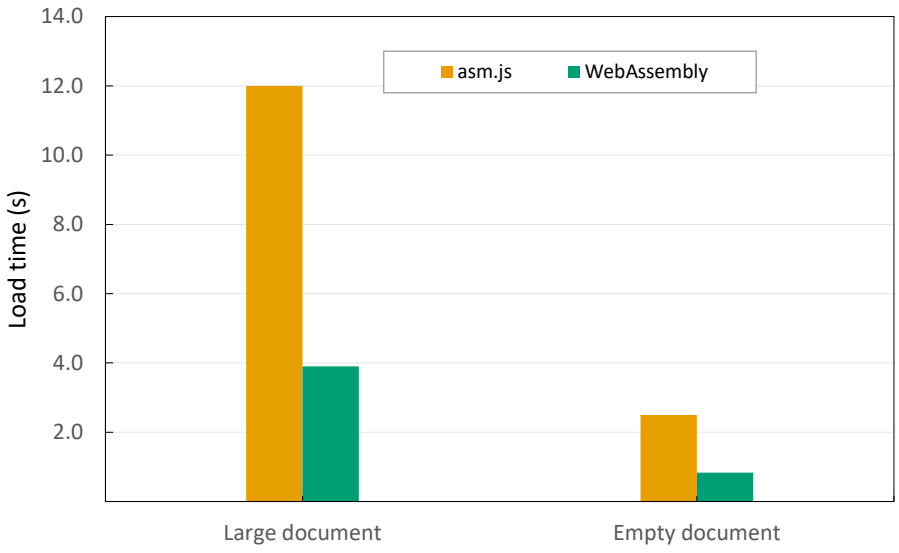

Figure 6. Figma load time comparison on Mozilla Firefox, adapted from [96]

The blog post introducing the usage of WebAssembly dates to 2017 and mentions that Figma only uses WebAssembly in Mozilla Firefox. This is because they ran into some showstopper issues in Google Chrome's WebAssembly implementation making it unsuitable for a wider

audience. [96] As of now (2022), both issues related to WebAssembly that were stopping them seem to be fixed [97, 98]. Figure 7 shows network traffic upon opening a Figma document in Google Chrome, which highlights Wasm-related files being transferred — indicating that Figma is now using WebAssembly for Google Chrome too.

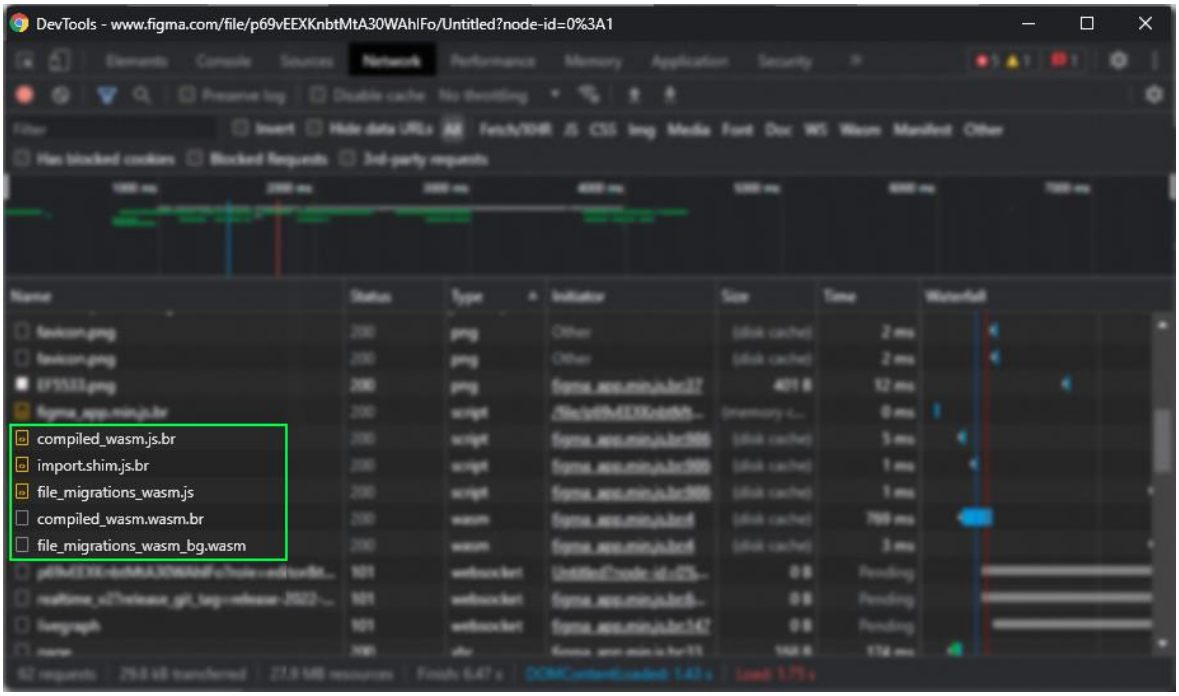

Figure 7. Network traffic upon opening a Figma document in Google Chrome

4.3 Micrio

Micrio is a storytelling platform [60] that is used by a number of broadcasting companies, museums, artists, and other professionals [61]. It enables creators to create visually appealing digital content in the form of high-resolution images and videos with minimal effort. The content can be further enriched with markers, tours, and audio to make it more engaging for end-users [62]. They also feature 360-degree photos, which simulate being in the actual place of where the photo was taken and having the ability to look around for more interactivity. Figure 8 shows an example of an application made possible by Micrio's technology. It is a digital tour of Rijksmuseum — "the Dutch national museum dedicated to arts and history in Amsterdam" [71], giving people a chance to explore the art from the comfort of their home.

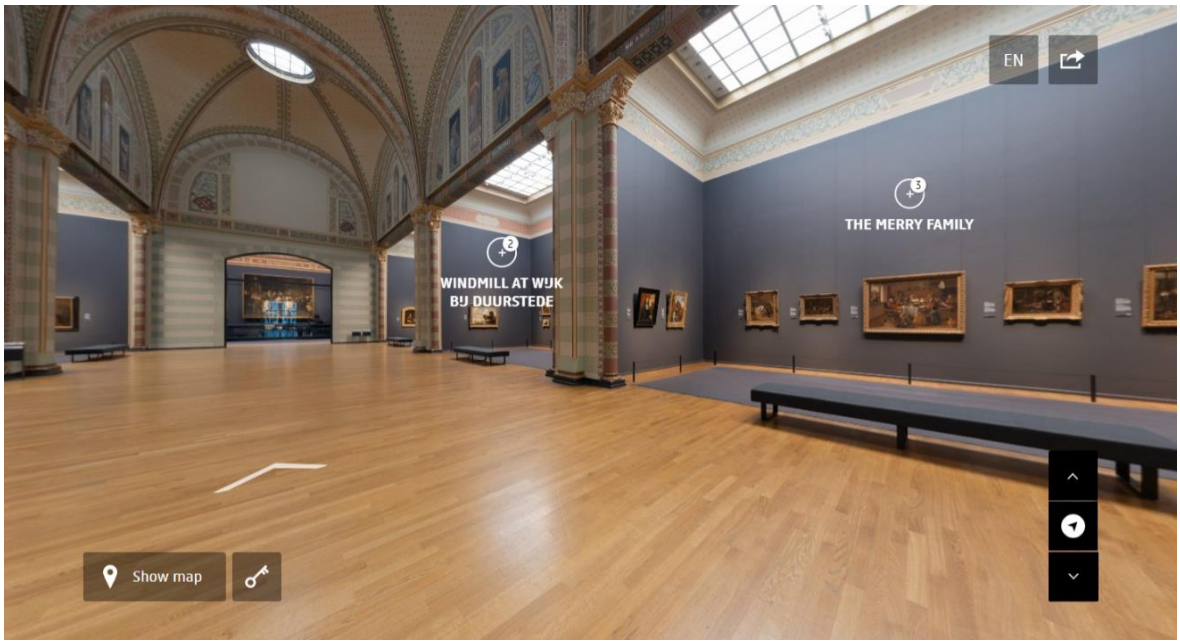

Figure 8. Digital tour of Rijksmuseum (screenshot from [72])

Rendering high-resolution images that users can freely zoom, pan, and navigate requires quite a lot of computational power, not to mention that the images can also be in 360 degrees, which means even more calculations must be done due to 3D matrix projection. This means delivering fast load times and a smooth interactive experience is a must. A frame drop happens when a Graphics Processing Unit (GPU) or a Central Processing Unit (CPU) fails to produce a new frame within the desired timeframe, which means that the display buffer is not updated, and the previous frame is still being shown [22]. A single frame drop in the processing of visuals can greatly hinder the user experience, for example in video games, or in this case, the Micrio library.

In late 2019, one of Micrio's developers had discovered WebAssembly and wanted to experiment if it could make the library run better than the current JavaScript version. The current library was done entirely in JavaScript: using three.js and WebGL for 360° images and Canvas2D for rendering two-dimensional images. Although there was nothing inherently wrong with the performance of the library at the time, the developer felt the need to try it out and simultaneously attempt to improve the library's code architecture. [24] They

saw the potential in WebAssembly as it could run code at near-native speed making a big difference in performance and could thus ensure a smooth browsing experience for the clients.

The transition to WebAssembly turned out to be anything but easy. Micrio first rewrote the core features of the library (i.e., a virtual zoomable and pannable high-resolution image, event handling, and rendering logic) to build a minimum viable product (MVP). This was done using C++ and Emscripten with Wasm as the compilation target. Having finished the initial prototype, they decided to benchmark it to see whether they were headed in the right direction. It turned out that the compiled binary was already at this point more than 3x larger than the feature-complete JavaScript version. Consequently, they concluded that it was not feasible to continue since the library size would grow drastically, thus hindering the user experience. [24]

Fast forward a few months later, the developer decided to give WebAssembly another try [24], this time using something a little more familiar than C++. Specifically, he stumbled upon the AssemblyScript language, which is just about the same as TypeScript [92]. TypeScript is a “strongly typed programming language that builds on JavaScript” [92] mainly used in web applications, which the developer had strong experience in. Having learned from the previous attempt, the new implementation started small to see if it would face the same problem. After Micrio realized the previous issue had been solved, they continued to add more functionality to the prototype until it would match that of the JavaScript counterpart. It was then again time for benchmarking to witness if it would yield better results. Initially, the performance improvement for the new version 3.0 was merely 14% less CPU usage but after lots of careful optimizing they got it down to 65% (see Figure 9). The final JavaScript bundle which includes the Wasm binary is 60% smaller in size, or when compressed still 12% lighter than the version 2.9. [24]

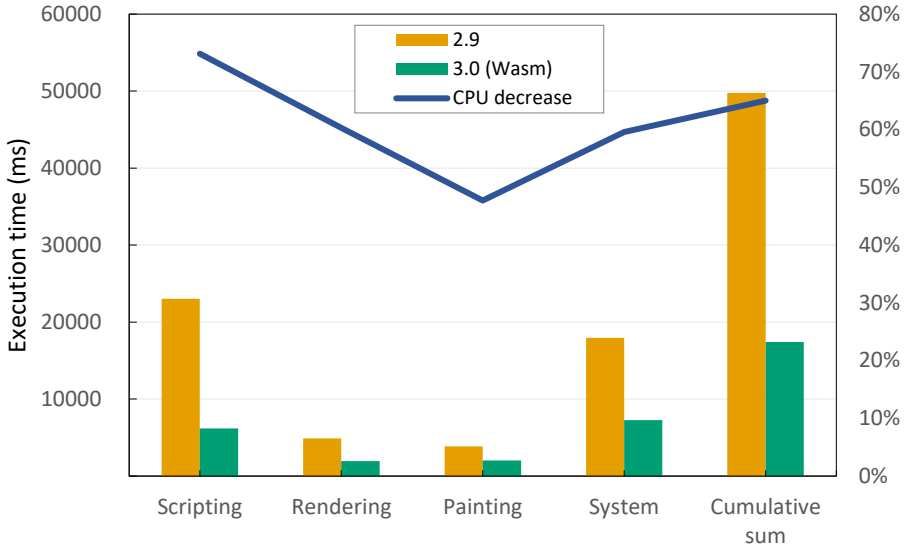

Figure 9. Benchmark of Micrio versions 2.9 and 3.0, adapted from [24]

It is worth noting that the performance comparison was done using only one sample on a single environment (i.e., the same device and browser), which means that the results could vary. Micrio has however provided hosted versions of the older and newer versions of their library, which are using the same sample. This can be used for benchmarking by a third-party author to verify the results. Furthermore, since the WebAssembly version uses WebGL rendering rather than Canvas2D rendering in the JavaScript version, the results are not directly comparable. Similarly, the version 2.9 already contains some basic math functions in Wasm, thus the improvement could be even higher than presented. In any case, Micrio has successfully migrated a pure JavaScript project to a native WebAssembly-built engine, making a huge difference in performance, cleaning up the code architecture, and getting rid of extra dependencies. They also reduced memory footprint and decreased the library size in the process. [24]

4.4 TensorFlow

Machine learning has become an important tool today — it can help find features and patterns in data to optimize processes and make decisions with minimal intervention from humans. TensorFlow is a state-of-the-art machine learning system that works seamlessly at

large scale and across heterogeneous environments [3, 86]. It is open source and has comprehensive tooling, libraries, and resources giving both researchers and developers easy access to start learning and building applications [86, 87]. Furthermore, TensorFlow provides different levels of abstraction for complex machine learning models [87], which makes it suitable for a wider audience regardless of their experience. Many companies today are utilizing TensorFlow to benefit from deep learning in their business operations, for example, Spotify, Airbnb, and PayPal [88].

TensorFlow supports several programming languages for constructing and executing graphs, one of which is JavaScript [84]. The JavaScript implementation is further divided in four different backends that implement tensor storage and mathematical operations: plain JavaScript, WebGL, WebGPU, and the latest addition, Wasm [89]. Though, one caveat is that the WebAssembly version does not have all the features as the JavaScript/WebGL counterpart [79] — but they have been adding more features since the release [111].

The WebAssembly backend was launched in 2020 as an alternative to the highly performant WebGL backend, allowing for faster CPU execution and increased performance on a wider set of devices [79]. WebGL is a JavaScript API to access the GPU for graphics operations and rendering [18]. It is widely known that GPUs provide high levels of parallelism [14, 34], and therefore its usage is increasingly popular in applications that require a large amount of computation power [35], such as machine learning. The performance of the Wasm backend is competitive with that of WebGL in some situations, yet it only has the second highest priority just after WebGL because of its inferior performance and lacking feature parity [83, 111]. Then, a couple of months later in 2020, TensorFlow announced a major performance update to the Wasm backend, featuring SIMD and multithreading, which highlighted up to 10x improvement. Though, noteworthy is that the default WebAssembly backend does not include SIMD nor multithreading, meaning developers have to explicitly declare if they wish to use them. [111]

As TensorFlow machine-learning models are based on algorithms, equations, and a slew of mathematical operations, the performance of these are critical and make a huge difference.

There are two main benchmarks that TensorFlow has used to compare the performance of different backends. One is BlazeFace, which is a light face detector model with 0.1 million parameters and roughly 20 million multiply-add operations [8, 90, 111]. And the other being MobileNet V2, a medium-sized model (3.5 million parameters and some 300 million multiply-add operations [111]) described as “a family of neural network architectures for efficient on-device image classification and related tasks” [91]. Benchmarks of the different backends are shown in Figures 10 and 11, where inference time refers to the time taken by the process of using a trained model to make a prediction [85]. From the results, it is clearly visible that Wasm is 10-30x faster than plain JavaScript. Although the results look bad for JavaScript, it is important to note that this is vanilla JavaScript, which does not have any optimizations or multithreading [83], meaning that the results would be closer with asm.js and SIMD. The results also outline that Wasm generally performs 2-4x worse than WebGL but with the addition of SIMD and threads it performs significantly better than all the backends.

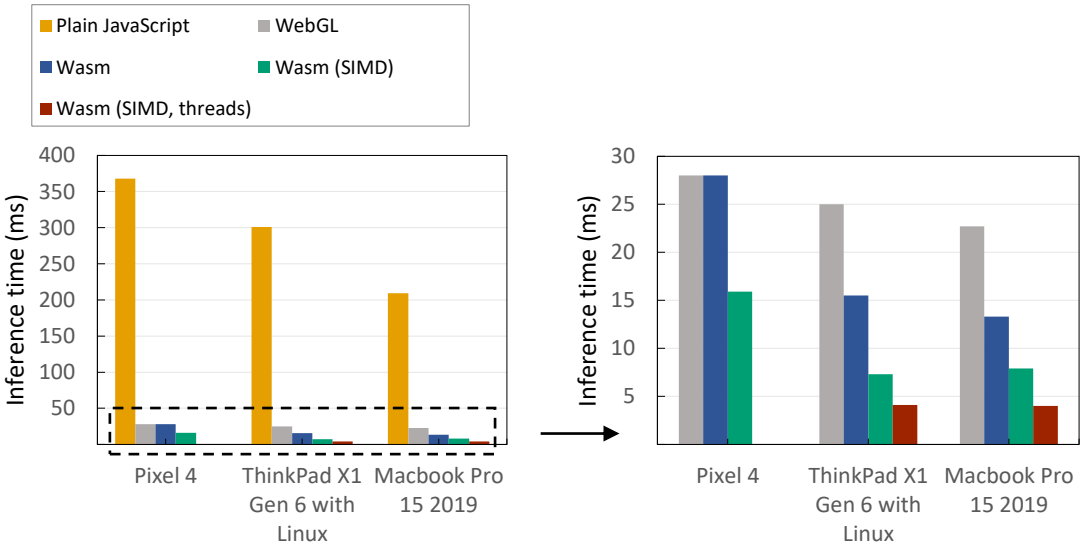

Figure 10. BlazeFace benchmark, adapted from [111]

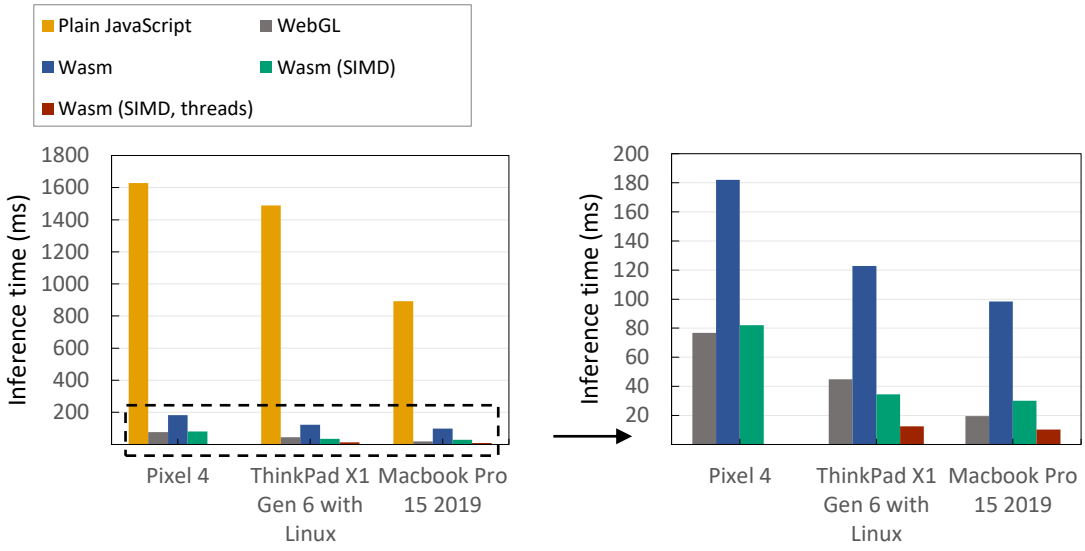

Figure 11. MobileNet V2 benchmark, adapted from [111]

4.5 Summary and Implications

The results from the conducted multiple-case study included four unique cases and showed varying results regarding performance. All the cases are very different in nature and have separate purposes, but it appears that complex tasks where similar operations are executed multiple times gained the biggest benefits. Namely, these include Micrio and TensorFlow, which rely heavily on mathematics and equations, meaning these factors have the greatest impact on performance. Furthermore, 1Password also gained notable benefits from their transformation in cases where data is substantially larger, taking more time to process it.

Figma did not report anything related to runtime performance, which is understandable as they were using asm.js already and there are many aspects to measure. However, they did find significant improvement in their load times. Overall, the results suggest that the performance of asm.js is much closer to WebAssembly than could be achieved with plain JavaScript. And, that WebAssembly has brought prominent benefits to each of the applications in regards of performance.

5. Discussion

This thesis aimed to explore the performance benefits of using WebAssembly to complement JavaScript in compute-intensive production applications. The results from the multiple-case study indicate that WebAssembly can be used to increase the performance of web applications. The observed improvements were in the range of 2-39x faster load or execution time depending on the inherent characteristics of the application, but most importantly all the cases showed positive results. The more demanding the application was performance-wise, the more benefit was seen from implementing WebAssembly. In addition, plain JavaScript is seen performing significantly worse than asm.js.

The results of this study are in line with those observed in previous research and confirm that WebAssembly has given web applications a tool to boost their performance even further. These results build on the existing evidence of Haas et al [41] and Jangda et al. [46], who used benchmarking suites to demonstrate the edge WebAssembly gives over JavaScript and how it competes with native code. While these benchmarking tools do present compute-intensive applications, they are primarily used for performance comparison, and thus they might not entirely present real applications. This research acknowledged that limitation and attempted to strengthen the claims. Furthermore, as presented by Hilbig et al., visualization tools were found as one of the most common use cases [45], which is supported by this study since both Figma and Micrio can be classified as such.

Multiple risks are identified for this study: firstly, the cases were selected from a single source of information that is dedicated to showcasing WebAssembly applications, therefore the reliability (i.e., due to favoritism and highlighting only positive outcomes) of the results is limited. Similarly, the validity of the results is impacted by relying on primary sources for the benchmark results and not conducting them independently. Organizations and companies have reported the results themselves, which could be biased or exaggerated to delight users. Furthermore, some of the benchmarks are performed on an improper setup (e.g., one device or browser environment and improper measuring tools) that might not be documented at all,

thus directly affecting the reliability, accuracy, and validity of the results. Generalization is also a thing to consider since the sample size was relatively small, and the examined applications are very different from each other. Though, many of the cases are credible in the sense that they provide the ability to run the benchmarks by other individuals, which means that the results can be validated.

As the study demonstrates, WebAssembly shows promising results, and with the recent additions of SIMD and multithreading [102], it could offer even more room for improvement and a broader range of use cases. The adoption rate is still quite low at the moment but is expected to rise since Wasm is an emerging technology with even more features planned to be added. However, because the results show varying boosts in performance, developers should use discretion to decide on its use, first starting from small and then building up. At the moment, the WebAssembly website recommends using it only for resource-intensive tasks and states that it is not intended to replace JavaScript [105]. They also mention various other use cases that were not entirely identified in the previous research [101], indicating that applications and developers have not yet fully utilized WebAssembly’s potential.

6. Conclusions

This research aimed to examine what performance benefits real-world web applications could gain from implementing WebAssembly to complement JavaScript in performance-demanding functions. To explore the subject, a quantitative multiple-case study on existing applications was conducted for better generalization of the results. Previous research has outlined greater performance in WebAssembly compared to JavaScript, but only through benchmarking tools that do not necessarily present real applications found on the Web. This study was meant to address this limitation and then compare the results to previous findings.

It was observed that the performance benefits are heavily depended on the characteristics of the application and that greater benefit was seen in more complex applications. In particular, the study showed varying results for different applications: from 2 to 39 times faster performance in terms of either execution time or load time. In addition, plain JavaScript was shown to perform far worse than asm.js. These findings are in line with the previous research and provide a clearer understanding of how WebAssembly can be beneficial for existing and future applications.

To better understand the implications of these results, future studies could address the limitations posed by this study, since it relied on the benchmarking results of primary sources instead of conducting them independently. Although the sources could be argued to be fairly credible, the results of the benchmarks may be impacted by inadequate testing setup due to, for example, using only a single browser and device.

References

[1] 1Password. Tour. Retrieved January 9, 2022 from https://1password.com/

[2] 1Password. Downloads. Retrieved February 5, 2022 from https://1password.com/downloads/chrome-os/

[3] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. and Zheng, X. 2016. {TensorFlow}: A System for {Large-Scale} Machine Learning. 265–283.

[4] Adams, A. and Sasse, M.A. 1999. Users are not the enemy. *Commun. ACM* 42, 12 (December 1999), 40–46.

[5] Auler, R., Borin, E., de Halleux, P., Moskal, M. and Tillmann, N. 2014. Addressing JavaScript JIT Engines Performance Quirks: A Crowdsourced Adaptive Compiler. In *Compiler Construction* (Lecture Notes in Computer Science), Springer, Berlin, Heidelberg, 218–237.

[6] Bastien, J. Going public launch bug. *GitHub*. Retrieved October 21, 2021 from https://github.com/WebAssembly/design/issues/150

[7] Baxter, P. and Jack, S. 2008. Qualitative Case Study Methodology: Study Design and Implementation for Novice Researchers. *The Qualitative Report* 13, 4 (December 2008), 544–559.

[8] Bazarevsky, V., Kartynnik, Y., Vakunov, A., Raveendran, K. and Grundmann, M. 2019. BlazeFace: Sub-millisecond Neural Face Detection on Mobile GPUs. *arXiv:1907.05047 [cs]* (July 2019).

[9] Bebenita, M., Brandner, F., Fähndrich, M., Logozzo, F., Schulte, W., Tillmann, N. and Venter, H. 2010. SPUR: a trace-based JIT compiler for CIL. 708–725.

[10] Berger, E.D. and Zorn, B.G. 2006. DieHard: probabilistic memory safety for unsafe languages. *SIGPLAN Not.* 41, 6 (June 2006), 158–168.

[11] Bonneau, J., Herley, C., Oorschot, P.C. van and Stajano, F. 2012. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *2012 IEEE Symposium on Security and Privacy*, 553–567.

[12] Brown, S. 2019. 1Password X 1.17: New brain, new menu, and even more accessible. *1Password Blog*. Retrieved January 9, 2022 from https://blog.1password.com/1passwordx-december-2019-release/

[13] Budge, B. 2019. Code caching for WebAssembly developers. Retrieved January 28, 2022 from https://v8.dev/blog/wasm-code-caching

[14] Cecilia, J.M., Nisbet, A., Amos, M., García, J.M. and Ujaldón, M. 2013. Enhancing GPU parallelism in nature-inspired algorithms. *J Supercomput* 63, 3 (March 2013), 773–789.

[15] Clark, L. 2017. What makes WebAssembly fast? *Mozilla Hacks – the Web developer blog*. Retrieved January 26, 2022 from https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast

[16] Clark, L. 2018. Making WebAssembly even faster: Firefox's new streaming and tiering compiler. *Mozilla Hacks – the Web developer blog*. Retrieved January 26, 2022 from https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler

[17] Clark, L. 2018. Calls between JavaScript and WebAssembly are finally fast 🎉. *Mozilla Hacks – the Web developer blog*. Retrieved February 2, 2022 from https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast-%F0%9F%8E%89

[18] Congote, J., Segura, A., Kabongo, L., Moreno, A., Posada, J. and Ruiz, O. 2011. Interactive visualization of volumetric data with WebGL in real-time. In *Proceedings of the 16th International Conference on 3D Web Technology* (Web3D '11), Association for Computing Machinery, New York, NY, USA, 137–146.

[19] Creswell, J.W. and Creswell, J.D. 2017. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications.

[20] dcode. 2021. *long.js*. Retrieved October 20, 2021 from https://github.com/dcodeIO/long.js

[21] Deveria, A. Can I use... Support tables for HTML5, CSS3, etc. Retrieved October 21, 2021 from https://caniuse.com/?search=webassembly

[22] Dey, S., Singh, A.K., Wang, X. and McDonald-Maier, K. 2020. User Interaction Aware Reinforcement Learning for Power and Thermal Efficiency of CPU-GPU Mobile MPSoCs. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 1728–1733.

[23] Dreimanis, G. 2021. Rust in Production: 1Password. *Serokell Software Development Company*. Retrieved January 9, 2022 from https://serokell.io/blog/rust-in-production-1password

[24] Duin, M. 2020. Going from JavaScript to WebAssembly in three steps. *Q42 Engineering*. Retrieved January 14, 2022 from https://engineering.q42.nl/webassembly/

[25] Edelberg, J. and Kilrain, J. 2020. Design Systems: Consistency, Efficiency & Collaboration in Creating Digital Products. In *Proceedings of the 38th ACM International Conference on Design of Communication*, ACM, Denton TX USA, 1–3.

[26] Ellul, J. 2017. *Towards WebAssembly for Wireless Sensor Networks*.

[27] Emscripten Contributors. Main — Emscripten 2.0.33-git (dev) documentation. Retrieved October 23, 2021 from https://emscripten.org/

[28] Emscripten Contributors. Building to WebAssembly — Emscripten 3.1.1-git (dev) documentation. Retrieved January 14, 2022 from https://emscripten.org/docs/compiling/WebAssembly.html

[29] Figma. The collaborative interface design tool. Retrieved December 14, 2021 from https://www.figma.com/

[30] Figma. You're in Good Company. Retrieved December 14, 2021 from https://www.figma.com/customers/

[31] Florencio, D. and Herley, C. 2007. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web* (WWW '07), Association for Computing Machinery, New York, NY, USA, 657–666.

[32] Fras, K. and Nowak, Z. 2020. WebAssembly – Hope for Fast Acceleration of Web Applications Using JavaScript. In *Information Systems Architecture and Technology: Proceedings of 40th Anniversary International Conference on Information Systems Architecture and Technology – ISAT 2019* (Advances in Intelligent Systems and Computing), Springer International Publishing, Cham, 275–284.

[33] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R.,

Bebenita, M., Chang, M. and Franz, M. 2009. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.* 44, 6 (June 2009), 465–478.

[34] Garland, M., Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y. and Volkov, V. 2008. Parallel Computing Experiences with CUDA. *Micro, IEEE* 28, (August 2008), 13–27.

[35] Garland, M. and Kirk, D. 2010. Understanding Throughput-Oriented Architectures. *Commun. ACM* 53, (November 2010), 58–66.

[36] Go contributors. The Go programming language. *GitHub*. Retrieved January 13, 2022 from https://github.com/golang/go

[37] Gong, L., Pradel, M. and Sen, K. 2015. JITProf: pinpointing JIT-unfriendly JavaScript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, Bergamo Italy, 357–368.

[38] Gong, L., Pradel, M., Sridharan, M. and Sen, K. 2015. DLint: dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (ISSTA 2015), Association for Computing Machinery, New York, NY, USA, 94–105.

[39] Google. 2021. *draco*. Retrieved October 20, 2021 from https://github.com/google/draco

[40] Ha, J., Haghighat, M., Cong, S. and Mckinley, K. 2009. A concurrent trace-based just-in-time compiler for JavaScript. (January 2009).

[41] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A. and Bastien, J. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, Barcelona Spain, 185–200.

[42] Harris, S. 2021. 1Password wins a G2 Best Software award. *1Password Blog*. Retrieved January 13, 2022 from https://blog.1password.com/1password-wins-a-g2-best-software-award/

[43] Herley, C. and Van Oorschot, P. 2012. A Research Agenda Acknowledging the Persistence of Passwords. *IEEE Security Privacy* 10, 1 (January 2012), 28–36.

[44] Herrera, D., Chen, H., Lavoie, E. and Hendren, L. 2018. Numerical computing on the web: benchmarking for the future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages* (DLS 2018), Association for Computing Machinery, New York, NY, USA, 88–100.

[45] Hilbig, A., Lehmann, D. and Pradel, M. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021*, ACM, Ljubljana Slovenia, 2696–2708.

[46] Jangda, A., Powers, B., Berger, E.D. and Guha, A. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. 107–120.

[47] Khan, F., Foley-Bourgon, V., Kathrotia, S., Lavoie, E. and Hendren, L. 2014. Using JavaScript and WebCL for numerical computations: a comparative study of native and web technologies. *SIGPLAN Not.* 50, 2 (October 2014), 91–102.

[48] Kim, M., Jeong, H.-J. and Moon, S.-M. 2017. Small Footprint JavaScript Engine. In *Components and Services for IoT Platforms: Paving the Way for IoT Standards*, Georgios Keramidas, Nikolaos Voros and Michael Hübner (eds.). Springer International Publishing, Cham, 103–116.

[49] Leaning Technologies. C/C++ to WebAssembly Compiler | Cheerp. Retrieved January 24, 2022 from https://leaningtech.com/cheerp/

[50] Lehmann, D., Kinder, J. and Pradel, M. 2020. Everything Old is New Again: Binary Security of WebAssembly. 217–234.

[51] Lyu, S. 2021. High-Performance Web Frontend Using WebAssembly. In *Practical Rust Web Projects: Building Cloud and Web-Based Applications*, Shing Lyu (ed.). Apress, Berkeley, CA, 193–249.

[52] Made with WebAssembly. Home Page. Retrieved November 22, 2021 from https://madewithwebassembly.com/

[53] Made with WebAssembly. uBlock Origin. Retrieved January 2, 2022 from https://madewithwebassembly.com/showcase/ublock-origin

[54] MDN Web Docs. 2021. asm.js. Retrieved January 24, 2022 from https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js

[55] MDN Web Docs. 2021. Caching compiled WebAssembly modules. Retrieved January 28, 2022 from https://developer.mozilla.org/en-US/docs/WebAssembly/Caching_modules

[56] MDN Web Docs. 2021. WebAssembly.instantiate(). Retrieved January 25, 2022 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/instantiate

[57] MDN Web Docs. 2021. WebAssembly Concepts. Retrieved February 2, 2022 from https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

[58] MDN Web Docs. 2021. Compiling from Rust to WebAssembly. Retrieved January 13, 2022 from https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm

[59] MDN Web Docs. 2022. Understanding WebAssembly text format. Retrieved January 19, 2022 from https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

[60] Micrio. Ultra Resolution Storytelling. Retrieved January 14, 2022 from https://micr.io/

[61] Micrio. Why use Micrio? Retrieved January 14, 2022 from https://micr.io/why

[62] Micrio. About. Retrieved January 14, 2022 from https://micr.io/about

[63] Møller, A. 2018. Technical perspective: WebAssembly: a quiet revolution of the web. *Commun. ACM* 61, 12 (November 2018), 106.

[64] Musch, M., Wressnegger, C., Johns, M. and Rieck, K. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (Lecture Notes in Computer Science), Springer International Publishing, Cham, 23–42.

[65] Nater, M. 2021. *Hyphenopoly.js*. Retrieved October 20, 2021 from https://github.com/mnater/Hyphenopoly

[66] Nattestad, T. and Al-Shamma, N. 2021. Photoshop's journey to the web. *web.dev*. Retrieved February 5, 2022 from https://web.dev/ps-on-the-web/

[67] Oliveira, F. and Mattos, J. 2020. Analysis of WebAssembly as a Strategy to Improve JavaScript Performance on IoT Environments. In *Anais Estendidos do Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC)*, SBC, 133–138.

[68] Reiser, M. and Bläser, L. 2017. Accelerate JavaScript applications by cross-compiling to WebAssembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (VMIL 2017), Association for Computing Machinery, New York, NY, USA, 10–17.

[69] Rescorla, E. 2021. What WebRTC means for you. Retrieved January 2, 2022 from https://blog.mozilla.org/en/mozilla/leadership/what-webrtc-means-for-you/

[70] Richards, G., Lebresne, S., Burg, B. and Vitek, J. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI '10), Association for Computing Machinery, New York, NY, USA, 1–12.

[71] Rijksmuseum. About us. Retrieved January 14, 2022 from https://www.rijksmuseum.nl/en/about-us

[72] Rijksmuseum. Masterpieces Up Close. Retrieved February 5, 2022 from https://www.rijksmuseum.nl/en/masterpieces-up-close

[73] Romano, A. and Wang, W. 2020. WASim: Understanding WebAssembly Applications through Classification. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1321–1325.

[74] Salim, S.S., Nisbet, A. and Luján, M. 2020. TruffleWasm: a WebAssembly interpreter on GraalVM. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (VEE '20), Association for Computing Machinery, New York, NY, USA, 88–100.

[75] Saunders, M., Lewis, P. and Thornhill, A. 2009. *Research Methods for Business Students*. Pearson Education.

[76] Selakovic, M. and Pradel, M. 2016. Performance issues and optimizations in JavaScript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering* (ICSE '16), Association for Computing Machinery, New York, NY, USA, 61–72.

[77] Serebryany, K., Stepanov, E., Shlyapnikov, A., Tsyrklevich, V. and Vyukov, D. 2018. Memory Tagging and how it improves C/C++ memory safety. *arXiv:1802.09517 [cs]* (February 2018).

[78] Serrano, M. 2018. JavaScript AOT compilation. *SIGPLAN Not.* 53, 8 (October 2018), 50–63.

[79] Smilkov, D., Thorat, N. and Yuan, A. 2020. Introducing the WebAssembly backend for TensorFlow.js. Retrieved February 3, 2022 from https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html

[80] Stake, R.E. 2013. *Multiple Case Study Analysis*. Guilford Press.

[81] Sukamolson, S. 2007. Fundamentals of quantitative research. (January 2007).

[82] Teare, D. 2019. 1Password X: May 2019 update. *1Password Blog*. Retrieved January 9, 2022 from https://blog.1password.com/1password-x-may-2019-update/

[83] TensorFlow. 2020. Platform and environment. Retrieved February 4, 2022 from https://www.tensorflow.org/js/guide/platform_environment

[84] TensorFlow. 2021. API Documentation. Retrieved February 4, 2022 from https://www.tensorflow.org/api_docs

[85] TensorFlow. 2021. TensorFlow Lite inference. Retrieved February 4, 2022 from https://www.tensorflow.org/lite/guide/inference

[86] TensorFlow. Home Page. Retrieved February 4, 2022 from https://www.tensorflow.org/

[87] TensorFlow. Why TensorFlow. Retrieved February 4, 2022 from https://www.tensorflow.org/about

[88] TensorFlow. Case Studies and Mentions. Retrieved February 4, 2022 from https://www.tensorflow.org/about/case-studies

[89] TensorFlow. Wasm backend. *GitHub*. Retrieved February 3, 2022 from https://github.com/tensorflow/tfjs

[90] TensorFlow. Blazeface detector. *GitHub*. Retrieved February 4, 2022 from https://github.com/tensorflow/tfjs-models

[91] TensorFlow. Imagenet classification with MobileNet V2. Retrieved February 3, 2022 from https://tfhub.dev/google/imagenet/mobilenet_v2_100_224/classification/5
[92] The AssemblyScript Project. Home Page. Retrieved October 23, 2021 from https://www.assemblyscript.org/
[93] Turner, A. 2021. *Made with WebAssembly*. Retrieved November 22, 2021 from https://github.com/torch2424/made-with-webassembly
[94] uBlock Origin. Home Page. Retrieved January 2, 2022 from https://ublockorigin.com/
[95] Wallace, E. 2015. Building a professional design tool on the web. *Figma*. Retrieved January 3, 2022 from https://www.figma.com/blog/building-a-professional-design-tool-on-the-web/
[96] Wallace, E. 2017. WebAssembly cut Figma's load time by 3x. *Figma*. Retrieved December 14, 2021 from https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/
[97] Wallace, E. Implicit caching of WebAssembly compilation artifacts. *chromium*. Retrieved January 4, 2022 from https://bugs.chromium.org/p/chromium/issues/detail?id=719172
[98] Wallace, E. WebAssembly throwing "memory access out of bounds" error in Chrome but not in Firefox. *chromium*. Retrieved January 4, 2022 from https://bugs.chromium.org/p/chromium/issues/detail?id=729768
[99] Watt, C. 2018. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (CPP 2018), Association for Computing Machinery, New York, NY, USA, 53–65.
[100] Watt, C., Rossberg, A. and Pichon-Pharabod, J. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* 3, OOPSLA (October 2019), 133:1-133:28.
[101] WebAssembly. Use Cases. Retrieved February 6, 2022 from https://webassembly.org/docs/use-cases/
[102] WebAssembly. Roadmap. Retrieved January 19, 2022 from https://webassembly.org/roadmap/
[103] WebAssembly. Security. Retrieved January 22, 2022 from https://webassembly.org/docs/security/
[104] WebAssembly. Home Page. Retrieved January 22, 2022 from https://webassembly.org/
[105] WebAssembly. FAQ. Retrieved February 5, 2022 from https://webassembly.org/docs/faq/
[106] Wen, E. and Weber, G. 2020. Wasmachine: Bring the Edge up to Speed with A WebAssembly OS. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 353–360.
[107] Wirfs-Brock, A. and Eich, B. 2020. JavaScript: the first 20 years. *Proc. ACM Program. Lang.* 4, HOPL (June 2020), 77:1-77:189.
[108] Yan, Y., Tu, T., Zhao, L., Zhou, Y. and Wang, W. 2021. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, Association for Computing Machinery, New York, NY, USA, 533–549.
[109] Ye, D., Su, Y., Sui, Y. and Xue, J. 2014. WPBOUND: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 88–99.
[110] Yin, R.K. 2009. *Case Study Research: Design and Methods*. SAGE.

[111] Yuan, A. and Dukhan, M. 2020. Supercharging the TensorFlow.js WebAssembly backend with SIMD and multi-threading. Retrieved February 3, 2022 from https://blog.tensorflow.org/2020/09/supercharging-tensorflowjs-webassembly.html

[112] Yue, C. and Wang, H. 2009. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th international conference on World wide web* (WWW '09), Association for Computing Machinery, New York, NY, USA, 961–970.

[113] Zakai, A. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (OOPSLA '11), Association for Computing Machinery, New York, NY, USA, 301–312.

[114] Zakai, A. asm.js. Retrieved October 23, 2021 from http://asmjs.org/

[115] Zheng, S., Wang, H., Wu, L., Huang, G. and Liu, X. 2021. VM Matters: A Comparison of WASM VMs and EVMs in the Performance of Blockchain Smart Contracts. *arXiv:2012.01032 [cs]* (January 2021).

[116] Zoom. Web SDK. Retrieved January 2, 2022 from https://marketplace.zoom.us/docs/sdk/native-sdks/web