Samuli Ylenius

Mitigating JavaScript's overhead with WebAssembly

Faculty of Information Technology and Communication Sciences
M. Sc. thesis
March 2020

ABSTRACT

Samuli Ylenius: Mitigating JavaScript's overhead with WebAssembly
M. Sc. thesis
Tampere University
Master's Degree Programme in Software Development
March 2020

The web and web development have evolved considerably during its short history. As a result, complex applications aren't limited to desktop applications anymore, but many of them have found themselves in the web. While JavaScript can meet the requirements of most web applications, its performance has been deemed to be inconsistent in applications that require top performance. There have been multiple attempts to bring native speed to the web, and the most recent promising one has been the open standard, WebAssembly.

In this thesis, the target was to examine WebAssembly, its design, features, background, relationship with JavaScript, and evaluate the current status of WebAssembly, and its future. Furthermore, to evaluate the overhead differences between JavaScript and WebAssembly, a Game of Life sample application was implemented in three splits, fully in JavaScript, mix of JavaScript and WebAssembly, and fully in WebAssembly. This allowed to not only compare the performance differences between JavaScript and WebAssembly but also evaluate the performance differences between different implementation splits.

Based on the results, WebAssembly came ahead of JavaScript especially in terms of pure execution times, although, similar benefits were gained from asm.js, a predecessor to WebAssembly. However, WebAssembly outperformed asm.js in size and load times. In addition, there was minimal additional benefit from doing a WebAssembly-only implementation, as just porting bottleneck functions from JavaScript to WebAssembly had similar performance benefits.

Keywords: performance, native, web application, JavaScript, WebAssembly

TIIVISTELMÄ

Samuli Ylenius: Mitigating JavaScript's overhead with WebAssembly
Pro gradu -tutkielma
Tampereen yliopisto
Ohjelmistokehityksen maisteriopinnot
Maaliskuu 2020

Web ja webkehitys ovat kehittyneet merkittävästi niiden lyhyen historiansa aikana. Sen seurauksena monimutkaiset sovellukset eivät enää ole rajoitettu työpöytäsovelluksiksi, vaan niitä löytyy myös webistä. Vaikka JavaScript pystyy täyttämään suurimman osan web-sovelluksien vaatimuksista, sen suorituskyky on koettu puutteelliseksi sovelluksissa, jotka vaativat huippusuorituskykyä. Vuosien aikana on ollut monta yritystä tuoda natiivisuorityskykyä webiin ja näistä uusin sekä lupaavin on avoin standardi, WebAssembly.

Tässä työssä tavoitteena on tutkia tarkemmin WebAssemblyä, sen suunnittelua, ominaisuuksia, taustaa, yhteyttä JavaScriptin kanssa, sekä WebAssemblyn nykyistä tilaa ja tulevaisuutta. Sen lisäksi, JavaScriptin ja WebAssemblyn suorituskykyä vertaillaan toteuttamalla Life-peli kolmella eri tavalla, ensimmäinen versio täysin JavaScriptillä, toinen JavaScriptillä ja WebAssemblyllä, ja viimeiseksi täysin WebAssemblyllä. Tämä lähestymistapa mahdollisti JavaScriptin ja WebAssemblyn suorituskykyerojen vertailun lisäksi myös eri toteutustapojen arvioinnin.

Toteutettujen sovellusten perusteella huomattiin, että WebAssemblyn suorituskyky suoritusaikojen perusteella oli merkittävästi JavaScriptiä nopeampi. Vaikkakin, WebAssemblyn edeltäjällä, asm.js:lla oli samankaltaiset suorituskykytulokset kuin WebAssemblyllä. Kuitenkin, WebAssemblyn tulokset tiedostojen koissa ja latausajoissa olivat asm.js:a paremmat. Tuloksista havaittiin myös, että täys-WebAssembly toteutustapa ei tuonut merkittäviä hyötyjä suorituskykyyn JavaScript+WebAssembly toteutukseen verrattuna, jossa pullonkaula JavaScript funktioita siirrettiin WebAssemblyn puolelle.

Avainsanat: suorituskyky, natiivi, web-sovellus, JavaScript, WebAssembly

Contents

1 Introduction

The (World Wide) Web and JavaScript have gone through drastic changes over the course of its short history. What started as a simple scripting language for form validation and manipulating the HTML DOM (Document Object Model) has risen to one of the most in-demand programming languages.

JavaScript usage is not limited simply to the web anymore, but it has spread to other domains as well. With the help of Node.js, it's even possible to use JavaScript for server-side applications, and popular frameworks such as Electron, allow developers to write desktop applications with web technologies, for example, Skype, Slack and Visual Studio Code are all built with Electron. In mobile application development, the use of web technologies can be a very attractive choice, especially when targeting multiple platforms. The two most popular platforms, Google's Android and Apple's iOS typically require two separate applications to be developed but web technologies allow applications to share some parts of the code. Popular choices include using WebView, hybrid applications, and frameworks, such as React Native.

While JavaScript has enjoyed being the longest supported programming language by all major browsers, it hasn't been the only solution during its reign. Using other languages in the web has been proposed multiple times and it has been usually achieved by browser plug-ins. Plug-in-based solutions were very popular especially in the early 2000s, notably Adobe Flash, Java Applet and Microsoft Silverlight. While they did bring performance benefits of native code into the web, their reputation was tainted by the constant security problems. With the growing distrust towards plug-ins, the introduction of HTML5, the rapid growth of the mobile platforms, and big mobile platforms, such as Apple's iOS, dropping support for the plug-ins, their usage took a huge dive. Most of the major plug-ins are already deprecated or will be in the coming years. Due to these reasons, JavaScript has remained the number one programming language in the web.

The speed that the web has evolved has surpassed even the most optimistic predictions. Over the years, more and more applications have moved to the web from desktop applications. As such, JavaScript has even become a popular target language for other languages, however, as JavaScript was never designed for such purpose, as doing so usually has its own set of problems especially performance-wise. Implementing applications that require high execution power to the web, such as image/video editing and games, really puts the capabilities and limitations of JavaScript to the test. While JavaScript gained significant performance improvements with the introduction of JavaScript just-in-time compiler in browsers and other browser engine optimizations, its performance, especially in complex applications, has remained inconsistent. There have been multiple previous attempts to gain near-native speed in web applications, including Google Native Client and asm.js, with their own advantages and disadvantages. Newest promising technology in this field that solves the shortcomings of its predecessors, while performing at near-native speed, is WebAssembly. Notably, what makes it especially promising and unique, is that it's being designed and developed collaboratively and supported by all major browser vendors and it's the first open standard for running native code in a web browser.

The topic of this thesis is to take a deeper look into WebAssembly, its design, features, relationship with JavaScript, evaluate the current status of WebAssembly and its future. In addition, via sample application, different ways to implement WebAssembly into a web application are explored and evaluated.

Chapter 2 will cover related work regarding WebAssembly's performance and describe some of the predecessor technologies that influenced and essentially lead to WebAssembly design and development. In Chapter 3, WebAssembly will be covered in detail, including its design, key features, security, current status, and future. In addition, it will cover what WebAssembly brings over JavaScript, especially from a performance perspective. Chapter 4 will cover the implementation of Game of Life application from three different implementation approaches, full JavaScript version, a mix of JavaScript and WebAssembly and a full WebAssembly version. Chapter 5 will cover the results and Chapter 6 will wrap up with conclusions.

2 Background

The web has been an appealing platform for applications, especially during the last 15 years. Although many of the common features websites these days have, were originally only possible through plug-ins such as Adobe Flash and Java Applet. HTML5 allowed the web to natively handle most of the features that plug-ins brought, especially regarding multimedia. WebGL (Web Graphics Library) and just-in-time compiled JavaScript narrowed down the performance differences that plug-in-based solutions did bring and as such, many plug-in-based solutions weren't as needed anymore. While JavaScript can meet the requirements of most web applications, its performance has remained inconsistent in applications that require top-end performance, especially compared to native applications. Achieving native speed in the web has been a focus during the last decade and WebAssembly is not the first attempt to attain native performance in the web and diversify web development. One major example of these includes Google Native Client (NaCl), which was eventually deprecated in favour of WebAssembly. However, such previous attempts and other related technologies, including asm.js and Emscripten, played a big part in WebAssembly's development and design, as WebAssembly essentially continued on the path that NaCl, Emscripten, and asm.js had set on.

2.1 Emscripten

Emscripten is a source-to-source compiler that compiles languages such as C/C++ into JavaScript, or more specifically into a subset of JavaScript, known as asm.js. The idea and motivation behind such a compiler are not completely new, as it's a very tempting idea to convert an application to the web without having to rewrite the entire program, as doing so will require a considerable amount of time and money. However, especially when working with complex programs such as games, simply converting them into JavaScript can lead to multiple problems, as JavaScript is not, or at least was not, fit to handle such a large amount of lines of code. In addition, many of the APIs (Application Programming Interface), such as audio work very differently in the web than on desktop [Wagner, 2017]. But these are all problems that Emscripten also had to tackle during its design and development.

Compiling high-level languages into JavaScript has been done before, with tools such as Google's Web Toolkit, which compiles Java into JavaScript, and Pyjamas, which compiles Python into JavaScript. However, usually in these types of conversion processes, due to near 1 to 1 compile manner, there are often limitations. For example, in Pyjamas, integer division in Python would produce an integer, but in JavaScript, it might turn into a floating-point. Emscripten takes another approach in this, as it doesn't compile high-level language into JavaScript, but first into an LLVM (Low-Level Virtual Machine) assembly and then into JavaScript. This potentially allows Emscripten to be used for multiple languages, unlike many other compilers that only allow one language due to very direct translation. However, there are challenges in compiling from LLVM into high-level language, since high-performance JavaScript requires using natural code flow structures, such as loops and if statements, but they don't exist in LLVM. Therefore, Emscripten must construct accurate high-level representation from low-level data without sacrificing performance. This is achieved by a loop recovery algorithm, called Relooper, which aims to generate native JavaScript control flow structures rather than recreating the original source code. [Zakai, 2011]

Emscripten saw its first light at Mozilla when Alan Zakai started working on Emscripten as a side project for a few years. It was demonstrated by running a C++ written game in a browser with all functioning components, including graphics, audio and multiplayer interactions. The game performed well, even faster than a game manually written in JavaScript due to preserving the strictness of C++ when converting it to JavaScript. With such a successful demo, Zakai wanted to take Emscripten to the next level by testing how well would it perform with commercial software. Developers from Unity game engine were invited to work with them to attempt to get Unity game engine running in the web. Within a week, they were able to do so but it was riddled with performance problems, with slow start-up and the game stuttering. This was due to browser engine having difficulties keeping up parsing, analysing and optimizing the converted JavaScript. [Wagner, 2017]

Luke Wagner, who works on the performance of the JavaScript engine in Firefox, began to study the demo's performance problems and discovered a solution to them. By limiting the pattern of input, it's possible to get reliable performance out of the engine. Therefore, Zakai began modifying the Emscripten to output only those patterns that performed well, and Wagner optimized the JavaScript engine to work even faster with those limited patterns. This limited subset of patterns eventually came to be called asm.js. After months of development, it was ready for the next demo. This time it was time to get Epic's Unreal Engine running in Firefox. With the optimizations of asm.js and modifications of Emscripten, the demo was very successful this time, with smooth animations instead of stuttering. [Wagner, 2017]

2.2 asm.js

asm.js is a subset of JavaScript which is optimized for performance. It's not intended to be written directly, but instead; it's used as a target language for compilers. Common usage has been to use languages like C/C++ to convert libraries and frameworks to the web using asm.js and a source-to-source compiler like Emscripten. Notable examples of this have been the Qt application framework and popular game engines, Unity and Unreal Engine [Wagner, 2017].

One of the main performance boosts asm.js provides over standard JavaScript, is that validated asm.js enables ahead-of-time (AOT) compilation. This is achieved by limiting JavaScript features into a subset that can be ahead-of-time optimized. Advantages of the AOT compiler-generated code is that it doesn't have runtime type checks and garbage collection, however, code that doesn't pass validation must fall back to standard methods, such as regular interpretation and just-in-time (JIT) compilation [Herman et al., 2014].

```c
int f(int i) {
  return i + 1;
}
```

Listing 1. C function.

```javascript
function f(i) {
  i = i|0;
  return (i + 1)|0;
}
```

Listing 2. C function generated to asm.js with Emscripten.

asm.js model is based on two main points, integer and floating-point arithmetic, and a virtual heap [Herman et al., 2014]. Listing 1 shows a function written in C, that takes in an integer, adds a value of 1 to it and returns the result. Listing 2 shows the same function when it's generated into asm.js via Emscripten. There are few notable differences between these two code snippets. Generated asm.js doesn't have types specified as JavaScript is not statically typed. Instead, to ensure that parameter *i* is an integer, bitwise OR operator is used to coerce the value back to an integer. This bitwise coercion also ensures that undefined values, for example from out of bounds accesses, are coerced back to an integer, as the result of undefined|0 is 0. These types of conversions ensure that even when a function is called by external JavaScript code, the arguments are coerced to expected types. This allows the compiler to produce performant ahead-of-time code, knowing that the function body never needs to run runtime type checks [Herman et al., 2014].

asm.js was first introduced in 2013 and naturally, Mozilla Firefox was the first browser to implement asm.js specific optimizations, as it was developed within Mozilla. Mozilla published the full specifications of asm.js and began encouraging other browser vendors to implement asm.js specific optimizations, and within few years, other browser engines had included asm.js specific performance changes [Wagner, 2017]. It's currently supported by most major browsers, including Edge, Chrome, and Firefox, but not Safari, however, Chrome's V8 JavaScript engine doesn't support AOT compilation, but still gains performance benefits of asm.js compared to standard JavaScript [Can I use asm.js, 2019].

Since major browsers began to support and gain the performance improvements of asm.js, the number of games converted to asm.js by Emscripten began to grow, notable examples include Facebook games, such as Candy Crush Saga and Top Eleven. Although games are a popular usage for asm.js, it's not the only one. Facebook began to use asm.js to compress user images in the front end to save bandwidth and Adobe used asm.js for their web version of image manipulation software Lightroom. [Wagner, 2017]

While it's possible for asm.js to perform near-native speed, according to Zakai [2017] one of its problems has been consistency due to lack of a standard. Different browsers approached asm.js optimizations from different directions as it was designed by one vendor, Mozilla, unlike WebAssembly, that's design involved members from all major browsers. In addition, since asm.js is a subset of JavaScript, it's limited to features that JavaScript has. This includes important features like 64-bit integer operations and threads. Another issue is, especially when dealing with large codebases is that the asm.js compile-phase would take a long time on cold start-ups. On mobile, this could take up to 20 seconds, which can be detrimental to user experience. Eventually, these limitations, lead to the creation of a new web standard, WebAssembly, that would be even more efficient to load and allowed to solve the limitations that asm.js would be tied to.

2.3 Google Native Client

Google Native Client is an open-source project by Google, that allows safely running native code inside the browser by using sandboxing technology. Unlike web technologies, NaCl can perform near-native speed as it allows to take full use of the CPU capabilities, including multicore processing. NaCl allows developers to write languages like C++ to access system services such as graphics and sound hardware while gaining the security properties of the web, as well as having access to the same services that JavaScript has [Donovan et al., 2010].

Extensions running native code have a bad reputation safety-wise as browser extension architecture allows plug-ins, such as ActiveX, to bypass the typical security mechanisms that are used for website content. These types of applications that are running native code, must rely upon non-technical measures for security, such as manual installation or pop-up dialogs. However, such measures have been proven unreliable and have led to installing malicious code. [Yee et al., 2009]

Knowing the security caveats of such extensions, Yee et al. [2009] emphasises that one of the main design goals of NaCl from the beginning was safety while gaining the performance of native code. This was done by splitting code execution into two parts, constrained environment for native code without unwanted side effects and a runtime hosting native code extensions where allowed side effects may occur and be controlled safely. Unsafe instructions such as system calls are prevented by a code verifier. Although, these constants require that C++ code must be recompiled to work on NaCl, which provides customized toolchain for compiling. As with many other Google products, Google offered notable rewards ($5000-15000) for all bugs revealing NaCl vulnerabilities, notably sandbox escaping.

While NaCl has been platform-independent from the beginning, one of the downsides is that it requires generating different executables (.nexe) for each architecture, such as x86, ARM, and MIPS. This limitation essentially made it non-portable, thus making it not well suited for the web, as it threatens the portability of the web. Therefore, NaCl has been limited to applications installed via the Google Web Store. To improve portability and suitability for the web, Portable Native Client (PNaCl) was developed. PNaCl only generates a single executable (.pexe) that translates it into host-specific executable during module load and before code execution. [Google Native Client, 2019]

Due to portability, PNaCl greatly reduces the effort to support multiple architectures, since with NaCl, simply having access to multiple different architectures in addition to testing and supporting all of them requires significant effort [Donovan et al., 2010]. This model would only lead to fragment the web, especially if NaCl added an additional supported architecture or if developers only focused on supporting the most popular architectures. Due to these reasons, in most use-cases, PNaCl was recommended over NaCl.

Overall NaCl received mixed responses. Some praising its security, performance, and an alternative to JavaScript written applications. While others, notably, higher-ups from Opera and Mozilla, criticised its interoperability issues and that it's bringing an old platform to the web and shifting focus away from the web platform [Metz, 2011]. Google deprecated (P)NaCl in 2017 in favour of WebAssembly and even provides a WebAssembly migration guide on NaCl documentation. The reasoning behind the deprecation and switching focus on WebAssembly was low usage of PNaCl and a much more vibrant ecosystem around WebAssembly [Nelson, 2017].

2.4 Related work

WebAssembly is still a relatively new technology, therefore, works related to WebAssembly's performance overhead aren't that well covered yet. Currently, most of the performance evaluations done regarding WebAssembly's performance are versus native code, especially focusing on the execution times, while this work focuses on the overall performance overhead differences between WebAssembly and JavaScript.

In the paper that introduced WebAssembly, Haas et al. [2017] conducted measurements that compared the execution times of WebAssembly and native code using PolyBenchC benchmark suite in Google Chrome's V8 engine and Firefoxes SpiderMonkey. This benchmark suite consists of 30 various numerical computations, including image processing and physics simulation [Reisinger, 2016]. Haas et al. [2017] results showed that while there were differences in start-up times between V8 and SpiderMonkey, overall, WebAssembly's performance was near-native speed, as seven benchmarks were within 10% of native code's performance and most of them within 2x of native speed. In addition, the benchmark suite was run on asm.js to compare its performance to WebAssembly. This showed that on average, WebAssembly outperformed asm.js by 33.7%.

In response to measurements conducted by Haas et al. [2017], the results were criticised by Jangda et al. [2019], since the benchmark suite used in the evaluation was limited to benchmarks consisting of about 100 lines of codes each. Therefore, Jangda et al. [2019] performed a more large-scale evaluation to analyse the performance differences between WebAssembly and native code by running WebAssembly-compiled Unix applications in the browser using SPEC CPU benchmarking suite. Running this benchmarking suite, consisting of benchmarks for scientific applications, image and video processing, showed that WebAssembly is performing slower than native code in Mozilla Firefox by an average of 45% and in Google Chrome by 55%. While some of the performance differences are caused by the design constraints of WebAssembly, Jangda et al. [2019] do acknowledge that not all the performance issues in the results are significant, as they can be mitigated in future WebAssembly implementations. Results regarding asm.js and WebAssembly performance are similar to Haas et al. [2017], as both results showed that WebAssembly outperforms asm.js roughly by 30%.

3 WebAssembly

WebAssembly is a standard for portable low-level bytecode, that's designed to serve as a compilation target for languages like C/C++, enabling running applications written in native code to be run in the browser with near-native speed. As its main purpose is to be a compilation target, it's not intended to be written by hand, other than small snippets.

WebAssembly was first publicly announced in June 2015 and it's the first W3C (World Wide Web Consortium) based open standard for running native code in the web. As a collaborative effort, WebAssembly Community Group consisting of members from all major browser vendors, core features for Minimum Viable Product (MVP) were designed and implemented. After the Browser Preview period, these core features were all supported and enabled by default in all major browsers in March 2017. [WebAssembly, 2019]

3.1 Design

From the beginning, one of the main goals of WebAssembly's design has been high-performance without sacrificing safety or portability. As such, WebAssembly's design focused on delivering all the properties that low-level code format should have, but previous attempts have lacked. These properties and goals are safe, fast, universal, portable and compact. Safety is naturally an important goal especially in the web, where code is loaded from untrusted sources. Protection from such code has been usually achieved running code inside a virtual machine (VM), like JavaScript Virtual Machine or a language plug-in. Such techniques prevent programs from compromising user data and system state, but they come with a performance overhead, especially on low-level code. Also, since such runtimes are usually designed for specific programming language or paradigm, using other ways usually greatly reduces performance, thus falling short on universal property. Many previous attempts in portability have fallen short on the requirements that code targeting the web should have, as they have designed to work on a single architecture, like Google Native Client, or have suffered from other portability problems. WebAssembly has been designed to work on multiple machine architectures, operating systems, and browsers. [Haas et al., 2017]

WebAssembly has been successful on all these design properties, which Haas et al. [2017] credits on unprecedented collaboration from all major browsers and the community. Portability-wise, despite its name, WebAssembly has been designed to work on platforms outside the web, including mobile and Internet of Things (IoT), as such, WebAssembly has been designed to work with and without the JavaScript Virtual Machine. Non-web platforms can provide different APIs than the web and those can be discovered and used via feature testing and dynamic linking [WebAssembly, 2019].

As WebAssembly's goal has never been to completely replace JavaScript, but rather to work alongside it, WebAssembly's design has focused on integrating seamlessly with the existing web platform. This includes maintaining the versionless and backwards-compatible nature of the web. In JavaScript, this type of feature testing is usually handled with polyfills, but since WebAssembly is ahead-of-time validated, it requires a different approach. Currently suggested strategies include compiling several versions of modules, each providing different feature support [WebAssembly, 2019]. Integrating with the web also includes supporting calls to and from JavaScript, accessing the same web APIs that JavaScript has access to and supporting the View Source functionality.

3.2 Key concepts

Since WebAssembly is a compilation target rather than a new programming language, getting started differs from ways developers would traditionally approach learning a new programming language or a framework. Especially when WebAssembly and the technologies surrounding it will continue to advance, developers can compile into WebAssembly using their high-level language-of-choice without knowing the inner workings of WebAssembly. However, there are some key concepts, that will help better understand how WebAssembly runs in the browser, assist debugging and even further optimize WebAssembly's performance.

WebAssembly *Module* represents WebAssembly's binary format that has been compiled into an executable and a loadable unit. It contains a function, table, global, and memory definitions. The module itself is stateless, shareable with Web Workers and can be instantiated multiple times. When the module is paired with the state it uses at runtime, it's called an *instance*. [MDN, 2019]

```c
int add(int num1, int num2
    ) {
  return num1 + num2;
}
```

Listing 3. Function adding two integers in C.

```
(module
 (export "add" (func $add))
 (func $add (param $0 i32) (
     param $1 i32) (result i32
     )
  get_local $0
  get_local $1
  i32.add
 )
)
```

Listing 4. Function adding two integers in WebAssembly's text format.

Listing 3 shows a simple function that takes two integers as parameters, adds them and then returns the result. Listing 4 then shows the same function when compiled with Emscripten into WebAssembly's text format. WebAssembly's text format is meant to be a human-readable format of WebAssembly's binary format, which is designed to be exposed in text editors, developer tools, or even be written by hand [MDN, 2019]. The module is represented in S-expressions, which are a notation for tree-structure data, which are notably used in the Lisp programming language. The interesting part of the Listing 4 code is the part inside the func. The function has a following structure (func <signature> <locals> <body>), where signature expresses what parameters it takes and returns. Locals are similar to JavaScript var, but with declared types. The body then represents the instructions, as WebAssembly execution is defined in terms of a stack machine, where every instruction either pushes or pops a certain number to or from the stack [MDN, 2019]. In Listing 4 body, local.get $0 and local.get $1 pushes a local read value into the stack then i32.add pops two i32 values from the stack, calculates their sum and pushes the value into stack.

```
fetch('add.wasm').then(response =>
    response.arrayBuffer()
).then(bytes =>
    WebAssembly.instantiate(bytes)
).then(obj => {
    console.log(obj.instance.exports.add(3, 4));  // "7"
});
```

Listing 5. Fetching and calling WebAssembly function inside a browser.

In order to actually call the code in Listing 4, the function must be exported as stated with (export "add" (func $add)) line. WebAssembly's text format is represented in .wat files, but the binary format which the following must be converted into to be able to use it in a browser is represented in .wasm file format. The code snippet in Listing 5 shows one way to load the wasm and call the function. Currently, the wasm needs to fetched using Fetch API or similar ways of fetching network resources, which are susceptible to CORS (Cross-origin resource sharing) errors, but in the future, WebAssembly modules will be loadable using the <script> tags and ES2015 import statements [Clark, 2017a]. In Listing 5 code, after fetching the wasm, the response is converted from bytecode into ArrayBuffer and then the module is instantiated and finally it can be called and the result (7) is printed into the console. The instantiate function also takes in a second parameter, importObject, that can hold values, function closures, memory or tables [MDN, 2019].

WebAssembly uses a large array of raw bytes as its main storage, which is more commonly referred to as linear memory. Each module can only define one memory and the memory can be shared with other instances [Haas et al., 2017]. At creation, the amount of memory allocated is given, with the lowest amount being the size of a page, that in WebAssembly has a constant size of 64 KiB (Kibibyte). But since the memory is dynamic, the memory can be expanded via memory.grow method, which takes in the number of pages as a parameter. Linear memories real addresses are not available directly to WebAssembly, but the memory is accessed via instructions by requesting data at an index of memory and the browser will handle figuring out the real address. Out-of-bounds accesses result in a *trap*, which immediately aborts the current computation [Haas et al., 2017]. Since the memory handling is basically a JavaScript object, it makes it easy to pass values between WebAssembly and JavaScript, and objects can be garbage collected when they go out of scope.

Due to WebAssembly not having direct access to the contents of memory, it's one of the reasons why WebAssembly is safer than pure native code. Because at worst, malicious WebAssembly program can only corrupt its own memory, which allows to safely execute even untrusted modules in the same memory address space [Haas et al., 2017].

In order to have something similar to C/C++ function pointers, while keeping memory addresses hidden, WebAssembly *table* was added. The table is an array that is outside of WebAssembly's memory, which values contain references to functions. Values are accessed similarly as the linear memory array, with indexes, as the actual addresses are hidden. Modules can call these functions by using call_indirect function that takes in the index as a parameter. Although, currently table is only for storing function references, however, in the future, there are plans to expand its capabilities. [Clark, 2017b]

3.3 Security

When dealing with native code in the web, security should be of main priorities since code in the web originates from unknown sources and especially considering the history of security vulnerabilities in its past. Plug-in based solutions, especially Adobe Flash was infamous of its many critical vulnerabilities [Wressnegger et al., 2016]. In addition, since native code typically has access to machine's underlying components, it raises additional safety concerns especially regards to memory. In browsers, JavaScript runs in a sandboxed environment, thus isolating browsers from the rest of the system. This allows not exposing any dangerous API's and limiting the possible damage in the case of an exploit.

Security in WebAssembly is designed with two major goals in mind, to protect the users from malicious and buggy modules and to provide developers with a means to develop safe applications. Similarly to JavaScript, WebAssembly modules run in a sandboxed environment separated from its host runtime. In addition, each module is constrained by the security policies of its embedder. In the development side, WebAssembly limits dangerous features from execution semantics. To enforce control-flow integrity (CFI), modules must declare all callable functions and their types at load time. CFI is a technique used to prevent attacks that redirect the flow of execution of a program. Since compiled code is immutable and non-observable at runtime, WebAssembly programs are protected against control-flow hijacking attacks. [WebAssembly, 2019]

While WebAssembly runs in a sandbox and thus isolating the impact of exploits, however, this doesn't prevent memory safety bugs from compromising WebAssembly code or the data it uses. While common mitigations such as data execution prevention and stack smashing protection are not necessary for WebAssembly programs due control-flow integrity and protected call stacks, classic memory safety vulnerabilities still exist when compiling from C/C++, namely buffer overflows and use-after-free. While new processor features, such as tagged memory and pointer authentication, can assist in detecting, mitigating and preventing such vulnerabilities with minimal overhead, WebAssembly's just-in-time compiler cannot take advantage of these features since critical high information for this is lost when compiling into WebAssembly. [Disselkoen et al., 2019]

Overall, WebAssembly is designed to be safe with its sandboxed environment and being unable to do anything it doesn't have permission to do. However, it doesn't mean that using unsafe languages will be safe in WebAssembly, which itself technically was never a design goal of WebAssembly and some might argue, that it should not be. Then again, even JavaScript, that's designed to be safe, vulnerabilities are found from time to time. Sandboxing mitigates some of the damage caused by the security bugs, although even sandbox escape vulnerabilities are sometimes discovered. In the end, WebAssembly's security is good, but as with all new technologies, there are always new potential security issues that need to be considered.

3.4 WebAssembly and JavaScript

One of the main advantages that WebAssembly brings over JavaScript is performance. But performance is not always such a clear term, especially when modern JavaScript's performance is already rather fast with the addition of just-in-time compiler in 2008 and many other browser engine optimizations over the years. In order to properly understand and evaluate how WebAssembly's performance differs from JavaScript, the following sections will cover the basic functions of JavaScript's just-in-time compiler and how it managed to give many-fold performance increase to JavaScript execution. In addition, it will cover where highly optimized modern JavaScript even with the JIT compiler might lack in performance and how WebAssembly can cover those caveats.

3.4.1 JavaScript just-in-time compiler

JavaScript is an interpretive language, meaning rather than compiling the code beforehand into machine language, the interpreter will execute the code line-by-line on the fly. This allows for a faster development cycle, as it's much faster and easier to make changes to the code without having to compile the entire program. While this makes a dynamically typed language, like JavaScript, convenient to write, such features will typically take a toll on the performance of the program compared to compiled code. It's especially notable when running the same code, such as a loop, as the interpreter needs to translate each loop all over again.

In order to solve some of the interpreter's limitations and to increase performance, browsers started implementing just-in-time compilers. Although, JIT itself was not a new concept, as it was first seen in 1960 in LISP and heavily used in Java but implementing JIT to browsers and JavaScript hasn't been done before. First one to do so in browsers was Google with their Google Chrome V8 JavaScript engine. This was motivated by the rise of heavy JavaScript applications in the web, such as Google Maps, which highlighted the performance problems of JavaScript. This led to other browsers to support JIT compiling as well, and browser's JIT compilers to developed and improved faster, as competition usually does.

JIT is basically a combination of an interpreter and a compiler. While browsers have slightly different implementations of the JIT compiler, the basic idea stays the same. Browsers started tracking how many times the code is run and what types are used by using a monitor or a profiler. In the beginning, the monitor runs everything through the interpreter and the code is marked cold. If the same code is run multiple times, it will be marked warm and finally, if it's run often, then it will be marked hot. [Clark, 2017a]

Warm functions will be compiled by the baseline compiler and stored. Each line of code inside the function will be compiled as a stub, which is then indexed by the line number and variable type. This will allow the monitor to detect if the code is used again with the same variable types, allowing it to just return the compiled version, which can improve the performance by quite a bit. [Clark, 2017a]

When the monitor detects that part of code is called often, it's marked hot and can be even further optimized, but in order to do so, the compiler must make some assumptions. These assumptions are based on the data that the monitor has gathered about the code usage. Previously mentioned loops, which especially suffer from interpreter's nature, can be heavily optimized. If in a loop something has been true for all previous loop iterations, it can be assumed to continue to be true. However, if the assumption turned out to be false, for example when variables coming into the loop differ from previous iterations, JIT will abandon the assumptions and the optimized code and bail out to either interpreter or the baseline compiled version that was created from warm code. [Clark, 2017a]

```
businessLogic = (value) => {
    const x = value * 3 * 5;
    const y = value * 1 * 2;
    return x % y;
}
performanceTest = (index) => {
    let sum = 0;
    if (index % 25 === 0) {
        sum = 1.1;
    }
    for (let i = 1; i <= 10000; i++) {
        sum += businessLogic(i);
    }
}
logPerformance = () => {
    for (let i = 1; i <= 100; i++) {
        const start = performance.now();
        performanceTest(i);
        const timeDuration = performance.now() - start;
        console.log(timeDuration);
    }
}
logPerformance();
```

Listing 6. JavaScript JIT optimization example.

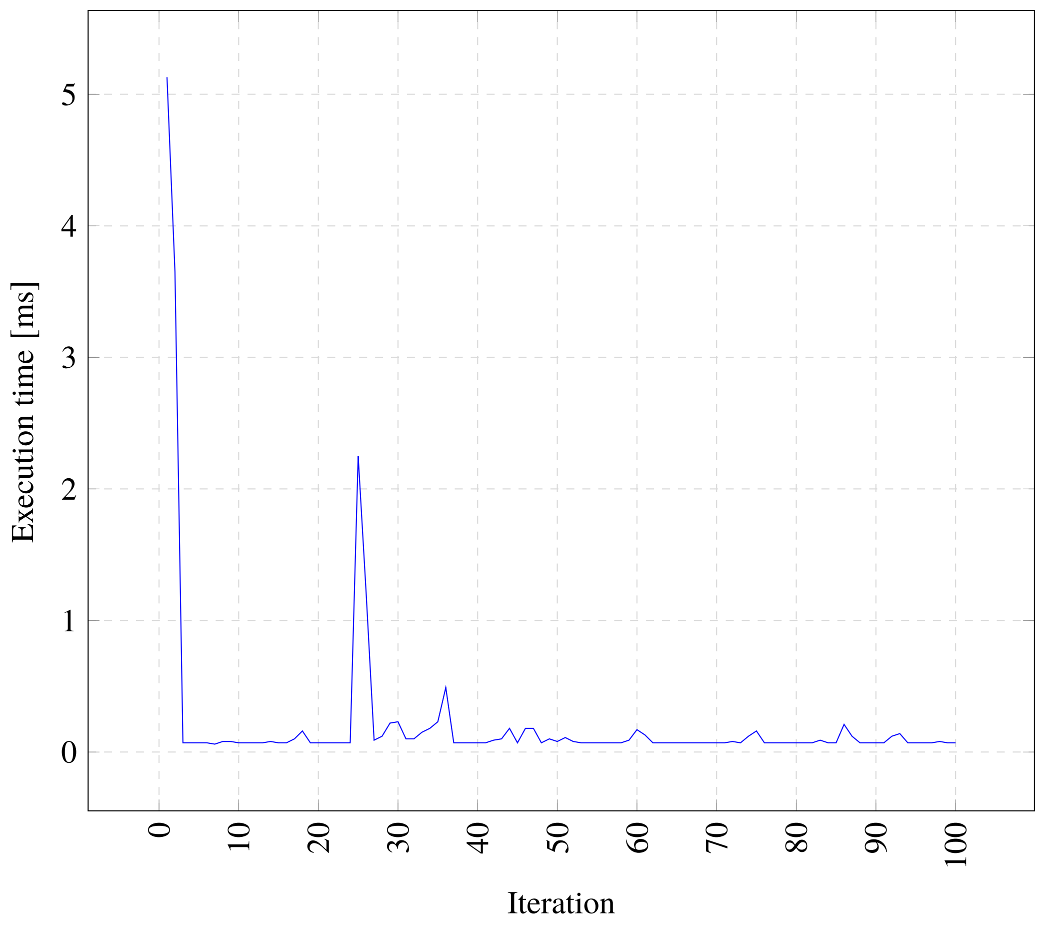

Figure 1. JIT optimization code execution measurements.

The code example in Listing 6 shows one of the ways how JIT compiler can perform type optimizations on JavaScript code. The main loop in logPerformance tracks measurements how long does it take to perform the performanceTest function. In performanceTest function, businessLogic function is called 10000 times. Function businessLogic simulates business logic by performing various calculations on function parameters. Figure 1 shows how many times each for loop in logPerformance function took to execute. As can be expected, the first and second loop took the longest time to execute, with 5.13 ms on the first loop and 3.65 ms on the second loop. Starting from the third loop, execution time dropped to 0.07 ms with only minor differences in the consecutive loops. However, the 25th loop took 2.25 ms to execute, since performanceTest function had a modulo operation check, so every 25th loop, sum variable value would change to 1.1 floating number instead of the integer value of 0. This causes the JIT optimizer to drop the assumptions it has made for the types of the function, causing de-optimization to occur. The 26th loop has an execution time of 1.12 ms and all loops after that had roughly the same execution time. Even the 50th and 75th loops performed well since the compiler had already made optimizations for floating types in the 25th loop.

JIT compiler has given JavaScript a huge performance boost, especially since the optimizations have evolved and improved over the years. However, this performance is not always consistent or predictable and it has added some overhead. Since the JIT compiler has to keep track and store the code execution statistics, recovery information when a bailout is needed due to wrong assumptions, and store both baseline and optimized versions of a function, it has increased the memory usage of the browsers. One cause of unpredictable performance is when part of the code is being optimized and de-optimized multiple times causing slower performance than just executing the baseline version. Although, browsers contain limits to how many times this type of cycle can happen. [Clark, 2017a]

3.4.2 WebAssembly compiler

WebAssembly differs from typical assembly due to the nature of the web. Since browsers run on multiple different types of machine architecture, there isn't a specific target architecture that the code is targeted for. Typically programming languages, like C or C++, don't get compiled directly into assembly languages as doing so would require multiple different language-to-assembly translators but instead, compilers have at least one layer between the translation process. This layer is called *intermediate representation* (IR). Compiler's front end analyses and translates the source code into an intermediate representation. The middle end, also sometimes referred to as the optimizer, performs performance and quality optimizations on the IR and then finally the compilers back end is responsible for architecture-specific optimizations and translating the code to target's CPU architecture, which is typically either x86 or ARM. However, in WebAssembly's case, instead of going from IR to assembly, it will go from IR to WebAssembly and then when the browser downloads WebAssembly, it will be translated into target machines assembly code. [Clark, 2017a]

3.4.3 Performance

Writing JavaScript that has high-performance is possible. However, as much of the modern JavaScript performance lies in the JIT compiler, writing highly performant code requires knowledge and expertise about the ins and outs of JIT compiler and the JIT optimizations the browser makes. But not many developers know how to specifically optimize their code for JIT compiler and even doing so isn't always so straightforward. Writing JIT optimized code can negatively affect the readability of the code, as many of the common code patterns used by developers can hinder the compilers ability to optimize the code, such as combining common tasks into a function that can take in different types [Clark, 2017a]. As mentioned before, browsers have different types of JIT optimizations, therefore, while optimizations can give a performance increase in one browser, they might reduce the performance in others. This is one of the major reasons why WebAssembly typically performs faster than JavaScript since those types of optimizations aren't necessary for WebAssembly. As WebAssembly is statically typed and compiled ahead-of-time, the compiler doesn't need to infer and monitor the code for optimizations. But instead, optimizations can be made ahead-of-time and the compiler doesn't need to make multiple different optimizations of the same code, unlike JIT, that's working with inferred types and under a time constraint due to optimizing during runtime. This greatly reduces the overhead that JIT optimizing and de-optimizing causes. Due to working with inferred types, especially applications dealing with true integer types, such a graphics software, image manipulation, and audio data applications, it can be especially difficult to reach even close to native performance. In addition, as there is no need to keep track of code execution statistics, WebAssembly typically requires less memory [Clark, 2017a].

When dealing with web applications, simply executing the code is not the only step. The code needs to be downloaded from the site first and while it's not a big concern with high-speed connections, it can take a quite long time with slow bandwidth to be able to use the page on the first load. Minified and compressed JavaScript can already be quite small in size, especially when using techniques like tree shaking (dead code elimination), but WebAssembly can even improve on that with its compact binary format. This binary encoding has been designed for both small size and decoding time, as it allows for streaming compilation [Haas et al., 2017]. Streaming compilation allows the WebAssembly to be compiled while it's downloading since it doesn't come as one file but rather in a series of packets. This allows the compilation to happen even faster than its downloading [Clark, 2018].

Once JavaScript is finished downloading, browsers parse it into Abstract Syntax Tree (AST), which represents the syntactic structure of JavaScript code. As it will directly delay how soon the user can interact with the site, browsers typically do this lazily, only parsing functions that are called first and just creating stubs for the rest of the functions [Clark, 2017a]. After parsing, AST can be converted into an intermediate representation, which in JavaScript's case, is bytecode. However, as explained in Subsection 3.4.2. WebAssembly compiler, WebAssembly is already in an intermediate representation stage when it's downloaded to the browser, therefore, it doesn't need to be parsed but rather decoded which is much simpler and faster than parsing, especially since it can be split across multiple threads [Clark, 2018]. This can, depending on the size and complexity of the program, greatly reduce the start-up time of the web application, especially on cold start-up.

JavaScript parsing task can also be split into threads outside of the main thread; however, the compiling happens on the main thread and it requires that the parsing has been completed. But in WebAssembly, Clark [2018] states that the decoding and baseline compiling can be parallelized, therefore, the main thread can focus on executing the code, rather than pausing to compile. After baseline compiled is served to the main thread, the thread can start producing even more optimized code and then swap it to the main thread. With eight compilation threads, this has brought 5-6x improvement in compilation speed to Google's V8 JavaScript engine and Mozilla's SpiderMonkey JavaScript engine [Haas et al., 2017].

As JavaScript doesn't have manual memory management, clearing out old variables are handled by the garbage collector. While modern garbage collectors are quite advanced and good at scheduling the garbage collection, it is still an element that adds to the inconsistent performance of JavaScript. WebAssembly currently doesn't support garbage collection, therefore memory must be manually managed [Clark, 2017a]. While manual memory management can make the performance more consistent, it requires additional expertise from the developer to fully gain the advantages over a garbage collector. Support to provide access to the browser's garbage collector is an important goal for WebAssembly and it's being actively designed and developed [Clark, 2017a].

3.4.4 Future

As WebAssembly is designed with performance as one of the main goals, it's natural that at least on paper, it seems to outperform and improve on every section of performance compared to JavaScript. However, it's not meant to replace JavaScript, but rather to work alongside with JavaScript. Rather than it being a choice of either JavaScript or WebAssembly, many developers may choose to use WebAssembly to rewrite parts of their current JavaScript code for better performance. Although rewriting parts as WebAssembly doesn't automatically bring performance increases but focusing on swapping compute-heavy functions from JavaScript to WebAssembly can bring notable performance increases. But then again, performance isn't the only measurement when choosing a programming language or a technology stack, especially when some applications might not gain much from a slight performance boost if there isn't any heavy calculation in the application. There are some features that are on the feature specification but not yet implemented in WebAssembly, such a DOM manipulation and web API calls aren't currently even possible without going through JavaScript first, which depending on the number of calls, can actually notably worsen the performance [Clark, 2017a]. Eventually, as usual, the choice of technology will come down to the requirements of the application. If the application already has native code implementation or there are useful native libraries, then WebAssembly is a very suitable choice. Then again, applications that heavily modify the DOM should stick to JavaScript stack, while considering porting heavy computations to WebAssembly if needed.

As the web history has shown, it's very unlikely that JavaScript will be replaced in the coming years. The community and ecosystem around JavaScript are very thriving and the web is filled with JavaScript applications, that need to be maintained. However, in addition to performance increases, WebAssembly diversifies web development as it provides other programming language options to web development in addition to JavaScript. As WebAssembly and the software related to it continue to evolve, many steps regarding WebAssembly might get abstracted away and developers might write code that gets turned into WebAssembly without doing so intentionally.

3.5 Development

As WebAssembly is not exactly a new programming language, there are few more choices to be made when starting out. There are multiple languages with each slightly different approaches and toolchains. Although, while multiple programming languages can be used to compile into WebAssembly, many of them are still in experimental phases at this stage of development. A community-contributed list maintained by Akinyemi [2019] lists 48 languages that currently compile into WebAssembly or have their VMs in WebAssembly. 15 of these are marked as stable for production use, 21 as unstable but usable and 12 is marked as very early stages.

Out of these languages, since one of the goals of MVP was to prioritise support for C/C++, therefore, it's one of the most used and supported languages for targeting WebAssembly. In addition, especially Rust has been a popular choice for compiling into WebAssembly due to its active development and community around it. Furthermore, Rust provides extensive tutorials and documentation for using Rust and WebAssembly together. In C#, there are few options for WebAssembly, Mono and Blazor. Mono is Microsoft's open-source project, which enables running .NET applications cross-platform. Blazor is also a Microsoft's open-source project that enables hosting components that handle UI (user interface) events and execute rendering logic in the browser using WebAssembly based .NET runtime. Blazor essentially allows writing rich UI applications without having to write JavaScript. Another interesting choice, especially for web developers, is AssemblyScript, which compiles a strict subset of TypeScript into WebAssembly. Since AssemblyScript uses very similar syntax to TypeScript, it makes it a very appealing choice for web developers as it doesn't require necessary learning a new language for developing WebAssembly programs.

Since there are multiple complex steps in compiling existing code into WebAssembly, there are some compiler toolchains to help perform these tasks. In order to compile source language into WebAssembly, one popular approach is to use LLVM Clang front end to compile into LLVM intermediate representation and then using the compiler's back end, to go from LLVM IR to WebAssembly. As of LLVM version 8, WebAssembly has been supported by default [LLVM, 2019].

Different languages use different tools to compile to WebAssembly, such as AssemblyScript relies on compiler infrastructure and toolchain library called Binaryen to compile to WebAssembly, which uses its internal IR, called Binaryen IR [Wirtz and Graey, 2020]. Rust instead uses a tool called wasm-pack to build rust-generated WebAssembly packages, that can be published into the npm (Node Package Manager) registry or just used alongside regular JavaScript packages [Rust, 2019]. Since WebAssembly is designed to be compact and efficient, it requires a fair number of optimizations to do so. Typically, these types of special optimizations require language-specific optimization, in addition to standard ones, therefore, many languages like Rust, have their own IRs and optimizers before they feed into LLVM [Zakai, 2018].

In C/C++, Emscripten is a popular tool-of-choice when targeting WebAssembly. Zakai [2019] states that Emscripten's goal is to act as a drop-in replacement for standard compilers like gcc. Emscripten's compiler front end, emcc, also uses Clang and LLVM to compile into WebAssembly. For the back end, Emscripten has two options, fastcomp and upstream LLVM wasm back end. Fastcomp uses asm.js back end together with asm2wasm to first compile into asm.js and then converting it into WebAssembly. Fastcomp was designed to work as a temporary solution until upstream LLVM wasm back end is completed since upstream LLVM wasm back end provides much faster linking and the code supports new WebAssembly features. In addition, Emscripten also does much more than just compiling, as it includes multiple additional tools and libraries to allow porting entire C/C++ codebases, such as emulating file systems by using IndexedDB [Clark, 2017a].

3.6 Current state of WebAssembly

WebAssembly is still a relatively new technology as it was publicly announced in 2015, especially compared to JavaScript, which has been around for decades. WebAssembly is still in constant development and while in its current state it manages to offer many improvements to some of JavaScript's shortcomings, it still lacks some main features that might not make it such an attractive option for all web applications.

WebAssembly [2019] is designed and implemented incrementally and as such, Minimum Viable Product was specified and agreed by all four major browser vendors, Chrome, Firefox, Edge and WebKit. The MVP was released in March 2017 and its release marked the end of Browser Preview. It was the first major release, WebAssembly 1.0, that set the point where all future features would be implemented and designed in a backwards-compatible manner. The MVP contained core features, such as support for modules, virtual and linear memory, and set of data types, such as 32 and 64-bit integers and floating-points. The focus of the MVP was to provide similar functionality as asm.js and to provide good support for C/C++.

With WebAssembly still being a relatively young technology, there is a good amount of speculating and experimenting on how WebAssembly can benefit their applications, but there are few actual real-world use cases of WebAssembly's benefits. One example of this is from eBay, where Jha and Padmanabhan [2019] describe using WebAssembly to compile their existing C++ barcode scanner library into the web. They had already used the same library for their native Android and iOS applications, however, for the mobile web they had to use a different solution, that performed very inconsistently. But since C++ libraries are ideal for compiling into WebAssembly, they were able to raise the previous solution's success rate from 20% to close to 100%. This was achieved by compiling C++ barcode scanner libraries into WebAssembly and then multithreading and racing three web workers to each other, one for their own C++ scanner library, one for ZBar, open-source C++ barcode scanner library and third for their original JavaScript solution. Another success story is from bioinformatics, where they were able to achieve 20x speed up to their DNA sequencing application by replacing slow JavaScript computations with calls to WebAssembly equivalents [Aboukhalil, 2019].

While the above-mentioned use cases showcased the positive effects of WebAssembly, they aren't the only occurrences of WebAssembly in the web. A study conducted by Musch et al. [2019] in June 2019 shows that among the Alexa Top 1 million websites, 1 out of 600 sites use WebAssembly. However, 56% of those sites use WebAssembly for malicious purposes, such as cryptocurrency mining and malicious code obstruction. This type of mining in the browser without the consent of the user has been around for a few years, especially using JavaScript. However, WebAssembly is especially suitable for mining purposes due to its performance and as it allows compiling cryptographic primitives into the browser. Rest of the

WebAssembly usages was mostly found in games, JavaScript libraries, that use WebAssembly under the hood, and tests that simply test if WebAssembly is available in the visitor's browser.

3.7 Future

Outside of the core features of the MVP release, there were still multiple key features in the high-level goals of WebAssembly that are in various phases of design and development. As these features continue to evolve, it will broaden the usage areas and programming language selection of WebAssembly. An especially important feature is the support of garbage collection, that would provide access to the advanced garbage collectors that exist in all browsers. This would expand the programming language option as it would allow typed languages, such as TypeScript, to be compiled into WebAssembly.

Other key features include exceptions, threads, and Single Instruction Multiple Data (SIMD) instructions, that all would furthermore enhance the performance of WebAssembly. All features go through a specification process, which all browser vendors participate in. While these features are important and bring many upsides, due to the nature of WebAssembly, there are multiple challenges in designing such complex features. For example, in exception handling, both embedder and WebAssembly have different notations of exceptions, but both must be aware of the other. However, as WebAssembly doesn't have knowledge of the embedder, nor does it have direct access to embedders memory, so WebAssembly must defer exception handling to host VM. Currently, exceptions are mostly emulated by compilers, although it can reduce performance. All these features and future proposals are tracked in detail in the WebAssembly GitHub page. [WebAssembly, 2019]

Depending on the application domain, some of these key features can heavily affect the performance, such as threading support and SIMD. SIMD allows performing the same operations on multiple data points simultaneously, like a vector. This is very useful when heavily working with multimedia, as one common application for SIMD is to use it for adjusting the brightness of the image. Applications requiring complex calculations gain much from SIMD, such as games, but it might not bring that much to average web applications. Currently, WebAssembly's SIMD feature is split into two parts, focusing first on providing support for short SIMD, that focuses on fixed-width 128-bit types, similar to that is in PNaCl's SIMD and SIMD.js, and then, later on, focusing on long SIMD [WebAssembly, 2019]. As a result, SIMD.js has been removed from EcmaScript's active development, as WebAssembly deemed more adequate for future SIMD development [SIMD.js, 2018].

These days, many web applications revolve around libraries or frameworks, like React.js or Angular, for building their front end applications. While these applications can gain performance increases by implementing their business logic in WebAssembly, there isn't, at least currently, many benefits to using WebAssembly for their UI interactions. This is due to WebAssembly not having direct access to web API yet but also, slow parts of the UI libraries are usually caused by interacting with the DOM. However, teams working on UI libraries, like React.js, could implement their core code, such as the reconciliation algorithm with WebAssembly, which could bring performance increases to all React.js applications [Clark, 2017a].

Outside of future features, there are other elements that continue to develop as WebAssembly progresses. One of these is the developer tools, that can aid and improve the code quality and development. Currently, browsers have some support for debugging WebAssembly but it’s not nearly as advanced or sophisticated as it’s for JavaScript, which has been developed for decades. Supported debugging features vary from browser to browser and currently it mostly requires some manual steps from the developer as many steps are not yet automated, including source map generation. Therefore, to access all the debugging features, it’s currently required to be using multiple browsers and jumping between them. Although WebAssembly can always be debugged in the original language that the developer is writing to generate WebAssembly and in most cases, it might be enough.

WebAssembly can run on non-web embeddings, however, currently, the priority has been in the web. Thus, compared to the web, the tooling for non-web embeddings is lacking, but it continues to evolve. Usage areas in the non-web environment include virtual reality, serverless computing, embedded systems, such as wearables. In addition, embedded scripting inside games, web applications and other sandboxed environments is another option, as WebAssembly possibly has much to offer there. Currently, making mods for multiplayer games, are usually done with Lua or Java. However, Java is generally usually considered to be too bloated for this purpose and Lua lacks performance. Therefore, WebAssembly can offer another suitable option while solving these problems.

4 Sample Application: Game of Life

For this thesis, the Game of Life application was chosen to be implemented for WebAssembly and JavaScript evaluation and comparison. Typically, the bigger and more complex the application is, the more opportunities and possible performance gain WebAssembly porting can bring. However, the purpose of this application is to evaluate how much can WebAssembly bring to simpler and smaller applications. In addition, Game of Life can become quite performance heavy depending on the life setup, overall grid size, and the duration of each generation, and can be even be extended with additional features outside of core features. In addition, WebAssembly lists casual games as one of the WebAssembly use cases inside the browser [WebAssembly, 2019].

The application was implemented in three phases, first JavaScript-only version of Game of Life was developed and then parts of the JavaScript implementation was ported into WebAssembly. In the final phase, a WebAssembly-only version was implemented.

4.1 Game of Life

Game of Life is a cellular automaton designed by John Horton Conway in 1970. The game itself is a zero-player-game, as it requires no additional interaction from the user after the initial board state has been set. [LifeWiki, 2018]

The universe in Game of Life is infinite, which consists of a two-dimensional grid of square cells, that each has a state of either alive or dead. Every iteration step, each living cell interacts with its eight neighbours using the following rules, as explained in LifeWiki [2018], the Game of Life Wiki:

1. Any live cell with fewer than two live neighbours dies (referred to as underpopulation or exposure).

2. Any live cell with more than three live neighbours dies (referred to as overpopulation or overcrowding).

3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.

4. Any dead cell with exactly three live neighbours will come to life.

4.2 Implementation

As mentioned in the Game of Life description section, the universe in Game of Life is infinite. However, in reality, computer memory is finite. This results in multiple different approaches to implementing the Game of Life algorithm. Popular choices in this field include using a two-dimensional array to hold the universe with a fixed universe size and assuming every cell outside the universe is dead. While this is simple to program, it will naturally cause some inaccurate results when life reaches the universe border. Another choice when using a two-dimensional data structure is to consider the edges stitched together, resulting in a toroidal array. In this approach, life cell will cross from one edge to the opposite edge. Other approaches include dynamically expanding the array holding the universe or using different data structures, such as vector.

As Game of Life has been around for quite a long time and has been researched thoroughly, there are even Game of Life specific algorithms. One of the notable algorithms that LifeWiki [2018] lists is called HashLife, which is a memoized algorithm, that uses hash tables to store the data. The algorithm allows taking advantage of the repetitive tasks in Game of Life by storing subpatterns in a hash table, so they don't have to be recalculated whenever the same pattern shows up in the future. However, storing multiple complex patterns will naturally increase the memory consumption of the program. All in all, there is no correct or incorrect approach, but they just lead to a different Game of Life variant.

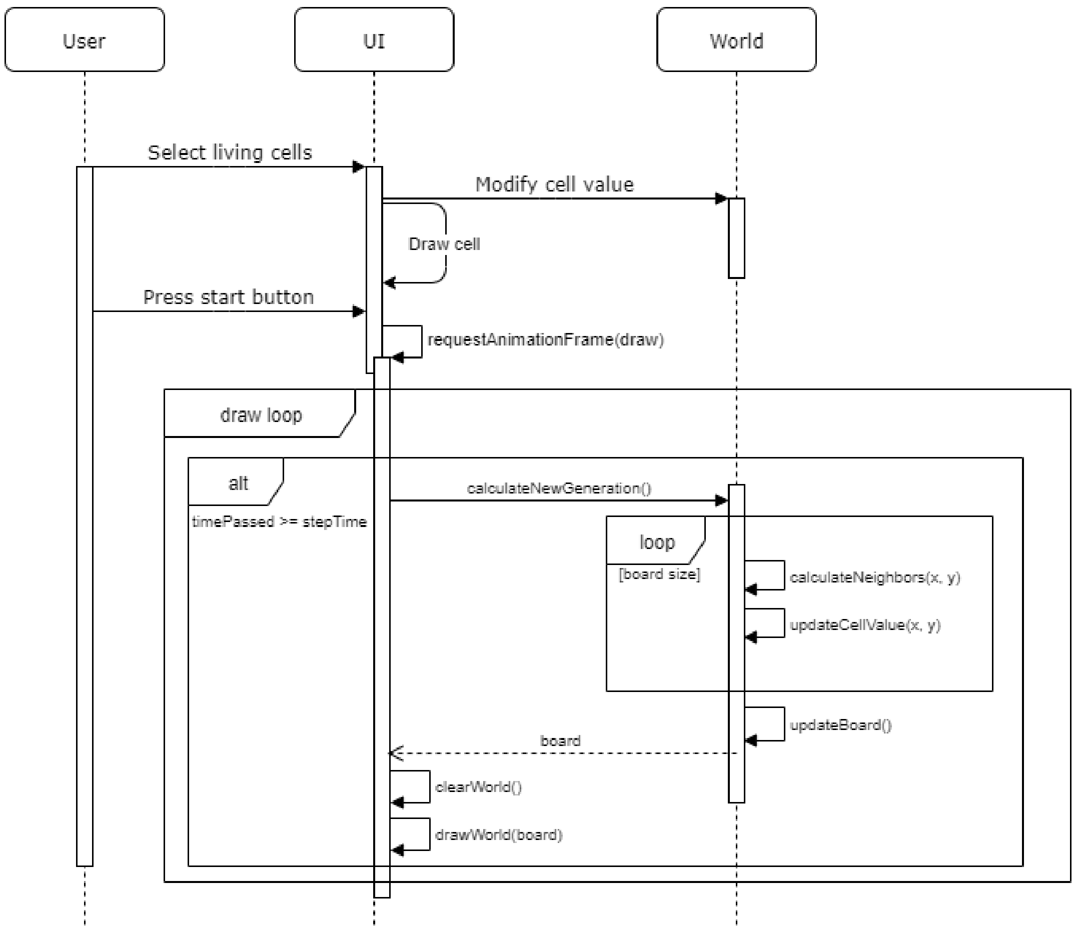

Figure 2. Sequence diagram depicting the overall Game of Life application operations.

For this Game of Life application, the toroidal array approach was chosen as it seemed simple enough to program while proving interesting and unique Game of Life patterns, that might not the possible in typical infinite Game of Life universe. However, for the purpose of this application, the approach choice doesn't matter that much, since regardless of the approach, the algorithm would be implemented in both JavaScript and WebAssembly.

Figure 2 shows the overall operations of the application from the start of user interaction to continuous Game of Life generation simulation. After the user has selected the initial Game of Life board state, the user clicks the start button to start the game, which will cause the program to enter the main drawing loop. This draw function is called by the browsers requestAnimationFrame method, which will call the requested function depending on the monitors refresh rate, which usually is at least 60 times per second. However, inside the draw function itself is a check to see if enough time has passed for a new generation to be calculated and drawn. This value, called stepTime, is configured before starting the game. For example, the value of 1000 ms would cause the current board to stay on screen for 1 second before the new generation is calculated and drawn. When enough time has passed, the new generation is calculated, the old board is cleared and a new one is drawn. These generation calculation related functions are shown and explained in more detail in Subsection 4.2.2. WebAssembly porting.

4.2.1 JavaScript implementation

Game of Life is a relatively straightforward application to implement, but there are few design choices to be made regarding the implementation of the application. Notably, the data structure used for holding the Game of Life board state and the choice of graphics API. Although, in JavaScript, these choices are much more straightforward than in some other languages. For data structure, JavaScript array or some array variant is a natural choice but there is still the choice of deciding how data is stored in the array. Representing the board state in a two-dimensional array is easy to grasp and develop, but the WebAssembly porting phase should be kept in mind. As passing simple data types, such as integers, is simple between JavaScript and WebAssembly, but the more structured the data is, the more complex the process is, and it can add some overhead. Therefore, to simplify the porting phase, the board state was implemented as a one-dimensional array using *Int32Array* typed array, with values 0 and 1, where value 0 represents a dead cell and 1 alive cell. Performance-wise, there aren't any meaningful differences between using a one-dimensional or a two-dimensional array.

```
for (let y = 0; y < yMax; y++)
{
  for (let x = 0; x < xMax; x++)
  {
    const cur = board[y][x];
    ...
  }
}
```

Listing 7. Iterating a two-dimensional array.

```
for (let y = 0; y < yMax; y++)
{
  for (let x = 0; x < xMax; x++)
  {
    const i = x + xMax * y;
    const cur = board[i];
    ...
  }
}
```

Listing 8. Iterating a one-dimensional array like a two-dimensional array.

Listing 7 shows a typical two-dimensional iterating and Listing 8 shows how to iterate one-dimensional array as it was two-dimensional. yMax represents the board's grid height value and xMax the width value. Listing 8 only has one extra step to calculate the index in a one-dimensional space.

The second major choice is the choice of graphics API. JavaScript has both Canvas API and WebGL API. Canvas API is much simpler and easier to learn and use than WebGL, especially when starting from scratch, both coding and knowledge-wise, as Canvas API requires minimal math knowledge and is more straightforward to use. Then again, while being more complex, WebGL is faster and has more capabilities than Canvas. But, the WebAssembly porting again makes the choice clear, as WebAssembly doesn't yet have direct access to the web APIs, which includes Canvas API, but OpenGL (Open Graphics Library) code written in C++ gets converted into WebGL equivalents when compiling into WebAssembly, therefore it makes more sense to use WebGL in the JavaScript implementation.

4.2.2 WebAssembly porting

Before planning which parts of the JavaScript implementation will be ported into WebAssembly, language used for writing WebAssembly had to be decided. While there are multiple languages that can be used to compile into WebAssembly, many of them are still on experimental phases at this stage. C/C++ was specially designed to be well supported right from the start of MVP, but in addition, notably, Rust has been a popular choice for compiling into WebAssembly. Another interesting choice is AssemblyScript, which compiles a strict subset of TypeScript into WebAssembly. For this application porting, C++ was chosen due to previous experience with the language and since it's well supported and documented.

```
const len = board.length;
const bytesPerElement = board.BYTES_PER_ELEMENT; // 4

const inputPtr = module._malloc(len * bytesPerElement);
const outputPtr = module._malloc(len * bytesPerElement);

module.HEAP32.set(board, inputPtr / bytesPerElement);
module._calculateNextGeneration(inputPtr, outputPtr,
    gridCountY, gridCountX);
nextGeneration = new Int32Array(module.HEAP32.buffer,
    outputPtr, len);

// deallocate memory
module._free(inputPtr);
module._free(outputPtr);
```

Listing 9. Calling WebAssembly function from JavaScript.

In Listing 9, WebAssembly function calculateNextGeneration is called with the current board state, next generation board state, boards height value gridCountY and width value gridCountX. As explained in Section 3.2. Key concepts, the module represents the WebAssembly binary format, which contains a function, table, global and memory definitions. Since WebAssembly works in a sandbox and doesn't have direct access to memory outside of it, in order to pass data structures between WebAssembly and JavaScript, they must be written directly into memory. In this code body, it's done by using *malloc*, an exported function from C++. Malloc function allocates the requested amount of memory, which in this case is the size of board array times the size of a 32-bit signed integer and returns the pointer to it. By calling module.HEAP32.set, the array is then written into WebAssembly memory. The set method takes a second parameter, which is the offset value. As malloc function returns the offset from the beginning of the WebAssembly memory, but since we are dealing with 32-bit integers (4 bytes each), the actual length is the offset divided by the bytes. After the WebAssembly function calculateNextGeneration is called, the data is then extracted into another JavaScript Int32Array. Afterwards, the allocated memory is deallocated by using free, which is an exported function from C++.

```cpp
EMSCRIPTEN_KEEPALIVE
void calculateNextGeneration(int *board, int *nextGeneration,
    int yMax, int xMax)
{
  for (int y = 0; y < yMax; y++)
  {
    for (int x = 0; x < xMax; x++)
    {
      int i = x + xMax * y;
      nextGeneration[i] = 1;
      int nCount = calculate1dNeighbors(board, i, yMax, xMax);
      int current = board[i];
      // Rules 1 and 3
      if (current == 1 && (nCount <= 1 || nCount > 3))
      {
        nextGeneration[i] = 0;
      }
      // Rule 4
      else if (current == 0 && nCount == 3)
      {
        nextGeneration[i] = 1;
      }
      // Rule 2
      else
      {
        nextGeneration[i] = current;
      }
    }
  }
}
```

Listing 10. C++ function that calculates the next generation according to Game of Life rules.

Listing 10 shows the calculateNextGeneration function which was called from JavaScript in Listing 9. It iterates over every element of the board to calculate the next generation of cells according to the four Game of Life rules that were listed in Section 4.1 Game of Life. As the current generation and the next generation states are passed from JavaScript to WebAssembly as pointers, there isn't need to return them. EMSCRIPTEN_KEEPALIVE flag keeps the compiler from possibly removing the function in optimization phase. Listing 10 and 11 ported functions differ very little from their JavaScript equivalents outside of type definitions, as they don't contain anything unique to either language.

```
EMSCRIPTEN_KEEPALIVE
int getValue(int *array, int x, int y, int yMax, int xMax)
{
  int xIndex = ((x % xMax + xMax) % xMax);
  int yIndex = ((y % yMax + yMax) % yMax);
  int index = xIndex + xMax * yIndex;
  return array[index];
}

EMSCRIPTEN_KEEPALIVE
int calculate1dNeighbors(int *arr, int i, int yMax, int xMax)
{
  int xIndex = i % xMax;
  int yIndex = i / xMax;
  int nCount = 0;
  nCount += getValue(arr, xIndex - 1, yIndex - 1, yMax, xMax);
  nCount += getValue(arr, xIndex, yIndex - 1, yMax, xMax);
  nCount += getValue(arr, xIndex + 1, yIndex - 1, yMax, xMax);
  nCount += getValue(arr, xIndex - 1, yIndex, yMax, xMax);
  nCount += getValue(arr, xIndex + 1, yIndex, yMax, xMax);
  nCount += getValue(arr, xIndex - 1, yIndex + 1, yMax, xMax);
  nCount += getValue(arr, xIndex, yIndex + 1, yMax, xMax);
  nCount += getValue(arr, xIndex + 1, yIndex + 1, yMax, xMax);
  return nCount;
}
```

Listing 11. Two C++ functions that check and add all 8 neighbour counts together.

Listing 11 contains two functions, calculate1dNeighbors and getValue. Function calculate1dNeighbors is called for every cell in the board and it calculates the neighbour count by checking all eight neighbour cells. Function getValue returns the requested (x, y) cell value, however, since the array holding the board state is one-dimensional and the neighbour checking is done in a toroidal approach, it requires few extra steps to get the actual value of the cell. First x % xMax is calculated, then xMax is added to it in case the previous calculation returned negative index and then finally, the modulus division is done to the result, in case the previous step made the index greater than xMax. Y index is similarly calculated. These calculations make the indexes wrap around the grid. The final index calculation is done to calculate the actual index in one-dimensional space. Alternatively, the toroidal neighbour check could have been implemented with multiple if statements instead of modulus division.

Emscripten [2019] provides three different modes for OpenGL. One providing support for emulating older desktop OpenGL legacy features, one emulating some of the OpenGL ES (OpenGL for Embedded Systems) features that are not in WebGL and then finally WebGL-friendly subset that provides direct mappings to WebGL 1/2 features. The latter one was chosen for this porting, as it's the most efficient one and recommended one for new code.

Graphics-wise, this Game of Life implementation is relatively simple, as there is only a single colour square being drawn every generation on the 2D surface without animation. Therefore, only one vertex and fragment shader are needed for the entire drawing process. As the drawing process is mostly done by GPU (Graphics Processing Unit), it was kept as simple as possible, since the focus of these comparisons and evaluations is on the CPU calculation side.

```
// Vertex shader
attribute vec2 aVertexPosition;
uniform vec2 uniformResolution;
void main()
{
    vec2 zeroToOne = aVertexPosition / uniformResolution;
    vec2 zeroToTwo = zeroToOne * 2.0;
    vec2 clipSpace = zeroToTwo - 1.0;
    gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}

// Fragment shader
void main()
{
    gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
}
```

Listing 12. Shaders written using the OpenGL ES Shading Language (GLSL).

Listing 12 contains the applications shader program, which includes the vertex and fragment shader. The vertex shader is responsible for calculating the vertex positions. For every shape being rendered, the vertex shader is run for each vertex in the shape. Vertex shader needed to draw a square isn't too complicated, but since, OpenGL clip space coordinates go from -1 to +1 and the actual application works with pixel coordinates, there are few extra lines in the vertex shader here to convert from provided pixel values to clip space coordinates. Attribute aVertexPosition contains the x, y values, and the uniform uniformResolution contains the canvas resolution. The vertex shader's main function converts the pixel values to clip space coordinates and then sets them to vertex shader's special variable gl_Position. After the shape's vertices have been processed by the vertex shader, the fragment shader is called for every pixel of the shape being drawn. Since there isn't any lighting to be applied, the fragment shader used here is really simple. Using the fragment shader's special variable, gl_FragColor, black colour is set as the colour of the shape.

On the WebAssembly side, to use the shaders in Listing 12, the shader's source code is first passed to OpenGL, then compiled and linked together. In addition, in the initialisation, buffers are created to pass then vertex data onto the GPU and OpenGL is provided information on how to get the data from the buffers and provide it to the attributes in Listing 12.

```cpp
void drawScene(int count) {
  glViewport(0, 0, canvasWidth, canvasHeight);
  glUniform2f(uniAttrib, canvasWidth, canvasHeight);
  // Draw squares from the vertices
  glDrawArrays(GL_TRIANGLES, 0, count);
  SDL_GL_SwapWindow(window);
}
void setRect(int x, int y, std::vector<GLfloat> &arr) {
  const GLfloat _x1 = (x * gridWidth) + 1.0f;
  const GLfloat _x2 = (x * gridWidth) + gridWidth;
  const GLfloat _y1 = (y * gridHeight) + 1.0f;
  const GLfloat _y2 = (y * gridHeight) + gridHeight;
  arr.insert(arr.end(), {_x1, _y1, _x2, _y1, _x1, _y2, _x1,
      _y2, _x2, _y1, _x2, _y2});
}
void drawLife() {
  clearLife();
  std::vector<GLfloat> vertexes;
  for (int y = 0; y < gameGridCountY; y++) {
    for (int x = 0; x < gameGridCountX; x++) {
      int i = x + gameGridCountX * y;
      if (gameBoard[i] == 1) {
        setRect(x, y, vertexes);
      }
    }
  }
  glBufferData(GL_ARRAY_BUFFER, vertexes.size() * sizeof(
      GLfloat), vertexes.data(), GL_STATIC_DRAW);
  drawScene(vertexes.size() / 2);
}
```

Listing 13. C++ functions that are responsible for rendering the board.

Listing 13 contains three functions that are responsible for creating the buffer containing all the vertex positions and rendering the scene. Function drawLife is called after the neighbour calculations have been completed. One effective optimization technique in rendering is to reduce the number of draw calls. Due to the fact, that all shapes being drawn each scene are exactly the same shape, it's easy to batch all the vertexes together and only draw once. However, if there were multiple different shapes, then drawing would have to be called for every shape. In the drawLife function, the board is iterated for every alive cell and their coordinates are saved into a vector inside setRect function. The rectangle that's drawn consists of two triangles, with each having three points. Therefore, there are 12 values being inserted into the vector, as each point has x and y value. After all rectangle points have been inserted into the vector, glbufferData is called, which copies the data to the position buffer in the GPU. Finally, the scene is drawn by calling glDrawArrays function.

The listed listings contain the most meaningful functions of the codebase in regard to performance evaluation conducted in Chapter 5. Results. Majority of the initialisation code has been omitted, especially regarding WebGL/OpenGL, but they can be found from the thesis application repository. [Ylenius, 2020]

5 Results

5.1 Performance evaluation

In this chapter, the different application implementation splits are evaluated in terms of performance. However, the performance isn't only about the execution times, but also about the overall overhead. Therefore, in addition to execution times, the evaluation includes the memory usage, size, and load times. C++ implementations will, in addition to WebAssembly, be converted into asm.js as well to compare its performance to standard JavaScript and WebAssembly.

The application has two main parts that are most interesting and meaningful to perform performance evaluations. The first one being the performance of Game of Life neighbour calculation iteration that was shown in the functions of Listings 10 and 11. JavaScript-only and WebAssembly-only implementations simply calculate the next generation but JavaScript+WebAssembly mix implementation has one extra step since the board state has to be passed from JavaScript to WebAssembly, as shown in Listing 9. The second interesting part to evaluate is the functions related to drawing the new board into the Canvas element. This part includes clearing the current board, the work to batch all the vertexes together, copying the vertex data into position buffer and drawing the squares from the data, as shown in Listings 12 and 13.

5.1.1 Test setup

Web feature support and performance can vary from browser to browser. Especially since the technology in question here is still relatively young, the performance evaluation includes results from both latest the versions of Google Chrome (77.0.3865.120 (64-bit)) and Mozilla Firefox (70.0 (64-bit)). In all performance tests, the WebAssembly and JavaScript code has been built for production. For JavaScript, this is done by using webpack v4, which minifies the bundle and optimizes the assets loading. For WebAssembly, Emscripten with optimization flag -O3 is used for a production build. All performance tests are conducted using Lenovo ThinkPad T440s with Intel® Core™ i5-4210U CPU @ 1.70GHz (4 CPUs), 12 GB of DDR3 memory and Intel® HD Graphics Family.

There are few adjustable variables in the Game of Life implementation that will directly affect the performance requirements to smoothly run the game at 60 FPS (frames per second). The variables include the size of the single grid square and the duration of a single generation. As the Canvas width is set to a fixed value of 1440 pixels and height to 960 pixels, the number of grids in the board are calculated from the grid size and canvas dimensions. The smaller the size of grid square is set to, the more grids there are in the board which will determine the array size holding the board, amount of neighbour calculations to be done and the number of grid squares to be drawn each generation iteration. In addition to the grid square size, the generation duration will affect the game's FPS. For all the performance tests, the grid square size has been set to 2 pixels, resulting in a 720x480 board (345600 grid squares) and the generation duration has been set to 20 ms. To get the most accurate results, every test was run 3 times and the best result for each implementation split was selected.

Figure 3. Zoomed-out screenshot of the 1000th generation of Game of Life.

The starting Game of Life setup was a single long horizontal line in the vertical midpoint of the board starting from x-index 1 and ending in x-index xMax-1. The left and right edge cells were left dead due to toroidal neighbour checking. If the edge cells were left alive then the consecutive generations would lead to more horizontal lines growing to both up and down directions, therefore, leaving the edge cells dead leads to more interesting and unique patterns. Since the board is 720x480 in size, this results in 718 alive cells in the initial board setup. This initial pattern allows causes the cells to expand in both up and down directions while creating common Game of Life shapes along the way. The performance tests were run for 1000 generation iterations and on average there were 11280 alive cells each generation with a peak of 93068 at 127th generation. Figure 3 shows the 1000th generation when there were 7564 alive cells on the board.

5.1.2 Execution

The implementations have two major portions where the majority of the time is spent in terms of execution time. These are the neighbour calculation and the calculations related to rendering the board.

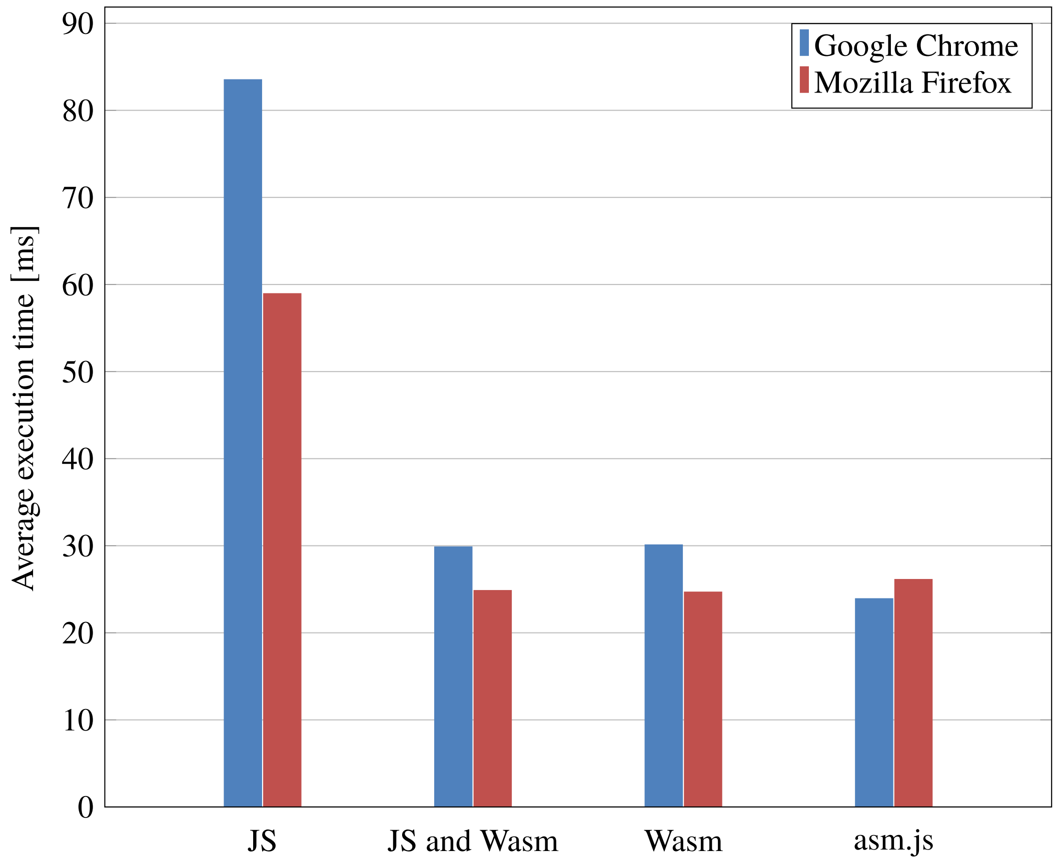

Figure 4. Average neighbour calculation execution times (lower is better) for 1000 iterations in Google Chrome and Mozilla Firefox.

Figure 4 shows the results of average neighbour calculation execution times for 1000 iterations in both Google Chrome and Mozilla Firefox. As expected, JavaScript-only implementation has worse performance compared to WebAssembly and asm.js implementations. In addition, Chrome has much worse performance compared to Firefox in the JavaScript-only implementation, with Firefox having an average of 58.94 ms and Chrome 83.51 ms. This can be caused by the different JIT optimizations in the browser or even the JavaScript typed array implementation. Other implementations have very similar performance to each other, JavaScript+WebAssembly having 24.85 ms for Firefox and Chrome 29.86 ms. As expected, WebAssembly-only implementation has the almost exact same value as the JavaScript+WebAssembly implementation, with Firefox 24.73 ms and Chrome 30.35 ms which shows that passing the board state between JavaScript and WebAssembly in JavaScript+WebAssembly implementation is really fast and has minimal effects on the performance. Surprisingly, the asm.js implementation has the best performance in Chrome with 29.90 ms average time, which is the only implementation where Chrome had better numbers than Firefox. Firefox's asm.js had an average execution time of 26.11 ms and it's slightly worse than in Firefox's WebAssembly implementation.

In terms of consistent performance, WebAssembly and asm.js based implementations had much less variation in execution times, but JavaScript execution times were much spikier as can be expected from JIT compiler. Overall, the browser's performance varies in every implementation. In this particular case, Firefox has the upper hand, however, in another case, Chrome might come ahead as the strengths and weaknesses of the browser in implementation and optimizations can be quite different.

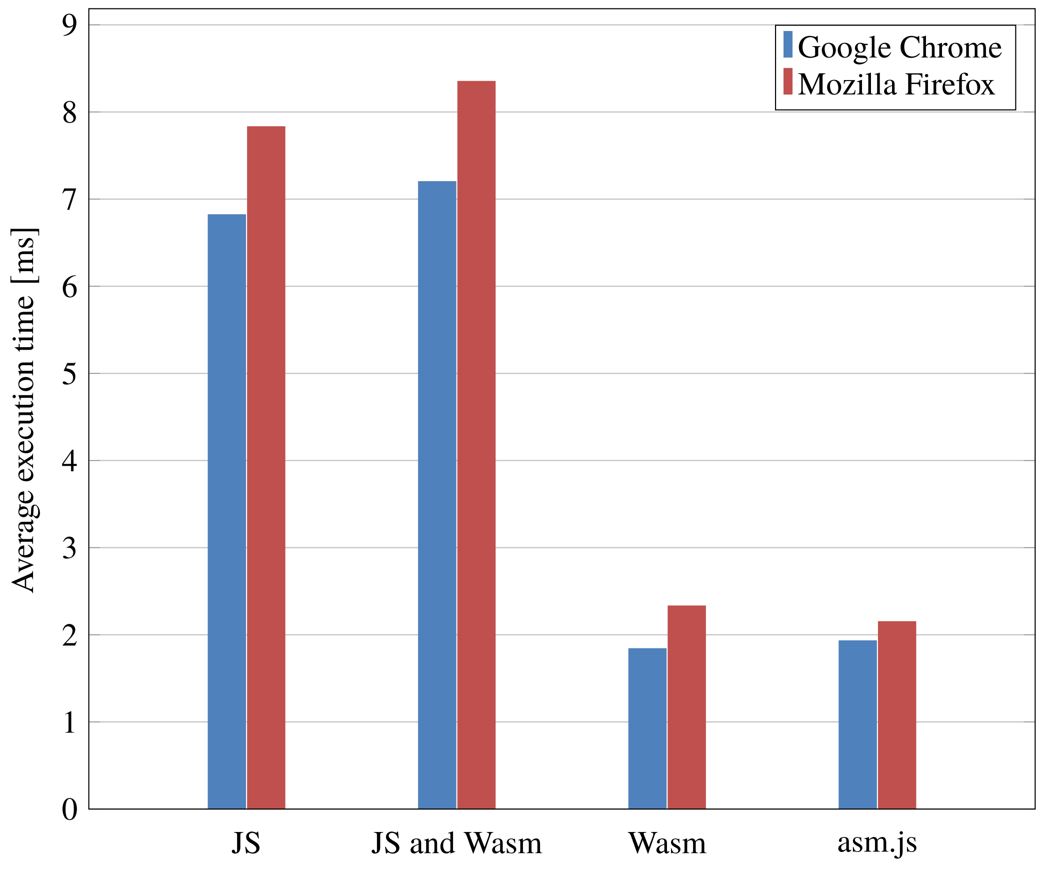

Figure 5. Average board draw times (lower is better) for 1000 iterations in Google Chrome and Mozilla Firefox.

Figure 5 shows the results of average board draw times calculation for 1000 iterations in both Google Chrome and Mozilla Firefox. This includes the work to batch all the vertexes together in addition to actual drawing. JavaScript and JavaScript+WebAssembly times are very similar as is to be expected, with JavaScript-only having an average of 7.83 ms in Firefox and in Chrome 6.82 ms. JavaScript+WebAssembly Firefox average is 8.12 ms and Chrome 7.19 ms. However, in WebAssembly-only and asm.js implementations, the average draw times are notably lower than in JavaScript-only and JavaScript+WebAssembly implementations. WebAssembly-only had an average of 2.30 ms in Firefox and in Chrome 1.84 ms, which is the lowest of all implementations. Asm.js only coming slightly behind in Chrome with an average of 1.92 ms and in Firefox 2.14 ms.

While WebAssembly graphics rendering is written using OpenGL, it's still getting converted into direct WebGL counterparts, therefore all the implementations use WebGL for rendering. However, as these times include the vertex calculations, this is where the majority of the time differences come from. As shown in Listing 13, the functions include a large amount of calculating, including both integers and floating-points, where WebAssembly and asm.js especially excel in compared to JavaScript.

Unlike in neighbour calculations, where Firefox had better numbers in almost all cases, Chrome had better numbers in all these implementations. While Chrome typically has had a slight edge over Firefox in WebGL performance, the results depend on the operating system, graphics card and the calculations being run.

5.1.3 Memory

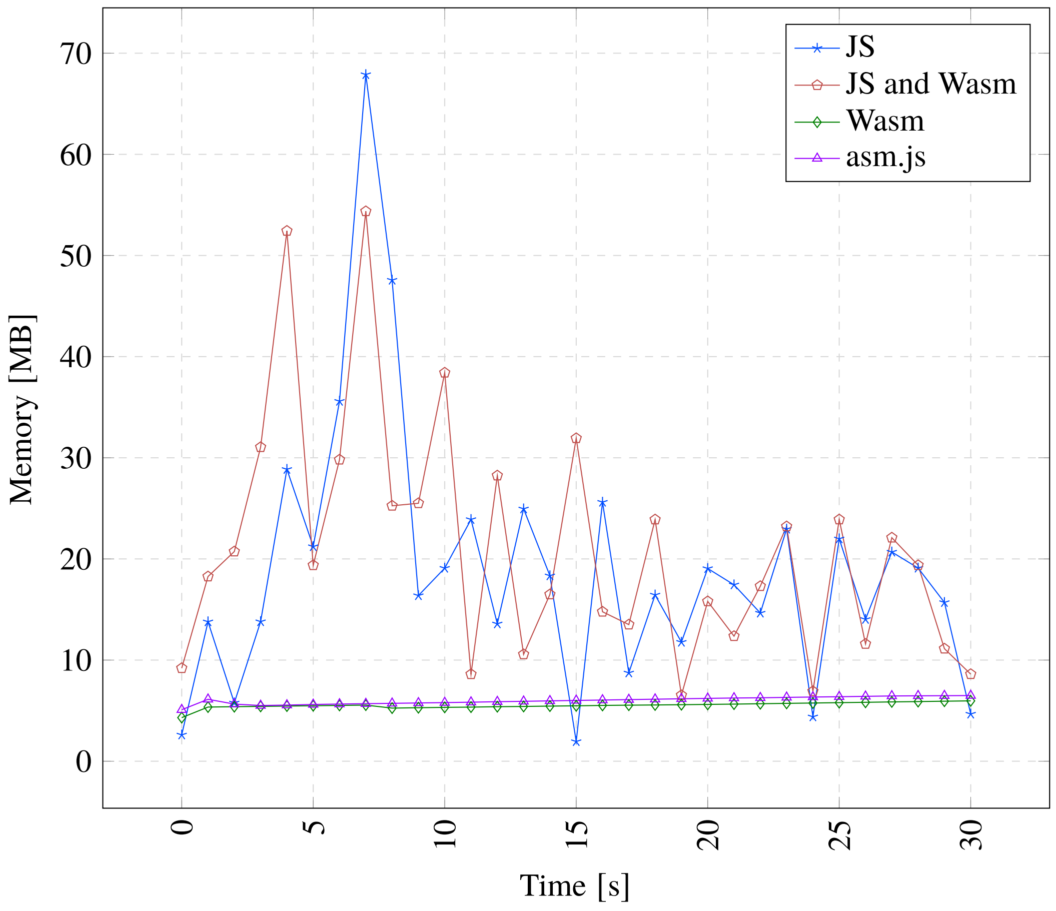

Figure 6. Overall memory usage in all implementations.

Figure 6 shows the rough memory usage of the application implementations. JavaScript results include the JavaScript heap size from the Chrome profiler. Web-

Assembly results include the JS heap size, Emscripten static memory area, which contains the constants and globals allocated at application start-up, and the results also include the dynamic memory area, which contains the dynamic heap allocations, when new or malloc is called. WebAssembly results are gathered from the Chrome profiler, but since the browser doesn't track WebAssembly memory usage, Emscripten's memory profiler is also used for additional information. However, it should be noted that the Chrome JS heap also includes Chrome's JavaScript V8 engines memory usage and memory usage can vary from run to run, therefore, it's more useful to research the memory usage trend rather than the specific values. The time for memory profiling is set to 30 seconds, and for all implementations, this is enough for over 1000 iterations of Game of Life.

The JavaScript-only and JavaScript+WebAssembly memory usage are much spikier than in the WebAssembly-only and asm.js implementations. This is due to the garbage collector and that, WebAssembly and asm.js heap size are set to a fixed size in the compile phase. For these implementations, the heap size is set to 32 MB. The trend in JavaScript-only and JavaScript+WebAssembly implementations are very similar in terms of the spikes in memory usage, although, mostly due to garbage collection, they are never exactly the same. WebAssembly-only and asm.js memory usage is almost exactly the same with both small spike in the beginning and then dropping and growing in an upward trend. The small spike, in the beginning, is mostly due to freeing the temporary compilation memory. Since this application is very small in terms of the codebase, the spike is very small, however, in bigger applications, this spike can be mitigated by separating the codebase instead of having a single large JavaScript file.

5.1.4 Size and load times

WebAssembly format is more compact than JavaScript, however, the WebAssembly file sizes can easily grow in size depending on how many standard libraries are used. In addition, currently, there has to be some JavaScript glue code to communicate between JavaScript and WebAssembly. JavaScript-only implementation doesn't use any additional third-party libraries and JavaScript+WebAssembly doesn't require any standard libraries, but WebAssembly-only implementation includes standard libraries such as vector and array and the libraries needed for OpenGL.

	Size (B)	48 Mbit/s (ms)	24 Mbit/s (ms)	1 Mbit/s (ms)
JS	8.18	3.33	3.68	58.97
JS and Wasm	22.04 (8.9)	7.18	10.45	176.3
Wasm	785 (571.12)	132.55	259.09	6270
asm.js	1240	218.36	420.56	10049

Table 1. Sizes and download times for all implementation splits.

Table 1 shows the total sizes of JavaScript (.js) and WebAssembly (.wasm) files and how long on average it took to download them with 48, 24 and 1 Mbit/s connections. Download speeds were tested using Chrome's network throttling profiles. JavaScript-only and asm.js implementations only consist of 1 JavaScript file, but JavaScript+WebAssembly and WebAssembly-only implementations have both WebAssembly and JavaScript files. WebAssembly file sizes are shown inside brackets. Even WebAssembly-only implementation requires some JavaScript glue code to handle JavaScript to WebAssembly calls. All these implementations are release versions of the implementations, with JavaScript minified and all comments removed. It should be noted that all WebAssembly and asm.js implementations have been compiled with Emscripten -O3 flag, but Emscripten offers additional flags for reducing code size, although some of them may reduce performance in favour of code size. However, in this application, these optimization flags only had at best 20-40 B differences between code sizes.

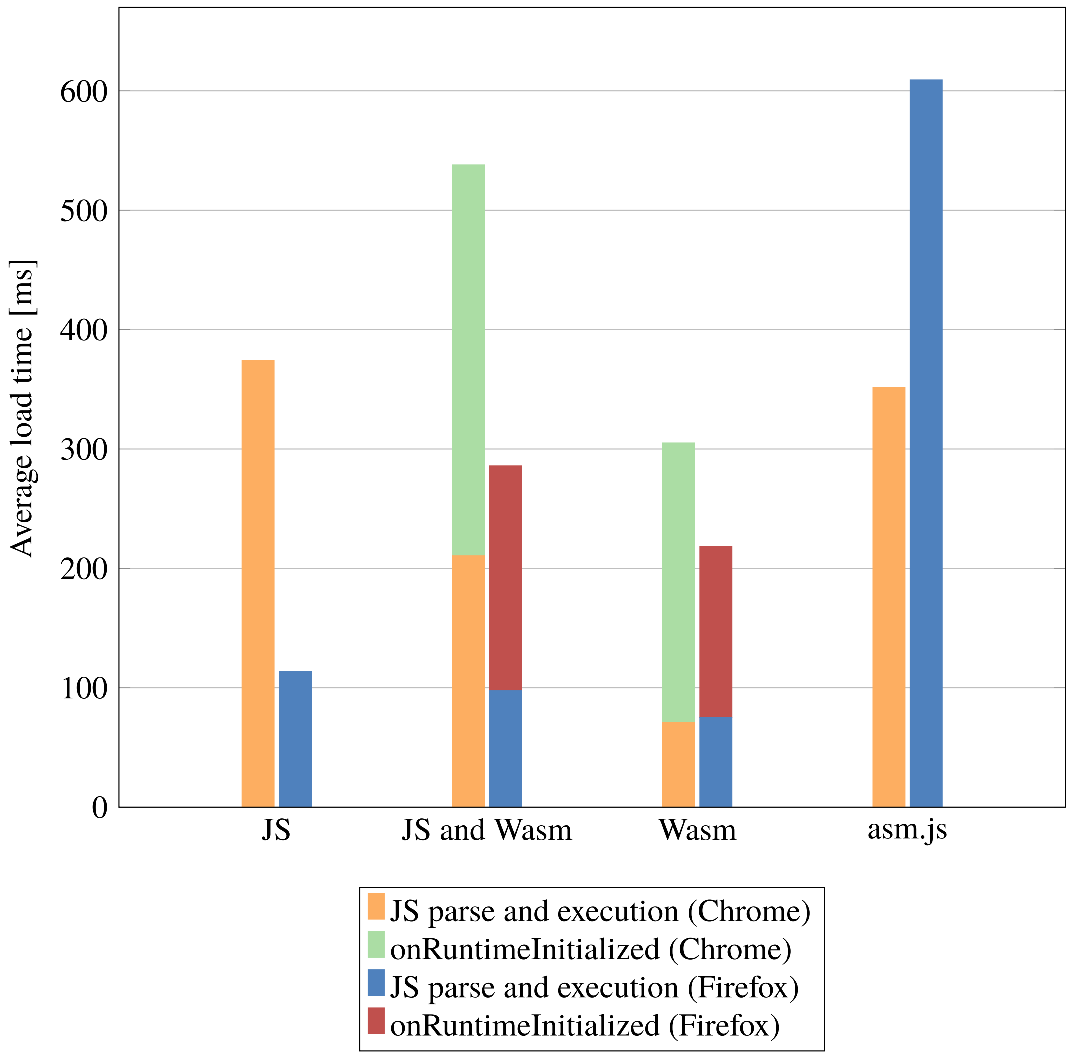

Figure 7. Average JavaScript parse and execution times and Emscripten runtime initialisation times in Google Chrome and Mozilla Firefox.

By far the lowest in size is the JavaScript-only implementation, with only 8.18 B. JavaScript+WebAssembly implementation is nearly triple in size compared to JavaScript-only implementation, with 22.04 B. However, WebAssembly-only and asm.js implementations are significantly greater in size. WebAssembly-only implementation with size of 785 B and asm.js implementation with size of 1240 B. This size difference is due to the JavaScript glue code and additional libraries such as SDL2 and OpenGL ES 2.0 libraries used for graphics rendering. JavaScript-only file is very small due to having no additional libraries or third-party packages and naturally, not requiring any glue code. With high internet speed, all implementation files get downloaded relatively fast without having detrimental effects on user experience. While both WebAssembly and JavaScript files can be loaded from cache after the first visit, however, the initial load time is still important, especially with slow connections. As expected, with a slow speed of 1 Mbit/s it takes a very long time to able to see anything on the page, which can even lead to users interrupting the download. Especially since even after the files are downloaded, there is still parsing/decoding to be done before actual user interaction can occur.

Figure 7 shows the average time it takes to parse and execute the downloaded JavaScript file and the time it takes for Emscripten runtime to be fully initialized on the first load. Runtime is fully initialized when all compiled code is safe to run, which includes asynchronous start-up operations such as asynchronous WebAssembly compilation. Results contain both Chrome and Firefox results, however, the parsing and execution strategies can vary a fair amount between browsers, therefore, it's more meaningful to compare different implementation split load time results within the browser rather than comparing them to other browser results.

Chrome has surprisingly slow JavaScript parse and execution time for JavaScript-only implementation, with 374.20 ms, especially considering that as Table 1 showed, it's the smallest in size of all implementations. In Firefox, the load time is only 113.62 ms. In JavaScript+WebAssembly implementation, the JavaScript parse and execution are lowered from JavaScript-only implementation, even though there is additional JavaScript glue code added. However, runtime initialization significantly adds to the total time, raising Chrome's total time to 537.90 ms and Firefox to 285.81 ms. While WebAssembly solution overall file size was much larger than JavaScript-only and JavaScript+WebAssembly implementations, the load times for both browsers are relatively short. Chrome total time is 305.02 ms and Firefox only 218.27 ms, with the majority of that time, is spent on runtime initialization, as expected. Asm.js implementation has worse load times in both Chrome and Firefox than the WebAssembly implementation, in addition to having bigger overall file sizes. Firefox has much higher load time than Chrome due to asm.js compilation not being supported in Chrome like it's in Firefox. Majority of Firefox's 609.09 ms is spend compiling, while Chrome's 351.29 ms is spend parsing and executing.

5.2 Discussion

As expected, WebAssembly performs very well in the calculations needed for the Game of Life application compared to JavaScript. Even for a relatively small application, WebAssembly provided a meaningful performance increase. In both Google Chrome and Mozilla Firefox, there was around 2 times performance increase in neighbour calculation times and in addition, WebAssembly performance was much more stable, while JavaScript performance numbers had much more variation. However, similar benefits were also gained from asm.js. Although, in addition to execution benefits, WebAssembly is more compact in size and faster to start-up compared to asm.js, due to WebAssembly's binary format and streaming compilation. In terms of memory, there isn't any major differences between WebAssembly and asm.js, as it to be expected since both have their memory size set to fixed size in compile phase, although in larger applications, it might become more noticeable, especially if asm.js codebase is all in single file, as it means that JavaScript engine must use memory to parse and compile the asm.js, thus causing a spike in memory usage. If the file is big enough, it might even cause the device to run out of memory.

Another interesting comparison is the JavaScript+WebAssembly versus WebAssembly-only implementation approach. As the results show, at least in this particular small application, in terms of pure execution numbers, there was a minimal additional benefit for doing WebAssembly-only implementation, as just porting bottleneck functions from JavaScript to WebAssembly had similar performance benefits. Especially when dealing with existing JavaScript solutions, the porting approach seems much more viable than fully porting, when the goal is to just achieve runtime performance increases. However, in memory and load time, WebAssembly-only implementation came ahead. WebAssembly-only implementation had much more stable and predictable memory usage compared to JavaScript+WebAssembly implementation. This is due to the lack of garbage collection and not needing to constantly pass around the board state using malloc. While JavaScript+WebAssembly had the upper hand in total file sizes, WebAssembly-only implementation was much more efficient to load.

Both porting and full WebAssembly implementation approaches are viable depending on requirements and priorities. Completely new applications might consider starting with WebAssembly especially when dealing with more complex data structures than simple data types or arrays. As not only there is a slight cost to JavaScript to WebAssembly calls but writing and reading them from memory isn't so straightforward yet. Although this will most likely change in the future, as it might be abstracted away, especially with utility libraries. But currently, especially without previous experience from manual memory management, it can be easy to create memory leaks without realising it. Then again, especially if there are multiple standard libraries being used the WebAssembly solution, the overall file size and the load time of the applications can grow quite quickly. Whether this is a big concern, depends on the requirements of the application. If the application is a complex application, such as a game, users might be more tolerant to wait the extra time. However, if the site is frequently visited by mobile devices, then it might be worth keeping the application light in size.

6 Conclusion

The goal of this thesis was to research WebAssembly's design, development, background and especially its relation to JavaScript notably from a performance perspective. WebAssembly's background covered related technologies that eventually lead to design, and development of WebAssembly, notably Emscripten, asm.js, and Google Native Client. WebAssembly was covered from multiple topics including its design, key concepts, security, development, its relation and future with JavaScript. In addition, a glance at its current status in terms of features, usage areas and future were covered.

In order to properly compare and evaluate the performance differences between JavaScript and WebAssembly, a sample application was implemented. This sample application, Game of Life, was done by implementing three versions of the same application. At first JavaScript-only version was implemented, then parts of the version were ported into WebAssembly and finally, full WebAssembly version was implemented as the last phase. These three implementations were then evaluated in terms of execution times, memory, size and load times.

From a pure execution performance perspective, the results were as expected. WebAssembly had much better performance compared to JavaScript-only implementation, in both neighbour calculation and drawing related functions. However, asm.js had similar performance to WebAssembly. Though, size and load times are where WebAssembly managed to outperform asm.js. WebAssembly-only implementation files, including JavaScript glue code, were much smaller compared to asm.js files and had nearly twice faster initialization times. JavaScript-only and JavaScript+WebAssembly implementations had by far the smallest files, however, their initialization times were relatively slow considering their small file sizes, especially in Chrome. In memory usage, both JavaScript-only and JavaScript+WebAssembly implementations had very spiky memory usage trends, mostly due to garbage collection, while WebAssembly-only and asm.js had very stable memory usage due to heap size set to fixed size during compilation and manual memory management.

WebAssembly is still a relatively new technology and there are still many parts to be researched as it continues to develop. It terms of this thesis and its application, there are multiple ways that it could be expanded and evaluated further. In addition to new features to be added to the game and compared, once multithreading is fully supported in WebAssembly, its performance compared to Web Workers could be an interesting evaluation. Furthermore, once other WebAssembly features continue to develop, less and less JavaScript glue code might be needed, especially once the web API calls directly from WebAssembly become possible. Therefore, expanding the application and re-evaluating the results might result in quite a different outcome in a few years. In addition, this thesis didn't touch much on the actual developing experience in terms of how easy it was to get WebAssembly to interoperate with JavaScript or what was lacking compared to traditional web developing methods. This could be an interesting research topic, as there is currently still a fair amount of differences in terms of developing tools and support.

Currently, most usages of WebAssembly in the real world are naturally from web applications, as its one of the main purposes of WebAssembly. However, as WebAssembly can run on non-web embeddings, it will be especially interesting how it will be utilized. Currently, compared to the web, the tooling for non-web embeddings is lacking, as the web had priority. Usages in the non-web environment include embedded devices, serverless, cross-platform applications and embedded scripting inside games, web applications and other sandboxed environments.

References

Aboukhalil, R. (2019). *How We Used WebAssembly To Speed Up Our Web App By 20X (Case Study)*. URL: https://www.smashingmagazine.com/2019/04/webassembly-speed-web-app/ (visited on 11/10/2019).

Akinyemi, S. (2019). *Awesome WebAssembly Languages*. URL: https://github.com/appcypher/awesome-wasm-langs (visited on 11/20/2019).

Can I use asm.js (2019). *Can I use asm.js*. URL: https://caniuse.com/#feat=asmjs (visited on 08/08/2019).

Clark, L. (2017a). *A crash course in just-in-time (JIT) compilers (6-part series)*. URL: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/ (visited on 08/19/2019).

Clark, L. (2017b). *WebAssembly table imports... what are they?* URL: https://hacks.mozilla.org/2017/07/webassembly-table-imports-what-are-they/ (visited on 03/02/2020).

Clark, L. (2018). *Making WebAssembly even faster: Firefox's new streaming and tiering compiler*. URL: https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/ (visited on 08/26/2019).

Disselkoen, C., J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan (2019). Position Paper: Progressive Memory Safety for WebAssembly. In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '19. Phoenix, AZ, USA: ACM, 4:1–4:8. ISBN: 978-1-4503-7226-8. DOI: 10.1145/3337167.3337171. URL: http://doi.acm.org/10.1145/3337167.3337171.

Donovan, A., R. Muth, B. Chen, and D. Sehr (2010). PNaCl: Portable native client executables. In: *Google White Paper*.

Emscripten (2019). *Emscripten documentation*. URL: https://emscripten.org/docs (visited on 11/25/2019).

Google Native Client (2019). *Native Client and Portable Native Client documentation*. URL: https://developer.chrome.com/native-client/nacl-and-pnacl (visited on 08/14/2019).

Haas, A., A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien (2017). Bringing the Web Up to Speed with WebAssembly. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, pp. 185–200. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062363. URL: http://doi.acm.org/10.1145/3062341.3062363.

Herman, D., L. Wagner, and A. Zakai (2014). *asm.js specification*. URL: http://asmjs.org/spec/latest/ (visited on 08/08/2019).

Jangda, A., B. Powers, E. D. Berger, and A. Guha (2019). Not so Fast: Analyzing the Performance of Webassembly vs. Native Code. In: *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '19. Renton, WA, USA: USENIX Association, pp. 107–120. ISBN: 9781939133038.

Jha, P. and S. Padmanabhan (2019). *WebAssembly at eBay: A Real-World Use Case.* URL: https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/ (visited on 11/10/2019).

LifeWiki (2018). *Conway's Game of Life Wiki.* URL: https://www.conwaylife.com/wiki/Conway's_Game_of_Life (visited on 12/10/2019).

LLVM (2019). *LLVM 8.0.0 Release Notes.* URL: https://releases.llvm.org/8.0.0/docs/ReleaseNotes.html (visited on 03/02/2020).

MDN (2019). *WebAssembly - MDN web docs.* URL: https://developer.mozilla.org/en-US/docs/WebAssembly (visited on 09/03/2019).

Metz, C. (2011). *Google Native Client: The web of the future - or the past?* URL: https://www.theregister.co.uk/2011/09/12/google_native_client_from_all_sides/ (visited on 08/14/2019).

Musch, M., C. Wressnegger, M. Johns, and K. Rieck (2019). New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In: pp. 23–42. DOI: 10.1007/978-3-030-22038-9_2.

Nelson, B. (2017). *Goodbye PNaCl, Hello WebAssembly!* URL: https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html (visited on 03/02/2020).

Reisinger, M. J. (2016). *PolyBench/C benchmark suite.* URL: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1 (visited on 02/29/2020).

Rust (2019). *The Rust and WebAssembly Book.* URL: https://rustwasm.github.io/docs/book/ (visited on 03/02/2020).

Schwartz, M. (2015). *WebAssembly- Explained.* URL: https://moduscreate.com/blog/webassembly-explained/ (visited on 08/07/2019).

SIMD.js (2018). *SIMD.js GitHub.* URL: https://github.com/tc39/ecmascript_simd (visited on 03/02/2020).

Wagner, L. (2017). Turbocharging the web. In: *IEEE Spectrum* 54.12, pp. 48–53. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2017.8118483.

WebAssembly (2019). *WebAssembly.* URL: https://webassembly.org/ (visited on 08/28/2019).

Wirtz, D. and M. Graey (2020). *AssemblyScript documentation.* URL: https://docs.assemblyscript.org/ (visited on 03/02/2020).

Wressnegger, C., F. Yamaguchi, D. Arp, and K. Rieck (2016). Comprehensive Analysis and Detection of Flash-Based Malware. In: *Detection of Intrusions and Malware, and Vulnerability Assessment.* Ed. by J. Caballero, U. Zurutuza, and R. J. Rodríguez. Cham: Springer International Publishing, pp. 101–121. ISBN: 978-3-319-40667-1.

Yee, B., D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar (2009). Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In: *2009 30th IEEE Symposium on Security and Privacy*, pp. 79–93. DOI: 10.1109/SP.2009.25.

Ylenius, S. (2020). *Game of Life WebAssembly JavaScript comparisons.* URL: https://github.com/samuliyle/Game-of-Life-WebAssembly-JavaScript-comparisons/ (visited on 01/29/2020).

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*, pp. 301–312. DOI: 10.1145/2048147.2048224.

Zakai, A. (2017). *Why WebAssembly is Faster Than asm.js*. URL: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/ (visited on 08/09/2019).

Zakai, A. (2018). *Compiling to WebAssembly with Binaryen*. URL: https://github.com/WebAssembly/binaryen/wiki/Compiling-to-WebAssembly-with-Binaryen (visited on 12/02/2019).

Zakai, A. (2019). *Emscripten and the LLVM WebAssembly backend*. URL: https://v8.dev/blog/emscripten-llvm-wasm (visited on 12/02/2019).