What effect does applying WebAssembly have on a compute intensive client-side application versus JavaScript?

Dennis Kievits
0946572

A Thesis in the Field of Computer Science
for the Degree of Bachelor of Science

Rotterdam University
Quintor
June, 2021

Abstract

The purpose of this research is to get a better understanding about WebAssembly and its effects when applied to a web project in comparison to JavaScript. The research is based on quantitative as well as qualitative research methods, as information was gathered via desk research, experimental research, comparative research, and field research. A prototype has also been realized that compares the performance differences between an implementation in JavaScript and WebAssembly. WebAssembly offers unique advantages for specific type of applications, where mainly performance and the ability to leverage existing code or libraries can play a large role. However applying WebAssembly does also come with some downsides, such as adding additional complexity to the project, or requiring developers with specific knowledge of WebAssembly to be able to effectively use the technology. Therefore, WebAssembly does not fit every project, and should only be considered when specific requirements are met. This research does not include information on how WebAssembly can be applied on existing projects.

Acknowledgements

I want to thank Quintor for the opportunity to research such a young and exciting technology, and I appreciate the mentoring and critical feedback that Sander ten Hoor and Bram Walet provided during the graduation internship and their insight into the technical aspects of the prototype.
I also want to thank Henk Bakker for allowing me to conduct the interview to gain more insight in what really matters for Quintor.

Furthermore, I also want to thank Jacques de Hooge and Judith Lemmens as they also have played a large role in supervising and guiding me towards performing the correct research and how to effectively write a thesis of good quality.

Last but not least I want to give my thanks to my cat, Minoes, for giving me the motivation to keep working on the thesis.

1 Table of Contents

List of Figures

List of Abbreviations

Abbreviation	Meaning
DOM	Document Object Model
NPM	Node Package Manager
CAD	Computer-Aided Design
WASM	WebAssembly
HTML	Hypertext Markup Language
JS	JavaScript
MVP	Minimum Viable Product
AOT	Ahead-Of-Time
JIT	Just-In-Time
CIL	Common Intermediate Language
MSIL	Microsoft Intermediate Language
API	Application Programming Interface
WWW	World Wide Web
VM	Virtual Machine
ASM	Assembly Language
WASI	WebAssembly System Interface
OS	Operating System
CLI	Command Line Interface
AST	Abstract Syntax Tree
POSIX	Portable Operating System Interface
IR	Intermediate Representation

Glossary

Term	Definition
Transpiling	Transforming source code from one programming language to another programming language with a similar level of abstraction
Compiling	Transforming source code into something else with a different level of abstraction, usually a lower level of abstraction such as byte code.
Minimum Viable Product	The early iteration of a product which presents sufficient features for launch
Compilation target	The target of which a particular source could be compiled into
Polyfill	Code that implements a feature on web browsers that do not support the feature, usually for compatibility with older web browsers
Hot path	Code that gets executed frequently in contrast to other parts of the code
JSX	A syntax extension to JavaScript that produces React “elements”, similar to a template language but it harnesses the full power of JavaScript
Web browser	A computer program that facilitates entry to and usage of the internet
World Wide Web	An information system where documents and other web resources are identified by Uniform Resource Locators (URLs)
Uniform Resource Locator	A reference to a web resource that specifies its location on a network; Colloquially termed a web address
AOT-compilation	Compilation that happens prior to execution, generally referring to compiling bytecode into machine code
JIT-compilation	A way of executing code that involves compilation during the execution of a program at runtime; Also known as dynamic translation or run-time compilation
Document Object Model	An API for valid HTML and well-formed XML documents. It defines the logical structure of documents and the way it is accessed and manipulated
API	A computing interface that defines interactions between software components. It also provides what kind of calls or requests can be made, how to make them and what data formats should be applied
Performance	A measurement of responsiveness and stability under a particular workload
Concurrency	The act of processing more than a single task at a time
Parallelism	The act of processing more than a single task at a time simultaneously
Virtual Machine	The virtualization or emulation of a computer system
Low-Level	A set of instructions that corresponds to an instruction in machine language or allows to address specific memory locations or other architectural elements

Overhead	Any combination of excess or indirect computation time, memory, bandwidth or other resource that is required to perform a task
Operating System	System software that manages computer hardware, software resources, and provides common services for computer programs
POSIX	A family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems
Intermediate Representation	The data structure or code used internally by a compiler or virtual machine to represent source code
Entrypoint	The position in a computer program that is first executed when it is being ran; it is also the place where the program could receive command line arguments if applicable
Monomorphization	Monomorphization is the process of turning generic code into specific code by filling in the concrete types that are used when compiled

2 Introduction

The research performed and documented in this thesis is part of the graduation project in the field of Computer Science for the degree of Bachelor of Science. The assignment is overseen and supported by Quintor, a company that builds software solutions for a wide variety of complex enterprise projects, such as the system for student grants of DUO or the system used by Enexis to manage electricity meters and gas meters [1].

The assignment is created because Quintor has interest in the possibilities WebAssembly unlocks for their current and future software solutions, such as bringing applications with compute intensive characteristics to the web browser which would not be a viable option with current widely used technologies.

2.1 Problem analysis

Quintor currently has developed an application using JavaScript named “ESDEC Calculator”. This application is used to calculate the materials required for solar projects based on provided roof and other relevant properties [2]. The ESDEC Calculator also automatically recommends the most suitable mounting system.

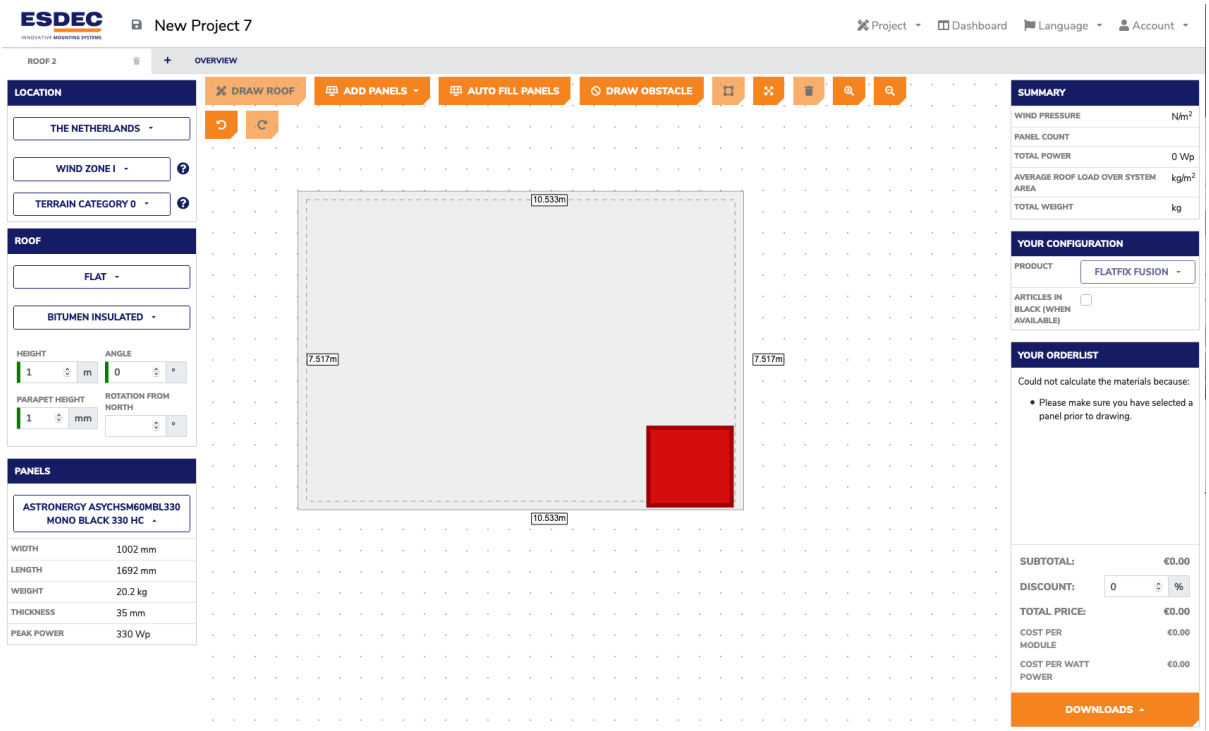

Figure 1: A sample project that showcases a drawn roof with a single obstacle.

However the application in question has compute intensive characteristics that negatively influence the user experience. Quintor has indicated that the application sometimes freezes up during the calculation process. Because Quintor is active within the innovation of cloud native development, they had come across WebAssembly and were intrigued by the potential this technology could provide for their projects. One of the most pronounced advantages WebAssembly has to offer is the ability to increase performance in web applications. Therefore, Quintor wants to know if WebAssembly could provide a solution to solve the issue at hand.

2.2 Objective

A solution should be realized that could improve the user experience of the ESDEC Calculator, and the research should be documented in this thesis as well as in an advisory report.

2.3 Main question and sub-questions

Main question:

What effect does applying WebAssembly have on a compute intensive client-side application versus JavaScript?

Sub-questions:

- What practical applications of WebAssembly in the browser currently exists?
- What are the pros and cons of WebAssembly when compared to JavaScript?
- What are the preconditions of the environment for executing WebAssembly?
- What are the performance differences of using WebAssembly versus an implementation in JavaScript?
- What are the performance differences between the available programming languages and frameworks for the same implementation?
- What are the performance differences between JavaScript and WebAssembly within a prototype of the ESDEC Calculator?

2.4 Scope

The scope of this research includes anything related to the main question as well the sub-questions. Therefore, anything related to WebAssembly but not directly related to these questions shall be not be further researched. The scope also includes prototyping and experimenting with the ESDEC Calculator application.

2.5 Products to be delivered

The products that have to be delivered are as following:

- A proof-of-concept shall be realized that solves the issue as described under Problem Analysis
- An advisory report shall be created that describes all conclusions and advice based on the research that was performed
- This thesis

3 Methodology

In this research quantitative as well as qualitative analysis has been used to analyze the effect of a compute intensive client-side application built using WebAssembly versus a similar implementation of that same application built using JavaScript. To get a more profound understanding of these technologies, a literature review has been performed and prototype applications were built for experiments to analyze the differences between these technologies. An interview with experts on frontend development of Quintor was also conducted in order to decide what the focus of the research should be put on further down the line.

3.1 Data Collection

For the literature review the world wide web was consulted, mainly for using services such as Google Scholar to look up other papers that have already done research in WebAssembly or a similar field. Web pages were also accessed for information on WebAssembly or other relevant information that was used in this document. The selected sources were compared to other sources if multiple were available based on the credibility of its author.

During the interviews that were held a note taking application named Notion had been leveraged in order to store the interviewee's answers on the questions that were asked. These interviews were conducted via Quintor's Jitsi platform which can be used to video call with others for e.g. doing meetings. Jitsi offers similar video calling functionalities to an application such as Microsoft Teams.

The results of the benchmarks that were performed were stored inside an Microsoft Excel table object which is a functionally directly available from within Microsoft Word.

3.2 Data-analysis

All analysis had been performed manually on collected data as the amounts of data were rather small, therefore not requiring specific applications in order to get better insights of the correlations within that data.

3.3 Research methods

The research methods that were applied during the research consists of:

- Desk research
- Experimental research
- Comparative research
- Field research

Sub-question	Type	Type of research	Method
What practical applications of WebAssembly in the browser currently exists?	Descriptive	Desk research	Literature study
What are the pros and cons of WebAssembly when compared to JavaScript?	Comparative	Desk research	Literature study
What are the preconditions of the environment for executing WebAssembly?	Descriptive	Desk research	Literature study
What are the performance differences of using WebAssembly versus an implementation in JavaScript?	Comparative	Experiment, performance tests	Prototyping
What are the performance differences between the available programming languages and frameworks for the same implementation?	Comparative	Experiment, performance tests	Prototyping
What are the performance differences between JavaScript and WebAssembly within a prototype of the ESDEC Calculator?	Descriptive, Comparative	Experiment, performance tests, Desk research	Prototyping, Expert interview

- Desk research was chosen in order to leverage existing information on WebAssembly that is available on the web.
- Experimental research will be used in order to prove whether the given hypotheses defined by the sub-questions are correct and to see whether the results such performance characteristics are in line with the expectations of Quintor.
- Comparative research will be used to analyze the differences in performance as well as other metrics between an implementation written in JavaScript versus WebAssembly.
- Field research will be used in order to conduct an interview with the experts on frontend development from Quintor.

4 Theoretical Framework

4.1 Assembly Language

Assembly language (or assembler language) is a low-level programming language in which there are abstractions for low-level operations [3]. Machine language is a series of numbers, which is hard to understand by humans; by using assembly language, programmers can write human-readable programs that correspond almost exactly to machine language. Assembly languages are tied to one specific computer architecture and for that reason are not portable to other architectures. One of the advantages of writing in an assembly language is that it allows for very precise control of what operations should be performed. Therefore, by writing assembly language it is possible to achieve optimal performance for a specific use case. That said, it is likely that a modern-day compiler is able to produce just as well optimized assembly language as a human can with the use of a higher-level language such as Rust or C++.

4.2 WebAssembly

WebAssembly (Wasm) is a binary instruction format for a stack-based virtual machine. A stack based virtual machine uses a stack machine where the Arithmetic Logic Unit's (ALU) primary function is to push and pop values from the stack instead of relying heavily on processor registers. This reduces the required number of processor registers significantly and therefore opens WebAssembly up to a wider variety of processors. WebAssembly is developed and designed by the W3C Community Group as a portable compilation target for programming languages, enabling deployment on the web for client and server applications [4]. Therefore, adding another option for executing code in the web browser besides JavaScript.

The W3C Community Group includes representatives from all major browsers as well as a W3C Working Group. Anyone that has a W3C account is able to join the WebAssembly Community Group to contribute to the development of WebAssembly.

The Minimum Viable Product was completed in March 2017 and shortly after made available on the public release of major browsers such as Chrome and Firefox.

Most modern-day major web browsers have support for WebAssembly [5], and there even exists two early prototypes with demos to potentially polyfill WebAssembly by using ASM.js [6].

WebAssembly was created with a few high-level goals in mind [7] [8] [9]:

- Harness common hardware capabilities, be portable and efficient
- Produce modular binaries which can be consumed in a similar way to JavaScript objects
- Integrate into the existing web platform
- Non-Web Embeddings support

WebAssembly sees the most practical application for software solutions such as:

- Video editing
- 3D-rendering

- Video games
- Music streaming
- Encryption
- Image recognition

5 What practical applications of WebAssembly in the browser currently exists?

5.1.1 Figma

Figma is a vector-based design tool that is accessible from the browser. They are using WebGL's image processing in order to allow the compute intensive software to run in the browser which is enabled by WebAssembly. Without WebAssembly the application would have not been ready for the use in the web browser [10].

5.1.2 Blazor

Blazor is an application framework where developers can create their frontend application using C# instead of JavaScript [11]. Blazor apps are composed using reusable UI components that are based on a stateful lifecycle. When the state changes, the components re-render. Similar frameworks or libraries to Blazor in terms of application structure would be React [12], Angular (from version 2 and up) [13] and Flutter [14].

5.1.3 Yew

Yew is a modern Rust framework for creating multi-threaded front-end web apps with WebAssembly [15]. It uses a syntax inspired by JSX, which is a syntax that is being used by React to declare interactive HTML. It is similar to Blazor where it allows the developers to create their frontend application entirely using Rust whilst also providing the option of interoperability with JavaScript, therefore also allowing the use of existing NPM packages or integration with existing JavaScript applications.

5.1.4 Autodesk AutoCAD web app

AutoCAD is a commercial computer-aided design (CAD) tool and drafting application. That is nowadays also accessible via the web, powered by WebAssembly [16]. Kevin Cheung has a practical guide where he covers moving a 30-year existing code base to the web using WebAssembly [17]. This showcases how WebAssembly can be leveraged to port over existing desktop class applications to the web.

The practical frameworks such as Blazor and Yew, as well as the use cases such as Figma and AutoCAD give a good idea what is possible using Web Assembly. The practical frameworks allow for the use of different programming languages hat have not to be transpiled to JavaScript to be able to run in the web browser, and the use cases show compute intensive applications being run in the browser, as well as an old code base being leveraged and being made web ready in just a few months' time.

6 How does WebAssembly work?

WebAssembly comes in two forms, a human-readable text format (.wat file) and the raw binary format (.wasm file). These formats are equivalent to each other as they are a representation of the same set of instructions. Web browsers and server applications use the binary format of WebAssembly as the size of the file would be smaller than the human-readable format.

WebAssembly is often confused with being similar to assembly, but WebAssembly is not an assembly language. WebAssembly describes an abstract syntax tree (AST) whereas an assembly language just consists of a low-level code representation. WebAssembly has higher level constructs such as loops and branches. Therefore, it is possible to write WebAssembly by hand, but usually WebAssembly is just being used as a compilation target by other languages. This means that you can write code in a language such as Rust or C++, which would then be compiled into WebAssembly.

```
(module
   (func $add
      (param $a i32)
      (param $b i32)
      (result i32)
      get_local $a get_local $b i32.add
   )
   (export "add" (func $add))
)
```

Snippet 1: Example of WebAssembly code that showcases a function to add two numbers together.

The supported web browsers make use of the same virtual machine that executes JavaScript to execute the WebAssembly instructions and provides the logic for allowing WebAssembly to call into JavaScript and vice-versa [18].
JavaScript and WebAssembly do not have direct access to each other's memory and can only pass primitive types such as integers as parameters or receive these as a result of function calls. They do however share a shared memory area to store more complex types such as strings, which then could be referenced by a pointer and passed around as it is represented as a primitive integral type.

7 What are the pros and cons of WebAssembly when compared to JavaScript?

The most notable advantage of WebAssembly in comparison to JavaScript is performance. It is correct that by leveraging WebAssembly in a project the possibility of getting greater performance does exist, but that does not mean that this is always the case.

7.1 Significant gains in parsing performance

WebAssembly has significant gains in terms of parsing performance compared to JavaScript, as cited from the website of WebAssembly:

> *The kind of binary format being considered for WebAssembly can be natively decoded much faster than JavaScript can be parsed (experiments show more than 20× faster). On mobile, large compiled codes can easily take 20–40 seconds* just to parse*, so native decoding (especially when combined with other techniques like streaming for better-than-gzip compression) is critical to providing a good cold-load user experience.*
>
> - Web Assembly FAQ [6]

This means that WebAssembly can be understood faster by the browser as it does not have to parse JavaScript, then create a tokenized format of the code which could then be translated into actual machine code. JavaScript also is a dynamically typed language; therefore, the types of variables might not be known ahead-of-time by the compiler. The compiler has to guess what the types of each variable will contain in order to efficiently store the data, if the data could not be inferred ahead-of-time, it might lead to less efficient machine code compared to what WebAssembly could offer considering there is no guessing involved [18].

7.2 Ability to choose any programming language that has WebAssembly as a compilation target

The web has only been supporting JavaScript as the only major language for a very long time, and with WebAssembly, this changes. Because WebAssembly can be used as a compilation target, any language that supports it can now be used for web development. This enables developers that have knowledge of other programming languages, such as Java to be able to target the web or any other platform that supports WebAssembly as well.
Besides that, certain programming languages might have concepts or advantages that JavaScript or other languages targeting the web are just not be able to offer [18].

7.3 The ability to leverage existing libraries or code bases in other programming languages

Because WebAssembly allows for the use of other programming languages, it also means that libraries that were written in the other programming languages also become available for the use in web development, as long as these libraries are written in a platform agnostic manner. The same advantage can be used to target the web using existing code bases written in languages that support WebAssembly as a compilation target, or to even transform a whole desktop class application such as AutoCAD to the web with relative ease [7].

7.4 WebAssembly has better portability

Because WebAssembly is a standardized binary instruction format, it will be able to run on any platform that supports it. This would be similar to technologies such as the .NET Framework or Java Runtime Environment (JRE), where code that was written in languages like C# or Java would be compiled into Common Intermediate Language (CIL) or byte code [19]. By using CIL or byte code instead of machine code, the produced binary would not be reliant on the platform it is running on, as the interpreter or Just-In-Time (JIT) compilation of the byte code would be able to efficiently translate each instruction into the machine code that could be executed on the platform. Whereas a binary produced targeting an AMD64 platform would not be able to run on a platform that only supports i386 or ARM. This essentially means that any language that supports a WebAssembly interpreter is able to call into the WebAssembly code from the foreign language, whereas in comparison to JavaScript you would need to spin up an entirely different runtime and glue layer to consume the functions it exposes, which comes with performance drawbacks and other difficulties, such as multi-threading.
Whilst JavaScript has seen good support by web browsers on various different platforms, and since recently also in 'native' mobile software development by technologies such as React Native, interoperability between other languages or platforms leaves much to be desired compared to what WebAssembly could offer [20].

7.5 Increased runtime performance

The runtime performance of an implementation in WebAssembly compared to JavaScript seems to almost always be better based on a set of benchmarks that can be found online [21]. The performance increment may however vary depending on what kind of task should be performed.
There are cases where an implementation in JavaScript outperforms the implementation that is targeting WebAssembly, most notably in scenarios where the benchmark uses a small array to compute a number. What should also be taken into account is that creating small WebAssembly functions that are being used in hot paths are not ideal, this introduces call overhead which can take a significant hit at performance as well [22].

WebAssembly also supports multi-threading by a feature named WebAssembly Threads to potentially increase performance of an application [23], as it allows for parallel computation. JavaScript by itself is a single-threaded language, that relies on a construct called the event loop to perform asynchronous tasks. However, asynchrony in JavaScript does not mean that code is executed in parallel. The event loop only allows for concurrency to process more than one tasks at a time. For this reason, most modern-day browsers support a feature called Web Workers [24]. Web Workers are threads that can perform tasks that do not interfere with the user interface. They are able to communicate via message passing between the main JavaScript thread or other Web Workers and allow for parallelization of JavaScript code in the browser. However, Web Workers do not share mutable data between them, whereas WebAssembly Threads are able to do so, reducing overhead that is introduced by message passing between threads. Further increasing the potential runtime performance over what JavaScript is able to offer.

7.6 Smaller download sizes

Because WebAssembly is usually downloaded and interpret in a binary format, the size of the binary file is almost always smaller than the equivalent JavaScript implementation which

would be shared in text format. This is possible because the optimization is done ahead of time. However, there is a big catch, it does also depend on what language or compilation parameters that were used. For example, an implementation of Fibonacci in Rust compiled to a WebAssembly binary will be far smaller than the same implementation written in Go to a WebAssembly binary. This occurs because Rust does not require a runtime to function, unlike Go. Depending on project size, and the chosen technology that targets WebAssembly, this might be an advantage or not [25].

7.7 Conclusion

WebAssembly as a technology offers significant benefits for the web as JavaScript simply cannot offer the same performance or benefits by the nature of the language. It enables web developers to bring entirely new experiences to the web that simply were out of reach beforehand.

8 Why WebAssembly might not be the solution for everything

8.1 No direct access to the DOM

WebAssembly is faster than JavaScript, but not in every use case. If WebAssembly wants to access or manipulate the DOM, it has to ask JavaScript to do so on its behalf. This introduces extra overhead which may slow down the performance of WebAssembly significantly [22]. Therefore, WebAssembly usually sees more use in compute intensive tasks that are being offloaded from JavaScript. When looking at a framework such as Blazor, they provide an API called JS Interop in order to access the DOM.

8.2 Slow interoperability between JavaScript and WebAssembly

Similar to having no direct access to the DOM, calling functions from either direction (JavaScript to WebAssembly and WebAssembly to JavaScript) introduces some overhead. If these calls are frequent, such as a hot path, this could introduce a significant performance hit. For this reason, these constructions are usually avoided; instead, calls are made to instruct the WebAssembly module to start or stop a process which writes to a shared memory area. This same memory area can then be directly accessed by JavaScript, avoiding the overhead [26].

8.3 Development requirements

One of the downsides of WebAssembly is that it requires the types of variables to be determined at compile time. Therefore, languages that are unable to provide these types during compilation, are unable to support WebAssembly. Some of the most popular languages are dynamically typed, such as JavaScript and Python 2. They are widely adopted and known by developers, but are unable to create WebAssembly modules with due to the lack of type information [27]. There however still is the possibility to use dynamically typed languages within WebAssembly, but that would require the runtime for these languages to be compiled to a WebAssembly module and the code to be interpret at runtime, removing some of WebAssembly's main advantages. This means that to use WebAssembly in a project, there are specific developer requirements to be able to write code that compiles to WebAssembly. This also means that developers that are only proficient with frontend development using technologies such as JavaScript, HTML, and CSS, are unable to create WebAssembly modules with their current skillset.

8.4 Not as performant compared to native binaries

WebAssembly is usually more performant than its JavaScript equivalent implementation, however compared to native binaries, it is usually less performant. WebAssembly is an intermediate language, which requires the binary format to first be compiled down to native machine code. This process takes time, and therefore reduces the speed at which the binary would be executed.

Compiling to native binaries also allows for optimizations that are not available for the use in WebAssembly (yet).
For example, a feature called Single Instruction, Multiple Data (SIMD) has the ability to boost performance by exploiting data level parallelism [28]. This feature is available for use when

compiling to native binaries, but as of writing this document is still in development stage for WebAssembly by major browsers such as Microsoft Edge and Mozilla Firefox [29].

8.5 Harder debugging and more complexity

By using WebAssembly next to another language, a new layer of complexity is introduced. It is yet another thing that has to be managed by the developer, which may require extra setup for techniques such as CI/CD or extra code to glue the two isolated modules together.
WebAssembly itself is also a less human-readable format than JavaScript, even when viewed in its textual form. This makes debugging harder as the developer will require extra tooling to understand what the WebAssembly instructions are, and even if they do have the required tooling to do so, without the embedded debug information the code is still harder to understand than JavaScript in most scenarios.
Google Chrome does have support for debugging WebAssembly under experimental features using DWARF, a debugging information format [30].

8.6 Conclusion

WebAssembly does not suit every project. It increases project complexity, increases developer requirements or simply is unable to perform well on the web with currently available APIs.

9 What is the security of WebAssembly like in comparison to JavaScript?

Because WebAssembly allows for the execution of code on client's devices, other aspects like security have to be taken into consideration as the code to be executed might have malicious intentions. When looking at the security of web technologies from the past there have been some incidents when it comes to security in the browser; namely Java-applets or Adobe Flash were a big concern in terms of security [31].

9.1 Sandboxing

All modern-day browsers use a technique called sandboxing, where each website that is visited within the browser has its own process and set of functionalities to play with, hence the name 'Sandbox'. It is an enclosed environment where the access to features is restricted so that the application is not allowed to access certain information or APIs. JavaScript and WebAssembly both run on the same runtime that is being isolated by a sandbox specific to each browser window, therefore WebAssembly has exactly the same security footprint as JavaScript has.

In contrast to other web technologies such as Adobe Flash, which is loaded in a separate process by the OS, will execute the code in it's own process therefore adding a potential security risk within the Adobe Flash itself. This means that if a vulnerability exists for Adobe Flash, a malicious piece of code written for Flash could bypass the secure environment of the browser by exploiting the separate process and therefore being able to access the victims machine. The features Adobe Flash or Java-applets offered for web applications are nowadays mostly able to performed by just using HTML5. For this reason and the fact that there is a big security risk within these solutions, all modern-day browsers disable support for these technologies by default.

9.2 Differences between the operating system and browser in terms of security

The main way of accessing the world wide web is by using the browser. By using the browser you are able to access any website that exists within the network, which also includes websites that have malicious intent. Because the browser simply downloads the content of the server when it is requested, that could include JavaScript or other executable code, it is very important to have good security in place when the code is being executed.
In comparison to an operating system, much more of the code that is being executed is done in a controlled and trusted environment. All non-trusted software that could be ran on the system has to be downloaded by the user, accepted through a prompt from the operating system, and then executed. And even then it is more likely that the operating system has a built-in virus scanner such as Windows Defender which implicitly scans, unless explicitly disabled by the user, the external software before being ran. Therefore, it is very important that the browser has even better security measures in place than the operating system, as it poses a bigger threat in terms of the potential accessibility of the code that has malicious intent.

Due to these concerns, the APIs and information that is available from within JavaScript and WebAssembly is very limited by default. If the application wants to access some more features the browser has to ask the user for permission on the behalf of the application, such as when

the application requires access to the microphone for speech-to-text. This in combination with the sandboxing technique browsers use to isolate the code in different processes creates a safe environment where even malicious code is not able to do much unless the user explicitly gives permission to do so.

Interesting to mention is that Windows in combination with Windows Defender also took some inspiration from the web when it comes to creating a more secure environment when executing software written by third parties. For example Windows Defender also uses a sandboxing technique to execute the potentially malicious software to analyze the actions the software performs before allowing it to be actually ran on the host machine, where it has access to all of the real information [32].

10 What are the preconditions of the environment for executing WebAssembly?

WebAssembly is an assembly language for a conceptual machine. This means that any environment that can virtualize this conceptual machine, is able to run the WebAssembly code. WebAssembly requires a system interface in order to interact with the physical machine it is running on and is usually provided by a virtual machine used in the supported web browsers. But even though WebAssembly was designed to run on the web, the W3C working group also found it desirable to be able to execute WebAssembly in other environments. Therefore, as of the time of writing this document, an average of 92.61% of users use a web browser that supports WebAssembly [5].

The web browsers that support WebAssembly are as following:

Browser	Total usage globally	Supported since
Edge	3.61%	17 October 2017 – version 16
Firefox	3.28%	7 March 2017 – version 52
Chrome	26.55%	9 March 2017 – version 57
Safari	3,93%	19 September 2017 – version 11
Opera	0,88%	21 March 2017 – version 44
Safari on iOS	13,58%	19 September 2017 – version 11
Opera Mobile	0,01%	16 February 2021 – version 62
Chrome for Android	37,03%	11 March 2021 – version 89
Firefox for Android	0,29%	23 February 2021 – version 86
Samsung Internet	3,2%	7 June 2018 – version 7.2
QQ Browser	0,22%	19 May 2020 – version 10.4

Table 1: WebAssembly support of major browsers

Web browsers that do not support WebAssembly are:

- Internet Explorer
- Opera Mini
- UC Browser for Android
- Baidu Browser
- KaiOS Browser

Please note that current versions of the Edge browser are based on Chromium instead of Microsoft's own engine EdgeHTML. Chromium is an open-source web browser project that serves as the base for Chrome and many other browsers [33]. Features such as WebAssembly are part of the engine which lives in the Chromium project, therefore a browser like Chrome will have a similar if not identical API or support for WebAssembly as Edge does.

10.1 WebAssembly System Interface

WebAssembly System Interface (WASI) is a standardization of a system interface for WebAssembly [34]. A system interface is an abstraction that forwards the calls from one system to the platform specific implementation. Using this technique WASI exposes an API similar to the API of the operating system, but instead of directly implementing these

functions, it will translate these calls into operating system calls based on the underlying operating system.

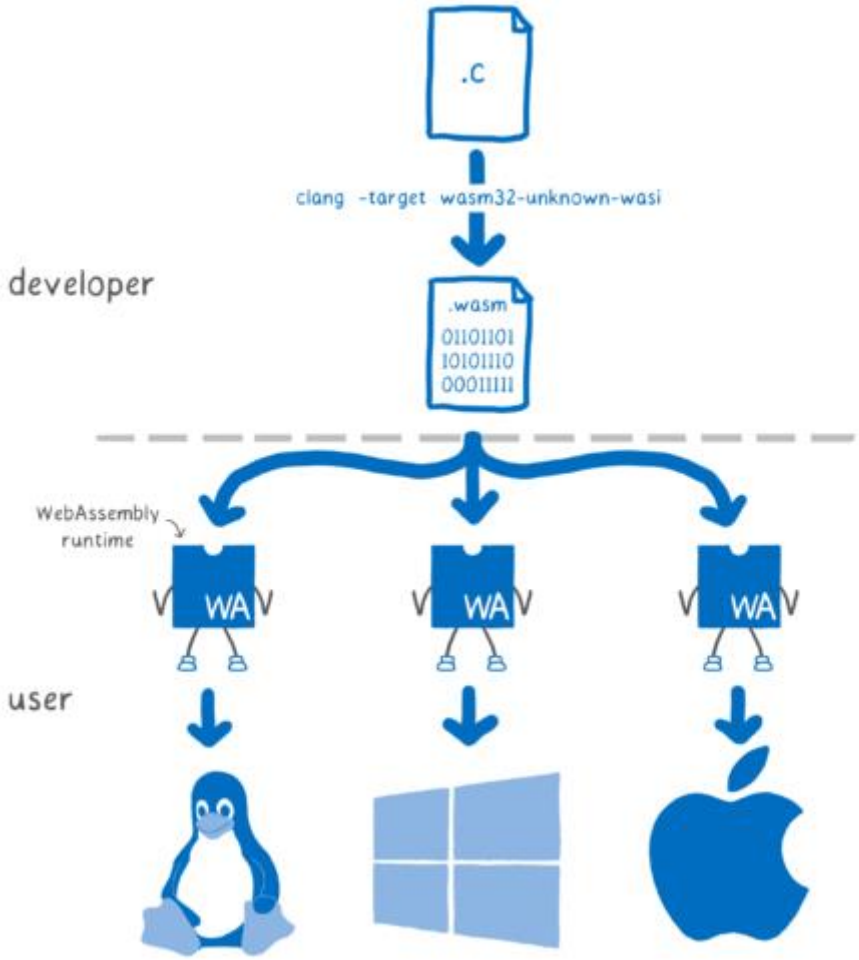

Figure 2: An example of the portability of compiling WebAssembly to target WASI [34].

As shown in Figure 2, the compiled WebAssembly file targeting WASI will call into abstractions depending on the underlying operating system.

Because WASI is just a standardization, multiple implementations exist for this standardization, such as Wasmtime and Lucet, both being developed by the Bytecode Alliance. The Bytecode Alliance is an open-source community dedicated to creating secure new software foundations [35]. Their founding members consists of Mozilla, Fastly, Intel and Redhat. There also exists a browser Polyfill for backward compatibility with browsers when targeting WASI.
Lucet is a native WebAssembly compiler and runtime. It has been designed to safely execute untrusted WebAssembly code inside your application [36].

10.1.1 Wasmtime

Wastime is a project by Bytecode Alliance that is a standalone wasm-only optimizing runtime for WebAssembly and WASI [37]. It is used to run WebAssembly code outside of the web and can be used both as a CLI utility or as a library that can be embedded in a larger application. Wasmtime has support for multiple languages through embeddings of the implementation:

- Rust

- C
- Python
- .NET
- Go

10.2 Conclusion

WebAssembly is able to run on most devices that include a modern web browser at the time of writing this document. However, WebAssembly is not limited by the web, as there also exists other standardizations that allow WebAssembly to be run on any devices or platform where this standardization would be implemented for, creating a very portable solution.

11 What are the performance differences of using WebAssembly versus an implementation in JavaScript?

In order to create these tests from multiple programming languages, a few tools were leveraged to create the needed WebAssembly modules from their respective source code as well as some JavaScript libraries/functions to measure the performance differences between different implementations.

The WebAssembly modules were built on the same machine the benchmarks were performed on.

To ensure that WebAssembly is able to execute at optimal performance all of the tests have been written in C/C++ as these two languages are known for having good performance characteristics.

LLVM has been used to compile the C/C++ source code into WebAssembly bytecode.
Most sources online reference the use of Emscripten to build WebAssembly bytecode from C/C++ source code. For the benchmarks in this research Emscripten was not leveraged as the features Emscripten provides are not required for the benchmarks and would only add bloat to the output. Emscripten aims to be a drop-in-replacement for a C/C++ compiler that enables code that was initially not meant to target the web to be able to run on the web. It does this by emulating an entire POSIX operating system where it is needed. Emscripten can intelligently extract what POSIX compatible features the C/C++ source code depends on and emulate the required feature in the browser. For example, if the source code depends on OpenGL, Emscripten will bundle code that creates a C-compatible GL context backed by WebGL. This requires a lot of work and more data that has to be send to the end user.
Because the benchmarks are rather simple, but yet compute intensive tasks, these compatibility features Emscripten provides are not required [38].

11.1 Steps to compile C to WebAssembly

Compiling C to WebAssembly starts of by compiling the source into an intermediate representation (IR), in this case LLVM IR. This can be done by using Clang, a language family frontend for LLVM. Clang includes a compiler for the C language family (C, C++, Objective C/C++, OpenCL, CUDA and RenderScript) [39]. MacOS comes bundled with Clang and should be available on other platforms as well. Since version 8 of LLVM the WebAssembly target is available by default.
In order to compile the source file into LLVM IR the following command can be executed:

```
clang --target=wasm32 -c -Os INPUT_FILENAME.c
```

We specify the target to be WASM 32 bit as well as the `-c` flag in order to tell Clang to just perform the compilation step. The `-Os` flag is used to tell the compiler to optimize the code for speed.
If the compilation is successful, an object file with the same name as the input filename will be produced with the `.o` extension. Next up is using `wasm-ld` to link the object file into an executable. `wasm-ld` is part of LLVM's linkers. Linking is performed by executing the following command:

```
wasm-ld --no-entry --export-all -o OUTPUT_FILENAME.wasm INPUT_FILENAME.o
```

The `--no-entry` flag is set because the source file does not contain an entry point (it is just a library with exposed functions). The `--export-all` flag is set to export everything exposed by the source file and the `-o` flag indicates where the linked output should be stored.
If the linking was successful a `.wasm` file should have been produced that contains the functions on the export table. Using this file, a WebAssembly runtime can instantiate the module and expose the functions to the codebase.

11.2 Benchmarks

To be able to benchmark the JavaScript and WebAssembly implementation a NPM package by the name of Benchmark.js (version 2.1.2). Benchmark.js has a suite of functionalities to effectively benchmark the implementation that supports high-resolution timers and consistently returns significant results [40].
All of the benchmarks provided were performed on a baseline MacBook Pro 2019 16" running MacOS Big Sur (version 11.1). The only applications that were used during the benchmark are JetBrains IntelliJ IDEA version 2020.3.3, iTerm2 (build 3.4.4), Google Chrome (version 89.0.4389.90 official build x86_64) and Mozilla Firefox (version 87.0 x86_64).

The benchmarks were performed on Google Chrome, Mozilla Firefox, and Safari.
These browsers were chosen because they cover almost all of the existing userbase based on the data as shown in Table 1 as well as they are all based on three different underlying engines. Google Chrome is powered by Chromium; Mozilla Firefox is powered by Gecko and Safari is powered by WebKit.
All benchmarks were performed without the developer tools open, as having this window open could drastically to affect performance of the browser. When benchmarking Safari with developer tools open there was a 5x slowdown on JavaScript performance compared to when it was closed.

All of the benchmarked results are based on a sample size that has less than 2.5% variation in total.

11.2.1 Benchmark 1: Fibonacci numbers

Fibonacci numbers is a function that forms a sequence, called the Fibonacci sequence, where the number that the function was invoked with is the sum of the two preceding ones, starting from 0 and 1 [41].
This function has a computational complexity that increases exponentially based in the input that is given, making it a good candidate for testing different computational complexities. This characteristic allows for easy testing that the influence of call overhead might introduce from calling the WebAssembly function a varied number of times from JavaScript.
The function is also recursive, which can be an interesting metric compared to what other benchmarks will cover.

Both implementations are simple and similar, the only real difference being that the C implementation requires type annotations.

```
const fibonacciJS = (n) => {
    if (n <= 1)
        return n;
    return fibonacciJS(n-1) + fibonacciJS(n-2);
};
```
Snippet 2: Implementation of Fibonacci in JavaScript.

```
int fibonacci(int n) {
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```
Snippet 3: Implementation of Fibonacci in C.

For this specific benchmark there is also a comparison made between optimized and non-optimized WebAssembly produced by Clang. There is a significant difference between the optimized and non-optimized variants of the WebAssembly module.

The result of the benchmark concludes a few interesting things when comparing the browsers:

- The JavaScript performance is worse compared to the optimized WebAssembly on every browser that was benchmarked, however the same cannot be said for the non-optimized variant of WebAssembly. The performance Google Chrome offers for non-optimized WebAssembly is very poor compared to JavaScript.
- WebAssembly executed on Mozilla Firefox always performs better than JavaScript executed on Mozilla Firefox in all of the benchmarks.
- Safari is the most potent browser in terms of performance for this particular benchmark.

This benchmark also shows the performance decrease that call overhead introduces when calling a WebAssembly function from JavaScript. Invoking `fibonacci(5)` is far less computationally demanding than invoking `fibonacci(30)`, which can be determined by the number of operations per second as shown below. Each time an operation is performed a call has to be made between the two environments which incurs some performance loss.
This becomes very obvious when comparing the operations per second from `fibonacci(5)` with `fibonacci(10)`. As the number of operations decreases the performance potential of WebAssembly increases as there is less overhead.

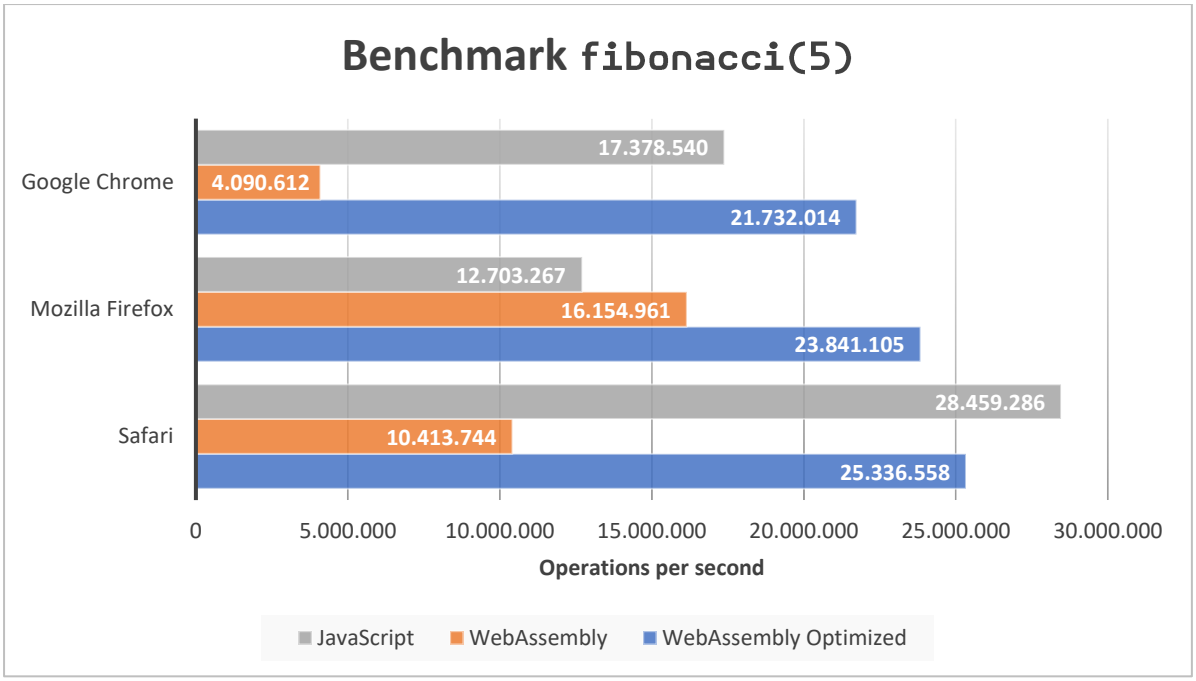

Figure 3: Benchmark results of invoking Fibonacci with input 5.

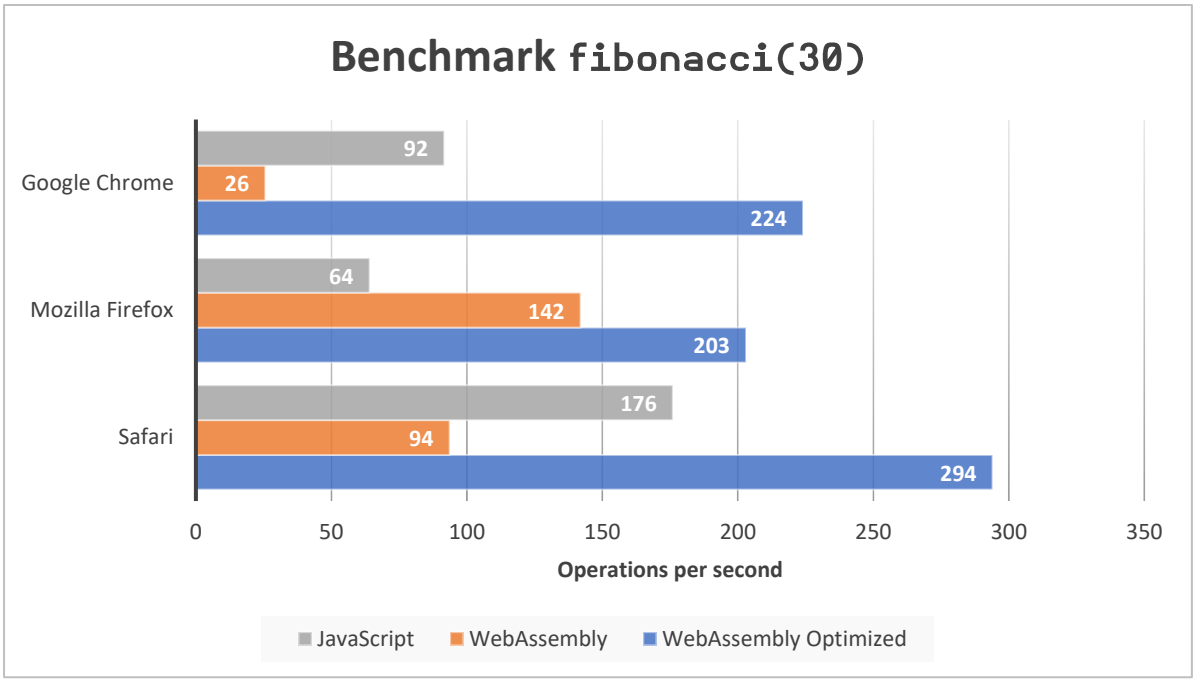

Figure 4: Benchmark results of invoking Fibonacci with input 30.

11.2.2 Benchmark 2: Grayscaling an image

This benchmark was performed in order to test performance when an array of different sizes is passed to the function that is being accessed as well as mutated during the invocation. In order to grayscale an image, every pixel needs to be read to acquire the underlying RGB values. These values will then be used to calculate the new value that will be assigned to all of the colors using the function `grayscale = (R + G + B) / 3`.

The JavaScript Canvas API that exposes the pixels and their respective RGB values are stored in the RGBA format. Grayscaling does not require the alpha (A) value of each pixel, but it still

takes up a single extra byte for each pixel. Therefore, each pixel will require four bytes to be stored in memory.

The object file produced by Clang has been linked using `wasm-ld` with an extra flag named `--initial-memory` to set up the required memory needed to store an image of resolution 3840x2160. The size of memory that was passed to this flag is based on the resolution multiplied by the number of bytes each pixel needs to store the values like so `3840 * 2160 * 4 = 33,177,600`. However, this exact size was not allowed by `wasm-ld` as the memory had to be 65536-byte aligned. For this reason the required memory size was extended to match the alignment requirement which was calculated like the following `(floor(33,177,600 / 65536) + 1) * 65536 = 33,226,752`.

The final command that was used to link the object file is as following:

```
wasm-ld --no-entry --export-all --initial-memory=33226752 -o
grayscalingimage.wasm grayscalingimage.o
```

```
const grayscale = (imageArr, width, height) => {
    const iterations = width * height * 4;
    for (let idx = 0; idx < iterations; idx += 4) {
        const r = imageArr[idx];
        const g = imageArr[idx + 1];
        const b = imageArr[idx + 2];

        const grayscaled = (r + g + b) / 3;
        imageArr[idx] = grayscaled;
        imageArr[idx + 1] = grayscaled;
        imageArr[idx + 2] = grayscaled;
    }
};
```

Snippet 4: Grayscale function implemented in JavaScript.

```
void grayscale(char* buffer, int width, int height) {
    int iterations = width * height * 4;
    for (int idx = 0; idx < iterations; idx += 4) {
        char r = buffer[idx];
        char g = buffer[idx + 1];
        char b = buffer[idx + 2];

        char grayscaled = (char)((
            ((int)r) +
            ((int)g) +
            ((int)b)
        ) / 3);
        buffer[idx] = grayscaled;
        buffer[idx + 1] = grayscaled;
        buffer[idx + 2] = grayscaled;
    }
}
```

Snippet 5: Grayscale function implemented in C.

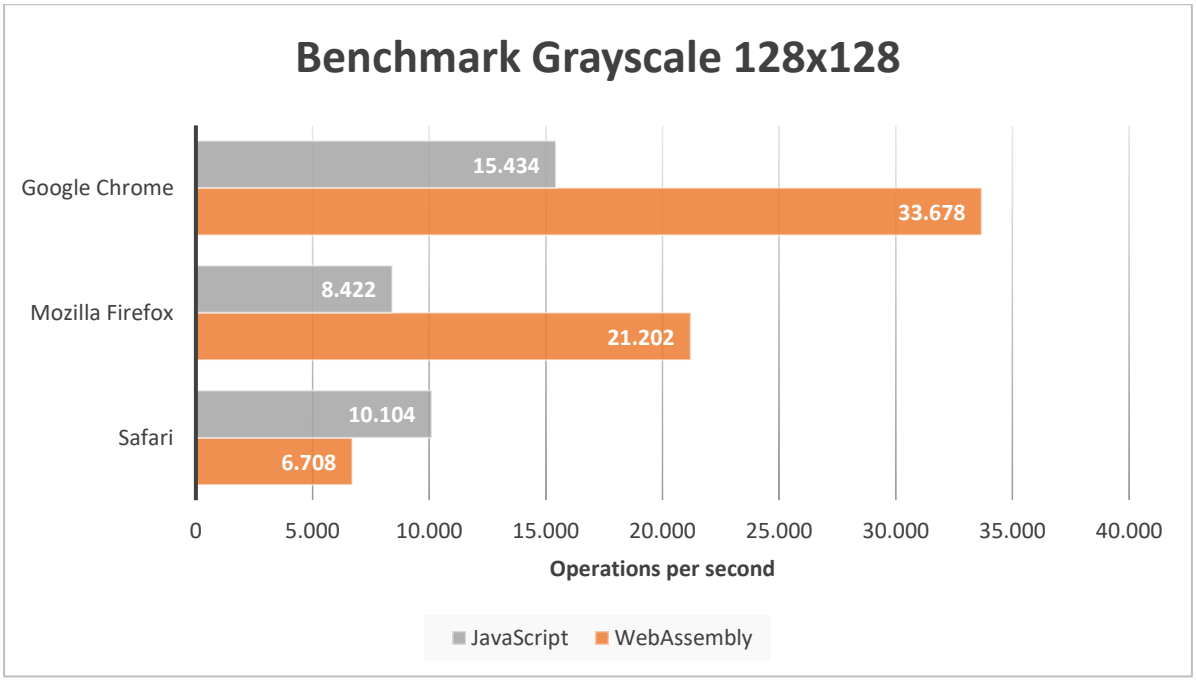

Figure 5: Benchmark results of invoking the grayscale function on an image of the resolution of 128x128 .

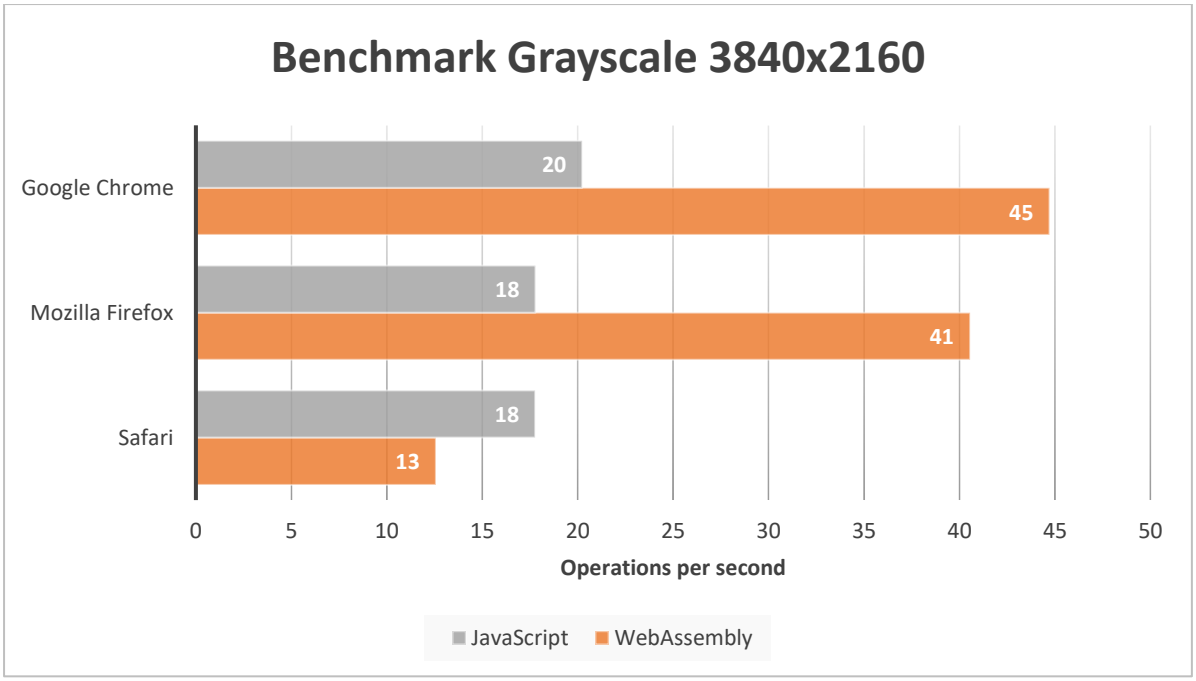

Figure 6: Benchmark results of invoking the grayscale function on an image of the resolution of 3840x2160.

11.2.3 Benchmark 3: Hashing using SHA256

To benchmark SHA256 two different implementations were acquired for JavaScript as well as C. For the JavaScript benchmark the core operation was part of a NPM package by the name of 'js-sha256', the only thing that was added was a small wrapper function in order to be consistent in comparison to the other benchmarks as shown in Snippet 6.

```
const sha256JS = (input) => {
    return sha256(input);
};
```

Snippet 6: File contents of benchmark.js which shows the single invocation that is made to the function exposed by js-sha256

The WebAssembly benchmark however required some more time to set up. To start off an implementation had to be found that was written for C and that did not rely on the standard library (`stdlib.h`). The standard library is not available when directly targeting WebAssembly, as access to e.g., the filesystem is not available. An implementation was found as a GitHub gist [42] that was written by the GitHub user bellbind [43]. The source of the executable that was part of this solution however did rely on functions such as `fopen` in order to read files to generate their respective SHA256 checksum. Therefore, this implementation has been used to only extract the computational logic from the source and has afterwards been combined with a function that only requires a buffer as an argument instead of relying on the filesystem as shown in Snippet 7.

```c
#include <stdint.h>
#include "sha256.h"

void sha256(char* inputBuffer, int inputLength, char* outputBuffer) {
  struct sha256 hash;
  sha256_init(&hash);
  sha256_update(&hash, inputBuffer, inputLength);
  sha256_digest(&hash, outputBuffer);
}
```

Snippet 7:File contents of benchmark.c which shows what invocations are made to the sha256 library.

This implementation however did not link using the commands listed in the steps to compile C to WebAssembly because it required access to the `memset` function that is usually provided by the C standard library. The `memset` function was required because the struct definitions in the C source code of the file `sha256.c` internally leverages this function during runtime to initialize the memory of the structure. Fortunately, an article was found that tackles this specific issue using WASI [44]. By using WASI it is possible to still target the standard library, which includes the `memset` function, even when running on the web. In order to do so, Clang has to be passed a few extra parameters on where it can find the other standard library to depend on. The C source code has been compiled and linked using the following command:

```
clang --target=wasm32-wasi -Os --sysroot $HOME/wasi-libc -nostartfiles -
Wl,--no-entry -Wl,--export=sha256 -Wl,--initial-memory=33226752 -o
benchmark.wasm sha256.c benchmark.c
```

Snippet 8: The command with arguments used to compile the WebAssembly module.

This command shown on Snippet 8 has a few different parameters that are interesting to note:

- The target parameter has been set to `--target=wasm32-wasi` instead of `--target=wasm32` to indicate that WASI should be used as the backend to compile to.
- The sysroot parameter has been added by setting `--sysroot $HOME/wasi-libc` to select a cross-compiler [45], in this case the one provided by WASI.
- The `-nostartfiles` flag indicates that Clang should not include default files (from e.g., the standard library).
- Every parameter prefixed by `-Wl,` indicates that the following value after that should be directly passed to `wasm-ld`.

The first benchmark is generating the SHA256 checksum of the input string ‘test’ and the other benchmarks are of generating the SHA256 checksum of input strings containing the character ‘A’ repeated n amount of times as shown in Snippet 9.

```
const SHA256_SIZE = 32;

const addBench = (name, input) => {
    suite.add(`sha256 javascript '${name}'`, () => {
        let data_copy = input.slice(0);
        sha256JS(data_copy);
    });
    suite.add(`sha256 WebAssembly '${name}'`, () => {
        const outputBuffer = new
Uint8Array(wasm.instance.exports.memory.buffer, 0, SHA256_SIZE);
        const inputBuffer = new
Uint8Array(wasm.instance.exports.memory.buffer, SHA256_SIZE + 1,
input.length);
        inputBuffer.set(input);
        wasm.instance.exports.sha256(inputBuffer.byteOffset,
input.length, outputBuffer.byteOffset);
    });
};
```

Snippet 9: Definition of the benchmarks which shows the invocation of the exposed functions.

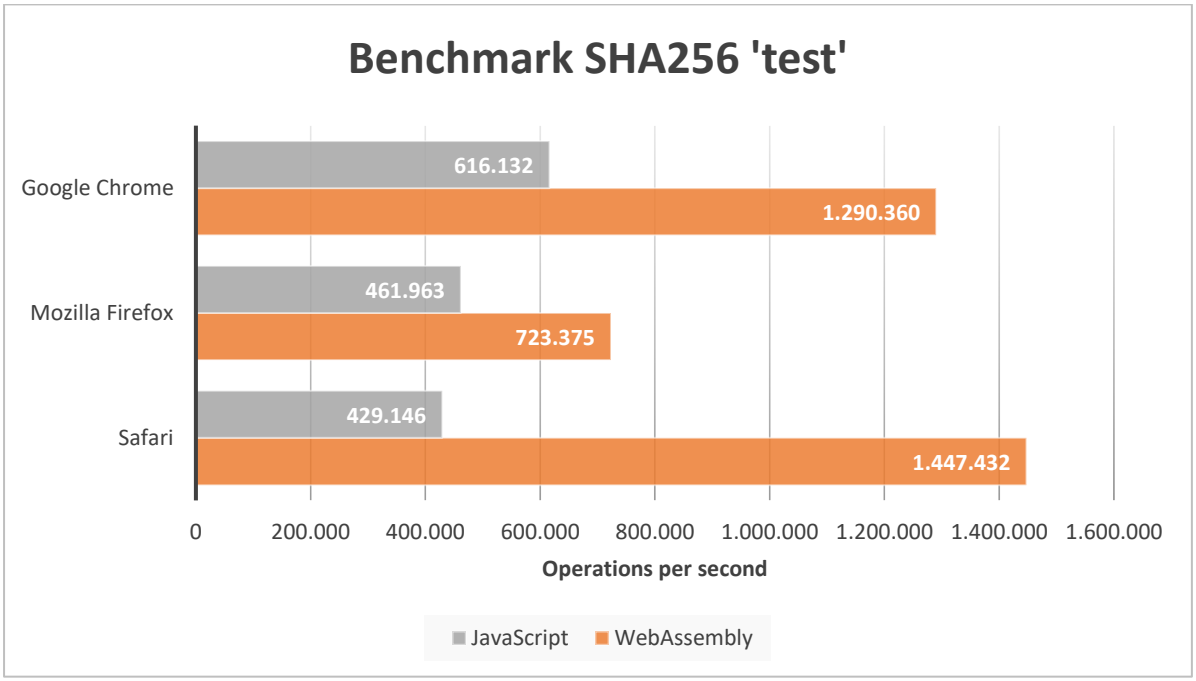

Figure 7: Benchmark results for generating a SHA256 checksum on the input string 'test'.

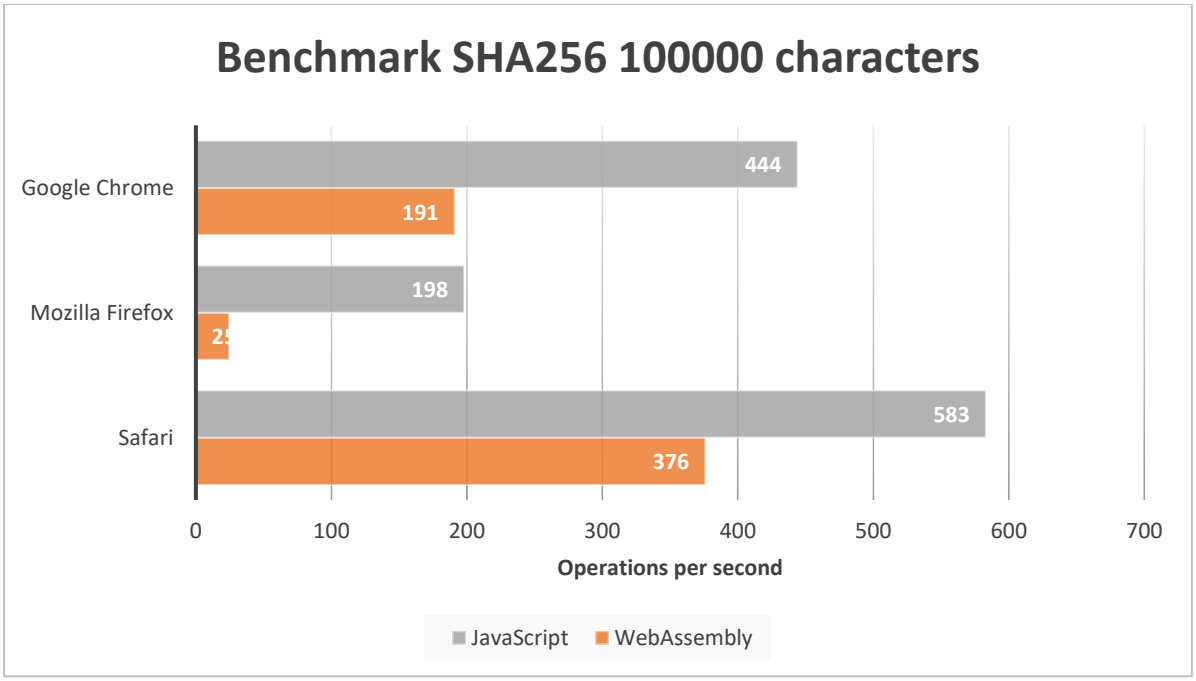

Figure 8: Benchmark results for generating a SHA256 checksum on an input string of the character 'A' repeated 100000 times.

11.3 Conclusion

WebAssembly is able to increase the performance of certain application workloads by a large amount when compared to a similar implementation in JavaScript, even when not testing other possible performance beneficial features such as multi-threading. This however is not always the case for every scenario, as some workloads, especially those where frequent calls are performed between the two environments, can create overhead which would completely eliminate the performance gain WebAssembly would provide when compared to JavaScript.

12 What are the performance differences between the available programming languages and frameworks for the same implementation?

In the previous chapter a single programming language, C, was leveraged in order to develop three benchmarks to showcase performance differences when comparing JavaScript to WebAssembly. However, WebAssembly is not just developed using only C, in fact there are a wide variety of choices of programming languages which all have their own trade-offs [46].
To compare these differences, three other programming languages were chosen that all have their own advantages and disadvantages. These languages had to conform to a few requirements; they either had to have a wide adoption in the software development market, had to be used extensively by Quintor or had to have unique features and advantages over other possible languages. Another requirement is that the language also should have good support for WebAssembly, as not every available language has matured support for the target yet. An exception was made for Java, as it does not yet have a stable API. However, as the language is widely adopted by Quintor, it does still seem relevant to explore and document what it can offer at the time of writing this document compared to the other languages.
To compare the performance differences between each language, an implementation of Fibonacci was written and compiled to WebAssembly. These implementations were then benchmarked using the same procedure as the previous chapter.

12.1 C# (Blazor)

C# is a modern, object-oriented, and type-safe programming language actively developed and maintained by Microsoft [47]. C# compiles into bytecode that can be executed by the .NET runtime, which means that when compiling C# to target WebAssembly, a runtime is required to execute the code.
As C# is a relatively mature language and the fact that it runs on the .NET framework, it also gains access to leverage a wide variety of libraries written in languages such as F# and Visual Basic. Quintor also has expertise in developing software that targets the .NET framework, which therefore means that developers may have an easier time writing software to target WebAssembly.

Blazor is a young framework that allows the developer to build interactive web user interface using C# instead of JavaScript. Using Blazor it is possible to develop the entire application, being frontend as well as backend, just using C#, HTML and CSS.
Alternatively, Blazor can also run the client-side logic on the server. In this scenario all UI events are being sent to the server using a real-time messaging framework, and after processing these events changes are sent back to the client to modify the DOM accordingly.
Blazor also allows for interoperability with JavaScript, which enables access to the existing libraries for JavaScript whilst still writing the logic in C#.
Blazor is based on a UI component ecosystem, similar to what React and Angular offers.

12.2 Rust

Rust gives programmers low-level control and reliable performance [48]. Unlike JavaScript and many other languages (like C# and Java), it does not require a garbage collector to free allocated memory; instead, it frees the memory when it goes out of scope, therefore allowing

for a much more deterministic performance footprint. Rust also provides constructs to control indirection, monomorphization and memory layout.

Rust lacks a runtime; this enables small binary (`.wasm`) file sizes because there does not need to be included extra bloat such as a garbage collector to function. Rust co-exists well with existing JavaScript code in the sense that it can easily be added to a project and used to port performance-sensitive code over to Rust, whilst not leaving the rest of the codebase as is. Rust also integrates well with existing tooling from JavaScript, such as NPM and Webpack.

Rust is unique in the sense that it brings in the ownership model which can guarantee memory-safety and thread-safety at compile time. These guarantees do no exists for comparable languages such as C/C++ in terms of speed and are usually the reason for the largest number of bugs that exists in code bases for these languages [49].

Rust also scores as the most loved language based on the StackOverflow Developer Survey that is held each year for five years running [50].

Because Rust is still a relatively young language, not many (stable versioned) libraries exist when compared to more mature languages such as C# or Java. Rust also has a steeper learning curve, making it more difficult for new developer to pick up the language and become proficient in using them in a timely manner.

12.3 Java

Java is a programming language and computing platform first released by Sun Microsystems in 1995 [51]. Java compiles into bytecode that can be executed by the Java Runtime Environment (JRE), therefore when a runtime is required when executing Java in WebAssembly.
Just like C#, Java is a mature language with a large number of libraries that can be leveraged when compiling to WebAssembly. Quintor has a lot of expertise in software development using Java, which is why it could potentially be a key technology for applying WebAssembly in current and future projects.

Java has a few interesting compilers that can transform simple functions to entire applications to the web. JWebAssembly is a compiler that uses Java class files as input, meaning that any language that compiles to Java bytecode like Clojure, Groovy, JRuby, Jython, Kotlin and Scala are able to target WebAssembly. This compiler aims to do a one-to-one conversion between the Java class files to WebAssembly, therefore the generated WebAssembly will be similar in size to the original Java class files [52].

Another interesting compiler that specifically targets the web, named CheerpJ, aims to convert any Java client application into a HTML5, JavaScript and WebAssembly application. This compiler also only requires Java bytecode, meaning no actual source code is required. It offers a full, unmodified runtime environment based on OpenJDK which supports Swing for UI, Filesystem, Audio and Printing functionality. CheerpJ is free for non-commercial purposes [53].

13 What are the performance differences between JavaScript and WebAssembly within a prototype of the ESDEC Calculator?

In order to showcase the researched topic in a real-world application. A prototype has been developed that leverages WebAssembly to perform a compute intensive task that will be compared with a similar JavaScript implementation. Both of these implementations have been benchmarked so the results can be compared.
This prototype is based on the ESDEC calculator and contains a subset of the functionalities of that application.
The application consists of multiple layers within two separate applications that together enable the prototype to retrieve information that can be modified without redeploying the application known as the backend as well as a separate presentation application known as the frontend.

13.1 Quality requirements of the end product and preconditions

Quintor has no direct requirements for the use of a specific programming language or framework. The programming language and framework should be chosen based the conclusions of an interview with experts on frontend development where the results of exploratory research that has been performed will be discussed.
Therefore, in order to be able to define some of the requirements ahead of time, an interview with the product owner of the project has been conducted which concluded in the following requirements for the minimum viable product that should be realized:

13.1.1 Functional requirements

- Ability to draw a rectangular plane to be further used as the roof on which solar panel layout should be performed
- Ability to draw boxes as obstacles on the rectangular plane
- Variables that influence the computation of optimal layout such as wind direction, wind speed, type of surface, angle of the roof and different kinds of roofs should be modifiable

13.1.2 Non-functional requirements

- Provide a better user experience through the advantages that this new technology (WebAssembly) has to offer when compared to the current ESDEC application, e.g. greater speed when performing computationally demanding tasks

The realized solution should also come with a document that covers the technical documentation of the project, this includes information such as:

- The technologies that were leveraged to build the solution
- The chosen project architecture
- Alternative technologies and architectures

Each of these topics should also be substantiated as to why these were chosen when compared to others.

13.2 Backend

The backend application was built using C# on ASP.NET Core. The application is used to expose data of the calculation process for the frontend application. The data is being stored in a PostgreSQL database and is being accessed via EntityFrameworkCore. The frontend application is then able to talk to a GraphQL API by making HTTP calls to the backend application, to which the backend application will respond with the requested data.
However, the application architecture is designed in such a way that all of these dependencies such as the database, or exposed API layer are all abstracted away in different application layers. This means that the application depends on abstractions instead of concretes, which allows for a very flexible and testable application. The backend application exists of the following layers:

- **Domain**: includes all domain specific elements of ESDEC such as models and business logic
- **Application**: provides logic that interacts on domain elements and abstractions that the API and Infrastructure layer can depend on
- **Infrastructure**: provides logic that interacts with the database by implementing abstractions exposed in the Application layer
- **API**: the final executable project that depend on all of the other layers to expose the information outside of the backend applications domain

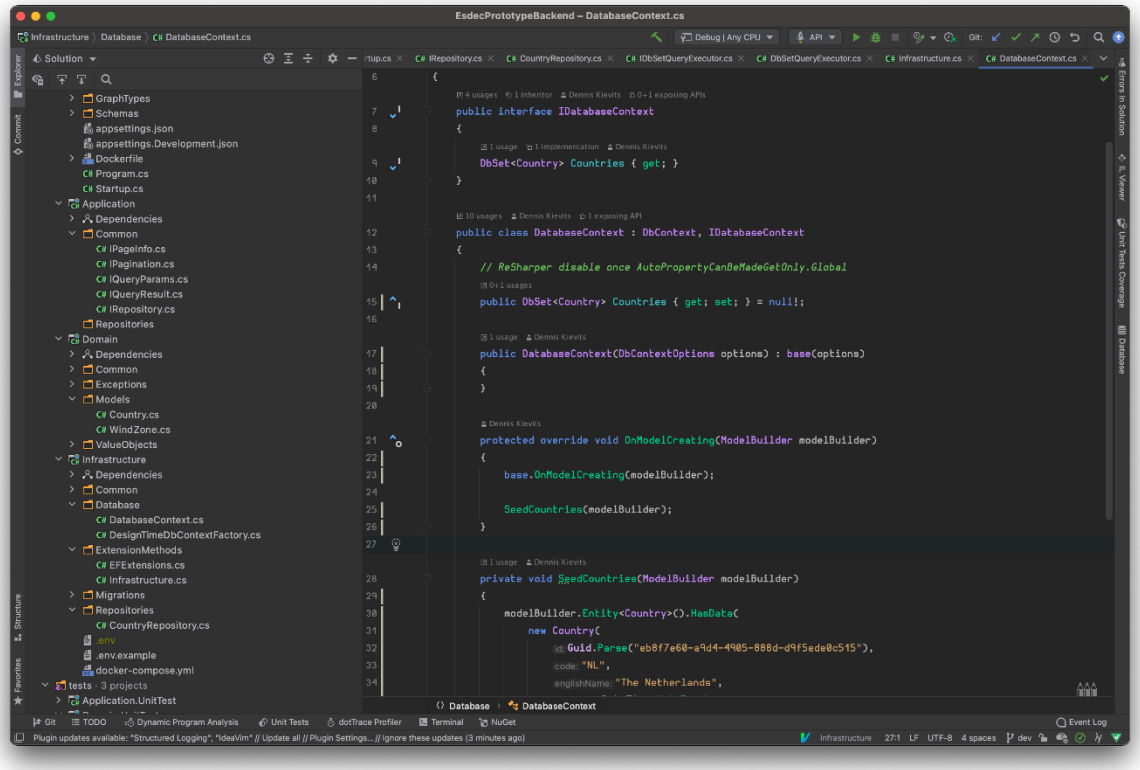

Figure 9: A screenshot of the IDE showcasing the backend project structure.

13.3 Frontend

The frontend application was built using TypeScript and the framework Svelte. Svelte allows to compose the application using component similarly to how ReactJS and Angular are developed. Svelte however has the advantage in terms of final bundle size and speed, as this framework only includes code that is actually used by the framework. The framework itself acts as a compiler, so there is no library bundled in the production file which saves a lot of space. This framework is chosen because the main issue that the current ESDEC application has is speed. Therefore, by using a framework that is also optimized for speed together with WebAssembly should bring forth the best possible result. To create the WebAssembly module that the frontend application relies on, the programming language Rust was chosen due to its excellent tooling and support to create fast and small WebAssembly modules.

The frontend application also follows best practices when it comes to application architecture as the components are defined in a modular fashion. This makes these components very testable and flexible in their use. These flexible components do not rely on dependencies internally but receive them via props which enables them to be used in other applications as well or to mock certain dependencies of which a certain component relies on.

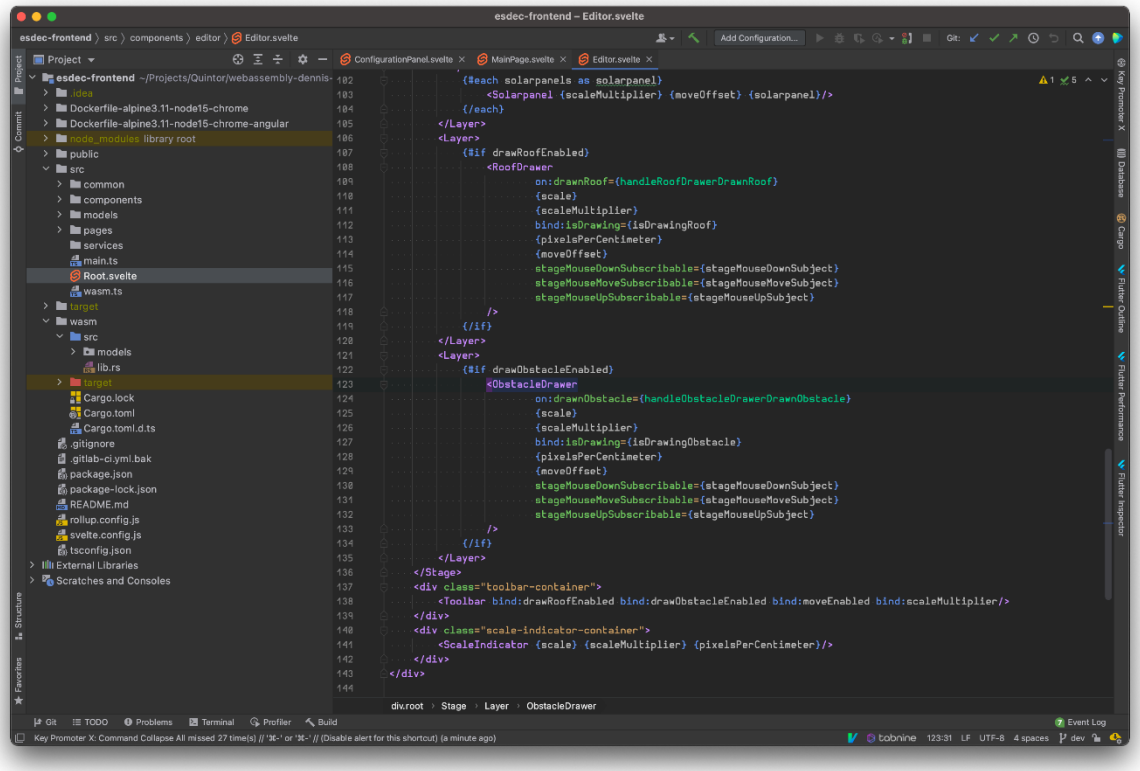

Figure 10: A screenshot of the IDE showcasing the frontend project structure.

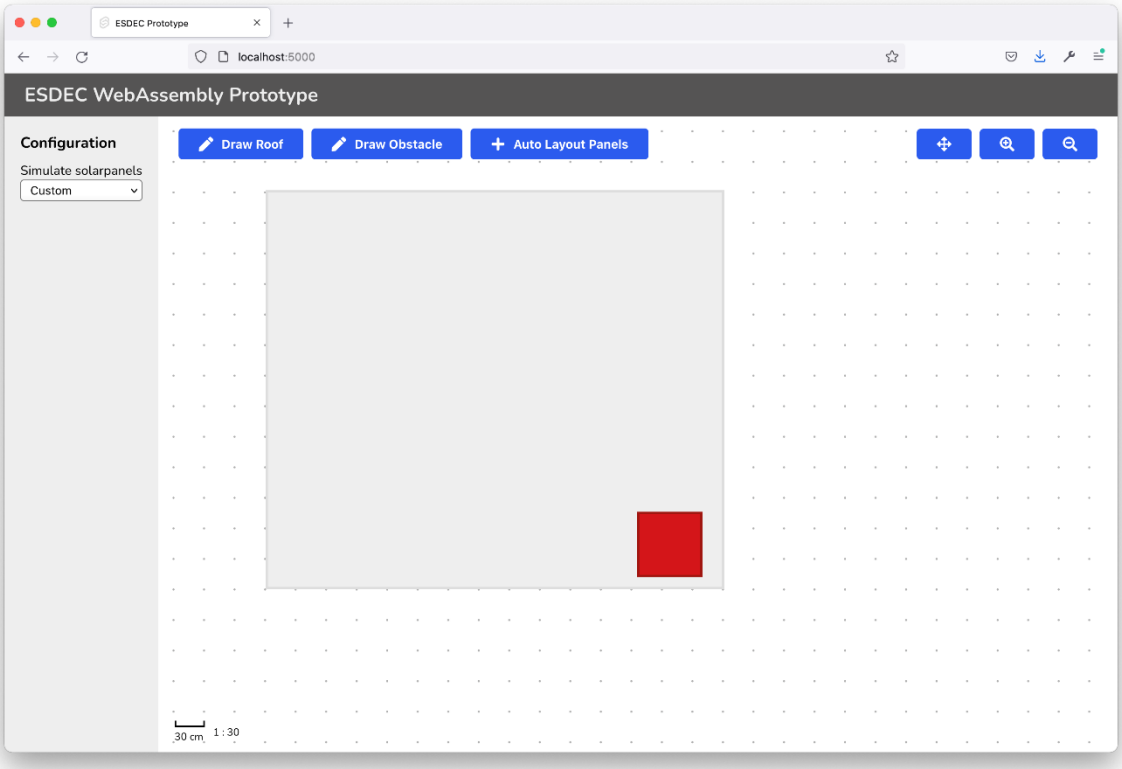

Figure 11: A screenshot of the prototype application showing a simple roof and obstacle.

13.4 Testing

Software testing is an important step in order to consistently improve and modify code with confidence. It also acts as self-documenting documentation of the project. Software testing for this prototype has been integrated within a continuous integration platform hosted by Quintor named Bamboo. Each time a commit has been made to the git repository this project has been hosted on, Bamboo will run the tests that are written for this project and notify the developers with the results of these tests.

13.4.1 Backend tests

For each layer in the backend application a separate C# project was added to the solution to be able to run these tests on. These C# projects follow the naming scheme of the application layer C# project's name suffixed by the type of tests that are contained within that C# project.

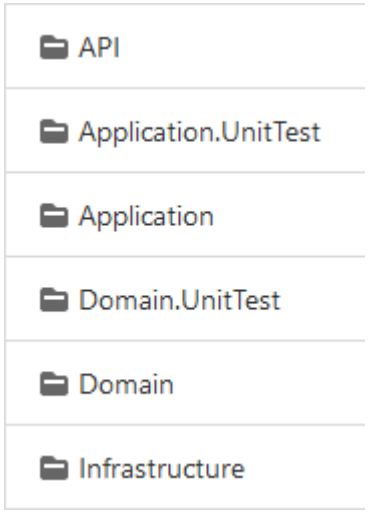

Figure 12: An example of the naming convention used to store these C# test projects in.

The backend currently only has unit tests as that suffices for this prototype, but in a real world scenario it would be wise to also write integration tests for the API layer as that publicly interfaces with other software outside of the backend domain, such as the frontend application.

```
using System;
using Domain.Exceptions.WindSpeed;
using Domain.ValueObjects;
using FluentAssertions;
using Xunit;

namespace Domain.UnitTest.ValueObjects
{
    public class WindSpeedTest
    {
        [Fact]
        public void ShouldThrowExceptionWithNegativeWindSpeed()
        {
            Action act = () => new WindSpeed(-1);
            act.Should().Throw<WindSpeedIsNegativeException>()
                .WithMessage("The provided wind speed (-1) was
negative");
        }

        [Fact]
        public void ShouldInstantiateSuccessfullyWithValidWindSpeed()
        {
            var windSpeed = new WindSpeed(29.5);
            windSpeed.ValueInMetersPerSecond.Should().Be(29.5);
        }
    }
}
```

Snippet 10: A unit test for testing whether a given WindSpeed type behaves as expected.

An example of how a unit test was written for a value object is shown in Snippet 10.
The unit tests leverage Nuget packages FluentAssertions and Xunit which enable to construct these unit tests using a "fluent" API which can be easily read by reading out that statement as *A should Be X* or *A should Throw Exception of Type X With Message X2*.
For all unit tests that were written for this project both the should work and should fail scenarios were implemented to ensure that all possible scenarios behave as expected.

13.4.2 Frontend tests

During the development process of the prototype a decision was made to change to from Angular to Svelte. The svelte version of this application does not contain any tests due to the time constraints of the research. Therefore the decision was made to share the tests that were written for the Angular application as most of the logic would translate over to the new application, and would still provide insights on how the previous tests were made.

The angular frontend consists of a few end-to-end (E2E) tests to ensure that the application functions properly when being tested from a similar environment as the user would be interacting with the application.

```
import { browser, logging } from 'protractor';
import { AppPage } from './app.po';

describe('workspace-project App', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage();
  });

  it('should display welcome message', async () => {
    await page.navigateTo();
    expect(await page.getTitleText()).toEqual('esdec-frontend app is
running!');
  });

  afterEach(async () => {
    // Assert that there are no errors emitted from the browser
    const logs = await browser.manage().logs().get(logging.Type.BROWSER);
    expect(logs).not.toContain(jasmine.objectContaining({
      level: logging.Level.SEVERE,
    } as logging.Entry));
  });
});
```

Snippet 11: An example E2E test for when loading the main application.

The angular frontend E2E tests make use of protractor to interact and fetch data from the DOM. Protractor runs tests against the application running in a real browser, interacting with it as a user would [54]. By running these tests automatically via the set up continuous integration platform Bamboo, it becomes easier to see when a change to the source code breaks some user functionality before it's too late.
Running these tests also require the environment to have a chromium browser installed as it leverages a browser to run these tests on, therefore a special docker image was specifically set up to include both the browser and NodeJS packages to automatically run these tests.

13.5 Conclusion

Due to time constraints of the research the actual WebAssembly implementation has not yet been finalized and therefore could not be shared as part of the thesis. The results of this prototype are aimed to be shared during the graduation session held on the 15th of July 2021.

14 Conclusion

The effect of applying WebAssembly on a compute intensive client-side application versus JavaScript is that it can potentially increase performance for the right type of application, and also have other benefits such as the ability to leverage existing code or libraries. Leveraging WebAssembly does however increase project complexity as well as increasing the developer requirements, which means it is not always suitable for all projects.

WebAssembly does definitely have a use case for certain types of applications, such applications being mostly performance critical applications. Think of applications that perform tasks such as video or audio editing. These computationally expensive type of applications can see the most benefit when applying WebAssembly to the application, especially considering WebAssembly allows for multi-threaded constructs instead of relying on just a single thread for workloads that can be parallelized. WebAssembly also could have potential when looking at re-using or transforming current (desktop class) applications to the web, as for example Emscripten could provide a relatively easy and effective way to achieve this, even going as far as to implement native OS APIs to play nice with the browser.

WebAssembly however is not perfect; it is still not very easy to integrate within your average frontend application. Adding WebAssembly to a project comes with a cost. There is more complexity, more parts to manage, more code to implement. Yet another thing that requires precious time that could have been spent directly for other activities. WebAssembly also requires a broader skillset when compared with the technologies that are mostly used for frontend development, forcing the use of a different language to build WebAssembly modules than what most frontend developers are comfortable with.

The ESDEC Calculator should most likely see performance improvements when implementing performance critical operations in WebAssembly. However during the development of the prototype it was also obvious that when creating a solar project with a very large (100+) amount of solar panels the performance bottleneck was more related to the visualization part rather of the application rather than the calculation. The visualization is currently performed by an external dependency which cannot be optimized using WebAssembly unless another library is leveraged or the visualization code is totally custom.

15 Recommendation

WebAssembly has the most potential in applications where performance is critical, or where currently existing desktop class applications are not feasible to be built from the ground up again when aiming to target the web. WebAssembly should therefore be used in applications where it is known ahead of time that the performance needs to be utilized. It could also be leveraged when the application is a port of a desktop application to re-use existing application code, or to simply be able to leverage certain critical libraries. It should however be known that including WebAssembly also changes the developer requirements for the project, as WebAssembly modules cannot be build using most programming languages that web developers use.

The decision does however not have to be made right way to include WebAssembly, it is also possible to add WebAssembly to a project at a later stage to handle the computationally intensive parts. In fact the advice would be to not prematurely optimize by e.g. including WebAssembly as it can only potentially increase project complexity where it is not required.

See Figure 13 in order to check whether a web project should consider applying WebAssembly.

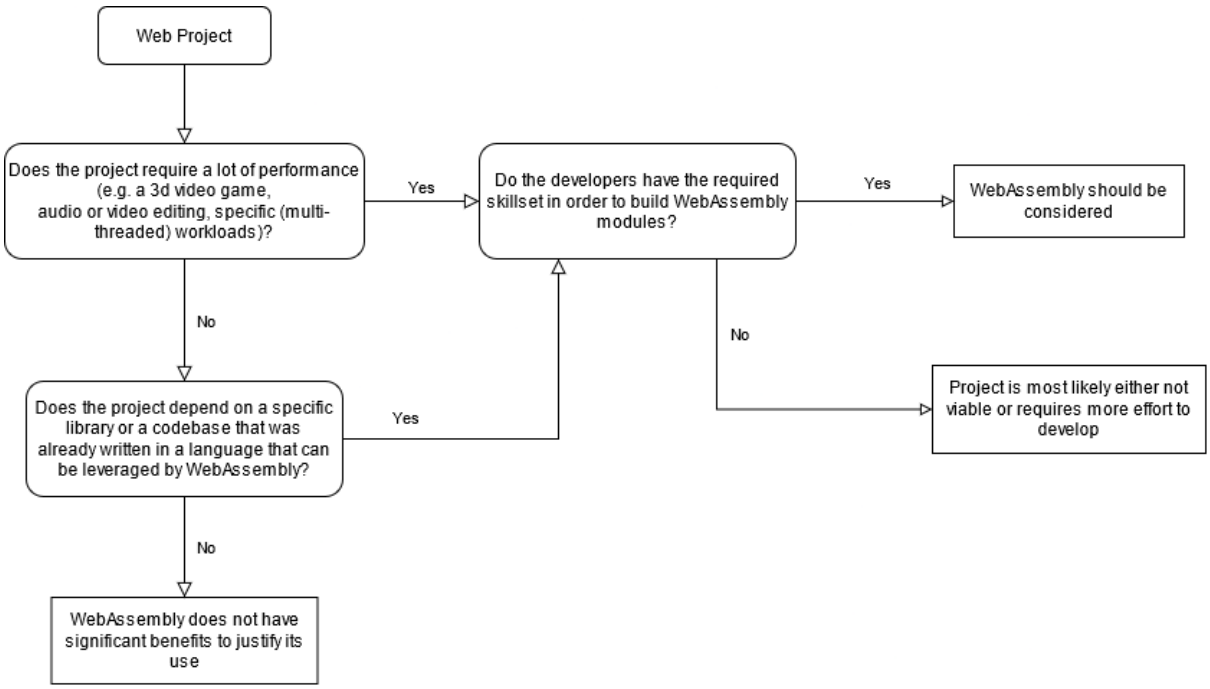

Figure 13: Decision tree on whether a web project should consider the use of WebAssembly.

For further research on WebAssembly, it would be advised to research on how to effectively leverage WebAssembly in a currently existing project, as this has been excluded from this thesis due to time constraints.

16 References

[1] Quintor Product B.V., “About us,” [Online]. Available: https://quintor.nl/over-ons/. [Accessed 9 March 2021].

[2] ESDEC, “Calculator - ESDEC,” [Online]. Available: https://eu.esdec.com/en/calculator/. [Accessed 25 June 2021].

[3] Computer Hope, “What is Assembly Language?,” 10 July 2019. [Online]. Available: https://www.computerhope.com/jargon/a/al.htm#different. [Accessed 16 March 2021].

[4] WebAssembly, “WebAssembly,” [Online]. Available: https://webassembly.org/. [Accessed 9 March 2021].

[5] CanIUse, “WebAssembly,” [Online]. Available: https://caniuse.com/wasm. [Accessed 10 March 2021].

[6] WebAssembly, “FAQ,” [Online]. Available: https://webassembly.org/docs/faq/. [Accessed 9 March 2021].

[7] Web Assembly, “WebAssembly High-Level Goals,” [Online]. Available: https://webassembly.org/docs/high-level-goals/. [Accessed 9 March 2021].

[8] WebAssembly, “Non-Web Embeddings,” [Online]. Available: https://webassembly.org/docs/non-web/. [Accessed 9 March 2021].

[9] B. James, “WebAssembly: What is it and why should you care?,” Hackaday, 4 April 2019. [Online]. Available: https://hackaday.com/2019/04/04/webassembly-what-is-it-and-why-should-you-care/. [Accessed 9 March 2021].

[10] Figma, “About,” [Online]. Available: https://www.figma.com/about/. [Accessed 9 March 2021].

[11] Microsoft, “Blazor | Build client web apps with C#,” [Online]. Available: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor. [Accessed 9 March 2021].

[12] Facebook, “React - A JavaScript library for building user interfaces,” [Online]. Available: https://reactjs.org/. [Accessed 9 March 2021].

[13] Google, “Angular - The modern web developer's platform,” [Online]. Available: https://angular.io/. [Accessed 9 March 2021].

[14] Google, “Flutter - Beautiful native apps in record time,” [Online]. Available: https://flutter.dev/. [Accessed 9 March 2021].

[15] Yewstack, “Yew - Rust / Wasm client web app framework,” [Online]. Available: https://github.com/yewstack/yew. [Accessed 9 March 2021].

[16] Autodesk, “AUTOCAD web app - Work in AutoCAD anytime, anywhere,” [Online]. Available: https://www.autodesk.com/products/autocad-web-app/overview?plc=ACDIST&term=1-YEAR&support=ADVANCED&quantity=1. [Accessed 9 March 2021].

[17] K. Cheung, “AutoCAD & WebAssembly: Moving a 30 Year Code Base to the Web,” InfoQ, [Online]. Available: https://www.infoq.com/presentations/autocad-webassembly/. [Accessed 9 March 2021].

[18] G. Tomassetti, “Understand WebAssembly: Why It Will Change the Web,” [Online]. Available: https://tomassetti.me/introduction-to-webassembly/. [Accessed 10 March 2021].

[19] PrograCoding, “What is Bytecode - PrograCoding,” [Online]. Available: https://www.pragracoding.com/bytecode/. [Accessed 10 March 2021].

[20] WebAssembly, “Portability - WebAssembly,” [Online]. Available: https://webassembly.org/docs/portability/. [Accessed 25 June 2021].

[21] Takahirox, “JS vs WASM,” [Online]. Available: https://takahirox.github.io/WebAssembly-benchmark/. [Accessed 10 March 2021].

[22] M. Bebenita, “WebAssembly is '30X' Faster than JavaScript,” 9 March 2017. [Online]. Available: https://medium.com/@mbebenita/webassembly-is-30x-faster-than-javascript-c71ea54d2f96. [Accessed 10 March 2021].

[23] A. Danilo, “WebAssembly Threads ready to try in Chrome 70,” Google, October 2018. [Online]. Available: https://developers.google.com/web/updates/2018/10/wasm-threads. [Accessed 15 March 2021].

[24] CanIUse, “Webworkers,” [Online]. Available: https://caniuse.com/webworkers. [Accessed 15 March 2021].

[25] Surma, “Is WebAssembly magic performance pixie dust?,” [Online]. Available: https://surma.dev/things/js-to-asc/index.html. [Accessed 25 June 2021].

[26] C. Reyes, “WebAssembly: Solving Performance Problems on the Web,” 5 June 2018. [Online]. Available: https://www.sitepoint.com/webassembly-solving-performance-problems/. [Accessed 25 June 2021].

[27] StackOverflow, “StackOverflow Survey 2020,” [Online]. Available: https://insights.stackoverflow.com/survey/2020. [Accessed 15 March 2021].

[28] B. Couriol, “Boosting WebAssembly Performance with SIMD and Multi-Threading,” InfoQ, 19 February 2021. [Online]. Available: https://www.infoq.com/articles/webassembly-simd-multithreading-performance-gains/. [Accessed 15 March 2021].

[29] Google, “Feature: WebAssembly SIMD,” 10 March 2021. [Online]. Available: https://www.chromestatus.com/feature/6533147810332672. [Accessed 15 March 2021].

[30] Google Developers, “Debugging WebAssembly with modern tools,” Google, 2020. [Online]. Available: https://developers.google.com/web/updates/2020/12/webassembly. [Accessed 15 March 2021].

[31] PC RIsk, “Why Adobe Flash is a Security Risk and Why Media Companies Still Use it,” 28 May 2021. [Online]. Available: https://www.pcrisk.com/internet-threat-news/10500-flash-security-risk. [Accessed 20 August 2021].

[32] Microsoft Research Team, “Windows Defender Antivirus can now run in a sandbox,” 26 October 2018. [Online]. Available: https://www.microsoft.com/security/blog/2018/10/26/windows-defender-antivirus-can-now-run-in-a-sandbox/. [Accessed 20 August 2021].

[33] J. Laukkonen, “What is Chromium?,” Lifewire, 28 April 2020. [Online]. Available: https://www.lifewire.com/chromium-web-browser-4171288. [Accessed 30 March 2021].

[34] L. Clark, “Standardizing WASI: A system interface to run WebAssembly outside the web,” Mozilla Hacks, 27 March 2019. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/. [Accessed 23 March 2021].

[35] the Bytecode Alliance, “About the Bytecode Alliance,” [Online]. Available: https://bytecodealliance.org/. [Accessed 23 March 2021].

[36] the Bytecode Alliance, “Lucet Repository,” [Online]. Available: https://github.com/bytecodealliance/lucet. [Accessed 23 March 2021].

[37] Bytecode Alliance, “Introduction Wasmtime,” [Online]. Available: https://docs.wasmtime.dev/. [Accessed 23 March 2021].

[38] Surma, “Compiling C to WebAssembly without Emscripten,” 28 May 2019. [Online]. Available: https://surma.dev/things/c-to-webassembly/. [Accessed 30 March 2021].

[39] LLVM Clang, “Clang: a C language family frontend for LLVM,” [Online]. Available: https://clang.llvm.org/. [Accessed 30 March 2021].

[40] M. Bynens and J. Dalton, “Benchmark.js,” [Online]. Available: https://benchmarkjs.com/. [Accessed 30 March 2021].

[41] MathIsFun, “Fibonacci Sequence,” [Online]. Available: https://www.mathsisfun.com/numbers/fibonacci-sequence.html. [Accessed 30 March 2021].

[42] bellbind, “sha256 implementation,” [Online]. Available: https://gist.github.com/bellbind/2fd66a09340942b9845ef432a5c8c419. [Accessed 6 April 2021].

[43] bellbind, “bellbind,” [Online]. Available: https://github.com/bellbind. [Accessed 6 April 2021].

[44] R. L. Apodaca, “Compiling C to WebAssembly and Running It - without Emscripten,” 16 October 2019. [Online]. Available: https://depth-first.com/articles/2019/10/16/compiling-c-to-webassembly-and-running-it-without-emscripten/. [Accessed 6 April 2021].

[45] LLVM, Clang, “Cross-compilation using Clang,” [Online]. Available: https://clang.llvm.org/docs/CrossCompilation.html. [Accessed 6 April 2021].

[46] appcypher, “Awesome WebAssembly Languages,” 20 March 2021. [Online]. Available: https://github.com/appcypher/awesome-wasm-langs. [Accessed 14 April 2021].

[47] Microsoft, “A tour of the C# language,” 28 January 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. [Accessed 13 April 2021].

[48] Rustwasm, “Why Rust and WebAssembly?,” [Online]. Available: https://rustwasm.github.io/docs/book/why-rust-and-webassembly.html. [Accessed 13 April 2021].

[49] V. Dasigi, “A Catalog of Common Bugs in C++ Programming,” December 1999. [Online]. Available:

https://www.researchgate.net/publication/2463564_A_Catalog_Of_Common_Bugs_In_C_Programming. [Accessed 12 April 2021].

[50] StackOverflow Developer Survey 2020, [Online]. Available: https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved. [Accessed 13 April 2021].

[51] Java, “What is Java technology and why do I need it?,” [Online]. Available: https://java.com/en/download/help/whatis_java.html. [Accessed 13 April 2021].

[52] i-net-software, “JWebAssembly,” 21 May 2020. [Online]. Available: https://github.com/i-net-software/JWebAssembly. [Accessed 13 April 2021].

[53] LeaningTech, “Java To WebAssembly Compiler | CheerpJ | LeaningTechnologies,” [Online]. Available: https://leaningtech.com/cheerpj/. [Accessed 13 April 2021].

[54] Protractor, “Protractor - end-to-end testing,” [Online]. Available: https://www.protractortest.org/#/. [Accessed 18 June 2021].

[55] N. Fitzgerald, “Oxidizing Source Maps with Rust and WebAssembly,” Mozilla, 18 January 2018. [Online]. Available: https://hacks.mozilla.org/2018/01/oxidizing-source-maps-with-rust-and-webassembly/. [Accessed 15 March 2021].

17 Appendix

17.1 Benchmark results

Benchmark	Environment	Google Chrome	Mozilla Firefox	Safari
Invoking `fibonacci(5)`	JavaScript	17,378,540 ops/sec (65 samples)	12,332,445 ops/sec (65 samples)	3,190,100 ops/sec (57 samples)
Invoking `fibonacci(5)`	WebAssembly	4,090,612 ops/sec (65 samples)	16,154,961 ops/sec (65 samples)	10,413,744 ops/sec (57 samples)
Invoking `fibonacci(10)`	JavaScript	1,377,071 ops/sec (66 samples)	924,206 ops/sec (66 samples)	330,646 ops/sec (64 samples)
Invoking `fibonacci(10)`	WebAssembly	389,370 ops/sec (63 samples)	2,071,223 ops/sec (66 samples)	1,309,456 ops/sec (64 samples)
Invoking `fibonacci(20)`	JavaScript	9,661 ops/sec (51 samples)	7,361 ops/sec (41 samples)	2,427 ops/sec (17 samples)
Invoking `fibonacci(20)`	WebAssembly	3,204 ops/sec (63 samples)	17,470 ops/sec (66 samples)	10,740 ops/sec (57 samples)
Invoking `fibonacci(30)`	JavaScript	91.96 ops/sec (56 samples)	60.1 ops/sec (53 samples)	19.58 ops/sec (36 samples)
Invoking `fibonacci(30)`	WebAssembly	25.57 ops/sec (46 samples)	142 ops/sec (62 samples)	93.61 ops/sec (54 samples)

Table 2: Results Fibonacci benchmark comparing JavaScript to WebAssembly without speed optimizations.

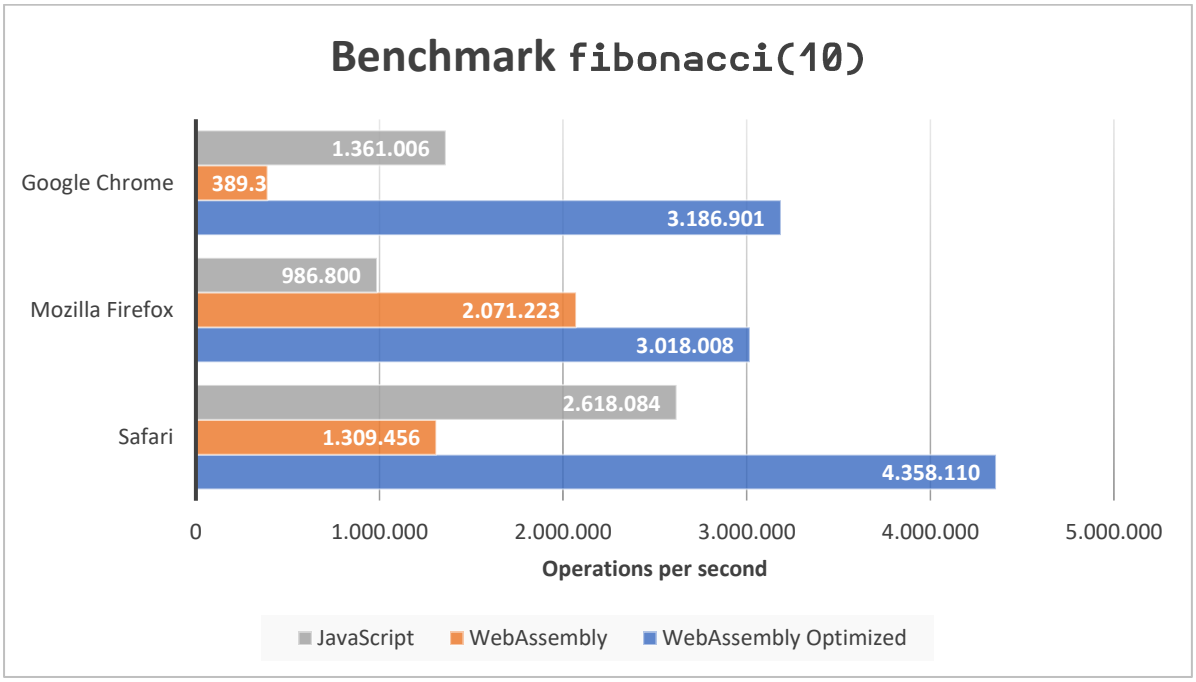

Figure 14: Benchmark results of invoking Fibonacci with input 10.

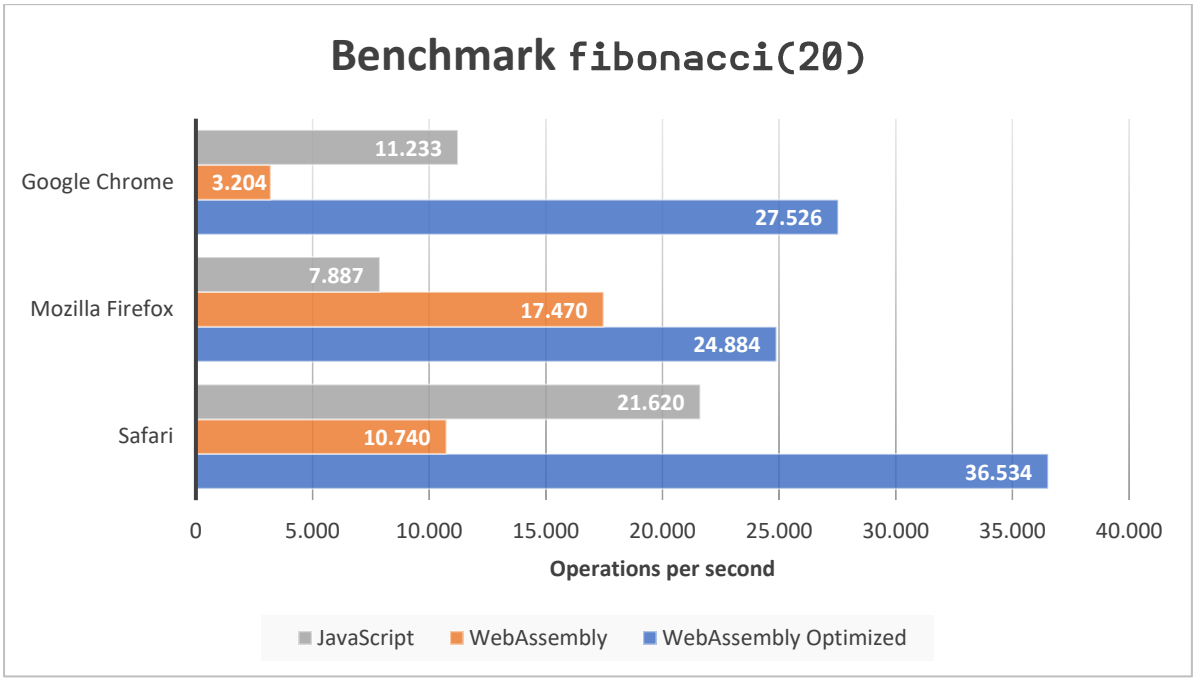

Figure 15: Benchmark results of invoking Fibonacci with input 20.

Benchmark	Environment	Google Chrome	Mozilla Firefox	Safari
Invoking `grayscale(IMG, 128, 128)`	JavaScript	14,417 ops/sec (63 samples)	10,100 ops/sec (52 samples)	3,433 ops/sec (61 samples)
Invoking `grayscale(IMG, 128, 128)`	WebAssembly	12,910 ops/sec (65 samples)	3,115 ops/sec (20 samples)	6,455 ops/sec (37 samples)
Invoking `grayscale(IMG, 760, 428)`	JavaScript	544 ops/sec (62 samples)	531 ops/sec (38 samples)	173 ops/sec (59 samples)
Invoking `grayscale(IMG, 760, 428)`	WebAssembly	638 ops/sec (62 samples)	158 ops/sec (56 samples)	329 ops/sec (60 samples)
Invoking `grayscale(IMG, 1800, 1200)`	JavaScript	71.43 ops/sec (53 samples)	64.39 ops/sec (38 samples)	26.01 ops/sec (39 samples)
Invoking `grayscale(IMG, 1800, 1200)`	WebAssembly	86.1 ops/sec (51 samples)	22.61 ops/sec (39 samples)	48.46 ops/sec (51 samples)
Invoking `grayscale(IMG, 3840, 2160)`	JavaScript	19.57 ops/sec (36 samples)	9.33 ops/sec (20 samples)	6.78 ops/sec (21 samples)
Invoking `grayscale(IMG, 3840, 2160)`	WebAssembly	21.77 ops/sec (40 samples)	6.11 ops/sec (19 samples)	12.18 ops/sec (34 samples)

Table 3: Results grayscaling benchmark comparing JavaScript to WebAssembly without speed optimizations.

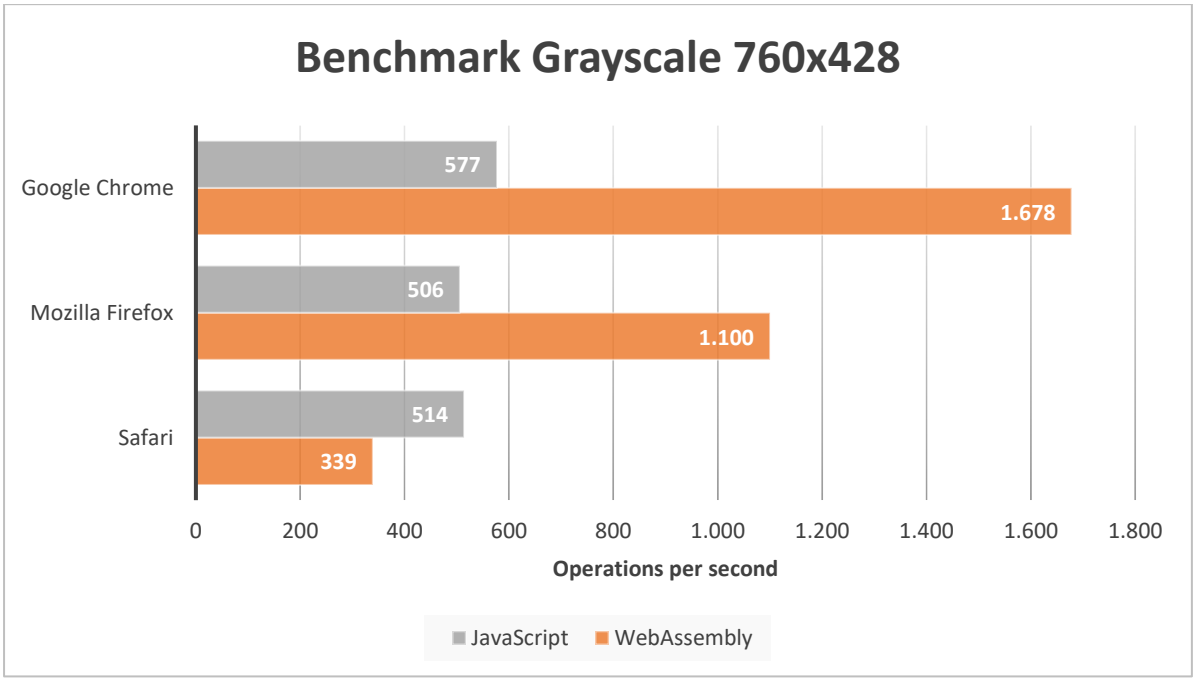

Figure 16: Benchmark results of invoking the grayscale function on an image of the resolution of 760x428.

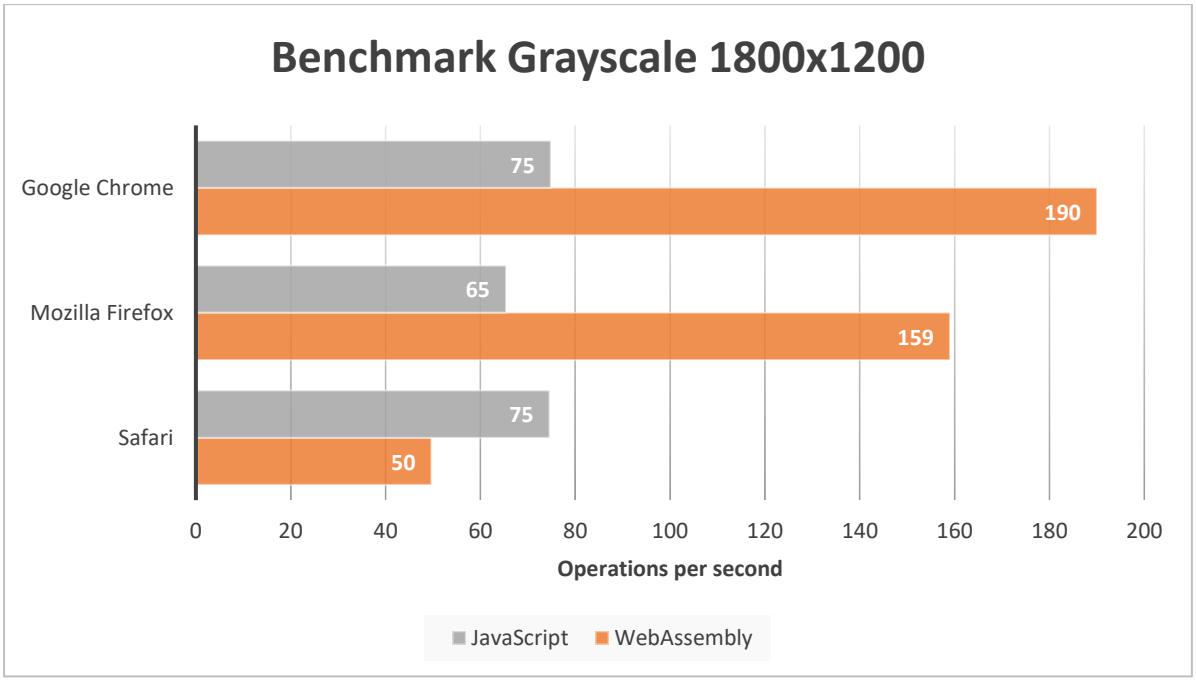

Figure 17: Benchmark results of invoking the grayscale function on an image of the resolution of 1800x1200.

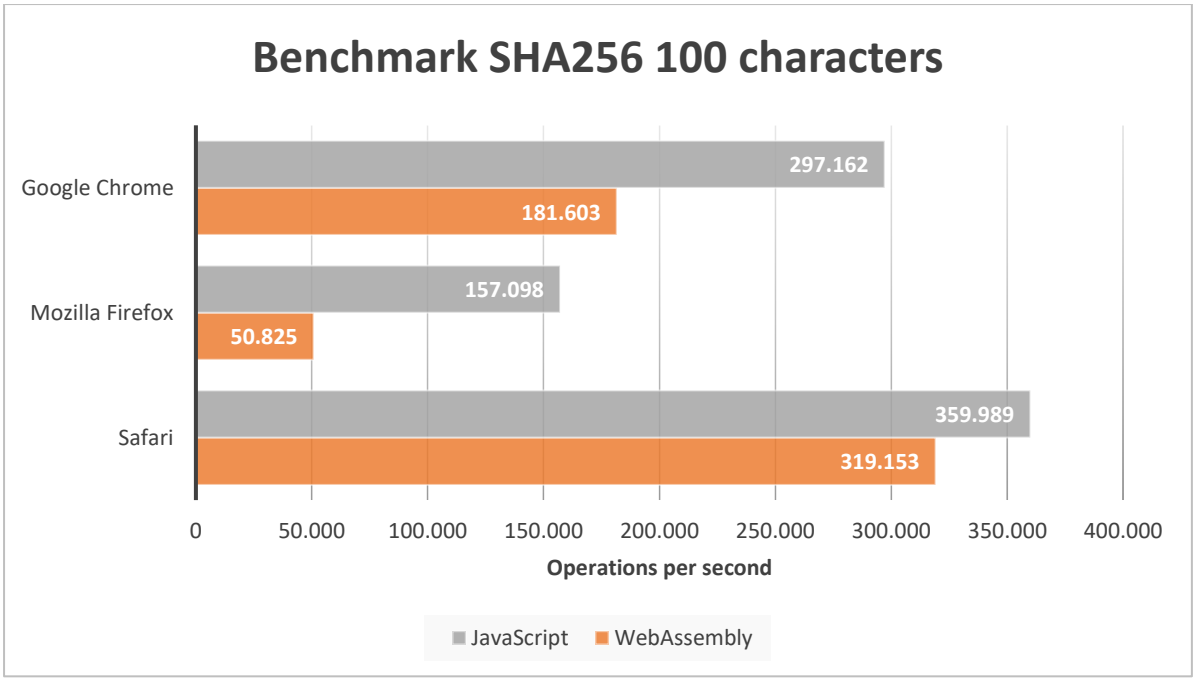

Figure 18: Benchmark results for generating a SHA256 checksum on an input string of the character 'A' repeated 100 times.

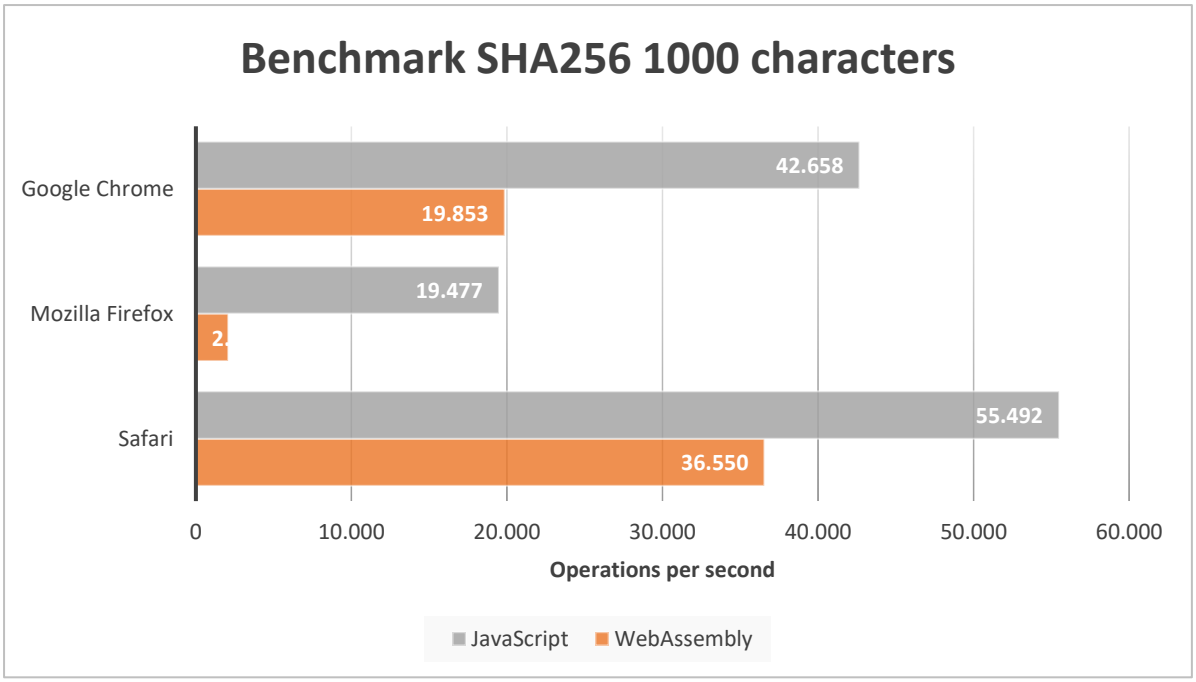

Figure 19: Benchmark results for generating a SHA256 checksum on an input string of the character 'A' repeated 1000 times.

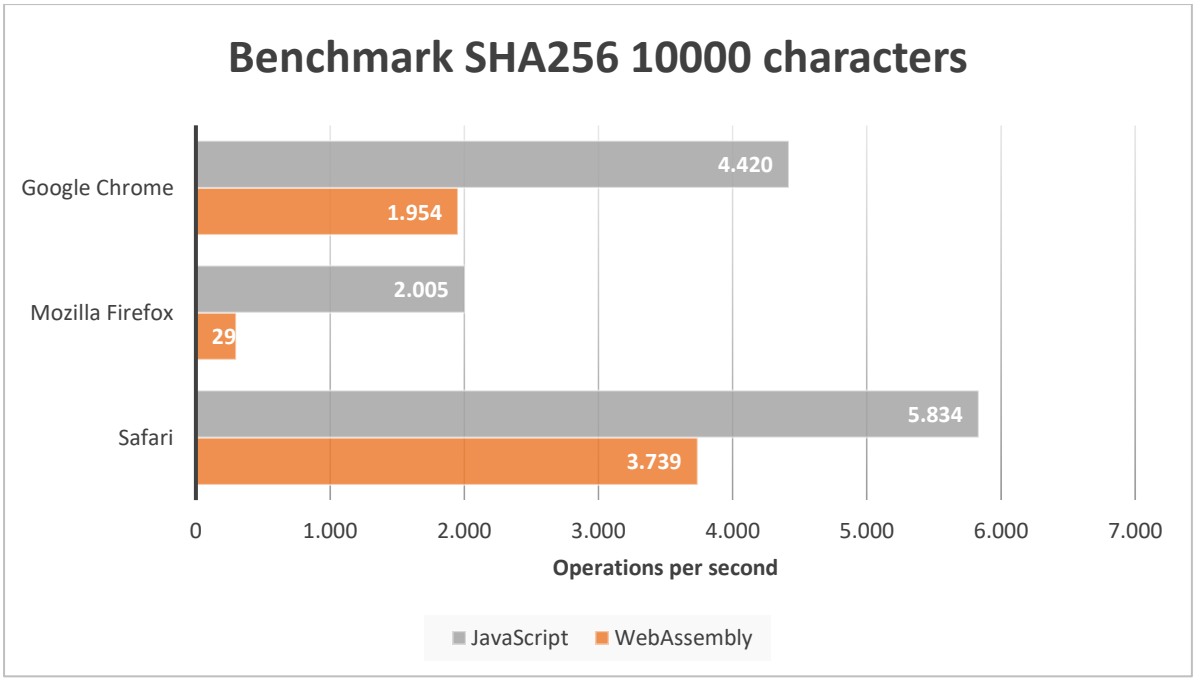

Figure 20: Benchmark results for generating a SHA256 checksum on an input string of the character 'A' repeated 10000 times.

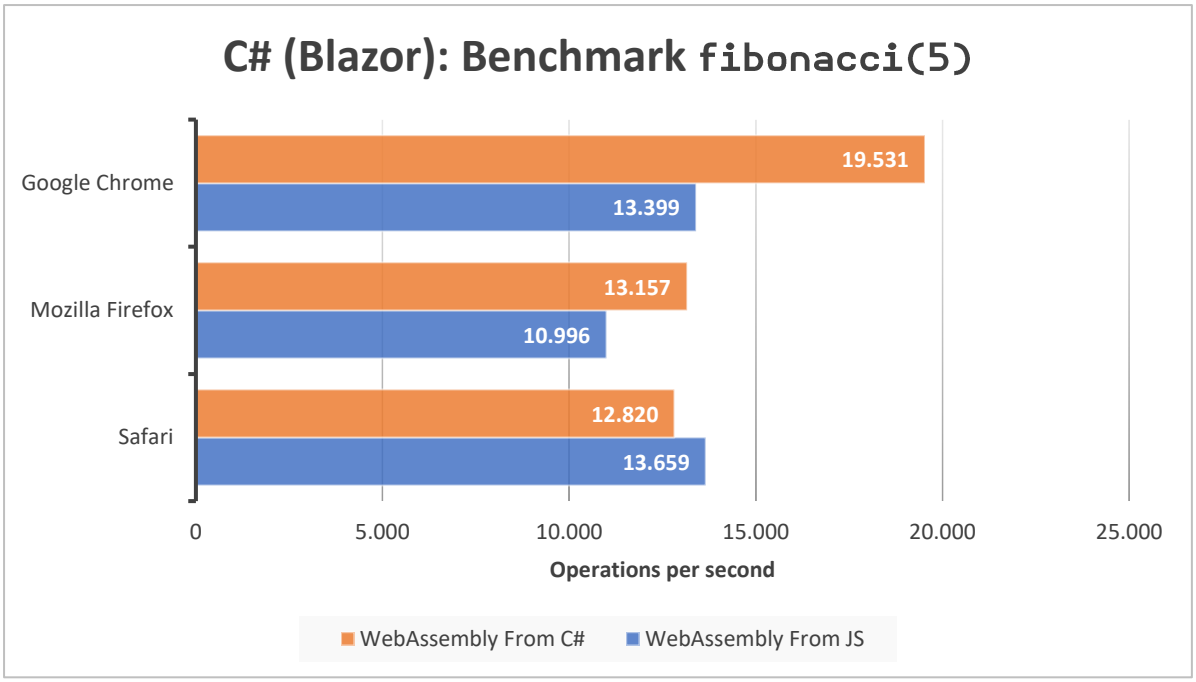

Figure 21: Benchmark results of invoking Fibonacci with input 5 on C# (Blazor).

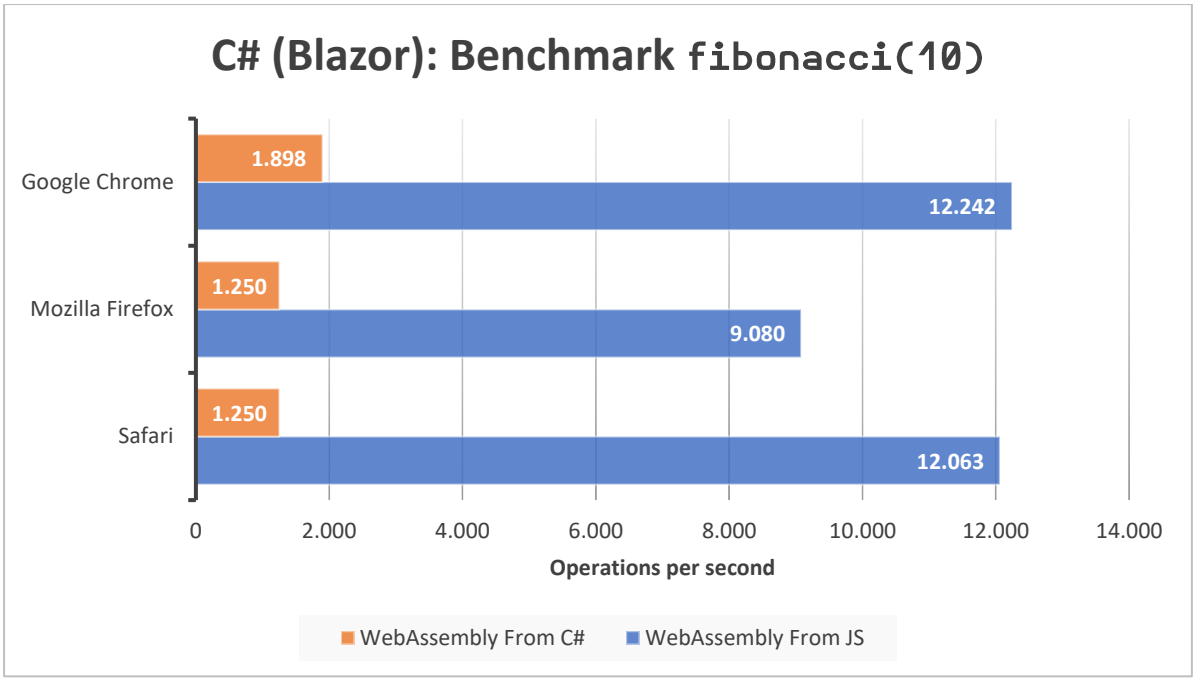

Figure 22: Benchmark results of invoking Fibonacci with input 10 on C# (Blazor).

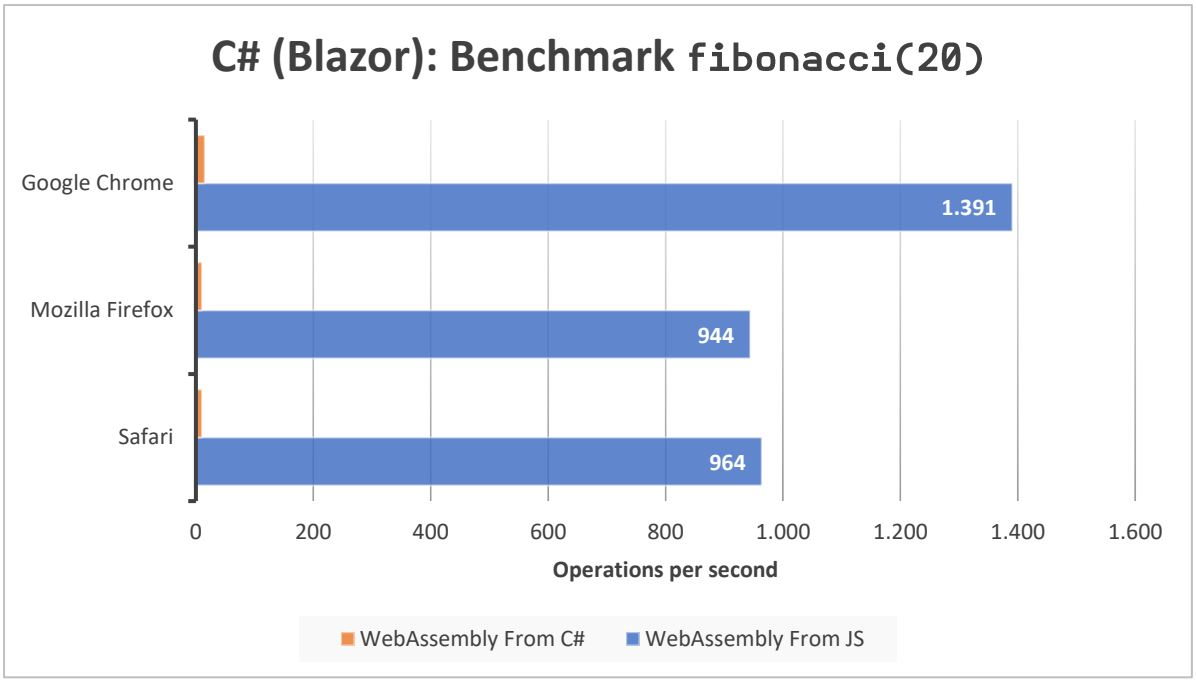

Figure 23: Benchmark results of invoking Fibonacci with input 20 on C# (Blazor). Operations per second when calling from C# is so low that it is barely comparable.

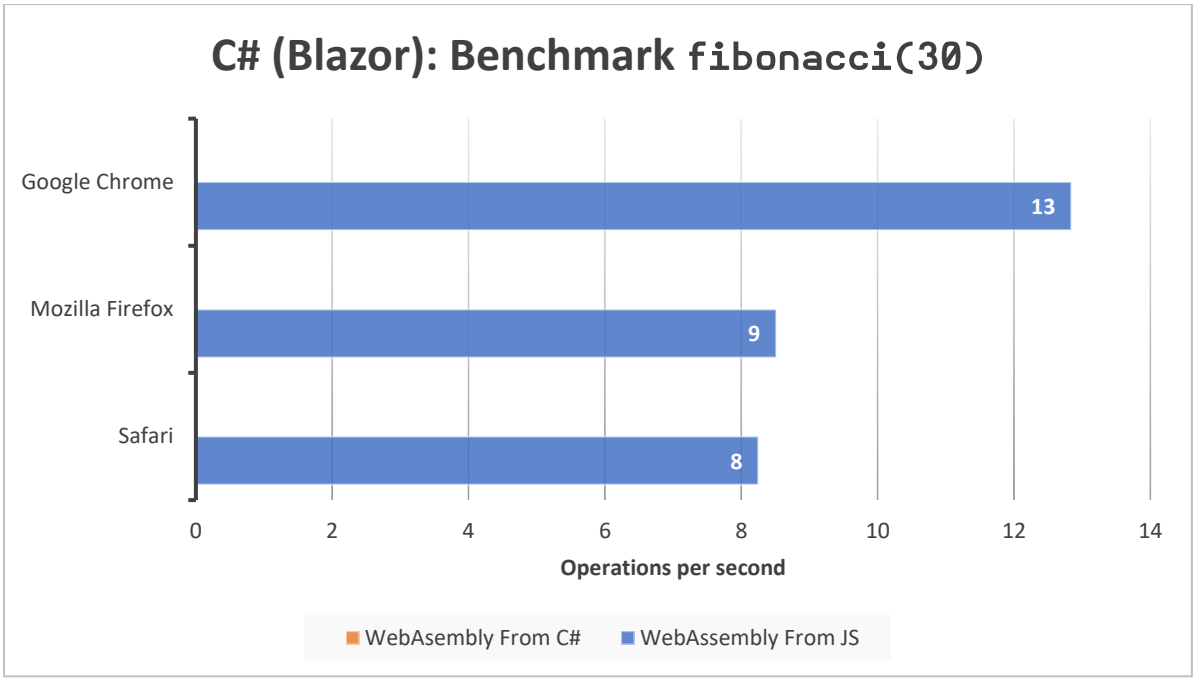

Figure 24: Benchmark results of invoking Fibonacci with input 30 on C# (Blazor). Operations per second when calling from C# is so low that it resulted in less than 1 op/s, therefore not being visible.

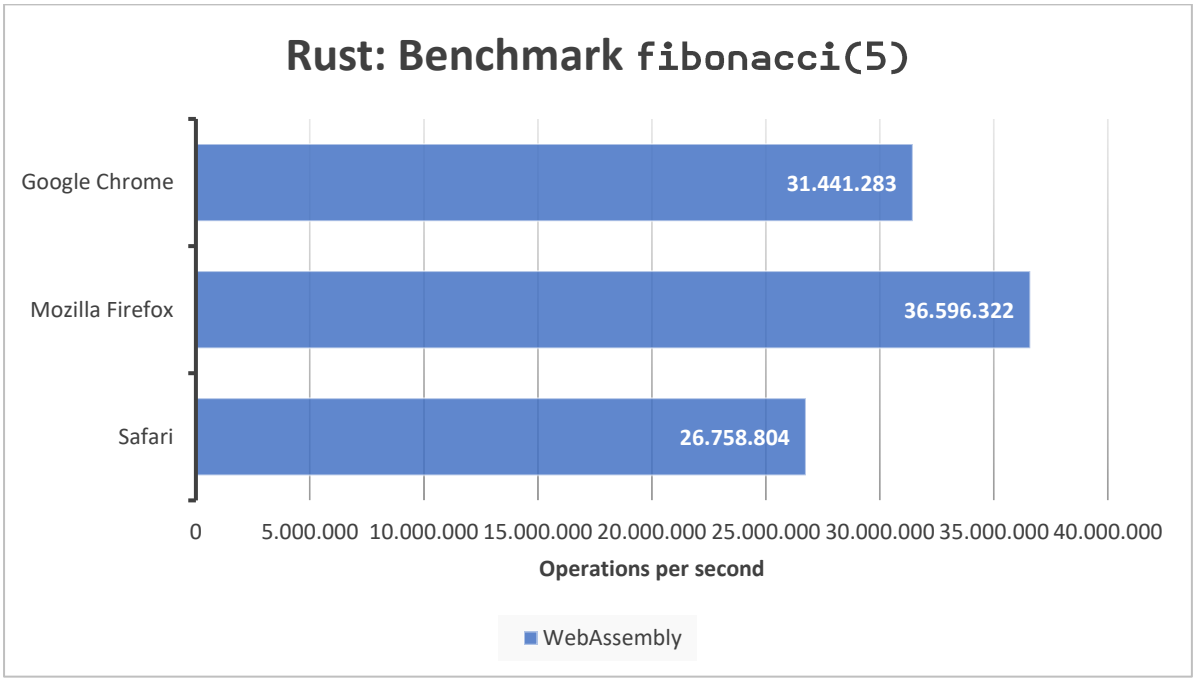

Figure 25: Benchmark results of invoking Fibonacci with input 5 on Rust.

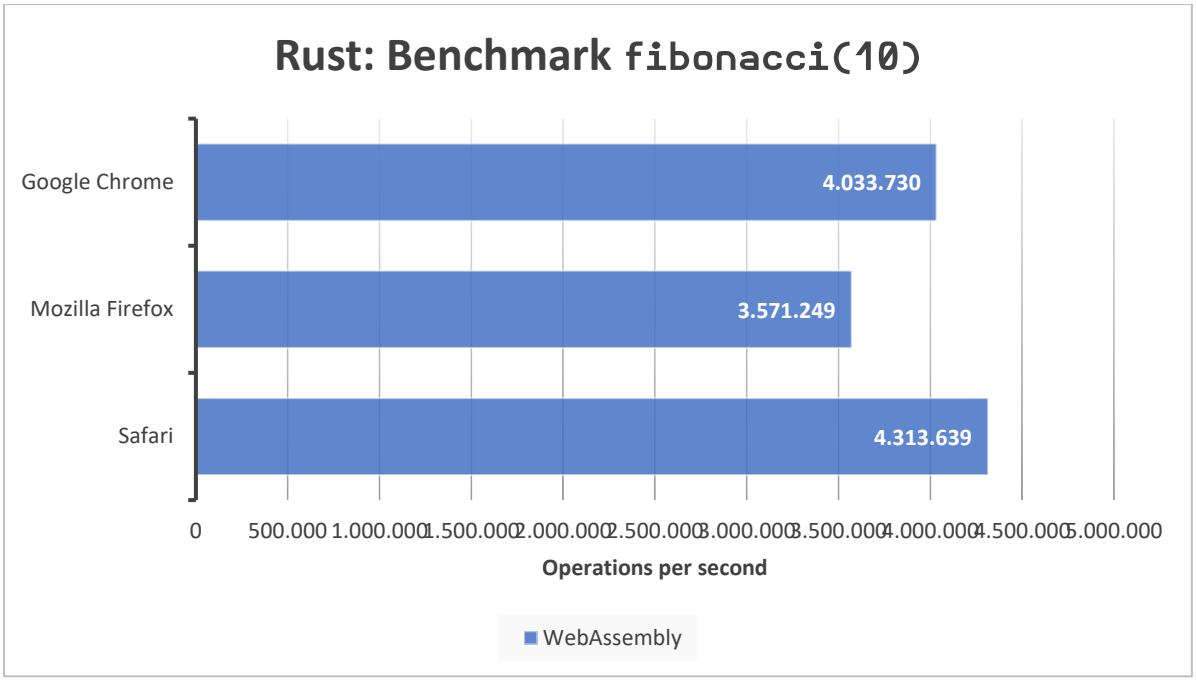

Figure 26: Benchmark results of invoking Fibonacci with input 10 on Rust.

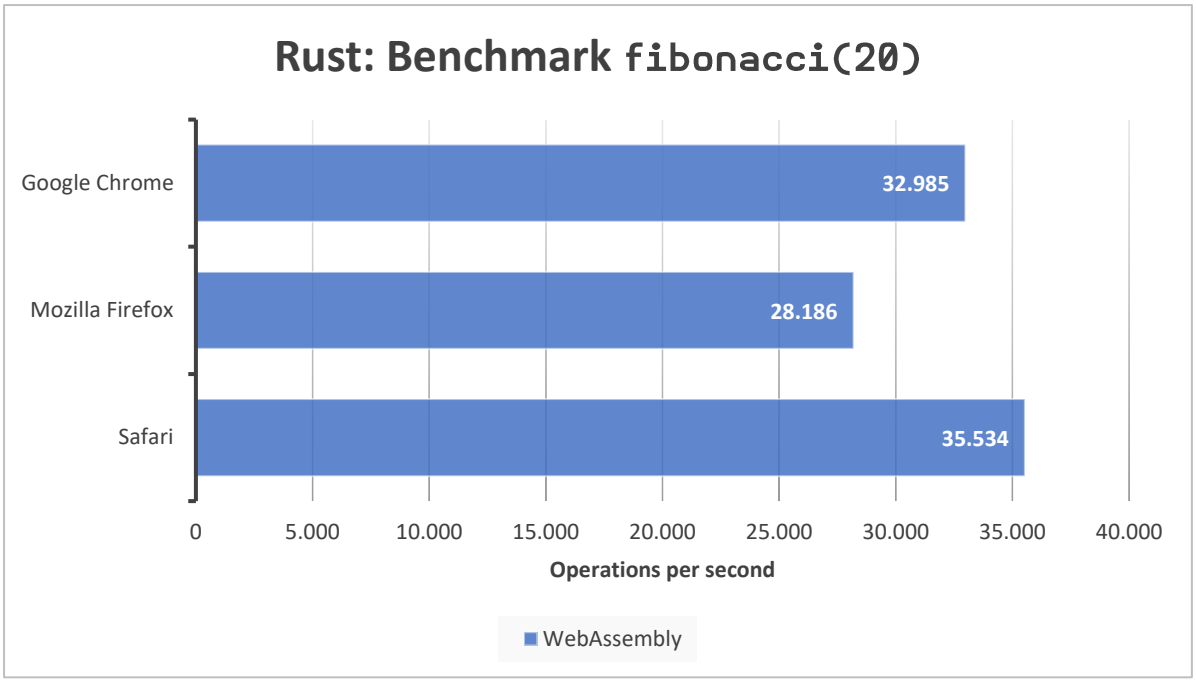

Figure 27: Benchmark results of invoking Fibonacci with input 20 on Rust.

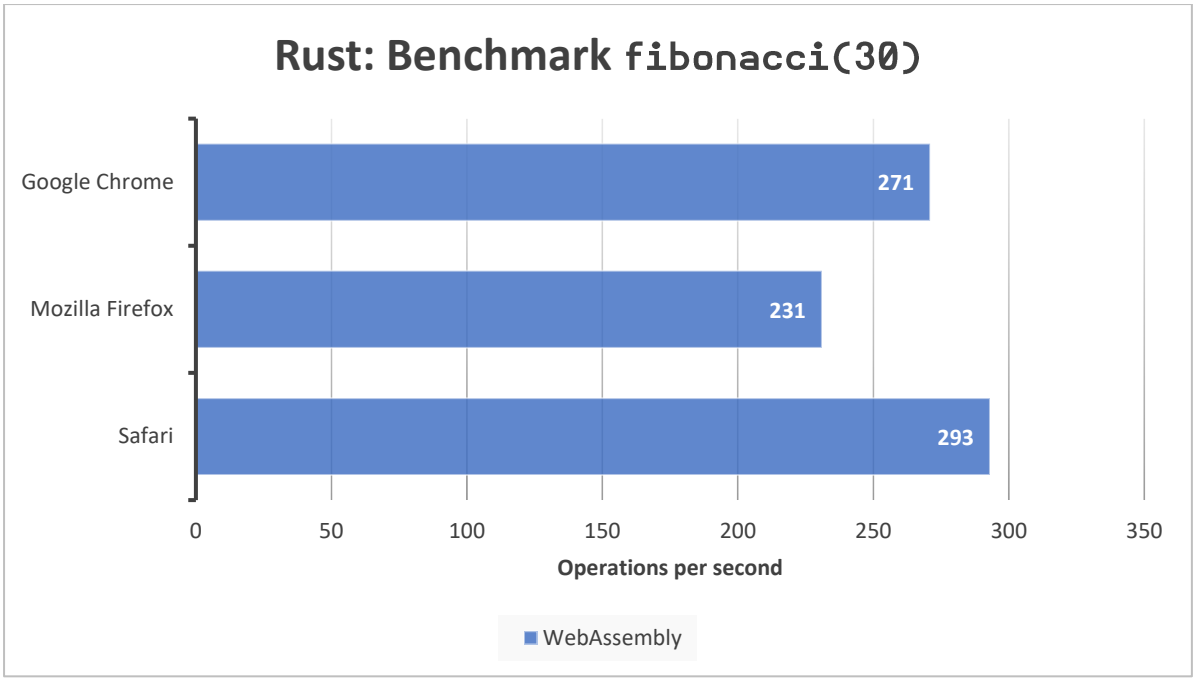

Figure 28: Benchmark results of invoking Fibonacci with input 30 on Rust.

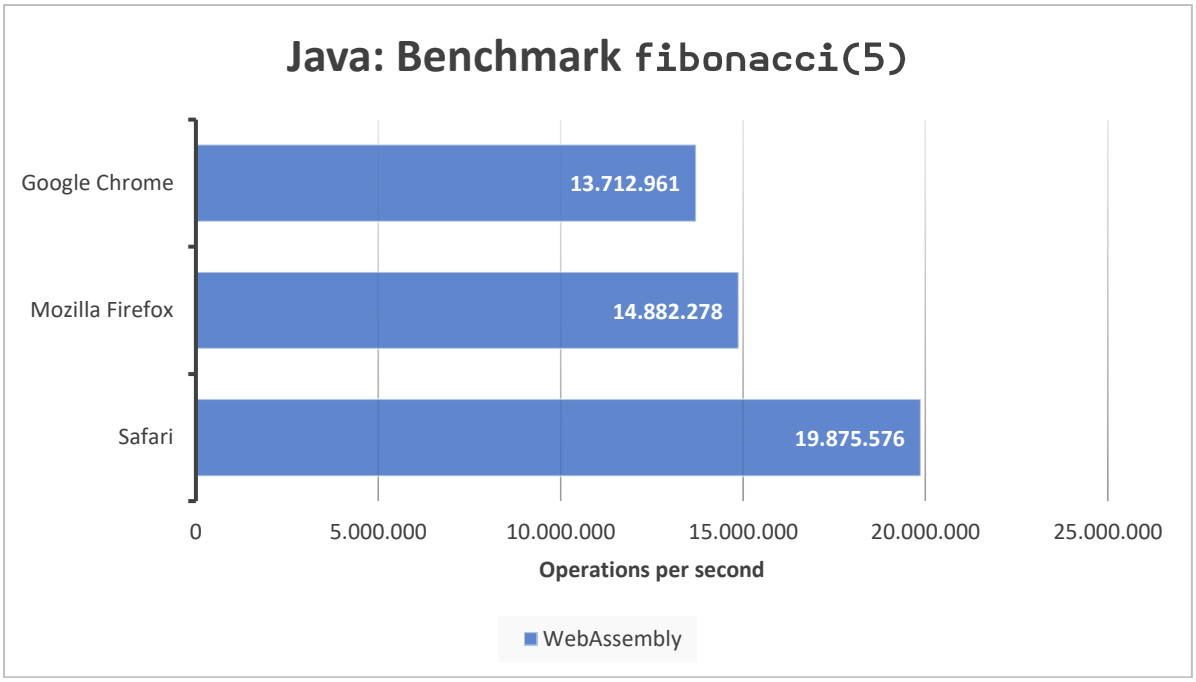

Figure 29: Benchmark results of invoking Fibonacci with input 5 on Java.

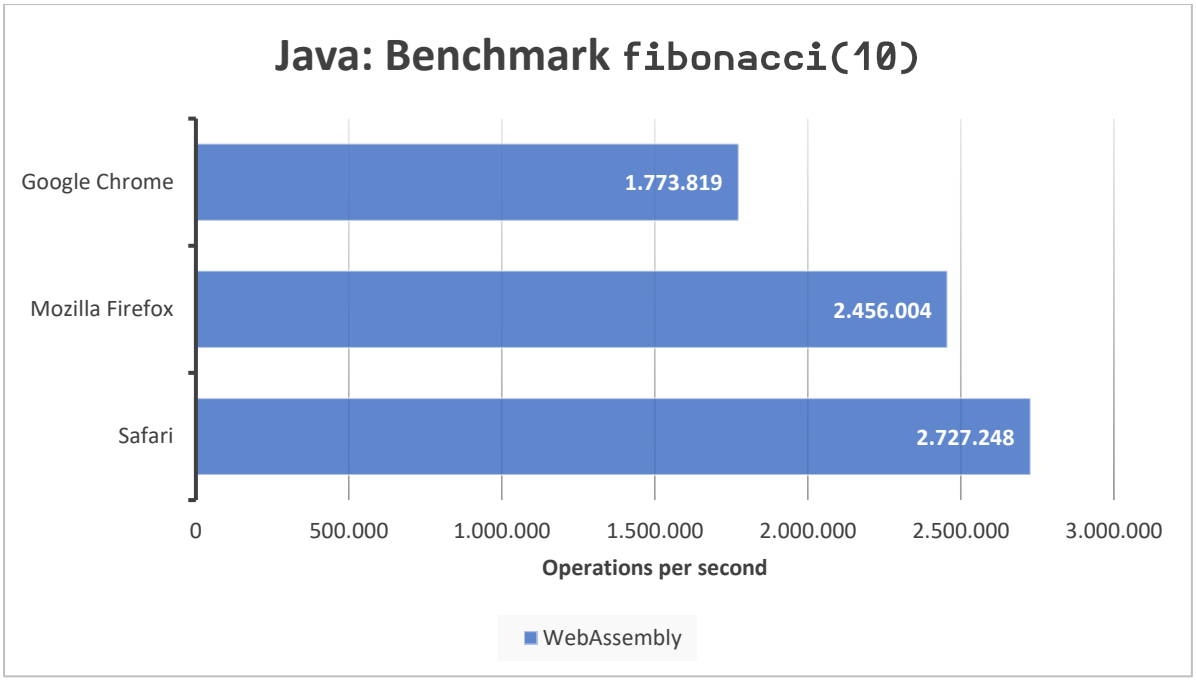

Figure 30: Benchmark results of invoking Fibonacci with input 10 on Java.

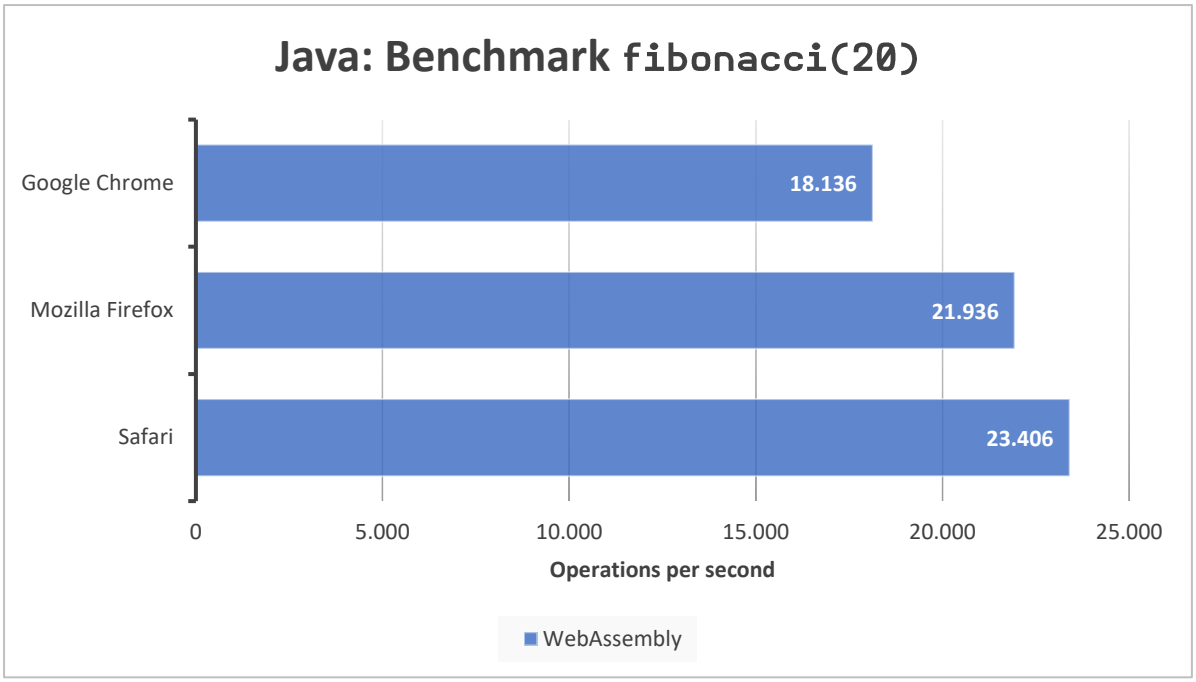

Figure 31: Benchmark results of invoking Fibonacci with input 20 on Java.

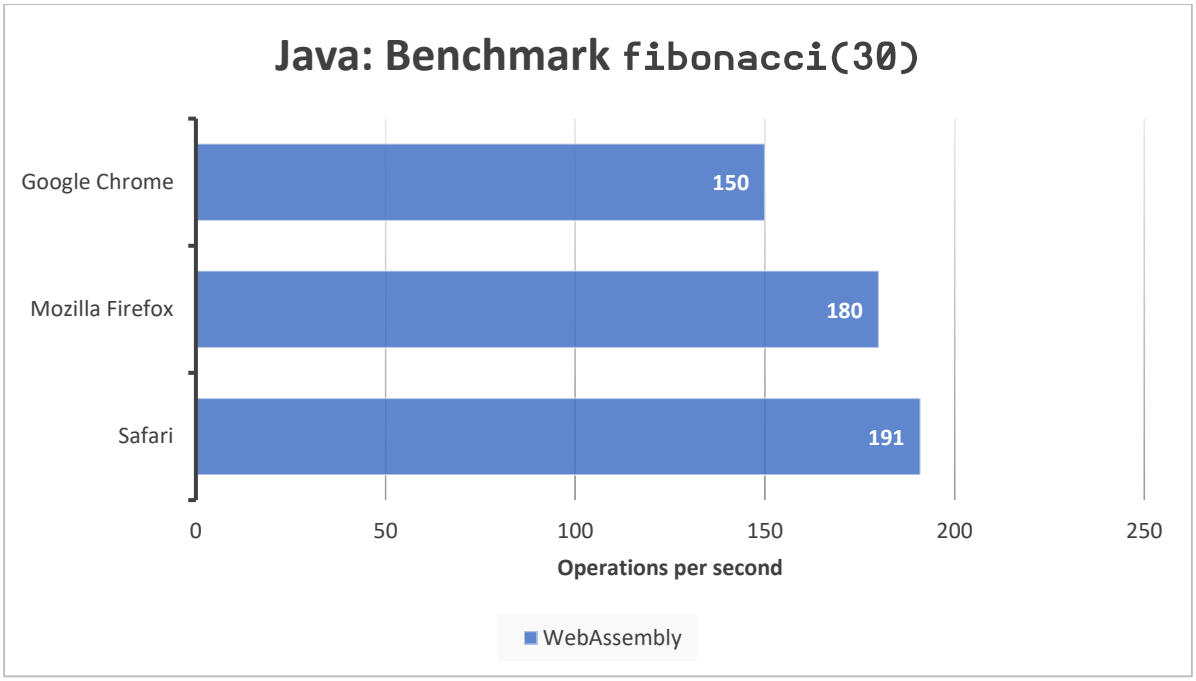

Figure 32: Benchmark results of invoking Fibonacci with input 30 on Java.

17.2 Interview: Important aspects of frontend development for Quintor

Important disclaimer: The actual interview has not been recorded, instead notes were taken during the interview and used to reconstruct the conversation as shown down below.
This interview was conducted in Dutch.

Interviewer: Dennis Kievits
Company: Quintor
Role: Software Development/Research Intern

Interviewee: Henk Bakker
Company: Quintor
Role: Frontend developer

Dennis Kievits: “Hello! First of all, thank you for wanting the conduct the interview with me.”

Henk Bakker: “Of course!”

Dennis Kievits: “First a little bit about myself; I’m Dennis, 23 years old and I am currently studying computer science at Hogeschool Rotterdam. For this study I’m currently having an internship at Quintor to research the effect that applying WebAssembly has on frontend application for Quintor.”

Dennis Kievits: “By conducting this interview I’d like to get a clear overview of what the important aspects of frontend development are for Quintor. Based on the information I’d receive here I’d be able to combine that with my research results to be able to give a better advice to Quintor and it would also help me in choosing the right technologies for the prototype to be built that showcases the effect of WebAssembly for a frontend application.”

Henk Bakker: “Alright.”

Dennis Kievits: “Do you have any experience with WebAssembly yourself?”

Henk Bakker: “I do have some experience with WebAssembly and have also done a couple of projects researching the potential of WebAssembly during Quintor’s Summercamp.”

Dennis Kievits: “Okay cool. I’d still like to share a little bit of information about WebAssembly just so we are on the same page.”

Dennis Kievits: “WebAssembly offers certain advantages, think of performance, smaller download sizes, better portability, more freedom of choice when it comes to programming languages or access to existing libraries and codebases. Some of the downsides being different development requirements (some developers simply don’t have the proper knowledge to build WebAssembly modules), some situations don’t really allow for the use of WebAssembly, such as where the application in question has to perform lots of interaction with the DOM, and there is also more complexity added to the project and debugging will become more difficult.”

Dennis Kievits: “My first question would be about testability of the written code of a project; Is testing of the written code, think of unit testing and code coverage, an important aspect for the frontend development of Quintor?”

Henk Bakker: “Testing is very important to us. It acts as a piece of documentation as well as quality control of the project. Code coverage is more of a helping hand and can be used to review how good the written unit tests are, in terms of what they cover in the codebase. The end goal of testing is CI/CD, automation of testing whilst also bringing the new updates to our clients as soon as possible. Testing is auto documenting of the written code and makes it easier for other developers to get comfortable within the codebase”.

Dennis Kievits: “Is debugging of the written code, think of a debugger of an IDE or other tooling to perform this process, an important aspect for Quintor?”

Henk Bakker: “Analyzing bugs is very important, it also helps us analyze bugs in software that we have in production environments. It’s useful to gather the state of an application (memory), especially when the code itself can in certain scenarios not be followed with relative ease. Most of our developers use a debugger.”

Dennis Kievits: “Is it also important for developers to be able to write code for WebAssembly with knowledge they currently have or that they could attain with relative ease.”

Henk Bakker: “Ease of use is definitely a big advantage. If the bar of entry is too high, it might cause adoptability issues.”

Dennis Kievits: “In terms of the chosen technology for writing WebAssembly code, would there be given a bias to programming languages or frameworks that have proven themselves over a longer period of time or have a larger adoption in the market?”

Henk Bakker: “For Quintor a strong adoption of a technology is important when choosing the right technology for the project. The developers need to have knowledge of these technologies to be able to effectively use them, but for specific use cases certain technologies might be chosen that we do not have the knowledge of right away.”

Dennis Kievits: “Would a relatively low adoption of a programming language for the use of creating WebAssembly modules where a significant advantage is known still be a reason not to use the programming language?”

Henk Bakker: “No way, that would still be underdeveloped. There already exists an ecosystem with native languages, no significant impact on resources. I honestly feel like WebAssembly plays a bigger role that are not in frontend development. I think its main advantages currently lie with serverless applications. As long as there are issues with interacting directly with the DOM from within WebAssembly. Or even simple things such as garbage collection or memory are not straightforward or solved. The only interesting WebAssembly technology as of right now would be Rust. As of right now I don’t think it is for web developers.”

Dennis Kievits: “Would there be a significant advantage for Quintor to be able to use the same programming language for frontend as well as backend development? This could have advantages such as sharing certain models in code, or leverage ‘backenders’ to also be able to write frontend code and vice versa?”

Henk Bakker: “Yes it does, but it depends on what you want to achieve. It can be a very easy solution for certain things. But most technologies that are currently out there are just too young, take for example Blazor, it is currently too much of a conceptual thing. We wouldn’t want to use that for an actual customer’s project, but I do see the potential use in creating a quick and dirty frontend to interact with some in-house application. So, conclusion: Yes, but the technologies simply aren’t there as of yet.”

Dennis Kievits: “Is the (initial) download size of a frontend application an important factor when choosing a certain framework / programming language? Some solutions can be of a size that is minimum like 8MB in size, because of the use of certain runtimes, these can usually be cached though, only causing the hit on first load”

Henk Bakker: “It depends on the type of application we would be building, for apps of internal use, we probably wouldn’t mind the size being somewhat larger. For high performance applications, or mobile first solutions we might consider this to be a more important factor.”

Dennis Kievits: “Thanks for conducting the interview with me!”

17.3 Prototype application code

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["API/API.csproj", "API/"]
RUN dotnet restore "API/API.csproj"
COPY . .
WORKDIR "/src/API"
RUN dotnet build "API.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "API.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "API.dll"]
```

Snippet 12: Dockerfile for Backend API project.

```
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Application.Common;
using Domain.Common;
using Infrastructure.ExtensionMethods;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Common
{
    public class DbSetQueryExecutor<T> : IDbSetQueryExecutor<T> where T:
ModelBase
    {
        public async Task<IQueryResult<T>> Query(DbSet<T> dbSet,
IQueryParams<T> queryParams, CancellationToken cancellationToken =
default)
        {
            var selector = dbSet.AsQueryable();
            var totalCount = (uint) await
selector.CountAsync(cancellationToken);

            // By default, if no ordering is applied, order by createdAt,
for consistency (especially with pagination)
            selector = selector.IsOrdered() ? selector :
selector.OrderBy(x => x.CreatedAt);
            if (queryParams.Pagination != null)
            {
                selector =
selector.ApplyPagination(queryParams.Pagination);
            }

            var elements = await selector.ToListAsync(cancellationToken);
            var pagination = queryParams.Pagination == null
                ? null
                : new LimitOffsetPageInfo(totalCount,
queryParams.Pagination.Limit, queryParams.Pagination.Offset);

            return new DatabaseQueryResult<T>(
                totalCount,
                elements,
                pagination);
        }
    }
}
```

Snippet 13: Backend infrastructure implementation to query a specific model from the database based on query parameters that can be provided via GraphQL directly.