MASTER'S THESIS 2022

Analyzing front-end performance using Webassembly

Jacob Nilsson, Andreas Trattner

ISSN 1650-2884
LU-CS-EX: 2022-27

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-27

Analyzing front-end performance using Webassembly

Jacob Nilsson, Andreas Trattner

Analyzing front-end performance using Webassembly

Jacob Nilsson
jacobnilsson97@gmail.com

Andreas Trattner
mister.trattner@gmail.com

June 29, 2022

Master's thesis work carried out at IKEA IT AB.

Supervisors: Lars Bendix, lars.bendix@cs.lth.se
Joakim Månsson, joakim.mansson@ingka.ikea.com

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

With an increasing demand for high quality web services, IKEA needs to be able to deliver new features while keeping their web services responsive. JavaScript, being the programming language used in the major web browsers, runs on the vast majority of websites today to enable complex behavior. However, being a dynamic, JIT-compiled language, its execution comes at the price of certain runtime overheads and best-effort optimizations. A new technology native to the web, WebAssembly, was created to address many of these shortcomings. The goal of this thesis is to evaluate the performance of JavaScript in comparison to WebAssembly for common algorithms that are not domain-specific.

To accomplish this, a literature study of the two technologies was conducted. IKEAs web solutions were explored through interviews in order to identify common patterns and techniques that should be generalizable to the majority of the web. A custom benchmarking framework was created in order to analyze multiple stages of WebAssembly execution in the browser. Finally, algorithms representative of the identified common techniques were benchmarked in both JavaScript and WebAssembly.

An unexpected result shows that WebAssembly is subject to significant data exchange overheads when non-primitive JavaScript data is passed between the contexts. A more expected result shows that if the data exchange phases are disregarded, WebAssembly executes faster than JavaScript in most circumstances, sometimes on an order of several magnitudes faster. Improvements in WebAssembly, such as using references to external data in an efficient way, could open up the technology to a broader use case.

Keywords: Performance, WebAssembly, JavaScript, Benchmarking, Front-end

Acknowledgements

Thanks to Magnus and Joakim at IKEA. Your constant encouragement, curiosity and support filled us with determination in this task. We would also like to thank all of the teams at IKEA that we worked with. The constant laughter and occasional fika made every office day a joy. Thank you for making us feel like a part of a family. We would also like to thank our friends and family for supporting us during this thesis.

Last but not least, a special thanks to our supervisor at LTH, Lars. Your experience, guidance and sense of humour saved us from many pitfalls and your love for metaphors provided endless moments of enlightenment.

Contents

Chapter 1
Introduction

This section presents an overview and the problem this thesis aims to solve, why it is interesting to solve it and the purpose of the thesis. The limitations and the structure of the report will also be presented.

1.1 Overview

The web browsers of today are tools made in order to parse HTML-documents and, within said documents, run scripts. The widespread adoption of the AJAX (Asynchronous JavaScript and XML) model has fundamentally changed the web landscape; By introducing a communication layer between the client and server, web pages stopped acting as static documents and allowed the same interactive behavior displayed in desktop applications. While the complexity of websites grows, introducing 3D rendering and other processing-heavy operations, the need to keep web pages responsive is more important than ever before. One bottleneck stands out; The "lingua franca" of the internet, JavaScript, is a scripting language. Features such as heap allocations, type speculations and the resulting overhead, garbage collection and “best-effort” JIT-compilation may lead to suboptimal execution times. The reason for the performance issues is not inadequately designed JavaScript engines, but mainly stems from the design of the JavaScript language itself that favors ease of use over performance. Looking upon these pitfalls in combination with the ubiquity of the web platform, an interest in a compilation target for high-level languages on the web is apparent. Over the years, several solutions to this issue have been proposed and explored to varying success. In this thesis, WebAssembly will be researched and benchmarked against JavaScript in order to see if it can execute processes faster. The result will be gathered from our tests and backed by literature studies in order to explain the test results and support the validity of the tests.

IKEA is a large organization that designs and sells ready-to-assemble furniture, kitchen appliances and home accessories, among other goods and home services. To provide their customers with the ability to shop from home their web platform offers web services that enable

that. Their services also include other features such as customizing a room with furniture, searching for furniture using a photo and others. Seeing as they are a large global company with many features on the web their web performance is important. Because of this IKEA is interested in improving performance in their web services and wants to investigate available methods. Since WebAssembly promises native-like performance, IKEA desires to explore its benefits and limitations. IKEA also wants to know where and how they can use it for enhancing their web service performance.

1.2 Problem definition

WebAssembly is a new technology which attempts to perform better than JavaScript by utilizing Ahead-Of-Time (AOT) compilation, static typing and code optimization. One could write code in one of several high-performance languages, such as C, C++ or Rust, compile the code to WebAssembly binaries, and execute them from a Javascript environment. Given the problem statement, the question becomes: Should IKEA adopt WebAssembly in their front-end landscape, why and for what kind of services and algorithms? Our main hypothesis is that WebAssembly will perform better than JavaScript for heavy-processes and time consuming algorithms. This is because WebAssembly code is closer to machine instructions and should therefore have faster execution time compared to JavaScript. However, because of WebAssembly's overhead and startup time JavaScript can be faster at executing some processes due to optimization in the browsers. This leads us to the sub hypothesis which is that the benefits of using WebAssembly highly depends on the type of algorithm, with regards to how the compiler can optimize the JavaScript code. By analyzing the collected data from the measurements, an investigation will be made to explore the reasons for the differences and their consequences.

1.3 Purpose

This master thesis expects to help IKEA know what type of their web services can be improved by using WebAssembly, also how and where to best use WebAssembly for improving performance. Since WebAssembly is quite new, developed in 2017, This paper will also serve as an investigation of a fairly unexplored area. Web developers can benefit from reading this research as a means to understand the current and future state of performance in the web browser. Reading this paper will give an understanding of how a new method for the web can improve the performance in a large company's web service. This thesis aims to increase the current knowledge base in regards to execution time performance in realistic use cases for WebAssembly. Also the WebAssembly community will gain knowledge about pros and cons of using WebAssembly and areas of improvement that could be explored. Substantial investigation has previously been carried out for fairly specific processes, namely those relying heavily on matrix, vector- and arithmetic operations commonly used in fields such as image- and audio processing. One goal of this paper is to analyze a broader spectrum of processes common in the web landscape which will include routines that process large data structures of arbitrary JavaScript data, in order to investigate whether or not WebAssembly will perform better and why.

1.4 Limitations

This thesis is conducted by two master students at Lund's university over the course of 20 weeks, it is impossible to cover all use cases where WebAssembly might perform better than JavaScript. Therefore the scope of the thesis has been limited to focus on a few selected algorithms/processes which is interesting for the IKEA front-end landscape. Utilizing WebAssembly on the server side is interesting and has gathered a lot of traction lately, but due to the manpower and time restriction it will not be in the scope of this thesis. Furthermore, the paper will focus on performance in regards to execution time only. Memory usage will not be explored. Competing solutions, mainly WebGL, will be discussed briefly but will not be taken into account when performance is compared. Considering the time restrictions the JavaScript engine in Chrome is the only one that will be researched. We will briefly look over the other engines but our guess is that the differences in execution time will be minimal.

1.5 Thesis structure

- Introduction
- Background
- Analysis
- Creating the benchmarking framework
- Running the benchmarks
- Discussion and related work
- Conclusion

Chapter 2
Background

This chapter will discuss the initial problem that IKEA faces and it will break it down into research questions that will be explored in order to answer the initial problem at IKEA. A roadmap will be presented about how this research would be best conducted to investigate the research questions. Furthermore there will be a chapter about the theory, concepts and principles that will be used in this thesis. After this chapter the manner of what to analyze and how is clear.

2.1 Context

This section will further explore the IKEA context and the perceived problem therein. Understanding where the problem originates and the problem domain will be necessary to solve it. Furthermore, It is necessary to set suitable research questions that shall be used to gather data to drive the investigation. The initiating problem will be analyzed to find out what areas are relevant for exploration in order to come to a conclusion.

Since it was first introduced, the world-wide web has grown exponentially in its usage and complexity. Simple collections of text documents and file servers have evolved into full scale applications. IKEA is no exception; public solutions such as Kitchen Planner have showcased that demanding 3D applications bring new potential to the web landscape and the IT industry as a whole. It is unlikely that the customers of tomorrow will come to expect less than the ones today do. Serving 490 million customers online from September 2019 to august 2020 [1] is testimony to the fact that IKEA has a large online presence. In order to stay ahead in the game, IKEA needs to keep their web services responsive and well-functioning.

Early websites were made up of simple static HTML documents and some websites still keep to this simplicity. In time, more advanced features in the browser window came in demand. JavaScript was introduced as the native programming language for the client-side of the web. As more features were introduced to the language, the gap between web pages and native applications was blurred. As of June 2022, JavaScript is reportedly used by 98% of all

web pages [2]. However, as the web became more dependent on scripting to drive increasingly advanced behavior, certain shortcomings of JavaScript got more apparent. Its design favors productivity over performance; features such as dynamic typing and automatic memory management results in an overhead during run-time. Early implementations relied on code interpretation, but were later transformed into JIT-compilers for increased performance.

The issue of software performance on the web and the flaws with JavaScript is not a recent discovery, as multiple solutions exist going back to the 1990s. Technologies such as Java Web Applets and Microsoft's ActiveX were created to address both inflexibilities in earlier iterations of HTML and bring the native software experience to the internet through plugins. By the end of the 2000s, the introduction of HTML 5 alongside powerful JavaScript JIT-compilers spelled the end for several of the earlier attempts at running high performance code in the browser window. In recent years, solutions such as Mozilla's ASM.js and Google's NaCL aimed to be the next step in the web landscape, but they failed in having a lasting impact. ASM.js, being a subset of JavaScript, gained some traction since it did not rely on external tools meaning that all browsers that manage JavaScript could run it. NaCL allowed native binaries to run in the browser, but was reliant on browser plugins to function properly and lacked portability.

In 2015, several of the key browser vendors, namely Google, Apple, Microsoft and Mozilla, started collaborating on what would become WebAssembly. It was developed with the intention to supersede ASM.js and get around its shortcomings. Similar to ASM.js, it is a compilation target rather than a language, and can be produced from code written in languages such as C, C++ and Rust. It is a virtual instruction set, closely mimicking X86 assembly, and can run through existing javascript runtimes such as V8, SpiderMonkey and Node.js. As WebAssembly promises performance similar to native programs, all the while running through the JavaScript runtime, it is an interesting potential solution to the initiating problem.

Both memory usage and energy consumption are metrics other than time behavior that can be used to describe the performance of a program. As both resource constrained mobile devices and low-energy IoT nodes are becoming more prevalent, these properties are of great concern and could have both a direct and indirect impact on execution time. Relating back to the initial problem, however, the responsiveness of a program is directly connected to the time it takes to perform tasks. As such, we consider that the research questions should reflect an investigation that uses program execution time as the main metric.

With execution time in mind, the next question becomes what factors could influence it. WebAssembly has, as one of its design goals, a fundamentally different execution model than JavaScript. It is assumed that any measured difference in execution time within the same context is a direct result of said differences, and as such it would be a useful area to research. Furthermore, JavaScript is only a standard, with different browsers that implement the technology. Differences in their implementation could possibly affect how efficiently a program can be executed. Following this it is necessary to explore differences between JavaScript and WebAssembly that affects the execution time and the first research question should therefore be;

1. What are the differences between WebAssembly and JavaScript that affect the execution time, why are there differences and what are the consequences?

    (a) Why do these differences affect the execution time?

(b) How is it dependent on the execution environment, compiler and choice of algorithm? To what extent?

(c) Why does WebAssembly have a start up time, can it be avoided? Is it negligible?

Similarly, all algorithms are not equal, and may utilize the hardware in different ways to process information. By identifying and analyzing various common algorithms used by IKEA, our exploration will answer the question whether or not WebAssembly could improve their services. By exposing potential strengths and weaknesses in certain software patterns when running WebAssembly, we will analyze the extent of the performance differences, as well as discuss the reasons for them and how to address them. This will also provide insight into areas of WebAssembly execution that can be improved by the WebAssembly community.

Connecting back to the IKEA context, there are many potential bottlenecks that could slow down the user experience. Websites often communicate through APIs with back-end systems that store and process data. This thesis is however concerned only with front-end technology in order to have a more focused discussion. All websites are not equal, as the driving processes depend on what kind of service the site provides. In order to answer IKEA's problem, it is necessary to explore how their front-end products function, and which kinds of algorithms that are prevalent in them. It is also necessary to find a suitable approach to evaluate the performance of WebAssembly and JavaScript. There exists multiple methods and frameworks to accomplish this. They should be discussed and possible caveats of their usage carefully analyzed. Therefore the second research question should be to explore IKEA's front-end landscape and current methods and frameworks to evaluate performance.

2. How much gain can be made with regards to execution time by implementing WebAssembly in the IKEA front-end landscape?

   (a) What are common algorithms and services in the IKEA front-end landscape? Are they suitable for WebAssembly? Why?

   (b) What is a suitable benchmark when evaluating code performance, in regards to time to complete different algorithms and processes and why?

   (c) What frameworks exist to benchmark WebAssembly- and JavaScript performance? What are their strengths and weaknesses?

2.2 Method

With the research questions in mind a need for exploring them in the most efficient and thorough manner possible is needed. In this chapter we will discuss different ways of investigating the research questions in order to answer the initial problem. The result of this discussion will be a detailed and motivated roadmap showing how to move forward with the investigation advantageously with regards to the research questions. The first research question focuses on the differences between WebAssembly and JavaScript and the second research question focuses more on the possible gains for IKEA and benchmarking. Instead of exploring the research questions in a sequential manner we have decided to explore them independently. It follows naturally that the benchmarking result will depend on the differences but the result from benchmarking will also provide insights into the consequences of these differences, which should be explored for research question (RQ) 1.

RQ 1 focuses on the differences between JavaScript and WebAssembly, RQ 1.b has a bit more focus on how it is affected by the environment and therefore will be explored later in the thesis. But in order to explore RQ 1.a and 1.c the first step should be to perform a literature study about JavaScript and WebAssembly. The goal will be to learn about different cases where performance, with regards to execution time, of the execution process is different and why. Another way of gathering knowledge about where the two competitors would perform differently could be to first run benchmarking tests and then do literary study to see the reasons for why. But this thesis aims to explore if- and where WebAssembly can be useful for IKEA and therefore simply doing measurements on random algorithms will not be sufficient. It would be more meaningful to first do a study of what type of tests has been performed by others and see what can be applied to the IKEA context. Then perform our own tests targeting IKEA services more accurately, in order to have more precise results for IKEA. We have therefore decided that performing a literature study first will be the best way of conducting this research. The literature study has the aim of helping us understand the differences between JavaScript and WebAssembly which is important to know when we move forward and try to adopt them in the IKEA front-end landscape. The literature study will be done by the authors by searching prior studies that compare WebAssembly and JavaScript, and search documentation about the two in order to find differences.

In order to explore RQ 2.b the second step should then be to understand more about the IKEA front-end landscape. One way of doing this could be to perform a survey and let the developers in the front-end landscape of IKEA answer it. This survey could contain questions about what type of algorithms that are commonly used, and other questions that have become important from the literature study in step 1. With this approach we would be able to reach many developers in a time efficient manner. However the responses would probably not be sufficient for our research. Since we have access to source code through IKEA's github, another way of doing it could be to analyze the source code ourselves in order to find algorithms. However this would be time consuming as we would need to spend time searching for where to look. We decided that the best way to understand more about the IKEA front-end landscape would be to interview developers/teams that work in that area. With this approach we would be able to ask our questions and interact with the developers to fully understand their answers. When deciding upon which we would interview we settled for:

- Product owners and engineering managers
- Web developers within the front-end landscape

The product owners is chosen as participants for the interviews since they know the product and can quickly explain an overview of what the product is for. They will not be able to give detailed answers to what type of algorithms that are prevalent in their products and where, therefore they will not be questioned about it. The product owners have the best insights of the future for the product and that is interesting for us to know, since that also answers the question of if and where WebAssembly can be used in the IKEA landscape. The questions to the product owners will therefore be with a focus on future prospects and goals for the product. The engineering managers are more concerned with people than technology in their day-to-day work. However, seeing as they mingle and network with multiple teams, they could have a great sense of how IKEA at large is moving forwards.

The questionnaire will be similar for both groups. When speaking to product owners specifically, they will be more tailored towards their product and how it will evolve. For engineering managers, the questions will be more about IKEA today and how it is expected to change.

We will also ask Web developers in order to get information about how WebAssembly could be adopted today into the IKEA front-end landscape. We have chosen to interview web developers currently working in that landscape. The selected developers will have knowledge about the specific product they are working with and know what type of algorithms they are currently using and how it performs. The result from this step together with step 1 will be the knowledge of what type of algorithms that would be interesting to compare in JavaScript and WebAssembly.

To explore RQ 1.b and RQ 2.a there is a need for implementations of the chosen algorithms. Therefore for the third step we need to implement the algorithms that have proven to be interesting from step 2. For the JavaScript implementation there is of course the possibility of writing the algorithm in JavaScript, but as a preference from the authors it will be done in TypeScript. It does not matter when comparing execution time since TypeScript will be transpiled into JavaScript in the browser. However for WebAssembly there are many different ways to write the algorithms. Since WebAssembly is a compilation target we need to select which language to write the algorithm in and then compile it into WebAssembly code. One option is to implement the algorithms in C or C++ and then compile it to WebAssembly using Emscripten. However Emscripten does not automatically produce bindings for using WebAssembly modules directly with JavaScript. This would have to be done manually and would be time consuming. An alternative to using C/C++ and Emscripten is AssemblyScript as language and Binaryen to compile it into WebAssembly. One of the biggest cases for AssemblyScript is that it is similar to TypeScript which means that it would be straightforward to implement algorithms in both TypeScript and AssemblyScript, compared to using a different syntax in C/C++. Since AssemblyScript has a narrow-use case, namely just compiling into a static WebAssembly binary, the documentation is smaller in size compared to C/C++. Another path could be to use Rust as the language together with wasm-pack, a plugin that binds rust source code to JavaScript in order to compile it into WebAssembly. A significant advantage of this approach is that wasm-pack automatically produces JavaScript bindings for WebAssembly modules. Rust is a well-known and well-documented language and, together with wasm-pack, is therefore chosen for implementing the algorithms in WebAssembly. The result of this step will be the chosen algorithms written in both JavaScript (TypeScript) and WebAssembly (Rust). This step is not directly connected to any of the research questions but the product, the implemented algorithms, from it will be used for measurements in the following steps.

For RQ 2.a there is a need to measure the difference in execution time for the implemented algorithms. Therefore RQ 2.b and RQ 2.c are explored in order to have the best possible way of comparing JavaScript and WebAssembly. The fourth step is therefore to find ways to measure the difference for these algorithms in both WebAssembly and JavaScript. As the initial problem is ultimately concerned with improving the user experience, one method to achieve this would be AB-testing; One web page is created using only JavaScript, and the other takes advantage of WebAssembly. Participants of the test could go through a set of procedures on the two versions, and finally answer a survey about the user experience. This would more directly answer the question whether or not IKEA should explore using

WebAssembly in their front-end. However, this has certain limitations, for example, loss of precision, anchoring the results in the subjective, and if the execution time for the various algorithms are short to begin with, the results could become inconclusive. Being a subjective measurement, the results would be locked on the ordinal scale- they are only meaningful in relation to each other. Gathering results through benchmarking with timestamps, on the other hand, is perhaps the better solution; It can capture discrepancies in execution time with a higher resolution and is objective. These measurements can be analyzed using many different scales, including the ratio scale, which further extends the ordinal scale by allowing the magnitude of the difference between results to be measured. There are potential issues, however; the resolution of the timestamp could introduce errors, as well as a certain code execution overhead that affects the running time of the program. The potential issues with timestamps, and what is considered a suitable depth of analysis will be further discussed in a later section. Benchmarking through timestamps will be the approach this thesis uses as it is considered more fruitful in exploring the research questions, while still noting that subjective results could be sufficient to solve the initiating problem. There are different ways to benchmark code performance. This step will result in a chosen benchmarking framework to collect metrics in.

The fifth step will be to further explore RQ 1.b, RQ 1.c and RQ 2.a. We will use the framework from step 4 and run the tests and collect metrics on the execution time for the algorithms in JavaScript and WebAssembly. Most importantly this step is also to explain and analyze the collected data and discuss the possibilities with WebAssembly. The data from the benchmarking will be analyzed and compared to existing knowledge in order to come to a conclusion and answer the initial problem.

To summarize the roadmap for exploring the RQs and with that be able to answer the initial problem:

1. Literature study regarding WebAssembly and the javascript execution process.
2. Analyze IKEA's front-end landscape.
3. Analyze and select appropriate framework(s) for benchmarking the algorithms
4. Create implementations of selected algorithms in both JavaScript and WebAssembly
5. Utilize selected framework for benchmarking algorithms and analyze the collected data.

2.3 Theoretical fundament

This section will present the theory, concepts and principles that will be used as foundation for the research in this thesis. It should be read in order for the thesis to be fully understood. The reader is encouraged to seek further information in the corresponding fields if needed, as this section only contains an overview.

First, concepts in the field of compilers are introduced. A brief introduction to concepts in web programming follows. Advanced programming introduces data structures and various sorting methods. Concepts of Programming Languages introduces the ideas of type systems. Time complexity is explained in the section algorithms, data structures and complexity. Finally, evaluation of software systems explains terminology and methods for benchmarking.

2.3.1 Compilers

Knowledge in compilers will be especially helpful in analyzing Research Question 1. Many of the discrepancies between JavaScript and WebAssembly's execution models can be explained with the use of concepts in this field.

Code execution models

There are two dominant ways to execute code. Compilation means that human-readable code is transformed into machine instructions before they are executed. Interpretation means that human-readable code can be executed directly through a program called "interpreter". Interpreted code is usually several magnitudes slower than compiled code.

Code compilation strategies

AOT-compilation means "Ahead-of-time", and produces the machine-readable code before program execution. It commonly involves several code optimization strategies, such as utilizing CPU registers instead of RAM whenever possible. The target architecture must be known beforehand as the resulting binary contains hardware instructions. JIT-compilation means "Just-in-time" and combines code interpretation with compilation to varying degrees. When certain parts of the code are called often enough, it is tagged as "hot"; the executing runtime could compile them into machine instructions for increased performance. One benefit of this strategy is that it could find optimization opportunities that are only revealed during runtime that would be impossible to find using AOT-compilation. The caveat is that it could take a while until the code has been properly optimized. Some languages, particularly javascript running through the major browser runtimes, use a "best-effort" strategy; Due to some properties of the language, the JIT-optimized code could become obsolete or erroneous, resulting in the runtime falling back on interpretation.

Virtual instruction set

One problem in software science is portability. Compilers such as GCC will produce machine code from C that is applicable to a certain computer architecture and operating system. If the program needs to run on another computer, it would most certainly need to be compiled again. Virtual instruction sets are a way to address this issue: Instead of creating hardware-specific instructions, the compiler creates an intermediate format, targeting an "ideal" computer. The result is a program that has undergone certain optimizations without being locked to a certain architecture. The virtual program can then be executed from any computer that has the appropriate runtime, usually with a minimal loss in performance. Adve et al. found that for short runs, translations from virtual object code to machine code resulted in a 1% additional execution time overhead[3].

2.3.2 Web programming

There are two broad divisions of web programming – front-end development which is also called client-side development and back-end development (server-side development). This

thesis focuses on the front-end development which mainly consists of JavaScript, HTML and CSS. JavaScript is used to interact with the user and make the website dynamic. JavaScript Object Notation (JSON) is a common data format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs (or other serializable values like arrays). JSON objects are widely used when sending data to and from a server. An example of a JSON object;

```
{"firstName":"John", "lastName":"Doe"}
```

If the website handles inputs and outputs such as gathering data from a server and then presenting it to the user, there is a need for using non-blocking function calls. It takes time for a function to fetch data from an API. Although multithreading is one approach that could solve this issue, an idiomatic way of handling this problem in JavaScript is the usage of asynchronous functions. Asynchronous programming allows a user to continue their business in an application, while processes run in the background, thus enhancing the user experience. It is common to use an asynchronous function with await where the fetch method is used to get data from an API. To make a function asynchronous simply write async in its declaration as below:

```
async function foo() {/*tasks*/}
```

2.3.3 Advanced programming

In this thesis concepts about common data structures will be important and since different sorting algorithms will be benchmarked a fundamental knowledge about them, mainly quicksort and heapsort is needed.

Data structures

Data structures is about organizing data in memory. One commonly used data structure is an array which is a collection of memory elements in which data is stored sequentially. Another example is a list or a queue where the data is stored sequentially, an example where the data is stored nonlinearly or not sequentially is a tree data structure.

Data structures can be classified as either static or dynamic. Static data structures are when the size of memory is allocated at the compile time, therefore the size is fixed. Dynamic data structures are when the size is allocated at the run time and therefore the size is flexible. This is important to note for this thesis since Rust/WebAssembly need to specify memory size at compile time whereas JavaScript manages to do it at runtime.

Sorting algorithms

Sorting algorithms are used to rearrange a given array or list elements by using a comparison operator on the elements. The comparison operator is used to determine the new order for all the elements in the respective data structure. Some common sorting algorithms that are widely used will be explained below with corresponding time complexity.

Quicksort Quicksort is what is called a divide and conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot element. There are different versions of quicksort that pick pivot in different ways. One is to always pick the first element as pivot. Another is to pick the last element as pivot, or to pick a random element as pivot or pick median as pivot. The time complexity depends on how good the pivot turns out to be. The best case for this algorithm is when the pivot will be in the middle of the sorted data structure then the time complexity will be $O(nlogn)$, in the worst case the pivot will end up either first or last and corresponding time complexity will be$O(n^2)$.

Heapsort The heapsort technique is based on binary heap data structure. Min-heap or max heap. For min-heap the root element is minimum and for max heap the root is maximum. For min-heap it consists of finding the minimum element and placing it at the beginning (root) and then repeating the process for the remaining elements building up a binary tree. By deleting elements from the root we can sort the whole array. The time complexity for heapsort is $O(nlogn)$.

2.3.4 Concepts of Programming Languages

Knowledge in Concepts of Programming Languages is useful when discussing RQ 1. JavaScript and WebAssembly have differences in their type systems that could affect the execution time.

Type systems

Statically-typed languages enforces that type information is available to a program at compile-time. The types are connected to variables or fields within the program. Beyond allowing a program to be guaranteed to be type-safe, knowing the size of the variables in the program allows the compiler to place fixed-size data in stack memory, which is more efficient than allocating heap memory. In Dynamically-typed languages, types are instead connected to run-time values, and a variable's signature may change during program execution. This usually results in a certain overhead. This is useful for research question 1.

Data passing

In programming, there are multiple ways to pass parameters to a function. **Pass-by-value** means that parameters are copied into the function, effectively duplicating data. **Pass-by-reference** means that the address of a parameter is copied into the function. Depending on what data is being passed into the function, pass-by-value can be a very expensive operation-passing data structures that are large or allocates data on the heap may lead to an overhead. Meanwhile, memory addresses are 32- or 64 bit integers, regardless of the size of the data structure they're referencing.

2.3.5 Algorithms, data structures and complexity

Knowledge in algorithms and time complexity is useful for both RQ 1 and 2. It gives the proper tools to reason about how execution time should scale in regards to certain algorithms.

Furthermore, it will be useful when performing an analysis on IKEA algorithms, in order to identify potentially demanding ones.

Time complexity and Big-O notation

In the realm of computer science, a common classification of algorithmic performance is time complexity. Rather than being a metric for the exact number of clock cycles until a routine terminates, it is an abstraction, giving information regarding the upper bound of the execution time in relation to the input data. It is written on the format $O(f(n))$, where n is the size of the input. As an example, if a program P is expected to take twice as long to execute when the input size is doubled, the program P has the time complexity $O(n)$ ($P \in O(n)$). Since then notation is an upper bound of how a program's execution time scales, it is also true that the same program $P \in O(n^2)$. In most discussions of this paper, the lowest possible bound is used, and as such, it is simply stated that P has the time complexity $O(n)$ in the previous example.

2.3.6 Evaluation of software systems

Knowledge in evaluation of software systems will be useful for RQ 2. It sets down standards for metrics and approaches to benchmarking. Various methods of evaluation are used to fuel the discussion concerning what manner of data extraction is the most suitable in order to best quantify differences in performance between the two solutions that are studied.

Metric scales

There are certain scales that can be used to evaluate properties of software systems. The Ordinal Scale applies when the only meaningful analysis of entities is their ordering. For example, in AB-testing, one could gather that most users prefer system A over B (A > B F(A) > F(B)), but the magnitude of this difference (|F(A) - F(B)|) cannot be quantified. The Ratio Scale is an extension of the ordinal scale- an ordering between entities is possible. However, it is also possible to measure the relative difference between entities that are mapped to the ratio scale. This thesis will mostly use the Ratio Scale to analyze results.

GQM model

The GQM stands for Goal Question Metrics and is a model for taking goal oriented measurements within software development. The GQM model is defined as first specifying the goals with the measuring (Goal), then the questions that will answer the goals (Question) and lastly specify the metrics that will answer those questions (Metrics).

The GQM blueprint: Object(s) of study This could be the code, product, process or a resource Purpose What is the purpose of taking measurements, could be to understand how something works or to improve the object(s). Quality focus What is the focus for measuring, could be cost, predictability, reliability or execution time. Perspective From whose perspective it is, developers, scientists and product owners are interested in different types of information. Context In what context has the measurements been conducted, for example at which company.

Data analysis

General purpose operating systems, such as Windows, MacOS and Linux are commonly used among web consumers. They have very intricate systems that prioritize and regulate processes, which could affect the execution time of programs by temporarily halting processes. This can result in variations in benchmarking tasks.

By running a benchmark multiple times, a series of measurements are acquired. Under ideal circumstances, they can be generalized into a Normal distribution the data series is assumed to converge towards a Central value. Standard deviance is a measurement of how close to the central value a randomly selected benchmark will likely be. A high standard deviance implies that the data series has a high degree of variation.

Outliers are measurements in the data series that are unusually large or small compared to the central value. What is considered an outlier can be rather subjective. One method of exposing potential outliers is using box plots, by identifying the median, upper quartile and lower quartile of the series, all values that fall outside of this range are considered outliers.

Chapter 3
Analysis

The problem with benchmarking algorithms is that if the algorithms were chosen randomly, which they would have been since the authors limited time and resources, the result of the benchmark would be vague and it would be difficult to reach a conclusion. Therefore this section aims to analyze the current literature together with the IKEA context and ultimately result in relevant algorithms that will be useful to benchmark. The reader learns about the current state of web technology, the root cause of the initiating problem and potential pitfalls in attempting to solve it. Interviews are conducted in order to find out what experiments are relevant. The result is a set of algorithms to explore, that have been motivated by their prevalence in the front-end landscape. Information regarding the algorithms and existing benchmarking tests will strengthen- or challenge the data obtained from our tests in chapter 5 “Running the benchmarks”.

3.1 Literature Study

In order to study the research questions we need to have algorithms that can be analyzed in both JavaScript and WebAssembly. This literature study will be done with the aim to understand differences between JavaScript and WebAssembly and how they might affect the execution time for different algorithms. With this understanding an analysis of algorithms that are dependent on those differences can be done. This will in turn result in a list of algorithms or processes that would be interesting to use for benchmarking given the differences of JavaScript and WebAssembly.

3.1.1 JavaScript

In order for the computer to understand and execute JavaScript code it must first be converted into machine-readable code. A JavaScript engine is a computer program that exists within a browser and that does this conversion. Since it is the JavaScript engine that does

the optimization when running javascript code, it is interesting for this thesis to study the engines.

It is no exaggeration to state that JavaScript was not an advanced language when it was created, rather just to interact with HTML-elements. Brendan Eich developed it in only 10 days [4], but it has still survived and is the main language used for websites. Since it is the optimization that has made JavaScript what it is today it is important for this thesis to study the execution process within the engines, in order to see how it can compete with WebAssembly. The aim of the study is to find cases where JavaScript has its strengths and can execute quickly and where it has weaknesses and executes slowly. Below in table 3.1 are common browsers and the name of their corresponding javascript engine:

Browser	JavaScript engine
Google Chrome	V8
Mozilla FireFox	SpiderMonkey
Edge (internet explorer)	Chakra
Safari	JavaScript Core Webkit

Table 3.1: Browsers and their corresponding JavaScript engine

To study all of the optimizations in the engines above in detail would be too time consuming. Therefore the choice was made to study in detail the V8 in Google Chrome since it is a widely used browser.

More detailed analysis of V8 engine

The V8 differs from other JavaScript engines in the way that it directly converts from an incoming JavaScript function to non-optimized native code. When the V8 processes JavaScript code there are three main steps taken; 1. Parsing the code, 2. Compiling the code, 3. Executing the code.

Step 1. Parsing the code During the Parsing phase, JavaScript code is decomposed into tokens. For example in: const sum = 4 + 1 the tokens are "const", "sum", "4", "+", "5". These tokens are then sent to the syntax parser which converts the code into an Abstract Syntax Tree (AST).

Step 2. Compiling the code The V8 engine uses one interpreter called Ignition and one compiler called TurboFan [5]. The ignition interpreter gets the AST and converts it into bytecode which proceeds into the execution step. On further executions the V8 engine finds patterns such as frequently executed functions, frequently used variables, and compiles them using TurboFan to improve performance. Suppose the performance degrades or the parameters passed to the function change their type, then the V8 decompiles the compiled code and falls back to the interpreter.

Step 3. Executing the code The bytecode is executed by using the memory heap and the call stack in the V8 engine's runtime environment. The call stack is where data is pushed onto for execution and popped from after their execution.

Just-In-Time compilation

The TurboFan compiler doing the optimization mentioned in step 2 above is a Just-In-Time compiler. One limitation with JIT compilers is that they need time to compile and optimize a program before it can execute. A benefit with JIT compilation is the optimization possibilities, since it is done at run time as opposed to Ahead-Of-Time (AOT), this means that the optimizations will be done dynamically with how the code is being used, such as for inline caching, which will be analyzed later in this chapter [6]).

Because of JIT compilation it could be motivated that running the same type of operations many times, allowing TurboFan to produce optimized code, should benefit the execution time of JavaScript. In "On the runtime and energy performance of WebAssembly: Is webassembly superior to JavaScript yet?", the authors performed benchmarking tests for different sorting algorithms with varying input sizes, they used c/c++ as the language for WebAssembly. They concluded that WebAssembly was faster at executing sorting algorithms, although not to an extreme extent. Interestingly they also found that the bigger the input size the smaller the performance gap between WebAssembly and JavaScript became, they argued that this was because of JIT compilation[7]. It would therefore be interesting to study where the breaking point for sorting algorithms is for WebAssembly compared to JavaScript. To build upon the data of the mentioned paper, sorting algorithms are added to the list of interesting algorithms for comparing WebAssembly and JavaScript.

Hidden classes and inline caching

Since JavaScript is a dynamically typed language, properties can be added dynamically to an object, the type of these properties can be changed dynamically, these possibilities pose a problem for the memory storage. In a statically typed language for example Java, the type of an object's property is known when compiled. The memory storing of the values for the properties/pointers to those properties are in a contiguous buffer with a fixed-offset based on the property's type. This is not possible in JavaScript since a property's type can change during runtime.

The ways that JavaScript stores the location of object properties are in dictionary-like objects, this makes the process of retrieving an object's properties time consuming. In order to optimize this process the V8 utilizes hidden classes [8]. Whenever an object is instantiated in V8 a hidden class is attached to it with the purpose of optimizing the property access time. When the created object is modified by an added or changed property a new hidden class will be created, with all the properties from the previous class, and include the new property. The old hidden class will be updated with a transition path, this path is important because it allows objects to share hidden classes if they are created in the same way. For example if two objects share a hidden class and both are assigned a new string property, the transition path ensures that the newly created hidden class is assigned to both of the objects. It is important to note that hidden classes depend on the order in which properties are added to objects. For example if two objects get the same type of properties but in different order they no longer share the same hidden class and there is no optimization in look up time.

Inline caching optimization makes use of hidden classes by keeping track of the addresses of the properties on objects. When a method is called the V8 keeps track of what type of objects that are passed as a parameter. It uses that information to assume what type of object will be used as a parameter in the future for that method. If its assumptions prove to be

correct it will be able to bypass the process of figuring out how to access the passed objects properties. It will instead use its cached information from previous lookups to the objects hidden class which entails memory penalties, however memory penalties is not the focus of this thesis.

Since inline caching with the help of hidden classes [9] is to improve the execution time remarkably it would be interesting to test its capabilities and how they can compete with WebAssembly since webAssembly cannot use run time optimizations in the same way since it is compiled and already optimized. In [10], the authors tested how much these optimizations improved JavaScript performance while running different benchmarks. Most notably they used the JSBench which is a benchmark consisting of real javascript code assembled from actual websites, they also tested Kraken, SunSpider and Octane which are common benchmarks for measuring javascript engine performance. They first tested JavaScript performance by only disabling the JIT compiler and then also the inline caching. Their first result with no JIT but inline caching resulted in an average instruction count increase by 6.4x, 2.7x, and 3.1x in Kraken, Octane, and SunSpider, respectively, while it decreases the count by 0.95x in JSBench. Disabling the inline caching as well increases the average instruction count by 84.6x, 43.6x, and 31.9x in Kraken, Octane, and SunSpider, respectively, and by only 1.4x in JSBench. As noted for Kraken, Octane and SunSpider there is a remarkable increase in instruction count, which in turn increases the execution time, when the inline caching is disabled.

This means that JavaScript should perform well with algorithms that are consistent with the input to their function calls and that are called many times. Also, many built-in functions in JavaScript are higher-order functions; functions that compare two objects usually take a function that specifies how objects should be compared. Before the engine can optimize this comparison, it will compare objects slowly, meaning a long time for sorting but after optimization it will compare quickly.

Type speculation and number optimization

To generate optimized code for code sections that are executed frequently the V8 uses type speculation. However these assumptions require to be frequently checked in order for the generated code to not be violated. In the case that the assumptions were wrong the program reverts back to the unoptimized code, this is called deoptimization and the frequent checks are called deoptimization checks [11]. An example of a simple javascript function is;

```
function add(a,b){
    return a+b;
}
```

Because of javaScript's kind and accepting syntax this is a valid function that can be called with two arguments of any type. JavaScript has defined double-precision floating-point as the number format. This means that there is no separate integer type and the number 37 consumes as much memory space as the number -9.75. If the add function above is called with two 32-bit integers it will still use double-precision floating point addition as number format, before optimization, which has double the latency of an integer addition, on a Cortex A76 core.

The interpreter in a JavaScript engine will collect dynamic type information for object access and type information. In the function "add" above, the interpreter will execute with floating point addition but if it finds that the function is always called with integers the

JIT compiler will use type speculation and make an assumption that add will be called with integers. It will then generate machine code which is optimized for integer addition instead of double addition. The deoptimization checks will be used in order for the assumptions not to be violated. If a function call comes with two floats then the optimized code can no longer be used, a deoptimization step or bailout is done and the code reverts back to unoptimized code. This means that JavaScript can continue running optimized code if the calls do not change the type of the arguments. The deoptimization checks that are there to verify assumptions have been tested to pose an execution overhead of 8%. More about type speculation and its consequences can be found in [11]. The type speculation is beneficial when optimizing away from using the JavaScript floating-point as number format and instead using optimized machine code for integer operations.

It is interesting to see wheteher or not type speculation can make JavaScript a contenter against WebAssemby since types are already known for WebAssembly. In thein the It can be In the case of always calling a function with the same type of the arguments, allowing the TurboFan to generate optimized machine code, JavaScript should therefore be close to WebAssembly with WebAssembly in regards to execution time. But to find data for this claim these types of algorithms should be tested.

Garbage Collection

In order for the browser to handle memory management, garbage collection was introduced as a feature in modern browsers. It provides the ability to allocate and free memory effortlessly in modern language runtimes. Furthermore, it has the ability to prevent some memory leaks that could be caused by programming errors, such as missed deallocation calls. Most garbage collectors (GC) have similar ways of handling memory. the common tasks that are done periodically includes:

- Identify live/dead objects
- Recycle/reuse the memory occupied by dead objects
- Compact/defragment memory (optional)

For traditional GC there is the concept of "stop-the-world" which is when the garbage collection tasks pauses the main thread while the tasks are performed. This affects the user experience in a bad way in the form of poor rendering and latency on the webpage. However the GC in V8 has come a long way with garbage collection. V8 uses the garbage collector Orinoco which makes use of the latest and greatest parallel, incremental and concurrent techniques for garbage collection [12]. However it is hard to identify what type of algorithms that will perform time inefficiently because of GC. It can be assumed that the general performance of JavaScript is affected by garbage collection in that it needs to pause a bit for clearing up memory. Rust does not utilize runtime garbage collection since it handles memory in a different way. But having an algorithm that adds a lot of objects to the heap without clearing it in WebAssembly will yield memory penalties and is therefore discouraged.

Memory layout

JavaScript does not use sequential memory but rather uses mappings to deal with the dynamic nature of objects. Each object is stored with a mapping, the first time the object is created

the engine assigns it a large slice of memory and stores it with an empty mapping. The engine remembers the mapping of property names to their stored value in the object array memory. When a property is added to the object it saves it in the first place of the object's memory array. After executing;

```
let p = {a:1}
```

P's memory array will be looking something like this

...	a	...	1	...

As can be seen by the table above the object's data array and the mapping array might be stored in different places in the memory. The left array is the mapping and and the right array holds the property values. When a property is added to the object the engine optimizes to reuse the old mappings and store the incremental changes with a link to the previous mapping, similar to a linked-list, even though this optimization, dynamically adding a property still takes time as compared to a statically typed language. A statically typed language will not need to worry about the mapping array or object's data array needing to be restructured since sizes are specified from the beginning [13]. This memory layout would suggest that algorithms that use objects with multiple properties will be executed faster in WebAssembly, since WebAssembly uses sequential memory.

3.1.2 WebAssembly

One of the motivations for creating WebAssembly was high performance, while still operating within existing web technologies. The purpose of this section is to discuss strengths- and weaknesses of WebAssembly, to which extent they may affect performance, and connect them to specific processes and algorithms. In order to achieve this, discrepancies between the technologies and features unique to WebAssembly are accounted for in order to motivate the discussion.

Execution model

The biggest difference between JavaScript and WebAssembly is their format. JavaScript is a high-level, human-readable text format. WebAssembly is a virtual instruction format, more akin to Java bytecode or x86 assembly. it is not supposed to be written manually. Rather, it is a compilation target for other languages, such as C, C++ and Rust. WebAssembly binaries are sent to the client computer, and executed through the browser JavaScript runtime [14].

WebAssembly aims to improve upon its predecessor ASM.js. It has been shown that WebAssembly running in the browser consistently outperforms ASM.js in regards to execution time, with a mean speedup of 1.54X and 1.39X in Firefox and Chrome respectively [15].

In V8, WebAssembly does not rely on the interpreter Ignition. Rather, it is immediately compiled into machine instructions using the baseline compiler Liftoff. After machine code generation, V8 runs the optimizing multi-pass compiler Turbofan, which aims to further improve the generated code. This is a slower process, but as soon as functions are optimized, they replace their Liftoff counterparts for the remainder of program execution [16]. This differs from JavaScript, which is always liable to the optimize-backoff process. An optimized javascript function may fail if the call signature changes from what has been speculated, falling back on the slower interpreter mode.

Claiming that WebAssembly is AOT-compiled is only partially true. In Chrome's V8 engine, WebAssembly code is further processed using JIT-compilation during runtime. Previous results have shown that while JavaScript's execution speed is significantly improved by JIT-optimizations, they are not as beneficial on WebAssembly code. Experiments on the polybenchC benchmark show that JIT-optimization improved JavaScript execution speed up to 120 times compared to interpreted JavaScript, with an average improvement on execution time around 38.4 times [17]. Meanwhile, WebAssembly was found to be on average 0.4% faster with JIT-compilation. It is partially credited to the fact that many optimizations that would benefit WebAssembly have already been applied during compilation from the source language [17].

Many existing WebAssembly compilers, such as Cheerp, Emscripten and RustC, are built upon the LLVM (Low Level Virtual Machine) compiler framework. In their report "Understanding the Performance of WebAssembly Applications" [18], Yutian Yan et. al. conclude that the efficiency of LLVM optimizations on WebAssembly is questionable, and might have a negative impact on its performance aspects.

These facts result in a technology which at least should be faster to execute than JavaScript before it has been optimized, regardless of what code is being executed.

Instantiation in JavaScript

JavaScript source code, as previously mentioned, is a high-level, human readable text format. WebAssembly, on the other hand, is delivered- and executed in a binary format. Experiments in Firefox show that it can be decoded an order of magnitude faster than ASM.JS, its direct predecessor which is based on a subset of JavaScript [19]. At the time of writing, WebAssembly does not use the common JavaScript instantiation mechanisms without the use of external tools such as JavaScript Web Bundlers like WebPack. Calling WebAssembly logic from JavaScript requires asynchronous instantiation of WebAssembly modules [20]. The module only needs to be instantiated a single time, a "cold-start". The module can then be reused without any additional start-up time. There is a WebAssembly standard proposal to implement a WebAssembly-ES Module integration [21], which would effectively initialize WebAssembly modules together with the JavaScript code. An investigation into the start-up time of WebAssembly modules shows that it it directly tied to the size of the module; In the experiment, reducing the size of a WebAssembly binary from 15.2 Mb to 28.6 Kb resulted in the startup time being reduced from 4.472 to 0.104 seconds [22]. The start-up time could be an issue. Algorithms which have a relatively short execution time and are not used frequently enough should execute faster in JavaScript than WebAssembly.

Data types and arithmetic

The WebAssembly standard defines a very limited number of numeric data types; integers- and floats of 32- and 64 bit width respectively, the 128-bit width vector type, specifically used for SIMD instructions, and reference types, holding function [23]. In unoptimized JavaScript, all numeric values are considered as double-precision floats if they can be represented using 64 bits.. This can prevent certain optimizations from applying until the runtime optimizes the code. An experiment on the javascript library Long.js, which provides 64-bit integer operations, shows that WebAssembly outperforms JavaScript in multiplication, division and

remainder operations, with 64-bit division being roughly twice as fast in WebAssembly. This is accredited to WebAssembly translating into fewer machine instructions due to a more specialized number type system [18].

As a consequence, it can be inferred that programs that rely heavily on arithmetic should perform better in WebAssembly than JavaScript. Examples of this include cryptographic hash functions, matrix- and vector operations, and signal processing algorithms such as convolution and fast fourier transforms. In 3D graphics, many algorithms such as inverse square root are executed for every single frame; even a small improvement in execution time can make a large difference for the frame rate.

SIMD

SIMD (Single Instruction, Multiple Data) is a data-level parallelism technique (in contrast to multithreading, which concerns process- or task parallelism). The instruction loads two sequences of operands, and applies some operator to the data, rather than processing the sequence of operands sequentially [24]. Originally in the works to be an extension of the ECMAScript standard, it was scrapped in favor of the feature being worked into WebAssembly [25]. In WebAssembly, this is realized with the vector type, consisting of four 32-bit integers, enabling two sequences of four operands to be processed simultaneously, if the underlying hardware supports the instruction. A recent study that benchmarked a HEVC video decoder showed that introducing SIMD-instructions resulted in a speedup of around 4x compared to the original program [26]. The feature is being further extended with the introduction of "Relaxed SIMD", and the Fused Multiply-Add operator, which can perform the operation $(a * b) + c$ in a single instruction (Relaxed SIMD proposal). On hardware that supports it, performing intensive numerical calculations on sequences of data should reach far higher performance. Examples of this include problems in signal processing, such as processing images, audio and video. Furthermore, many machine learning tasks rely on computations over large numerical sequences.

Memory

WebAssembly modules do not define their own memory. Instead, when a WebAssembly module is instantiated from JavaScript, a block of untyped, sequential bytes are allocated and shared with the instance [27].

WebAssembly does not have any mechanism for garbage collection, but it has been proposed as an extension to the standard [28].

Static typing

WebAssembly contains type information. As such, the runtime does not have to perform certain checks which are instead weaved into the program itself. Certain overheads are removed from program execution, which should result in a program that runs faster. The impact of this depends on how soon and well a javascript runtime can produce type information during execution. In theory, the same type information that is present in WebAssembly will eventually be found in JavaScript, leading to type-specific optimizations to occur which removes certain overheads. However, as stated previously, a failed deoptimization check in JavaScript

will lead to the runtime falling back on the slower interpreter mode. Furthermore, even well-optimized JavaScript runs deoptimization checks at every function call, which WebAssembly does not require.

Passing data

As previously mentioned, WebAssembly provides support for a fairly limited set of types. Since WebAssembly is a stack machine which works solely with numeric data, calling a WebAssembly function from JavaScript only allows input- and output in the form numbers. The key to getting around this limitation is using the shared memory between JavaScript and WebAssembly. For example, passing string data to a WebAssembly function would instead work by allocating enough space on its stack memory to hold the bytes of the string and write the data directly. The function would then receive an integer pointing to the string's position in memory, together with its length. Since this is a cumbersome mechanism to write by hand, many WebAssembly toolchains such as WASM-pack for Rust automatically generate the necessary helper code to allocate- and free fixed-size sequences from the shared memory.

A new data type, stringref, has been proposed for WebAssembly. Rather than requiring strings to be copied into a shared buffer, WebAssembly modules would be able to directly interact with JavaScript string without any overhead [29].

Another issue is JavaScript objects and generic arrays (which are syntactic sugar for objects). These are, fundamentally, mutable sets of key-value pairs. Furthermore, there is no limitation to what kind of data they may hold. Due to JavaScript's loose rules regarding memory layout, two objects could be structurally identical, yet position their properties at completely different memory offsets.

WebAssembly, aiming to be a minimal standard, has no method of directly interacting with JavaScript objects, despite their key importance to web applications. Instead, the programmer has to define how JavaScript data is decoded- and encoded in WebAssembly memory. There are many frameworks that enable generic encoding- and decoding between the instances, such as MessagePack for C++ and SERDE for Rust. SERDE encodes JavaScript data into JSON strings, shares it through WebAssembly memory, and then decodes it in WebAssembly [30]. This could potentially introduce a large overhead compared to javascript, where passing large data structures to functions is resolved through memory pointers or **pass-by-reference**, rather than using data duplication or **pass-by-value**. Algorithms that handle large amounts of data, such as sorting algorithms, mapping algorithms, and grouping algorithms are expected to scale poorly performance-wise as the volume of data increases in WebAssembly.

On the other hand, processes which rely on fixed-size, shared numerical data could go for another approach. An example of this is drawing on a HTML canvas element. The pixels could be represented with a byte buffer, instantiated from WebAssembly and placed in shared memory. There is no need to pass this data between the context with parameters and function outputs. Since the data has a fixed size and known word size, calculating image data in WebAssembly and then reading it back in JavaScript should come with little or no overhead. With this in mind, together with SIMD-operations for data parallelism when groups of bytes are processed together, handling- and generating image data in WebAssembly should be beneficial.

Summary

The very core of WebAssembly, being a low-level, static-typed instruction set, should theoretically increase performance over JavaScript across the board. WebAssembly modules are parsed faster than ASM.JS and require less overhead during runtime. Features such as SIMD-operations could potentially increase the execution speed of iterative calculations up to 4 times due to data parallelism. Data passing between a JavaScript context and WebAssembly context is cumbersome for data which is not numerical, such as strings and generic arrays, since it cannot directly share such data through references. Since WebAssembly needs to be loaded manually during runtime, short-running, single time processes are likely to perform worse in WebAssembly than JavaScript.

The following list outlines groups of algorithms that have been identified as interesting for further research.

- Sorting algorithms
- Grouping algorithms
- Mapping algorithms
- Filtering algorithms
- Machine learning algorithms
  - Back propagation (Deep learning)
- Signal processing algorithms
  - Convolution
  - video, audio- and image processing
  - compression
  - Fast fourier transform
- Cryptographic hash functions
- Matrix- and vector algorithms
  - Matrix multiplication
  - LU-factorization
  - QR decomposition
  - Newton-Raphson
- 3D graphics
  - Inverse square root
- Image data generation
  - QR codes
  - Animations

3.2 Interviews

In order to focus on algorithms that would be relevant for what IKEA is doing, it is required to see how they use web technology. Therefore, interviews were conducted with various IKEA employees to find out which algorithms previously mentioned should be of particular importance in this context. The literature study above resulted in both general groups of problems and specific algorithms that are interesting to study, however due to limited time and resources that list needs to be reduced further.

3.2.1 About the interviews

The interview phase of the thesis is done with the goal of finding out which processes are being used, or are likely to be used in the future, by IKEA. Given the scope of this thesis, it would not be feasible to implement algorithms in all of the groups mentioned in the previous section while providing a fruitful discussion around their time behavior. Furthermore, since the problem lies in the IKEA context, it is necessary to know how they use web technology in order to provide an adequate answer to whether or not WebAssembly is a viable solution.

Selection of participants

Since WebAssembly's presence in the web landscape is a recent development, many employees within IKEA have limited knowledge of it. The focus was therefore to divide participants into two categories that correspond with their position at IKEA. Namely,

- Product owners and engineering managers
- Web developers within the front-end landscape

In total, we interviewed 7 product owners, 3 engineering managers and 11 developers. See Appendix A for the interview guide.

Interview answers

Product owners and engineering managers We decided to ask more general questions about the product and what the future goals are since product owners are not deeply engaged with the code development. But the direction of a product is important for us as it allows us to analyze algorithms that will be used in the future. The interviews aimed to find out information about the current- and future state of technology at IKEA. Although interesting, a lot of information gathered was not directly relevant to the problem of responsive web pages. The following outlines common answers that are relevant to this thesis.

In regards to the question about future goals of the products, all interviewees had plans but they were mostly concerned with better integrations to other solutions. A key area of interest that was mentioned by most of them was machine learning. Specifically, solutions that perform classification tasks were mentioned to be on the horizon. On the question regarding bottlenecks, most mentioned API-calls and data being spread between solutions as the main offender, but nothing localized in their own product. Upon reflecting on the

question, one interviewee said “We do not design our solutions based on what is hard to accomplish, but what the customers desire”.

Web developers There are a lot of sorting, filtering, grouping and mapping that are being used in the front-end, and with an increasing amount of users on the online platform, this could prove to become a bottleneck in the future.

Some developers have (or are interested in introducing) animations in their product in order to improve the user experience. With the possibilities of WebAssembly for computation, it would be interesting to see how well it can perform computation of frames that could be used for animation. Furthermore, several developers mentioned QR-codes as being an interesting algorithm to benchmark.

Regarding the question if the product handles a lot of data the answer was usually yes, but not on the client-side. Since IKEA has many online customers with a profile, there is a lot of customer data. But customer data is regarded as sensitive information and should be kept on the server-side since it is dangerous to send it to the client-side. This means that although there is a lot of data some of it should not be on the client-side. Therefore there are not many performance heavy algorithms in the front-end that consume a lot of time.

List of algorithms to benchmark

In the current IKEA web landscape, certain algorithms were mentioned as being more interesting than others, namely algorithms for sorting, mapping, grouping and filtering data. From the literature study, these groups of algorithms are shown to have certain strengths- and weaknesses in both JavaScript and WebAssembly. Since they rely on repeated operations on similar data, JavaScript is expected to create optimizations both in function structure and property lookup, given enough computations occur. Meanwhile, WebAssembly has an initial startup time, and requires encoding and decoding the data to- and from memory rather than passing references to it. At the same time, the increased algorithmic performance in WebAssembly should result in the data being processed quicker. This group of algorithms is interesting to explore for multiple purposes. As they are used in IKEA’s front-end landscape, their investigation will help answer potential future performance problems at IKEA. Furthermore, as data passing is a complicated matter in WebAssembly, it could help understanding potential issues with the technology for future improvements.

Another group of processes that were mentioned as interesting was image data generation. QR-codes are prevalent in the IKEA client-side landscape. Furthermore, many teams have (or intend to introduce) animations in their products. It is beneficial to IKEA to see if these algorithms perform better in WebAssembly, certainly when large amounts of image frames are to be generated. Furthermore, tying back to the literature study, image data can be represented as a sequence of bytes and shared across the JavaScript and WebAssembly context, resulting in little overhead.

In regards to future development, Machine learning is very interesting for IKEA. Data privacy is becoming a large concern today; The traditional approach of data collection to create Machine Learning models might come to be threatened by laws such as the General Data Protection Regulation (GDPR) in Europe. Concepts such as Federated Learning, which attempts to solve this issue by distributing certain machine learning tasks to the client machine might become critical to the future of personalization and classification online [31]. Connecting to the literature study, many machine learning solutions rely heavily on repeated

and often complicated calculations, where WebAssembly shows great potential. It is a large field with many solutions, and as such, will be narrowed down in this thesis.

One problem that was mentioned as interesting for IKEA is the bin-packing problem, which entails positioning a set of items of different geometries within a a larger container. Although there exists multiple solutions to the problem, it is NP-hard, and as such, finding an optimal packing is time consuming. The complex nature of the problem makes it interesting from a practical standpoint, as WebAssembly is expected to run faster than JavaScript in general, and as such should complete the task faster.

Disclaimer concerning the algorithms

From the answer of the interviews, a collection of problem domains that are interesting for IKEA was accounted for. It should be noted, however, that the actual algorithms that are benchmarked are not taken from IKEA source code, and are instead publicly known algorithms representative of - or inspired by - the aforementioned problem domains. There are multiple reasons for this. First, the inner workings of IKEA shall not be made public due to a Non-Disclosure Agreement made between the authors and IKEA. second, by benchmarking well-known, generic algorithms, the results should be more generalizable to other contexts, all the while being valid in the IKEA context.

Summary

Going forward, several algorithms identified as interesting in section 3.1.1 and 3.1.2 have been discarded, such as cryptographic hash functions, algorithms in 3D graphics, signal processing and matrix algorithms. Although significant in the realm of browser performance, they were not mentioned as relevant within the teams that were interviewed. In summary, the following groups of algorithms- and processes have been selected as appropriate for further analysis.

- Sorting
- Mapping
- Grouping
- Filtering
- Machine Learning
- Image data generation
- QR codes
- Bin packing

Chapter 4

Creating the benchmarking framework

Believing that it would be simple to measure execution time for WebAssembly and JavaScript was quickly regretted. It proved to be a complicated task as there are many approaches and pitfalls to consider and it must be done correctly in order for the result to be useful. This chapter will discuss the problem of how to best measure execution time for the algorithms from the previous chapter. It will do this by first exploring existing solutions for benchmarking algorithms and their possible usage for the goal of this thesis. Second, an analysis of approaches to measuring execution time will be conducted in order to conclude the approach for this thesis. This will produce a requirement specification that needs to be met in order to produce valid results for this thesis. Finally, we will discuss and highlight interesting problems and how we solved them while implementing our solution for the best way of measuring execution time for this thesis.

In order to solve the problem of whether or not WebAssembly would speed up IKEA's services, the manner of measurement should at least capture the time it takes from initiating the WebAssembly module until the entire service has executed. However, there are three aspects of WebAssembly execution that have been previously identified that could alter execution time and as such, would be interesting to explore as they could give key insights into why WebAssembly's performance differs from JavaScript. First, WebAssembly modules have to be manually initiated through an asynchronous call WebAssembly.initiateStreaming(). Second, exchanging complex data between JavaScript and WebAssembly cannot be done through references, and requires writing- and reading data from a shared buffer. Third, switching between JavaScript and WebAssembly execution could add a certain runtime overhead. As such, finding a tool that allows certain limited introspection for the above stated parts of execution would be beneficial for further discussions.

4.1 Existing Solutions

A time efficient solution for measuring execution time would be to use an existing framework for benchmarking. However we must first conduct an analysis of available solutions in order to conclude their possibilities for this thesis. This section will analyze Wasabi, Jalangi and browser profiling by reading their documentation and how they fit with our goal for measuring execution time for our list of algorithms. We reason that existing solutions are inadequate for the purpose of this thesis, since they either gather data that is not relevant to execution time and could affect the execution time in an unclear manner, or require that programs in the two languages need to be run through separate frameworks, which would make it hard to compare differences in execution time since their influence on time behavior could vary.

4.1.1 Dynamic analysis frameworks

There are two prominent frameworks for analyzing WebAssembly and JavaScript: Wasabi for WebAssembly, and Jalangi for JavaScript. They are both based on the concept of runtime (or binary) instrumentation: A program is processed by the framework and transformed into a new program, with injected code that captures individual statements. Using a template system, one could define what information should be extracted during the capture phase of execution. For the purposes of this thesis, timestamps could be created as specific statements are executed. There is a slowdown associated with using dynamic analysis frameworks. Depending on the program there is an overhead of Wasabi which is 1.02 to 163x of the original runtime [32]. When performing analysis on the SunSpider benchmark suite, Jalangi shows an average slowdown of 30X the original execution speed [33].

There are some shortcomings with these solutions. For the purpose of answering whether or not WebAssembly is faster than JavaScript for the selected algorithms, breaking down execution time to statement level is unnecessary. In order to get accurate measurements, the benchmarks should be run through the same framework. Running code through Jalangi and Wasabi for JavaScript and WebAssembly respectively would alter the full execution time of the programs to potentially various degrees for the two languages, leading to inconclusive results.

4.1.2 Browser profilers

Most browsers have a developer tool suite, which usually includes a performance profiler. It can be used to capture the runtime behavior of a program down to function execution time, heap allocations and garbage collection, but also DOM manipulation such as rendering and adding new DOM elements.

Using a browser profiler results in the entire javascript engine being instrumented and could be used to measure the execution time of both languages. In order to understand how it would impact execution times, A simple test was performed on the heapsort algorithm to get an idea of its impact. First, the algorithm was run on 50.000 integers over 100 iterations in a fresh chrome instance, see figure 4.1). Following this, the same test was repeated but with the developer tools tab open, see figure 4.2.

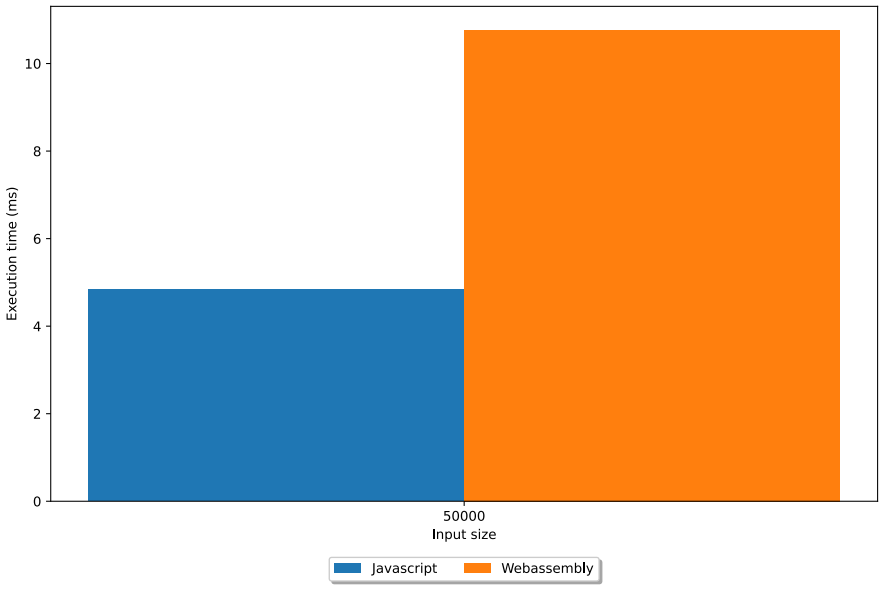

Figure 4.1: Running heapsort on 50.000 integers for 100 iterations

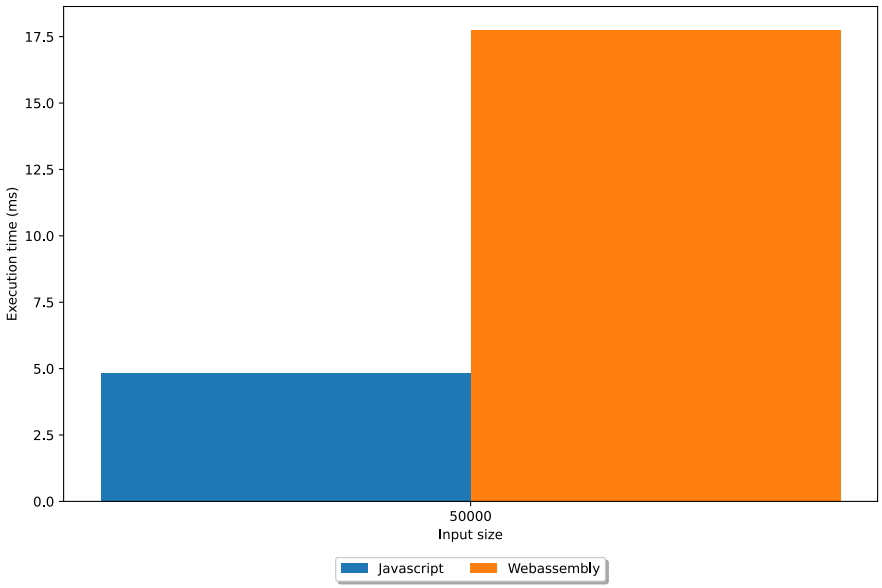

Figure 4.2: Running heapsort on 50.000 integers for 100 iterations with dev tools open in chrome

The results show that keeping the developer tools tab open has no discernable effect on JavaScript execution times, executing on an average in 4,833 ms in a clean window, and 4,846 ms when the tab is open. The WebAssembly execution times, however, are noticably greater: The execution time on average increases from 10,768 ms to 17,748 ms just by keeping the tab open. This pattern is shown for all input sizes and algorithms. The reason for this change in execution time is unknown, but is enough to conclude that it is not a suitable method to gather accurate data.

4.1.3 Summary

In conclusion, the above solutions are powerful toolboxes for program analysis, but alter execution too much or to varying degrees, which would make the results harder to analyze. An optimal solution should capture just the information that is relevant to this thesis and be usable for both JavaScript and WebAssembly.

4.2 Approaches to measuring execution time

Creating a custom tool that is tailored towards the problem this thesis attempts to solve is an alternative to using existing frameworks. However, there are various approaches and pitfalls that need to be addressed when execution time measurements are performed. This section will analyze two approaches, blackbox measurements and runtime instrumentation, highlighting potential issues. This will produce guidelines that will be beneficial when creating a custom framework.

The main issue when measuring execution time in software is that capturing timestamps comes with a cost, in the form of additional execution time overhead. An experiment on measurement overheads using JavaScript's performance.now() function shows that, within a 5% error margin, calling the function in Firefox 81 takes 21 µs on average, while in Chrome 84 it takes 5.5 µs on average, but these values vary from call to call [34]. The matter becomes more complicated, as several browsers have added a random jitter to their timers in order to prevent timing attacks such as Spectre [35].

4.2.1 Blackbox measuring

Blackbox measuring is the simplest solution; start a timer just before the program or function starts execution, and stop it right after. The main benefit from this approach is that the executing code is left completely untouched, yielding a result that is as close as possible to the actual execution time. This approach is adequate in answering whether or not WebAssembly could improve IKEA's services, but will not produce any insights to certain phases of execution, such as data exchange and switching between the JavaScript- and WebAssembly context. A simple example is provided in code listing 4.1.

4.2.2 Runtime instrumentation

Runtime instrumentation is the practice of injecting code into the source code or binary of the program, so called "instrumentation statements", which could be used to capture infor-

Listing 4.1: Example code illustrating how blackbox measurements are performed

```
const t0 = performance.now();
const result = someLongRunningFunction(arg1, arg2);
const t1 = performance.now();
const executionTime = t1 - t0;
```

mation regarding the execution on either a statement or a collection of statements. This would be required for program introspection, such as capturing the time it takes to decode and encode data between JavaScript and WebAssembly.

Given the program $P = S_1S_2...S_n$ where S_i is the i:th statement of the program, the instrumentation of the program, a program in itself, is $I(P) = I_1S_1I_2S_2...I_nS_n$, where I_i is the instrumentation code around statement S_i. Certain instrumentation statements may be left out, if the goal is to just gather information about certain groups of statements.

One major improvement over blackbox testing is that instead of measuring execution time on program-level, runtime instrumentations can measure execution time on statement level. There are some issues with runtime instrumentation, however. The major one is the phenomenon known as "software perturbation". Every call to an instrumentation statement, given that it is not empty, has a certain overhead [36]. A heavily instrumented program could potentially lead to a dramatically altered execution time, while a less instrumented program reveals less runtime information.

4.3 Requirements Specification

In section 3.1.2 it became evident that non-primitive data exchange is a crucial aspect for certain algorithms, which means that the benchmarking framework should be able to capture its influence on execution time. This requires that there are parts of the program that pass data between JavaScript and the WebAssembly module and that the data passing time can be measured.

In order for the result to be accurate it should not be affected by any long overhead. Cases where there must be an overhead should be limited. There should not be notable change to the executing code since it will affect the execution time.

In order to explore RQ 1.c the benchmarking framework needs to be able to measure the startup time for WebAssembly. To measure startup time a timestamp was made before initializing WebAssembly and then directly after instantiation.

Taking the above into account, in order to have a basis for fruitful analysis of the above stated aspects of WebAssembly execution, the following phases of execution should be captured, where applicable:

- WebAssembly instantiation
- Switching from the JavaScript context to WebAssembly
- Deserializing parameters in WebAssembly
- Executing the algorithm

Listing 4.2: Example code illustrating how the WebAssembly startup time is measured

```
const t0 = performance.now();
initWasmModule().then((module) => {
    const t1 = performance.now();
    const startup = t1 - t0;
    //run benchmark
});
```

- Serializing return value in WebAssembly
- Switching from the WebAssembly context to JavaScript

4.3.1 Lessons learned

Existing frameworks that were explored did not meet the requirement specification and therefore a decision was made to create a new benchmarking framework. In this section we will discuss the difficult parts of creating the benchmarking framework and experiences we now have that would have benefitted us greatly if we had known them at the start of the thesis.

Measuring execution time

To meet the requirements specification for measuring execution time it was decided to develop a benchmarking framework that would be a hybrid between blackbox testing and runtime instrumentation. The blackbox technique is the one used for measuring the execution time of the algorithms in both JavaScript and WebAssembly, as well as WebAssembly's startup time. Limited runtime instrumentation would allow gathering additional information regarding the execution time of various phases during WebAssembly execution.

Measuring the full execution time of the algorithms is done by simply creating timestamps before- and after function execution. However this would yield limited information about what it is during the execution of an algorithm that is time consuming. As we have seen, serialization and deserialization complicates data passing. To manually add all the timestamps to the algorithms would be time consuming and would decrease the scalability of testing if for example another interesting aspect of the algorithms would be interesting to measure time, therefore a framework is created. In order to fully explore RQ 1.c we need to be able to measure the start up time for WebAssembly. Therefore an important part about the requirement specification is to be able to measure this start up time. It was solved in the framework by having a timestamp right before and right after the instantiation of the WebAssembly module, see listing 4.2.

Furthermore, as previously discussed, it would be interesting to investigate the time behavior of WebAssembly functions that exchange data with the JavaScript context that cannot be directly used, such as JavaScript arrays. Since this will require data passing with serialization and deserialization, which we believe could be a common use case for WebAssembly. This will however require certain instrumentation statements inside of the WebAssembly functions. WebAssembly running inside of a javascript runtime does not have direct access

Listing 4.3: Example code illustrating how the benchmarking function is defined

```
//Benchmarking function in JavaScript
function benchmark(wasmFunction, nbrIterations, ...args) {
    for(let i = 0; i < nbrIterations: i++) {
      startTimer();
      wasmFunction(...args);
      endTimer();
      collectResults();
    }
}
```

to any timestamp functions, and as such needs to be imported from the JavaScript context. Listing 4.4 shows an example of how this could be done. First, four functions are declared as external, with a reference to which module the functions should be imported. These external functions are called at certain points during WebAssembly execution. When executing the functions, the program returns briefly to the JavaScript context and registers a timestamp.

The external functions trigger a call to custom functions in the JavaScript context, registering the appropriate timestamp before - and after - serialization and deserialization events.

We have identified several key parts of WebAssembly execution time that are important to capture in order to give a good view of what parts of the execution that allocates time. Using the above approach several of these key parts can be captured and studied:

1. Startup time
2. Time to enter function from JavaScript context
3. Time to deserialize parameters
4. Time to execute algorithm
5. Time to serialize output
6. Time to return to the JavaScript context

Instrumentation overheads

From the previous section, it is evident that benchmarking WebAssembly functions will use four additional timestamps that will not be present in JavaScript functions and as such, will give a result that is greater than the actual execution time. This section explains how the impact of the additional timestamps were analysed. In conclusion, the impact of four additional timestamps was found to be negligible.

Rust, C and C++ have support for pre-processor directives that may omit- or include code based on flags that are passed to the compiler, see listing 4.5. In the example, the instrumentation statement will only be added if the flag "instrumentation" is used while compiling. One could then run the benchmarks with- and without instrumentation to find out how much overhead is added.

Listing 4.4: Example Rust code illustrating how instrumentation statements were placed along with how data exchanges are performed

```
use wasm_bindgen::prelude::wasm_bindgen;
/*
    Declare external functions imported from the JavaScript
    context
*/
#[wasm_bindgen(raw_module = "../wasm-js-externals")]
extern "C" {
    #[wasm_bindgen(js_name = registerSerializeStart)]
    pub fn register_serialize_start();

    #[wasm_bindgen(js_name = registerSerializeEnd)]
    pub fn register_serialize_end();

    #[wasm_bindgen(js_name = registerDeserializeStart)]
    pub fn register_deserialize_start();

    #[wasm_bindgen(js_name = registerDeserializeEnd)]
    pub fn register_deserialize_end();

}
/*
    Implementation of a WebAssembly function which is exported
    to JavaScript through wasm-bindgen
*/
#[wasm_bindgen(js_name = webAssemblyFunction)]
pub fn webassembly_function(input_values: &JsValue) -> JsValue {
    register_deserialize_start();
    let deserialized_input: Vec<i32> = values.into_serde().unwrap();
    register_deserialize_end();
    let results = run_rust_function(deserialized_input);
    register_serialize_start();
    let serialized_output = JsValue::from_serde(&results).unwrap();
    register_serialize_end();
    serialized_output
}
```

Listing 4.5: Conditional compilation of instrumentation statements in Rust

```
#cfg(feature = "instrumentation")
register_serialize_start();
```

Since the average overhead of calling performance.now() was previously established, it could also be used to reason how much the execution time was affected with instrumentation. Using a similar setup as in section 4.1.2, heapsort on 50.000 elements over 100 iterations was performed with- and without instrumentation statements in the Rust code. The result shows that with the four additional instrumentation statements, the execution time of WebAssembly was on average increased by 0.163 ms. Since this impact is minimal, especially for longer running algorithms, the benchmarking framework is deemed to be accurate enough to gather reliable data.

The random timer jitter introduced in modern web browser would be a considerable problem to the purpose of this thesis. However, these can be temporarily disabled using browser configurations.

4.4 Toolchains used

To be as efficient as possible while developing the framework we need to have an environment that is easy to work with. The fairly broad spectrum of software problems to be tested could be time-consuming and unwieldly to work with without a modular, scalable setup. In this section we will discuss the toolchains we decided upon.

4.4.1 JavaScript toolchain

Since the authors have previous experience in Web development, it was decided early on that the benchmarking page should use a combination of React and WebPack. The time invested in setting up the environment was offset by the ease of scaling up and extending the page as more algorithms were implemented. Furthermore, it was considered a more realistic environment for testing, as many modern solutions rely on the technologies. As compared to only using a simple HTML-page, since the developer of a simple HTML-page would probably not consider using WebAssemby for execution time performance. When using WebAssembly modules with JavaScript an issue arises on how the communication between the two should be done. One useful tip we found was to use Webpack with its Wasm-pack plugin [37]. It became remarkably easier by utilizing Webpack + Wasm-pack plugin when using Rust code to generate WebAssembly modules that would be used with JavaScript. It allowed for an automated compiling trail which relieved the manual labor by generating appropriate javascript helper code, and supported imports of WebAssembly modules that did no longer require asynchronous instantiation. For the purpose of measuring startup time in this thesis, the last feature was disabled.

4.4.2 WebAssembly toolchain

When using WebAssembly in the browser it needs to be able to communicate with JavaScript. In order to use such an interaction wasm-bindgen is utilized, it is a Rust library and CLI tool used when compiling Rust code into WebAssembly code for high level interactions between the two [38]. It creates helper code between WebAssembly and JavaScript that can easily be used in JavaScript and proved very useful for this thesis.

Chapter 5
Running the benchmarks

In order to explore RQ 1.b, RQ 1.c and RQ 2.a we need to have concrete data that can be analyzed. Therefore this chapter will present the gathered data, together with a discussion, from running the benchmarking frameworks. The outcome of this chapter will be a recommendation for what type of algorithms that would benefit from using WebAssembly compared to JavaScript and potential weaknesses in WebAssembly that could be further improved.

5.1 Benchmarking setup

In this section we will discuss how we set up our benchmarks, our thoughts on iterations and input sizes for testing. How to handle outliers will also be discussed. The specifications of the machine running the benchmarks will be presented.

5.1.1 Setup

When measuring execution time it is important that the results are trustworthy which is why only one test measurement for each algorithm will not be sufficient. There could potentially be other processes running on the computer that are hard to control, and variations in execution time when running code that is object to a JIT-compiler, which could alter execution time suddenly and aggressively. In order to yield as correct data as possible the benchmarks will be done with multiple iterations. Since there is the possibility of outliers in the results, these can be handled in three ways:

- Scale up the number of test iterations, lessening the influence of outliers
- Discard test runs which contain outliers
- Ignore outliers

Using [17] as a starting point, which also directly explores differences in performance between JavaScript and WebAssembly, the number of iterations per benchmark is set to 10 where applicable, in order to lessen the impact of outliers. Furthermore, the PolybenchC benchmark specifies 5 different program input sizes: XS, S, M, L, and XL, which depend on the algorithm. Since this thesis is interested not only in the full execution time of the two technologies, but also several phases of WebAssembly execution such as data exchange with JavaScript, running tests on multiple input sizes could expose how these phases are affected as the input grows, leading to a more fruitful discussion.

5.1.2 Input sizes

The question is then what suitable input sizes are for the various algorithms. There are two key factors to take into account when the program input sizes are considered. First, they should be large enough that any potential measurement errors become minimal. Benchmarking an almost instantaneous function call would suffer the effects of micro-benchmarking: variance in the time to call to performance.now and execution being temporarily halted due to other processes running on the computer could have a severe impact on each iteration, leading to an erroneous result. Second, the difference between input sizes should be large enough that the result shows how the various phases of execution scale up as the input grows. For example, running algorithms on input sizes 1000 to 1005 in increments of 1 would vary little.

As a starting point, the average overhead of calling performance.now was established in section 4.2 to be 5.5 µs and 21 µs for Chrome and Firefox respectively. However, the potential influence of other computer processes while running the algorithms can not be estimated. As such, the smallest input size was selected empirically, by identifying a size at which multiple test runs yielded similar results. We increment a bit differently depending on the algorithm.

5.1.3 Metrics

As previously mentioned, multiple phases of WebAssembly execution will be subject to measurements. As such, it is necessary to establish a few ways in which metrics will be collected.

In order to lessen confusion going forwards, a discussion concerning the word "Algorithm" or "Algorithm phase" is in order. When discussing JavaScript, "algorithm" refers to the full execution of an algorithm, since this thesis does not break down JavaScript execution into multiple steps.

The same does not apply to WebAssembly: logic written in the language has certain extra work done before and after the actual algorithm. As such, when discussing WebAssembly, "algorithm" refers to the phase that occurs after all preparations are completed, and before work to return data to the JavaScript context and cleaning up memory is performed.

Since the benchmarking framework does not take into account when JIT-optimizations or garbage collection occur, the execution time of running the JavaScript functions will be captured in their entirety:

$$JS_{full} = JS_{effective} = JS_{algorithm}$$

WebAssembly execution has multiple phases that can be collected in two groups: Effective execution time and preparatory execution time:

$$WASM_{full} = WASM_{effective} + WASM_{preparatory}$$

The effective work in WebAssembly is when the algorithm actually runs:

$$WASM_{effective} = WASM_{algorithm}$$

The preparatory work in WebAssembly is all the phases that exchange data with JavaScript or cleans up memory:

$$WASM_{preparatory} = WASM_{Enter} + WASM_{deserialize} + WASM_{serialize} + WASM_{return}$$

For the purpose of using WebAssembly instead of JavaScript, the most interesting ratio is the full execution time of the two technologies compared to each other:

$$Speedup_{full} = \frac{JS_{full}}{WASM_{full}}$$

For the purpose of analyzing how well WebAssembly could perform if data exchange was theoretically absent, The execution time of JavaScript is compared to only the effective phase of WebAssembly execution, namely the algorithm time:

$$Speedup_{algorithm} = \frac{JS_{algorithm}}{WASM_{algorithm}}$$

Finally, the efficacy, or degree of execution time used to perform the actual algorithm, of WebAssembly can be shown through the algorithm time compared to the full execution time:

$$efficacy = \frac{WASM_{effective}}{WASM_{full}} = \frac{WASM_{algorithm}}{WASM_{algorithm} + WASM_{preparatory}}$$

The efficacy shows how severe the impact of data passing is to a given algorithm. Another metric that will be explored is how execution time scales with input sizes.

5.1.4 Plots

The plots that follow are grouped, stacked bar plots, placing JavaScript execution time in direct comparison to WebAssembly execution time. Since WebAssembly execution is broken down into multiple phases, an explanation for the different phases is presented in table 5.1.

5.1.5 Execution setup

When running the benchmarks, all applications were closed down except for two: The browser in which tests were performed, and a terminal window hosting the page. Each benchmark was run through a new Chrome incognito window in order to have a clean slate without any cached data. It should be noted that the benchmarks were run on a cold-start, meaning that the functions were not run in advance to warm up the JIT-compiler.

5.1.6 Machine specifications

The benchmarks were executed on a Windows machine using the chrome browser, see table 5.2 for a detailed specification.

Name	Executing context	Explanation
JS: Algorithm	JS	Running the full algorithm in pure JavaScript
WASM: Enter	JS	Placing references to JavaScript data on the shared heap and switching context to WebAssembly.
WASM: Deserialize	JS + WASM	Runs two steps: First, A call to the the javascript context transforms object data on the shared heap into bytes in WebAssembly linear memory through JSON encoding. Then, the WebAssembly context parses the string. At the end of this phase, WebAssembly has full ownership of a copy of the data
WASM: Algorithm	WASM	Running the algorithm in WebAssembly
WASM: Serialize	JS + WASM	The return value of the algorithm is first converted into a JSON string in WebAssembly Memory. Then, A call to JavaScript parses the JSON string in WebAssembly memory into an object on the shared heap.
WASM: Return	JS + WASM	If needed, WebAssembly cleans up data that will be discarded when the function ends. Then, the previously parsed data is removed from the shared heap and returned from the function.

Table 5.1: Table with an explanation for the different execution phases

Operating System	Windows 10 Pro, OS Build 19042.1706, x64
CPU	Intel i5-4670K @ 3.40GHz, 4 Cores
RAM	2 x 8 Gb DDR4
Browser	Chrome, build 102.0.5005.63, x64

Table 5.2: Specifications for the computer and browser used to run the benchmarks

5.2 Benchmarking results

To give an answer to if and where IKEA could use WebAssembly to speed up their client-side services, it is necessary to explore which services are improved by the technology. The analysis section argued in favor of- and against certain software patterns and algorithms. This section will present- and discuss the results from running the previously established list of interesting algorithms through the benchmarking framework that was previously discussed. Each explored algorithm is explored individually. Plots from running the benchmark are shown and discussed.

5.2.1 General results

As the results from the various algorithms were inspected, one thing to note was the lack of trends in execution time between iterations. Disregarding outliers, the measured execution times would vary a little but generally stay in a certain range.

5.2.2 Sorting

For the purpose of this thesis, the choice of sorting algorithms was limited to two: Heapsort and quicksort. They are both $O(nlogn)$ algorithms, with quicksort having the worst case scenario $O(n^2)$ but usually running faster than heapsort.

The initial input size was set to 50.000 elements, and scaled up to 800.000 elements. Smaller input sizes were tested, but the results varied heavily between test runs even when the number of iterations was high, with one language executing faster than the other almost randomly. At 50.000, the results were stable enough between test runs and as such was chosen as a starting point.

In all tests, data was passed as a generic JavaScript array. After setting the number of elements n, the benchmarking framework generated an array of integers $0, 1, ...n$ and then randomly shuffled the array. On each benchmarking iteration, the functions then sorted copies of that list. For the string variant, the list elements were instead integers that were parsed into strings.

It should be noted that the integer scenario could have been solved using fixed-size arrays such as JavaScripts UInt32Array, which are far easier to handle in WebAssembly and require little more than sharing address pointers and array lengths. However, they are not representative of the common use case where generic, dynamic arrays dominate.

Result

Heapsort - integer data The plot in figure 5.1 shows the different input sizes on the x-axis and the execution time in milliseconds on the y-axis. The blue bars show the full execution time of JavaScript. The other colors represent execution time in WebAssembly, with the red bars showing the algorithm work and the others various phases of data exchange or cleanup. It is notable that preparatory work is considerable for this problem, since it passes arrays which need to be parsed. At 50.000 integers, WebAssembly's algorithm phase shows a speedup of 1.9x compared to JavaScript. As the input sizes grow, this speedup decreases, ending at 0.97x compared to JavaScript at 800.000 integers. Accounting for the full WebAssembly execution time, WebAssembly starts with a slowdown of 0.5x which declines steadily to 0.45x at the largest input size. The efficacy of WebAssembly is 26.4% at 50.000 integers, and increases steadily to 46.8% at 800.000 integers.

Heapsort - string data The WASM: Return-phase becomes considerable when passing arrays of strings to the algorithm, see figure 5.2. The reason for this is not clear as the generated WebAssembly code is highly obfuscated and would require a significant time to study. One probable reason is that WebAssembly cleans up data that goes out of scope, see the discussion in section 5.4 for further information.

At 50.000 strings, WebAssembly's algorithm phase shows a speedup of 1.79x compared to JavaScript. As the input sizes grow, this speedup decreases, ending at 1.12x compared to

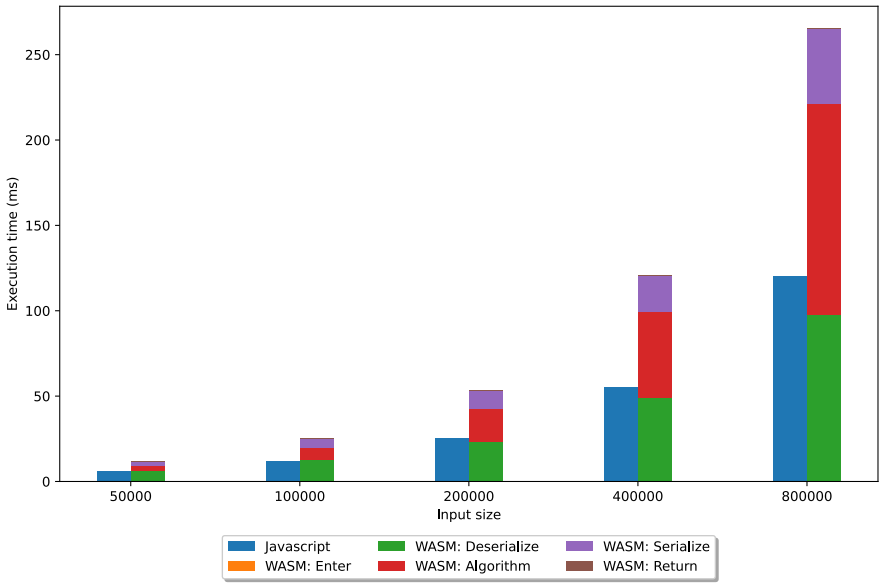

Figure 5.1: Heapsort - integer data

JavaScript at 800.000 strings. Accounting for the full WebAssembly execution time, WebAssembly starts with a slowdown of 0.86x which declines steadily to 0.74x at the largest input size. The efficacy of WebAssembly is 48.2% at 50.000 strings, and increases steadily to 65.8% at 800.000 strings.

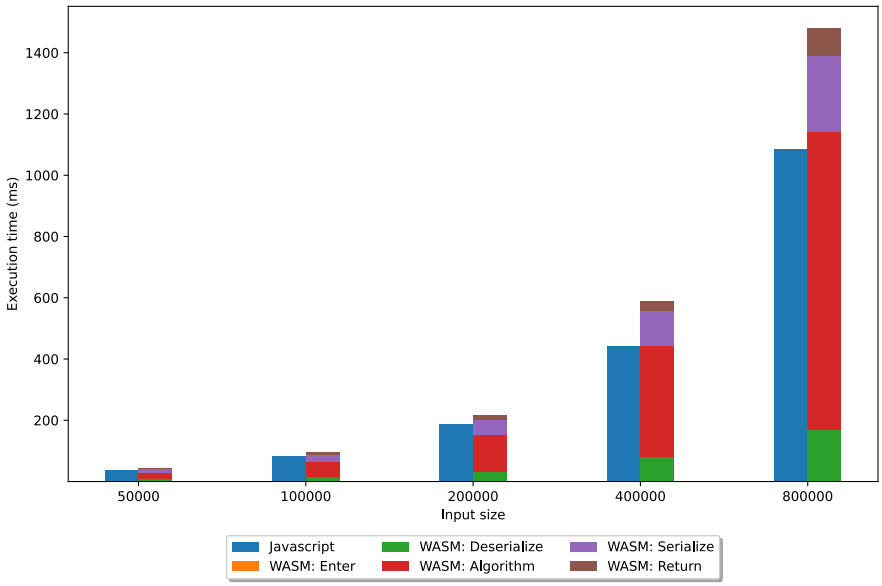

Figure 5.2: Heapsort - string data

Quicksort - integer data The plot for the benchmark is shown in figure 5.3. At 50.000 integers, WebAssembly's algorithm phase shows a speedup of 2.57x compared to JavaScript. As the input sizes grow, this speedup decreases, ending at 2.23x compared to JavaScript at 800.000 integers. Accounting for the full WebAssembly execution time, WebAssembly starts with a slowdown of 0.66x which declines steadily to 0.62x at the largest input size. The effi-

cacy of WebAssembly is 25.7% at 50.000 integers, and increases steadily to 28.1% at 800.000 integers.

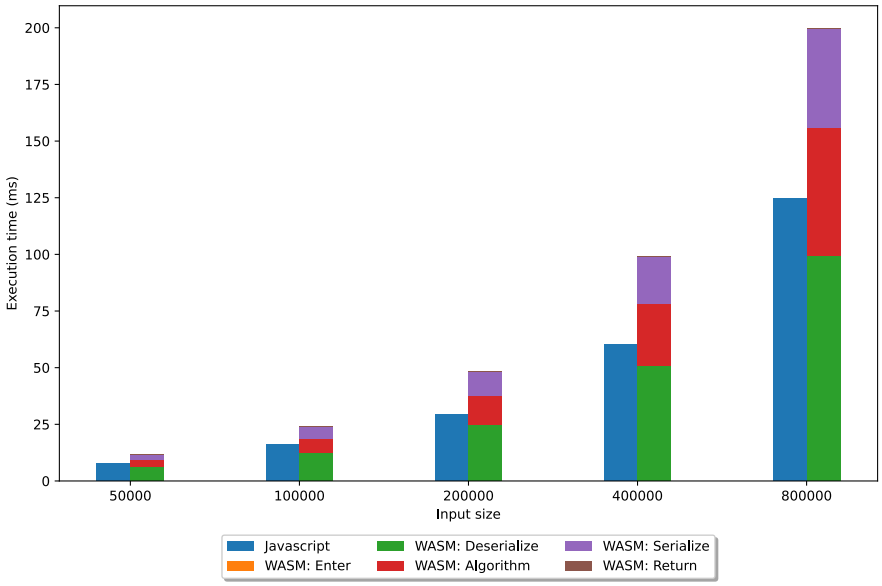

Figure 5.3: Quicksort - integer data

Quicksort - string data The plot for the benchmark is shown in figure 5.4. At 50.000 strings, WebAssembly's algorithm phase shows a speedup of 2.86x compared to JavaScript. As the input sizes grow, this speedup decreases, ending at 2.05x compared to JavaScript at 800.000 strings. Accounting for the full WebAssembly execution time, WebAssembly stays very close to JavaScript, staying between 0.98x and 1.05x JavaScript execution time up to the largest input size, where it declines to a slowdown 0.87x to JavaScript. The efficacy of WebAssembly is 35.3% at 50.000 strings, and increases steadily to 42.7% at 800.000 strings.

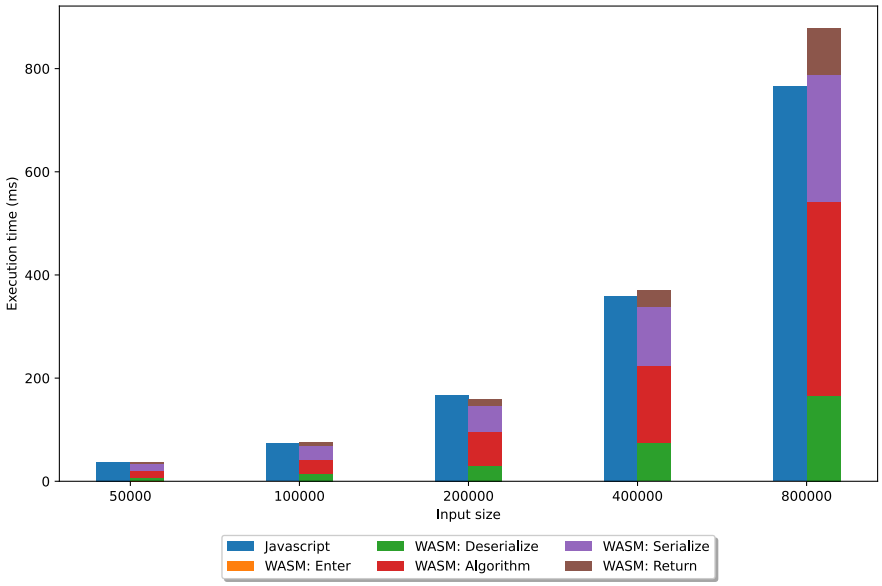

Figure 5.4: Quicksort - string data

Discussion

In both algorithms, sorting string data takes more time to execute than integer data. This is to be expected, as string comparisons are a more complex process than integer comparisons. Phases passing data to- and from WebAssembly is substantial, with the string sorting benchmarks having a greater execution time during the data passing phases due to the more expensive operation to read- and write strings as bytes in WebAssembly memory. The execution time of all phases seem to scale linearly with the input size.

In both algorithms and input types, there is a pattern where the algorithm phase of WebAssembly is quicker than JavaScript at the smallest input, but this speedup decreases as input grows, and when running heapsort on integers, JavaScript executes faster at the largest input size. Similar experiments performed by Yan et al. also increased input sizes while comparing WebAssembly and JavaScript. The results are similar: WebAssembly becomes slower compared to JavaScript as the input size increases. The authors attribute this to JIT-optimizations becoming more aggressive the longer a function runs, which has little effect on already optimized WebAssembly [18].

5.2.3 Mapping

mapping is a concept in functional programming, a higher order function. Given a list of values of type V, some function g that transforms objects of type V into K, it produces a list of values of type K.

```
createMapping<K, V>(values: Array<V>, g: (value: V) => K): Array
    <K>
```

The initial input size was set to 50.000 elements, and scaled up to 800.000 elements. Mapping functions have a time complexity that depends on the function g. In this case, both benchmarks use a function g with time complexity O(1), resulting in the full time complexity O(n). A simple constant time function g was chosen since transforming a collection of objects to a collection of one of the object's fields is a very common practice. The input data was a list of javascript objects.

```
{
    itemID: number;
    name:   string;
}
```

Two versions of the function were tested: One that transformed the list to a list of the integer value, and one that transformed the list to a list of the string value.

Result

Map objects to integers

The plot for the benchmark is shown in figure 5.5. At 50.000 objects, WebAssembly's algorithm phase shows a slowdown of 0.95x compared to JavaScript. As the input size grows, this slowdown becomes a speedup, ending at 2.14x compared to JavaScript at 800.000 objects. Accounting for the full WebAssembly execution time, WebAssembly starts with a slowdown of 0.04x which increases steadily to 0.07x at the largest input size. The efficacy of WebAssembly is 4.6% at 50.000 objects, and decreases steadily to 3.4% at 800.000 objects.

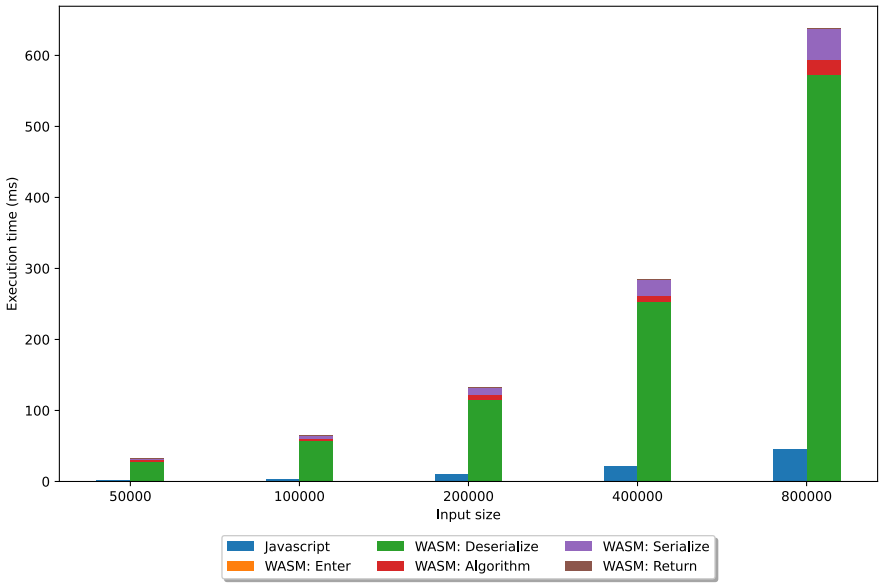

Figure 5.5: Map objects to integers

Map objects to strings The plot for the benchmark is shown in figure 5.6. At 50.000 objects, WebAssembly's algorithm phase shows a speedup of 19.07x compared to JavaScript. As the input sizes grow, this speedup increases, ending at 35.82x compared to JavaScript at 800.000 elements. Accounting for the full WebAssembly execution time, WebAssembly starts with a slowdown of 0.07x which increases steadily to 0.1x at the largest input size. The efficacy of WebAssembly stays around 0.3-0.4% at for all input sizes with no discernable trend.

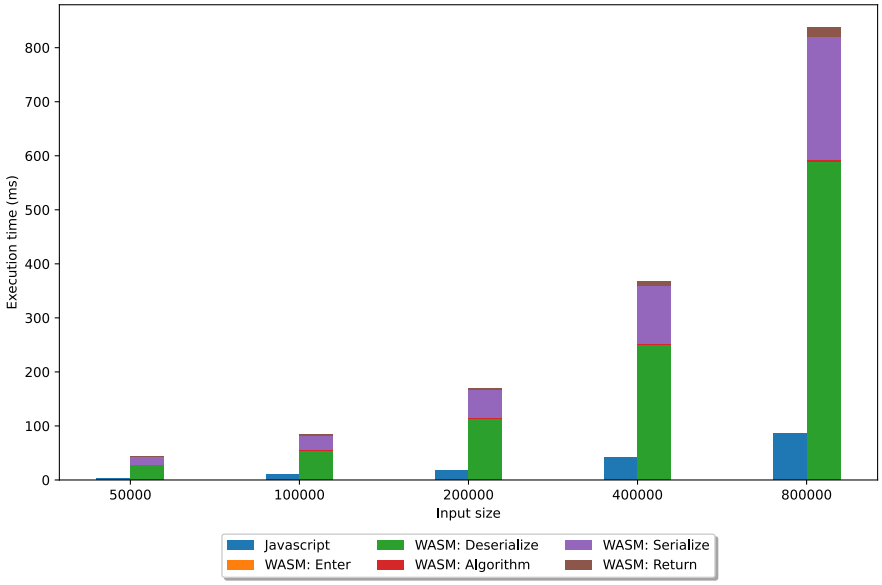

Figure 5.6: Map objects to strings

Discussion

The algorithm performance of mapping objects to strings in WebAssembly was striking, and initially believed to be due to an error in the algorithm. However, an inspection of the values that the call produced confirmed that it did indeed work. This could be attributed to the ownership system of the Rust language, which allows the mapping function to safely operate on the input list in an optimal manner since it is rendered invalid after the call, which in turn can produce highly optimized WebAssembly code. In order to confirm this, additional versions of the mapping function were produced that instead used non-destructive references, but were otherwise identical. The result shows that the destructive version does indeed perform far better during the algorithm phase, with a speedup compared to the non-destructive version ranging between 10.3x and 12.7x, see Table 5.3.

Input Size	50.000	100.000	200.000	400.000	800.000
Destructive (ms)	0.15	0.31	0.62	1.24	2.39
Non-destructive (ms)	1.8	3.37	7.87	12.83	25.5
Speedup	12x	10.9x	12.7x	10.3x	10.7x

Table 5.3: Table comparing WebAssembly algorithm execution time with both destructive and non-destructive mapping strategies, and the relative speedup for the algorithm for the destructive strategy

In the end, however, JavaScript outperforms WebAssembly in both benchmarks when the full execution time is considered, regardless of the input size. This could be attributed to well-performed optimizations using inline caching in JavaScript: The JIT-compiler quickly finds the memory offset of the field requested by the function g, and optimizes the property lookup.

5.2.4 Grouping

Grouping could refer to many things, but in this instance, it is a function which receives a collection of values, some function that transforms a value to a key, and as a result produces a Map-like structure from the individual keys to collections of values which produce that key.

```
createGrouping<K, V>(values: Array<V>, g: (value: V) => K):
    Record<K: Array<V>>
```

A simple example would be a collection of Person objects, and a function which transforms a Person to their first name. The result would be a Map with first names as keys, and lists of people with the same first name as values.

In this test, objects of the following structure were used:

```
{
  itemID: number;
  groupID: number;
  groupName: string;
}
```

itemID is a field which contains a number unique to the object. groupID is a number between 1 and 4. groupName is similar to groupID, being a string "Group A" to "Group D".

Applying the algorithm to either groupID or groupName would create 4 different groups. In order to see how performance was affected when the number of groups increase with the input, one benchmark groups items by itemID, which is unique for every item, resulting in as many groups as there are objects.

Result

Group array of objects by group id

The plot for the benchmark is shown in figure 5.7. At 50.000 elements, WebAssembly's algorithm phase shows a speedup of 1.23x compared to JavaScript. As the input sizes grow, WebAssembly's algorithm phase becomes even faster in comparison to JavaScript, growing steadily to 3.23x compared to JavaScript at 800.000 elements. Accounting for the full WebAssembly execution time, WebAssembly starts with a slowdown of 0.04x which increases steadily to 0.09x at the largest input size. The efficacy of WebAssembly is 3.5% at 50.000 integers, and decreases steadily to 2.8% at 800.000 elements.

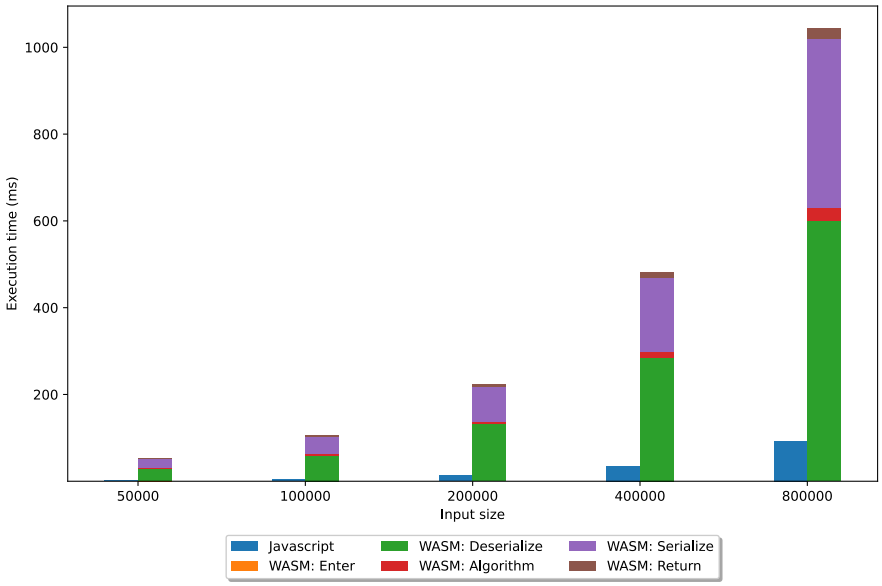

Figure 5.7: Group array of objects by integer

Group array of objects by item id The plot for the benchmark is shown in figure 5.8. The pattern in this test is interesting: WebAssembly's algorithm phase at 50.000 elements shows a speedup of 2.11x compared to JavaScript. at 100.000 elements, it decreases to 1.59x. Then it increases steadily to 2.06x at the largest input size. Taking the full execution time of WebAssembly into account, a similar pattern shows: at 50.000 elements there is a slowdown of 0.17x compared to JavaScript, which is further decreased to 0.13 at 100.000 elements, which then increases steadily to 0.15x at the largest input size. The efficacy of WebAssembly is at 8% at the smallest input size, and decreases steadily to 7.1% at the largest.

Group array of objects by group name The plot for the benchmark is shown in figure 5.9. At 50000 elements, WebAssembly's algorithm phase shows a slowdown of 0.56x compared to

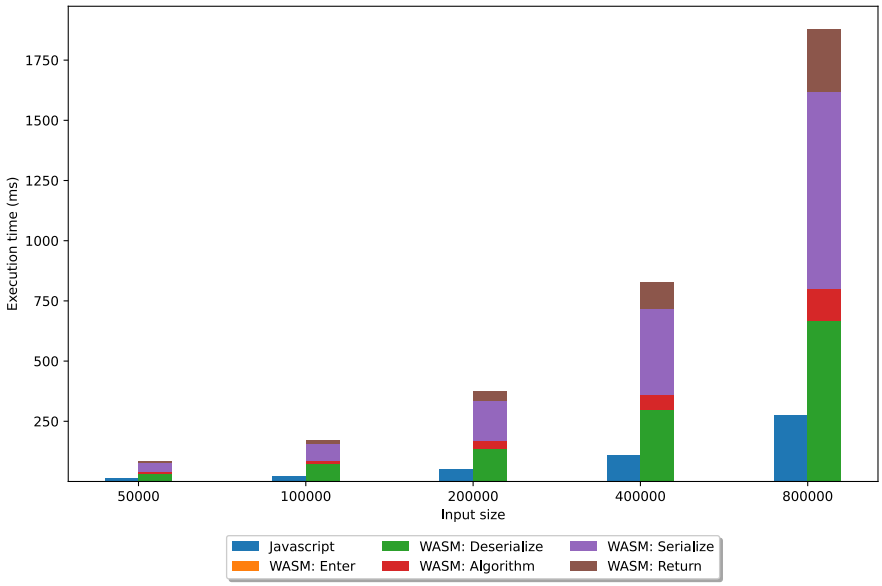

Figure 5.8: Group array of objects by unique integers

JavaScript. As the input sizes grow, WebAssembly's algorithm phase becomes even slower in comparison to JavaScript, ending at 0.33x compared to JavaScript at 800000 elements. Accounting for the full WebAssembly execution time, WebAssembly starts with a slowdown of 0.05x which decreases steadily to 0.03x at the largest input size. The efficacy of WebAssembly is 8.8% at 50.000 integers, and increases steadily to 9.5% at 800.000 elements.

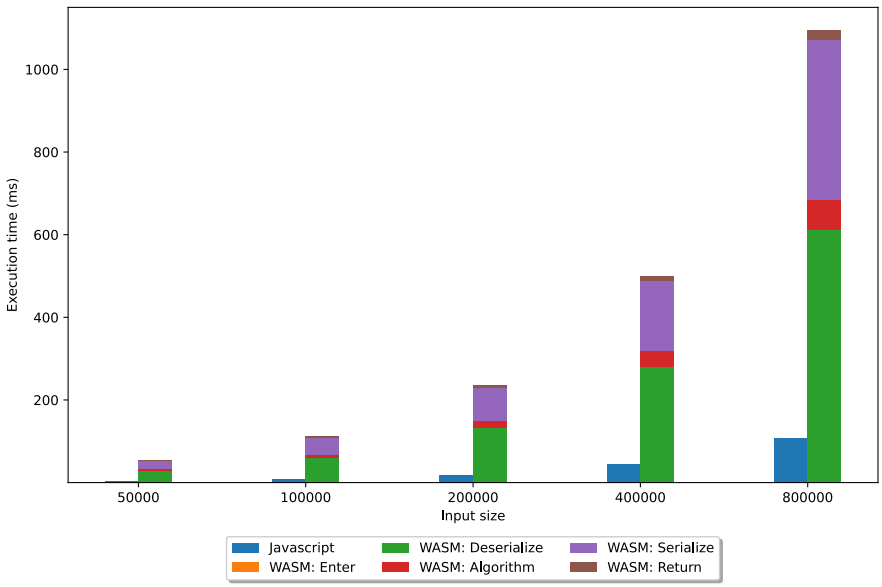

Figure 5.9: Group array of objects by string

Discussion

Grouping items into a combined object is exceptionally fast in JavaScript compared to WebAssembly, mainly due to computationally heavy serialization and deserialization steps. In the two benchmarks which grouped into a limited set of groups, JavaScript would be able to make use of inline caching optimizations: The signature of the grouped object would quickly receive all of its properties, and future deoptimization checks would not fail.

It is interesting to note that both the integer and string versions perform fairly similar, with the full execution time at 800.000 elements for JavaScript and WebAssembly were 93 and 1043 ms respectively for integers, for strings 107 and 1095 ms respectively.

Although the group keys are duplicated, the group values in JavaScript would be arrays of references rather than duplicated data. For WebAssembly, having been compiled from Rust which uses destructive optimizations, data duplication would be limited during the algorithm phase.

The results from the benchmark grouping by unique IDs is interesting, as both languages show a slowdown. This could be due to the additional heap allocations required to create a new array for each element. Furthermore, it would become impossible for JavaScript to create a hidden-class optimization, as each grouped element adds a new property. The results had a less stable trend between different input sizes for both languages. For JavaScript, requesting more heap memory repeatedly for the arrays could trigger garbage collection more often.

5.2.5 Filtering

There are two ways of filtering that are being benchmarked, one we implement ourselves for complete insight and one that is builtin in the language as that might be the common use case, the goal of both is however to filter through an array of string numbers [“1”, “2”, ”3”, .. input size] and only keep the ones that contain number “9”. However one is custom filtering which consists of going through the array and for each element comparing it to “9”, the other one uses JavaScripts builtin-filter function against Rusts builtin filter function. The input has been selected to be between 50.000 and 800.000 as it gives a good span and the execution time the algorithms reaches is measurably high.

Result

Custom filtering The plot for the benchmark is shown in figure 5.10. At 50000 elements, WebAssembly’s algorithm phase shows a slowdown of 0.99x compared to JavaScript. As the input sizes grow, the relative performance of WebAssembly’s algorithm phase compared to JavaScript increases, ending at 2.57x compared to JavaScript at 800000 elements. Accounting for the full WebAssembly execution time, WebAssembly starts with a slowdown of 0.13x which increases steadily to 0.27x at the largest input size. The efficacy of WebAssembly is 13.6% at 50.000 strings, and decreases steadily to 10.4% at 800.000 strings.

Builtin filtering The plot for the benchmark is shown in figure 5.11. At 50000 elements, WebAssembly’s algorithm phase shows a speedup of 1.09x compared to JavaScript. As the input sizes grow, the relative performance of WebAssembly’s algorithm phase compared to JavaScript increases, ending at 2.52x compared to JavaScript at 800000 elements. Accounting for the full WebAssembly execution time, WebAssembly starts with a slowdown of 0.15x

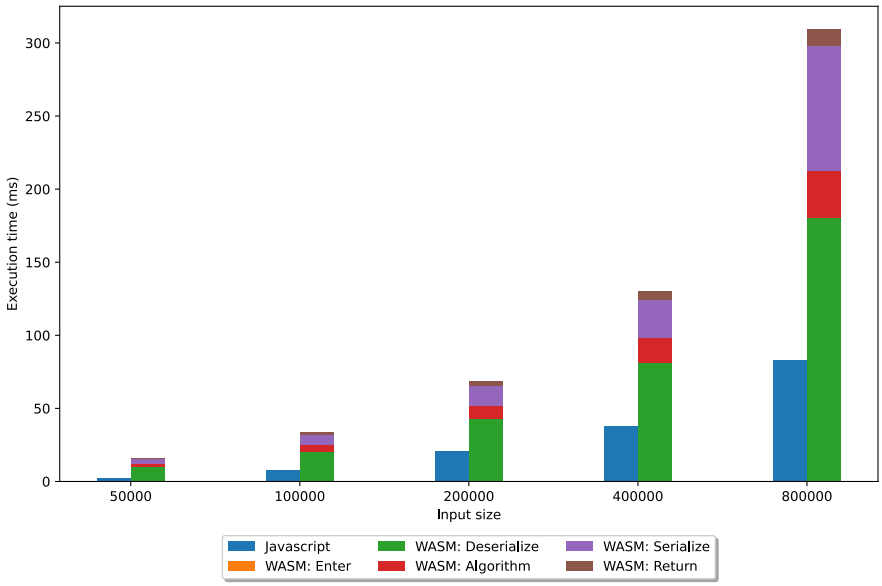

Figure 5.10: Custom filtering, where the input is an array of string numbers

which increases steadily to 0.27x at the largest input size. The efficacy of WebAssembly is 13.4% at 50.000 strings, and decreases steadily to 10.9% at 800.000 strings.

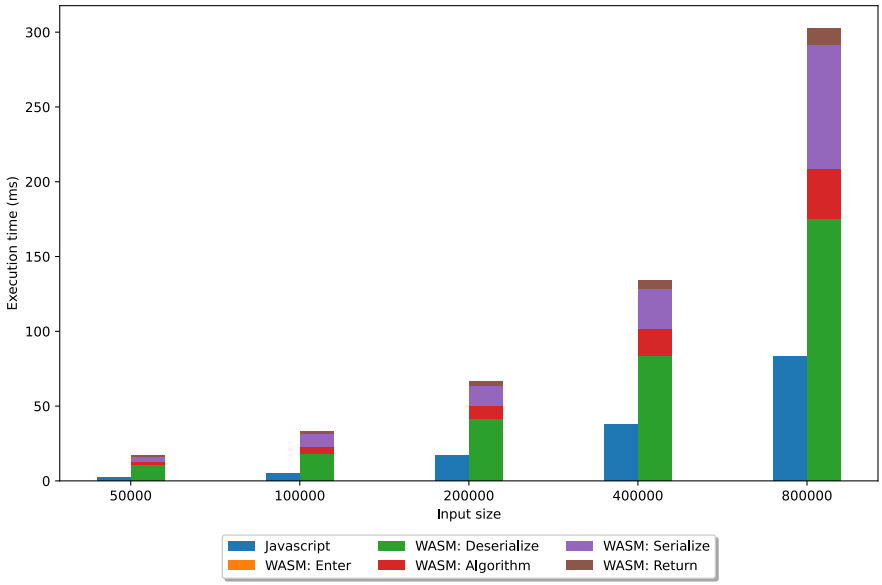

Figure 5.11: Builtin filtering, where the input is an array of string numbers

Discussion

WebAssembly performs the algorithmic part faster than JavaScript but the time consumption for serialization and especially deserialization slows down the execution process, resulting in

a remarkably faster execution time for JavaScript. As can be seen there is not much difference when using a customized filtering method compared to using builtin function, except for overall execution time for the different input sizes. But since this thesis is not for evaluating the performance of the builtin function it will not be discussed further.

Things to note though is that the time for deserialization and serialization is far too expensive, the reason is that it consumes a lot of time when writing an array of strings into WebAssembly, since it needs to be copied and converted into bytes for WebAssembly memory. The WASM: Return-phase is non-negligible, and becomes more of an issue when the input size increases.

5.2.6 Machine learning

For machine learning we have chosen to benchmark the K nearest neighbor (KNN) algorithm [39]. It is used for classification tasks, which IKEA is interested in. It is a supervised learning algorithm that tries to predict the correct class for test data by calculating the distance between test data and training data. Then use the K nearest points of training data to select the class of test data).

Since we do not have time to write a complex machine learning algorithm, the KNN was decided to be a good choice. The KNN was chosen because it is a simple and common algorithm in the space of machine learning, which can provide some insight into how WebAssembly could perform for general machine learning algorithms. It uses a JSON file with data for 10.000 people as JSON objects including their length, weight and gender.

The input sizes for the benchmark were chosen as 500, 1000, 2000, 4000 and 8000 data points. For each input size, 80% of the data set is used in training, and 20% for testing the classification.

Result

The plot for the benchmark is shown in figure 5.12. At 500 elements, WebAssembly's algorithm phase shows a speedup of 2.9x compared to JavaScript. As the input sizes grow, the relative performance of WebAssembly's algorithm phase compared to JavaScript increases steadily, ending at 3.4x compared to JavaScript at 8000 elements. Accounting for the full WebAssembly execution time, WebAssembly starts with a speedup of 1.83x which increases steadily to 3.38x at the largest input size. The efficacy of WebAssembly is 63.1% at 500 elements, and decreases steadily to 99.3% at 8000 elements.

Discussion

As can be seen from the figure 5.12 the execution time for WebAssembly beats JavaScript for all input sizes. One reason for this is the KNN algorithm's reliance on comparing distances between objects. Even though the data is in the form of JSON objects the overall execution time is faster in WebAssembly. As can be seen in figure 5.13 there is a deserialisation phase that consumes time but it becomes negligible when looking at the overall execution time. There is also time consumed for serialization and entering the function from JavaScript, but it is negligible compared to the deserialization phase. The algorithm is still representing between 63.1% and 99.3% of the overall execution time for WebAssembly though. Which

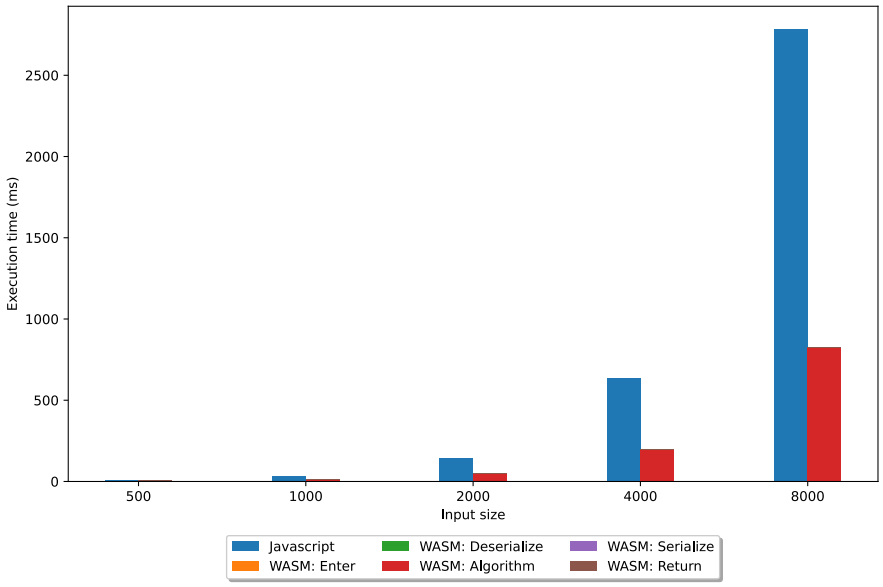

Figure 5.12: K nearest neighbor algorithm

means that, with limited data passing, if the time spent on algorithmic computing for both JavaScript and WebAssembly can be sufficiently high making the data passing time negligible WebAssembly can be used gainfully.

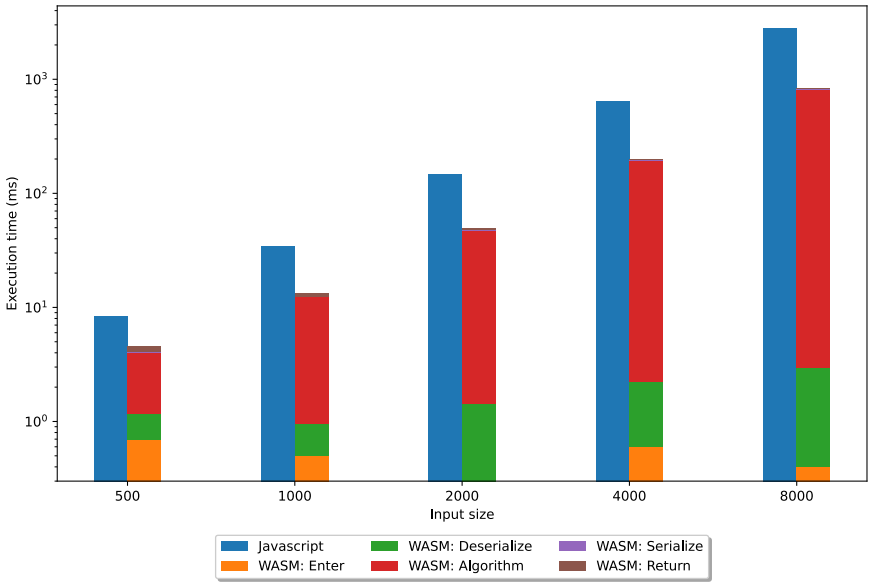

Figure 5.13: K nearest neighbor algorithm with logarithmic scale

5.2.7 Image data generation

For image data generation the algorithm that was selected as representable was the one implemented by Biffle in the guide *Making really tiny WebAssembly graphics demos* [40]. It was

chosen because it uses an algorithm for generating image frames and the need for spending time constructing our own algorithm for animation could be eliminated. The image is then rendered to the HTML canvas element which is commonly used by web developers and IKEA is no exception. Especially since IKEA is interested in improving user experience there is a need for rendering high quality images.

The input was chosen to rendering 50, 100, 200, 400 and 800 frames in both languages.

Result

Figure 5.14 contains the result for benchmarking the image generation algorithm, where the input size is the number of frames to compute with the given logical algorithm. The input sizes differ from what has previously been used in for example sorting, the reason is that the time consumed for generating 800.000 frames would be too long and with 50-800 there is still enough of a span for recognizing trends. The algorithm speedup in WebAssembly for the image generation function started at 1.91x for 50 frames, and increased steadily to 2.1x at 800 frames in WebAssembly. The preparation phases of WebAssembly (WASM: enter, deserialize, serialize, and return) had an average execution time very close to or equal to 0 ms since there was no need to pass data as parameters and function output between the instances. It is no surprise that the overall execution time and the algorithm execution time are the same since there is no need for serialization- or deserialization.

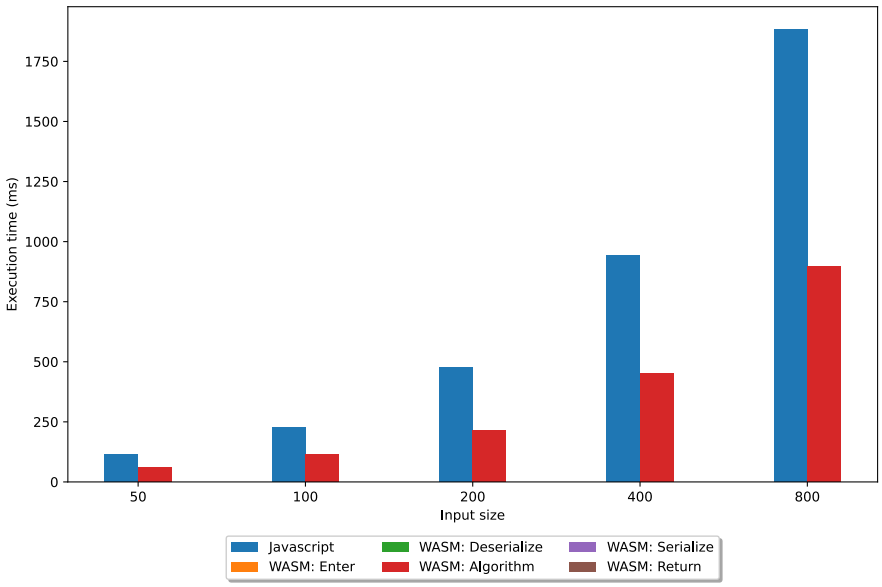

Figure 5.14: Image data generation

Discussion

This algorithm uses a shared buffer between JavaScript and WebAssembly meaning that there is no need for passing any data through parameters or function outputs between JavaScript and WebAssembly. It can be seen from figure 5.14 that there is no need for deserialization or serialization. It is an effective way of using WebAssembly together with JavaScript, since the

computationally heavy part can be ported to WebAssembly. This is for computing the frames using an algorithm that colors the frames. This does not do rendering but the time gained from using WebAssembly can be used for other processes, for example physics in gameplay. The time it took for JavaScript to generate image data for a set number of frames is about double compared to WebAssembly for the same number of frames.

We looked into using SIMD-operations for the image data generation but since the algorithm we used only operated on 32 bit integers for each frame, SIMD was not utilized. Because SIMD is used on the newly introduced vector type v128 which consists of 128 bits, it was decided that it would be too hard rewriting the algorithm with vector type. therefore SIMD was not used. But by rewriting the data generation algorithm making use of vectorized data, SIMD-operations should be able to speed up the execution process by using data parallelism, in theory further increasing WebAssembly performance up to four times.

5.2.8 QR code generation

When generating qr-codes there are libraries that are available for both JavaScript and Rust. When performing this comparison one library was chosen to be used [41] this contains implementation for both JavaScript and Rust and it was therefore chosen to be tested.

Result

Figure 5.15 shows the results from generating a number of QR codes which direct to “https://www.ikea.com”. The input was chosen to range between 500 and 8.000 and not as in sorting as that would have been excruciatingly time consuming and the chosen span was considered to be giving enough information. The algorithm speedup for the QR code generation algorithm starts at 3.11x at 500 QR-codes and keeps steady up until 8.000 QR-codes, where it decreases to 3.01x. Accounting for the full WebAssembly execution time, WebAssembly showed a speedup at 2.76x at 500 QR-codes and keeps steady up to 8.000 QR-codes, where it decreases slightly to 2.71x. Although difficult to see, there is a deserialization phase in the beginning, since there is a string input, however it can be negligible. But the serialization becomes time consuming for larger inputs and can therefore not be negligible. This is because the QR code is returned by WebAssembly as a large SVG string which needs to be converted into a JSON string in WebAssembly Memory and then parsed into an object on the shared heap, which is time consuming considering the size of the SVG string.

Discussion

The only data passing in the start phase is one string which needs to be deserialized. This argues that a good use case for WebAssembly is when there is a simple input, needing limited data passing, that is used for heavy computation and yields many outputs. Also for this case the SIMD-operation could be used which argues for an even faster execution time. However the size of the output SVG string is troubling and if it would have been much larger the time consumption for serialization would be remarkably large.

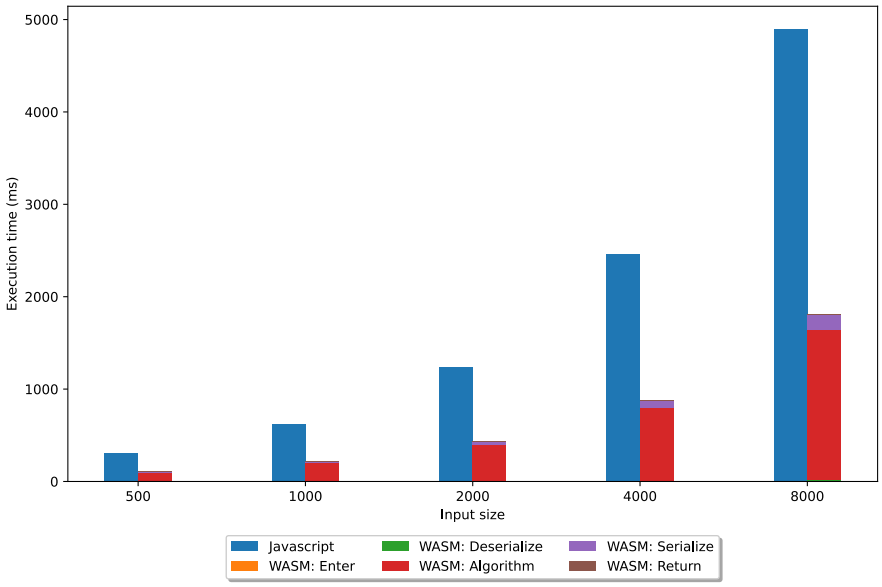

Figure 5.15: QR code generation

5.2.9 Bin packing algorithm

As stated in the analysis the bin packing problem is a computationally heavy algorithm. It was inspired by an online Rust example [42] by MacAulay. Using a First Fit Decreasing-heuristic, the algorithm attempts to place the item with the largest dimension in the smallest available space until no more boxes can be placed, or all items have been placed. This solution differs from the general bin packing problem in that it can only pack items in the initial bin, rather than creating a new one.

Since test data was hard come by, it was generated by using a modified bin-packer: Given a container of a fixed size, it calculated how many of each test item could fit inside if packed with the same algorithm, having a bias towards items with the most extreme dimensions.

The implemented algorithm uses a naive approach due to time constraints, only working on containers and items that were rectangular cuboids and rotations were limited to 90º. Because of its simplicity, smaller inputs resulted in very quick execution times in both JavaScript and WebAssembly. As such, the containers used for test data generation were larger: Cubes with sides 250, 500, 1000, 2000 and 4000 cm per run, resulting in a packing of 208, 1704, 14796, 110388 and 882216 items respectively. As such, in contrast to the previous experiments, the actual input sizes to the bin-packing benchmark is not doubled between two input sizes.

The input to the function was kept as simple as possible in order to minimize data exchange: The dimensions of the container and two lists of identical dimensions: One containing the dimensions of the items and the other the number of items of each type to pack. The output is an array of arrays, each pertaining to a certain item similar to how the input lists were structured. Each subarray contains the coordinates of the item's corner closest to origin, as well as a simple rotation vector.

The 8 possible items were selected randomly from IKEA's website, which provides the dimensions of the boxes they are shipped in.

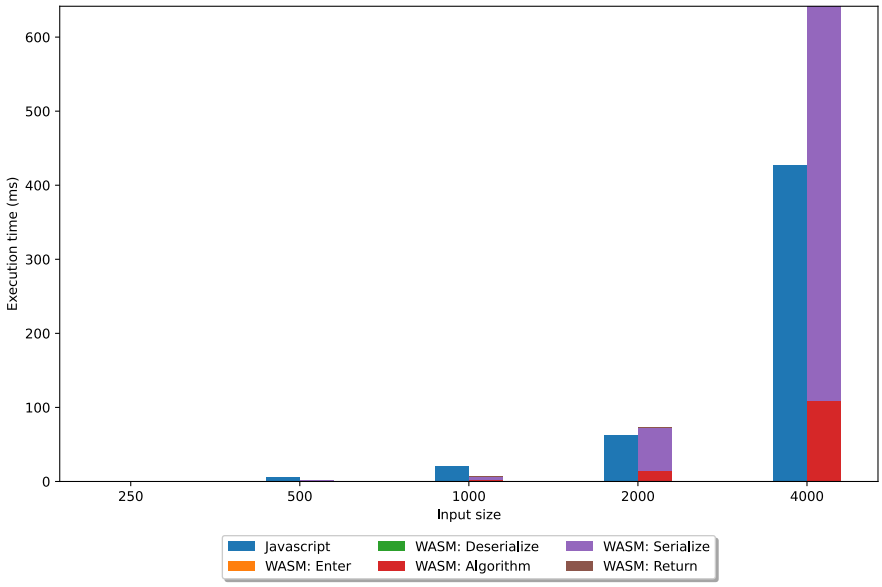

Figure 5.16: Bin packing

Result

Figure 5.16 shows the result of the benchmark.

At 208 items, WebAssembly's algorithm phase shows a speedup of 22.8x compared to JavaScript. As the input sizes grow, the relative performance of WebAssembly's algorithm phase compared to JavaScript decreases, ending at 3.94x compared to JavaScript at 882216 items. Accounting for the full WebAssembly execution time, WebAssembly starts with a speedup of 4.07x at 208 items, increases to 5.15x at 1704 items, then decreases to 3.0x, 0.87x and finally 0.67x at the largest input. The efficacy of WebAssembly varies as well: From the smallest to the largest input sizes, it is 17.9%, 32.1%, 33.1%, 19.1% and finally 16.9%.

Since it is hard to make out the values at the lower input sizes, figure 5.17 shows the same values on a logarithmic execution time scale.

Discussion

As expected, the Deserialization phase of WebAssembly is very limited due to how the input was defined, and ranged between 0.05 and 0.08 ms in execution time. Increased input size did not result in a longer deserialization phase. The same cannot be said about the serialization phase. Since the output grows with an increased number of boxes to pack, so does the serialization.

One possible explanation for the differences could be the data structures used in the solutions. JavaScript does not have a standard MinHeap implementation, and as such, it was implemented by hand. It could be the case that it is not as well optimized as the binary heap in Rust's standard library.

Furthermore, the algorithm is heavily oriented towards arithmetic, which has been shown to run faster in WebAssembly than JavaScript.

The fact that the algorithm performs well on smaller inputs is very promising: As previously stated, the large amount of items packed is a result of the algorithm being too naive,

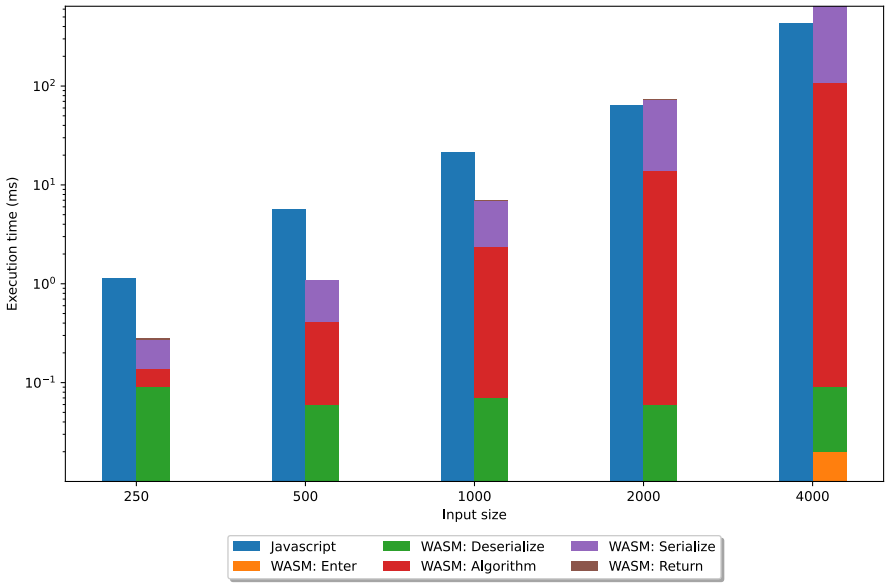

Figure 5.17: Bin packing, execution times on a logarithmic scale

requiring an increase in input size in order to have a less error-prone result. A more sophisticated algorithm that could work with other geometries than rectangular cuboids and right angles should perform better in WebAssembly, if the number of items is kept relatively low.

5.3 WebAssembly startup times

As part of the research questions, the time required to start up a WebAssembly module came into question. This section presents our findings, and discusses the results.

In conclusion, the startup time of WebAssembly is very limited for the binaries used in this thesis. Furthermore, ongoing work in the language standard aims to remove the need to manually instantiate the binaries. The used JavaScript bundler, Webpack, already supports the feature if specific flags are passed to wasm-pack on WebAssembly compilation. For the purpose of investigating the impact of startup time, however, this feature was disabled.

5.3.1 Results

The time to initialize the various modules was tested over 100 iterations per module, and the average time was calculated. Table 3.1 shows the results from this test.

5.3.2 Discussion

As the results show, the time required to initialize the module is very limited, and never exceeds 4 ms. In section 3.1.2 it was established that the instantiation time of a WebAssembly module is linked to the size of the binary. Our results seem to indicate the same behavior, as the smallest binary, "Canvas Rendering" with the size 372 bytes is the fastest to initialize. The size of the other binaries are far larger, starting at 82813 for the second smallest,

Module name	File size (bytes)	Startup time (ms)
Canvas Rendering	372	3.275
Filtering	82813	3.467
KNN	88661	3.580
Bin Packing	92588	3.569
Mapping	95343	3.667
Sorting	98841	3.583
QR-codes	104241	3.59
Grouping	104425	3.43

Table 5.4: Table showing the time to initialize the WebAssembly modules together with the size of the binaries ascending

"Filtering". This can be accredited to the fact that the only module that does not use wasm-bindgen is Canvas Rendering, which is represented using data which is natively understood by WebAssembly. The execution time increased slightly as the binaries increased in size, with minor variations. In previous work experimenting with different binary sizes, it was measured that starting a 29286 byte WebAssembly module took 10.4 ms [22], which is higher than our largest binary at 104425, taking on average 3.43 ms. There could be multiple explanations for this. First, the authors claim that they used the linux command *time* to measure the execution time of the program. Second, the authors run their tests in *Wasmtime*, a WebAssembly runtime intended to be executed outside of the web. In comparison, our data is gathered by isolating WebAssembly instantiation with calls to performance.now in a chrome browser. As such, it is possible that the authors captured more than just the instantiation time in their call.

5.4 Cross-algorithm discussion

There are general patterns that are clear when running benchmarks across multiple algorithms. This section will discuss some of those patterns in an attempt to explain the reasons for them. Finally, the section contains a short summary of recommendations as to how WebAssembly should be used given the results, and potential areas of improvements.

As can be seen in the benchmark results, the actual algorithm phase is rarely slower in WebAssembly than in JavaScript. In the cases where JavaScript is faster than WebAssembly, it is usually data exchanges that dominate WebAssembly execution time. The time consumption for data passing poses problems for the possible usage of WebAssembly as it slows down the execution time for processes that need to pass a lot of data between WebAssembly and JavaScript. As was discovered when running the benchmarks, filtering, sorting, grouping and mapping introduces a large overhead when passing data between JavaScript and WebAssembly. Even though the algorithmic part executes faster in most cases, this overhead makes the use case for WebAssembly discouraged. Improvements in the data passing might be performed and will be discussed in the next chapter “discussion and related work”, but in our case these algorithms should not be used with WebAssembly. In these cases it should also be noted that JavaScript still performs quite well and the gain for using the algorithmic part

in WebAssembly is in terms of milliseconds even for large inputs.

One interesting observation when executing code in WebAssembly was the “WASM: Return” phase. The initial speculation was that this phase only entailed JavaScript removing references to the data that was shared with WebAssembly in preparation for the deserialization step, since the timestamp was created immediately before returning from the function. However, some quick testing on the JavaScript side showed that this part of the process was almost instantaneous. This behavior was inspected using various test functions, receiving both simple data that was immediately understood by WebAssembly such as integers, more complex types such as strings, and finally arrays. The WebAssembly code was then inspected. Although highly obfuscated, it was clear that simple inputs such as integers were returned by simply removing the value from the WebAssembly stack, resulting in a minimal overhead. Meanwhile, the more complex inputs generated additional code before returning control to JavaScript. The most plausible explanation is that when WebAssembly takes control over complex data structures by copying them, it allocates internal memory to hold the data. At the end of the function, this data is cleaned up to free space. Comparing benchmarks in both the sorting and mapping category, arrays of either strings or integers were passed to WebAssembly. The "WASM: Return" phase of the functions working on integers had minimal durations, often measured to 0 ms, while the string versions lasted for several milliseconds. Since the internal data is stored as a vector, it is likely that cleaning a vector of integers requires simply marking the address of the underlying address pointer as free. Meanwhile, the string version contains a vector of string references which are allocated on the heap. These need to be individually deallocated, and as such, takes much longer to clean up.

Another interesting observation is that the “WASM: Enter” phase has little- to no impact on execution time. In most benchmarks, the duration of this phase was measured to 0 ms, with some instances lasting for a fraction of a millisecond. During this phase, non-primitive parameters such as strings, arrays and objects are added to the shared heap, before execution is handed over to WebAssembly. As such, context switching is shown to be of little consequence in long running algorithms.

In RQ 1.a “How is it dependent on the execution environment, compiler and choice of algorithm? To what extent?“, the aim was to explore factors that affect the execution time for WebAssembly and JavaScript. The results presented and discussed in this chapter shows that the differences in JavaScript and WebAssembly is heavily present and affects the execution time when performing different algorithms. Due to time constraints we did not have time to study other compilers and execution environments.

Connecting back to RQ 1.c “Why does WebAssembly have a start up time, can it be avoided? Is it negligible?”. Our findings show that the startup time for WebAssembly can be negligible and even eliminated when using WebPack and Wasm-pack. But even without WebPack and Wasm-pack the wasm module could be instantiated when loading the page which would remove the startup time from execution. It was found that the time to instantiate a WebAssembly binary is dependent on the size of the binary. The main problem is the time consumption of data passing that is present for some algorithms.

Algorithms that only perform simple tasks once, for example filtering through a semi-large array once, are probably not worth porting to WebAssembly because of data passing overhead and JavaScripts filtering function is sufficiently good.

One discovery from the results and discussion above is that serialization and deserialization becomes more time consuming when the input is an array of string type compared

to an array of integer type. The execution time is even higher when arrays of objects are passed. This can be seen in figures 5.1 and 5.2 (note different scaling on y-axis). The reason for this is because string is a more complex data type compared to integer and therefore needs more work in order to be understood by WebAssembly. For the use case of QR code generation, the result from that discussion was that generating many qr codes from one string is a good use case since it is a simple input, needing limited data passing, that is used for heavy computation. However if the input would have been an array of strings, simple enough in JavaScript, the time consumption for deserializing the array would have been devastating. For the use case of QR code generation, the results were in favor of WebAssembly compared to JavaScript. Both the input- and output from the function were strings, which requires some preparatory work. However, for such limited non-primitive data, the impact was extremely small in relation to the effective work of the algorithm.

RQ 2.a "What are common algorithms and services in the IKEA front-end landscape? Are they suitable for WebAssembly? Why?", is used for exploring if the chosen algorithms from the analysis chapter are suitable for WebAssembly and why. As we have found in this chapter, some of the algorithms that IKEA uses can be gainfully implemented in WebAssembly instead. Below we will give a list of recommendations for what algorithms WebAssembly performs well in and in which algorithms it performs worse than JavaScript.

5.4.1 Recommendations

From the results and discussions in this chapter the recommendations for algorithms that WebAssembly should be used for instead of JavaScript is:

- Image data generation, both for QR code where a string input yields many qr codes as output, for example qr code for a discount and also for generating images for HTML canvas elements.
- Machine learning that relies on computationally heavy algorithms where the time for computation is longer than the time for serialization and deserialization, in this thesis a classification algorithm was tested and executed faster than JavaScript.
- Simple algorithms that have limited data passing, or even utilizes a shared memory buffer not needing any data passing. The execution of the algorithm has been seen to always execute faster than the algorithm in JavaScript, meaning that data passing can be limited or removed WebAssembly can be beneficial.

5.4.2 Areas of improvement

What has become strikingly clear from this chapter, that has not been discussed in the research papers we read for our literature study, is that passing data between JavaScript and WebAssembly is time consuming. Previous work that we have read, have focused entirely on problems with data that can be easily shared between JS and WebAssembly. Anything more complex than floats and fixed-size arrays has yet to be seen. However our work is heavily oriented towards generic arrays and object data. Since there are many cases where WebAssembly and JavaScript needs to pass data between the two, this area needs to be improved in order for WebAssembly to be used beneficially in many algorithms.

Chapter 6
Discussion and related work

To come to a conclusion and solve the initial problem it is necessary to discuss the findings we have gathered and relate it to prior art. We will reflect upon our own work and method as to see what worked well and what should have been done differently. The threats of validity will be discussed in order to evaluate the strength of our claims. To motivate general use cases from the findings in this thesis the generalizability of the results and insights will be discussed. We will relate our result to related work and bring insights into future work that has been raised by this thesis.

6.1 Reflection on our own work and method

The following section analyzes various aspects of the method used to reach the results. Knowledge gained from the work process that could have altered the approach will be discussed, and alternatives to the method will be accounted for. In summary, most of the planned methods worked well. Some phases, especially when researching and creating a benchmarking framework, had some surprises in store that were hard to predict and required slight deviations in the plan.

6.1.1 Analysis and interviews

The interview phase of the work did not differ significantly from the planned approach. However, it consumed a substantial amount of time during the early phases of the work process, and the vast majority of the information gathered had to be discarded. The questions were deemed suitable, as it was directly concerned with what we aimed to learn. However, there were certain aspects of the interviews that could have been done differently. First, the interviewees did not properly learn about our goal until the interview started beyond a quick summary when we reached out. We should have been more explicit in our goal and show the questions ahead of time. Second, the majority of the interviewees did not know

what WebAssembly was and what it could do. It could have helped prime the interviewees if its capabilities were showcased better.

In hindsight, it would have been far more time efficient to have employees fill in a form with relevant questions, which could have reached the same result, and would have been a strong contender to the planned method. Another approach would have been to align the work process to a single team with computationally heavy goals in mind. This could have led to a more focused- and deep analysis, rather than the broader approach this paper used. However, this would have come with the disadvantage of the results becoming less generalizable. Furthermore, such teams were not found- for IKEA, the origin of this thesis is curiosity rather than necessity.

6.1.2 Creating the benchmarking framework and approach to instrumentation

In the process of analyzing different approaches to program instrumentation, no tools that could perform the same analysis on both JavaScript and WebAssembly were found. The existing solutions focused on other aspects of program analysis. Previous works such as “Empowering Web Applications with WebAssembly: Are We There Yet?”[17] and “Understanding the Performance of Webassembly Applications”[18] use blackbox measurements to capture execution time behavior in both languages. While this could have been one way of studying the algorithms in full, our analysis of how WebAssembly exchanges data with the JavaScript context led to suspicions that this approach would hide interesting details from the results. As such, creating a custom instrumentation framework is considered to have been the correct choice for the purpose of this thesis. Although the phases of our work were listed in a sequence, there was a significant overlap between designing the benchmarking framework and running the benchmarks. During this overlap, it was found that our approach using custom timestamps in rust code had little overhead. Using conditional compilation in Rust, the instrumentation statements could be switched on and off through flags, allowing us to compare execution time of instrumented WebAssembly and pure blackbox WebAssembly. The results were that using the four additional instrumentation statements inside the code had minimal impact on execution time, see section 4.3.1. Using Chrome’s builtin developer tools page, however, changed the execution time significantly: By simple keeping the tab open while running the benchmarks, the same heapsort algorithm test had a minor impact on JavaScript execution while WebAssembly execution times were impacted significantly, see section 4.1.2. The reason for this difference was unclear and we were unable to find information about this behavior. As such, this method of gathering data was found to be unreliable for our purposes.

Concerning generalizability and control over what was instrumented, however, the choice to implement the instrumentation statements manually in the rust code might not have been the best approach, as seen with the "WASM: Return" phase which did different things depending on the output. If this had been known beforehand, we would have created a customization of the wasm-bindgen tool that injects instrumentation statements programmatically, which would have increased control over what was measured. Furthermore, it would have allowed for a more general benchmarking framework that could be used in both client- and server side solutions and would not rely on a combined webpage- and benchmarking solution.

The issue of exchanging non-primitive data between JavaScript and WebAssembly contexts was not immediately obvious. During the analysis phase, it was noted that previous research papers- and resources provide information and data for problems that can be expressed using integers and fixed-size arrays which WebAssembly handles with relative ease. The problem of data interoperability is stated implicitly in that WebAssembly operates on numeric types, which JavaScript can either append to the WebAssembly stack or encode into shared memory. It can be discussed whether or not this knowledge would have changed our use of tools such as wasm-bindgen in this work: On one hand, using wasm-bindgen was a time efficient way to create a bridge between the contexts, which allowed us to utilize our time to implement- and test more algorithms. On the other hand, serializing- and deserializing data through wasm-bindgen's generic JsValue type is a questionable practice: it acts on arbitrary data and as such utilizes JSON encoding- and decoding, which is potentially costly. Another approach would have been to manually create logic to encode- and decode data on both the JavaScript- and WebAssembly side, which would have been a lengthier process but would have been tailored to the data, potentially increasing WebAssembly's total performance. Although potentially better for a highly tailored solution, it is not a recommended practice for very generic solutions.

Another concept that was briefly explored was WebAssembly's fairly recent addition, externref. Rather than duplicating data into WebAssembly memory, one could pass a reference to data in the JavaScript context. Two things became apparent from some quick testing. First, the feature is disabled by default in wasm-bindgen and it is required to pass extra flags for the tool to make use of it. This affected the WebAssembly optimizer wasm-opt and the JavaScript-WebAssembly bundler wasm-pack, neither of which currently support the feature. As such, externref could only be tested in a manually crafted binary without optimizations. Second, testing showed that although the data passing overhead was reduced dramatically, the execution time of the algorithm was increased to such a degree that its full execution time was even longer than the serialization-deserialization solution. The reason for this is not completely understood, but since the data only exists as a reference in WebAssembly, one possibility is that any usage of the data in WebAssembly requires switching back to the JavaScript context. The results show that context switches are fast in WebAssembly, as both the phases "WASM: Enter" and "WASM: Return" can be close to instantaneous phases. A study from Mozilla shows that this can be the case, as the execution time of 100.000.000 function calls from WebAssembly to JavaScript has been improved from the previous 5500 ms to 500 ms [43]. However, it is unknown if this applies to chrome and externref, as the study specifically investigates functions exported between the contexts in Firefox.

6.1.3 Running the benchmarks

The benchmarks were executed using two primary parameters: Number of iterations and algorithm input size.

Little information was found in regards to an adequate method of picking the number of iterations. Using similar research as a starting point, a common number of iterations per benchmark was 5 [18] or 10 [17][44]. As such, it was decided to use 10 iterations per benchmark in this work.

As the algorithms were to be implemented, it was decided early on that all algorithms should have known source code or otherwise kept simple to save time. As an example, the

choice of using KNN as a representation of machine learning algorithms. It is arguably the most simple machine learning algorithm, and has some odd properties such as no model training phase and a fairly demanding test phase. Similarly, the two sorting algorithms that were used, heapsort and quicksort, were created by hand rather than using the standard library sort functions available in the two languages. The motivation for this is that we wanted to know exactly what code was being run, rather than rely on a standard library or machine learning library, which could potentially hide away implementation details.

Concerning sorting algorithms, the chosen algorithms might not have been the best choice: both implementations are recursive, in-place sorting algorithms, with the most significant difference being that heapsort is more predictable in execution time. One of the algorithms should have been mergesort, which utilizes auxiliary arrays and as such, requires O(n) additional space to execute. Although the current choice does not threaten the validity of the results, it would have been interesting to see if the additional memory usage would have affected the two languages in any way.

6.2 Threats to validity

In order for the conclusion of this thesis to be valid, the data need to be questioned as to see if there could be other factors that influence the results. In this chapter our data will be evaluated and arguments opposing the result will be discussed.

The most important conclusion we assert, which we have not seen in related work, is that data passing can be detrimental for WebAssembly usage in many algorithms. However we have not read all papers touching upon WebAssembly and JavaScript interprocess communication, meaning that it might have been discussed in previous work. But it is deemed not common knowledge since we could not find it during our literature study. But since our literature search was done in order to find differences between and JavaScript and WebAssembly, we might have missed literature about data passing. To see the extent of this we decided to do a quick search where the criteria was changed in order to target data passing. Most previous work that explores the performance of WebAssembly does mention the fact that WebAssembly only supports float and integer data natively. However, it is common to see previous research being focused on domain-specific, demanding algorithms that can interface easily between Javascript and WebAssembly. Another threat to the conclusion that data passing is the killer of performance could be that our choice of using serialization and deserialization (SERDE) was wrong and that there is another way of passing data between JavaScript and WebAssembly that is more efficient. But when seeking ways of passing non-trivial data, SERDE was the most common and visible result online, which is why we decided to use it. We later found out about ExternRef and WasmAbi. We did a quick test using ExternRef. Although the preparatory phases of WebAssembly execution were far shorter, it resulted in dramatically increased execution times during the algorithm phase. However it was not a solution that was explored sufficiently in order to draw any conclusion to how it would compare to SERDE. Using wasmAbi, an interface in wasm-bindgen to provide logic as to how data structures should be encoded and decoded from memory, was discarded as an initial study of it pointed out that it would be time consuming to implement. Since there are other ways of avoiding/limiting passing of data our conclusion still stands that data passing should be limited, especially when using SERDE, however there might be other solutions that

can allow data passing in a more efficient way.

One problem with our claim about data passing is that a framework was used to generate helper functions that performed the data exchange process. If we instead had customized the data passing for each of the algorithms instead of using stringify/parse for all, we might have been able to reduce data passing time but we decided that this would not be a general use case for WebAssembly since customizing data passing would be time consuming. Since we could save a lot of time using wasm-bindgen it was decided that it would be the best way going forward.

A bottleneck in our ability to reason about the results is the nature of the WebAssembly code that was generated: The use of wasm-bindgen increased our overall productivity and saved time, but would introduce additional code to each WebAssembly function, specifically at the beginning and the end of them, and could depend on the in-put- and output of the function. It was presumed to be code meant to clean up memory used by data that was about the go out of scope due to not being present for very simple primitive data. But since it entailed upwards of a few thousand lines of code that were hard to analyze, the exact nature of this code was not understood and brings some uncertainty to our reasoning about it.

Our recommendations for which algorithms work well with WebAssembly and which should not be used with WebAssembly are affected by the designs of said algorithms. We claim that machine learning for classification can be beneficially implemented in WebAssembly as opposed to JavaScript, if de/serialization can be limited. However we only tested a simple machine learning algorithm, k nearest neighbor, which might not be the best representation for overall machine learning classification algorithms. Instead we could have chosen a more mathematical approach with vector computation, using SIMD-operations we would probably have seen that the mathematically heavy part of machine learning can be improved. But also for this case we argued that a simple machine learning algorithm would be most time efficient to implement which would result in the possibility to further explore other algorithms.

Although not found in literature it is possible that React could be contributing with additional overhead which might have skewed the results a bit. We do not think it is likely that it affected the result and if it did it was not by much, therefore we did not investigate this. Furthermore, the usage of React is very common in web development today, and as such any potential overhead could increase the generalizability of our results.

One general threat to the validity of this thesis is the fact that we decided to only investigate one browser, namely the V8. Although there are differences in optimizations between different JavaScript engines, we decided that it was more beneficial to fully understand the execution process of one commonly used engine instead of briefly investigating multiple. However this poses the threat that we can not guarantee that the results from running the benchmarking framework will yield equivalent results.

In order for the results to be accurate, and not biased by different computer specs, each of the benchmarks was done using one computer, namely the Windows machine, see specifications in section 5.1.6. However this poses a potential threat as the results become biased by the specs of that computer. In order to see the magnitude of this issue the benchmarking framework was used on another computer as well, namely a 2021 Macbook Pro with a 2,3 GHz 8-Core Intel i9 CPU, 2 x 8 GB DDR4 RAM. Although the ratio in the overall WebAssembly execution time stayed the same, there were some differences in overall execution time. For some algorithms where execution speed were almost the same, WebAssembly would win on

one computer and lose on the other. However we did not have time to further investigate the reasons for why.

Weaknesses in the benchmarking method could influence the gathered data, making it less trustworthy. One such weakness is that there are processes which could have consumed processing power in the background lessening the efficiency for the algorithms. One such process could be garbage collection since it pauses the execution in order to clear up memory. Although WebAssembly in itself does not perform garbage collection, it is quite possible that it could have been triggered during one of the multiple data exchange phases, where JavaScript is used to allocate and deallocate data which is shared between the instances. Another uncertainty with the framework is that there is no JIT-warmup in the tests. We decided that allowing JavaScript to use optimized machine code at the start of each test would be an unfair advantage. The consequence of this is that we can not know when - or even if - JavaScript optimizations occurred. Given the literature study we can with certain confidence say that V8 engine optimizations probably occurred, as the execution time of JavaScript did not change significantly between multiple iterations.

As raised in the reflection on our own work and method, choosing the number of iterations is no easy task as the data found addressing this choice was limited. Previous work used 5 and 10 iterations, but did not motivate their choices. As such, we settled for 10 iterations per test. The reason to use a certain number of iterations was to lessen the impact of variance between test runs. Although outliers were uncommon, they were present and could increase the execution time during certain phases significantly. Although our method of calculating the average of 10 iterations for each phase independently should bring the final result closer to a realistic number, it runs the risk of being too high for both JavaScript and WebAssembly. One argument in favor of this method, however, is that the outliers are not caused by measurement errors, but are a natural occurrence due to the nature of the JavaScript runtime. They could, however, give an unfair advantage to one language over the other.

The size of the smallest program inputs were chosen through testing various inputs until one was found that showed a result that was similar across multiple independent benchmarks, which could be a questionable practice. This number was fairly high for most benchmarks. Due to this, the report fails to provide data on very small inputs, such as sorting 100 elements. Previous results show that smaller program input sizes favor WebAssembly over JavaScript due to missing JIT-optimizations. Analyzing the trends from section 5.2.2, this would likely have been the case for most algorithms in our work, as the most common observation was that WebAssembly's speedup decreased as the input size increased. However, another observation is that the most common pattern is that the efficacy of WebAssembly increases as the input size grows, and as such, data exchange could have been far more dominant on smaller inputs. This report does not provide any data on this, and tests on very small inputs lead to wildly varying results in execution time and as such, were not deemed suitable for the purpose of this thesis. As a result, the claims made by this thesis cannot be generalized to very small inputs, which is certainly a valid case depending on the context.

Many algorithms such as the sorting methods in JavaScript mutate the input. Since the same function was called on each benchmark iteration, the solution to this was to pass a closure to the benchmarking function, which generated a new copy of the original input data before each test. This would likely have triggered garbage collection more often during the benchmarks, as references to the previous inputs were dropped between iterations.

6.3 Generalizability

The results and deliveries from this thesis should be viewed from the given context. However it is important to highlight the generalizability of the work in order for it to be useful and applicable in other contexts.

Many of the results derived from this thesis are highly generalizable. The only IKEA influence was when the long list of interesting algorithms was reduced in order to fit with the IKEA usability. The claim that data passing should be limited when using wasm-bindgen is therefore applicable for the general use case of WebAssembly for any algorithm.

The algorithms that were benchmarked are commonly used and not specific for IKEA. This means that the conclusions from this work can be applied in many contexts where WebAssembly performance is sought in regards to execution. Another important delivery of this thesis is the long list of algorithms that are interesting to benchmark given the differences in WebAssembly and JavaScript. This list kindles thoughts of other use cases where WebAssembly could be faster than JavaScript but which were not investigated due to time and resource restraint and given context.

The benchmarking framework was developed independent from the IKEA context and with the aim of providing enough information when benchmarking algorithms in JavaScript compared to WebAssembly. The language decided to target WebAssembly was Rust and this affects how the WebAssembly binary will be built compared to if the language used would have been C/C++. This means that the data and results the framework produced are specific for the Rust toolchain but the difference, had C/C++ been used, should be minimal. When designing the framework we had generalizability in mind and decided therefore to use React instead of simple html pages as most web pages, that depend on performance, are built using a framework.

6.4 Related work

The following section contains reviews of papers related to this thesis. A brief summary of each work is given in order to let the reader understand its contents. Then, each paper is discussed in how it relates to this thesis. The papers that will be reviewed are, in order, [44], [45], [17] and [46].

The Need for Speed of AI Applications: Performance Comparison of Native vs. Browser-based Algorithm Implementations

Summary

The authors address the problem with increasing demand on AI applications. The paper also explores the current state of client-side possibilities for computationally demanding AI applications, by comparing JavaScript, ASM.js, WebAssembly and native binaries. The authors claim that the current advances with browser based implementations such as JavaScript, ASM.js and WebAssembly challenge native binaries with regards to execution time. To prove

their claims the writers conducted 10 test cases with different computational heaviness, some being more extensive processes than others, on the different implementations.

In conclusion, the authors' tests yielded the result they were expecting for the computational and function-call oriented operations that do not heavily rely on memory management, but they were surprised by WebAssembly's capabilities. The result was that compiled C++ code executed fastest, secondly WebAssembly closely followed by ASM.js and JavaScript far behind. Interesting cases were when WebAssembly beat compiled C++ which it did in four cases. ASM.js beat compiled C++ in one case and JavaScript never beat compiled C++. The authors interpret that this speed difference could be because of WebAssembly's SIMD vectorization happening within the Emscripten pipeline.

Discussion

The authors in this paper only looked at overall execution time and therefore lost insights into how data passing affected the execution time. What we did differently was that we studied more in detail what parts of the execution process are time consuming. This proved to be novel as we have yet to find any prior art addressing this question. However their tests were made with an AI context in mind, and therefore the focus was on number crunching routines, which differs from our focus as we were seeking a more all around use case for WebAssembly. Furthermore they only used a simple html page for their tests as compared to our tests which were done on a react page. Since their focus was on number crunching algorithms their results differ from ours as it shows a speedup using WebAssembly compared to JavaScript for all algorithms.

Since they did not provide an explanation of how data is passed while running these algorithms it is hard to argue if data passing is the reason for the cases where WebAssembly does not completely outshine JavaScript. However for some of the algorithms they tested we can conclude that data passing is limited, for example the test of filling an array, and for this WebAssembly dominates with a faster execution time compared to JavaScript. Therefore their findings and ours does not stand in dispute, but rather that we provided more insight into reasons for why their number crunching algorithms performed well with WebAssembly.

Accelerate JavaScript Applications by Cross-Compiling to WebAssembly

Summary

The problem that is addressed is the insufficient performance for compute-intensive processes that exist with JavaScript. To solve it the authors present their cross-compiler Speedy.js, which is a subset of JavaScript/TypeScript that was designed to perform better than regular JavaScript when dealing with computationally intensive functions. The cross-compiler should only impose minimal restrictions on the JavaScript code and reduce runtime fluctuations. To prove that their cross-compiler Speedy.js enhances performance they conducted benchmarks from regular typescript compared to typescript cross-compiled with Speedy.js.

In conclusion the tests proved that there was an increase in execution speed for most cases and in some cases up to a factor of four. There were also less fluctuations in execution with

Speedy.js compared to an ordinary JavaScript engine, they claimed that this consistency only will grow with more support for WebAssembly in the browsers.

Discussion

One notable difference we did compared to this work is that instead of compiling JavaScript code to WebAssembly, we made use of Rust. Another difference is that instead of passing objects between JavaScript and WebAssembly, resulting in data passing, they made use of the heap and only passed object references between the two. This limits the useability as they must specify more information about how the stored object on the heap should be represented in WebAssembly, for it to operate on it. Our work brings new insights into how to pass objects instead of object references, and thereby freeing up space on the heap. With Speedy.js they only used 8 test cases which were all handling numbers, what we do differently is that we used more extensive tests. As their work was conducted in 2017, when WebAssembly was still in the birth stage, they did not have as much prior art to rely on as we did. Therefore the context from which their work was done differs and they did not explore use cases of WebAssembly rather just benchmarking simple numeric algorithms. One issue with their conclusion is that in order to use Speedy.js relaxations in regards to safety must be reduced, i.e array boundary checks can no longer be done, which will enhance performance at cost of safety. But it still points to an improved performance when using WebAssembly instead of JavaScript for numeric computations.

Empowering Web Applications Using WebAssembly

Summary

The authors attempt to shed some light on the question “what parts of your code should be executed through WebAssembly, and which parts need to be improved in WebAssembly execution?”. The authors claim to contribute to the scientific community by exploring the performance of WebAssembly applications in comparison to JavaScript in both execution time and memory usage. The goal is to explore shortcomings in the current WebAssembly implementation in Chrome in order to create opportunities to improve it. To substantiate their research, the authors use an experimental analysis of JS/WASM execution, by running a computationally heavy C benchmark cross-compiled into both JavaScript and WebAssembly in the Chrome browser. This is done using a C cross compiler, Cheerp. Their results show that for smaller program inputs, WebAssembly executes faster than JavaScript for all benchmarks, but more than half of the benchmarks lag behind as the input size becomes larger. The authors claim that this happens due to more aggressive JIT-optimizations on the JavaScript side, which greatly improves JavaScript performance while having negligible impact on WebAssembly. Furthermore, memory usage of WebAssembly is shown to be greater than that of JavaScript, being explained through JavaScript’s garbage collection in contrast to WebAssembly’s linear memory model which does not automatically reclaim memory.

Discussion

The authors claim that the sluggish nature of JavaScript on smaller input sizes can be attributed to missing JIT-optimizations. While this is true, the input sizes in our work were larger in general, with small differences in execution time between iterations for both WebAssembly and JavaScript. Furthermore, many of the algorithms that were tested in their work are geared towards a more specialized domain- matrices. Although computationally heavy algorithms, the data can be easily passed between the context and as such should result in minimal data passing overhead. Furthermore, another difference between our work and theirs is that they used the same C source code for both JavaScript and WebAssembly, using Cheerp to cross-compile to the two languages, while our algorithms were manually coded for both JavaScript and Rust.

The Cost of Speculation: Revisiting Overheads in the V8 Engine

Summary

The authors study the impact of type speculation in JavaScript. Introducing optimizations in the dynamically typed language comes with the price of requiring type information to be checked during runtime in order to ensure that the optimizations are still valid. The authors claim to contribute to the scientific community by studying the overhead that comes from type speculation and deoptimization checks. The goal is to identify potential improvements in how V8 handles deoptimization checks. To substantiate their research, the authors run the JetStream2 JavaScript benchmark suite, consisting of 51 JavaScript benchmarks. The generated machine code is studied to find which deoptimization checks were performed and their frequency. Their results show that when running the benchmark suite, 8% of all the generated instructions were deoptimization checks. The most common check is for Small Integers (not-a-SMI), an optimization which makes the assumption that data types are integers and not floats. Furthermore, the authors create an extension of the ARMv64 Instruction Set, by adding new instructions to immediately coerce small ints to machine integers. By doing this, a speedup of 3% is shown on average, with SMI heavy algorithms such as sparse matrix multiplication and dot product performing with a 10% speedup.

Discussion

A significant difference between this paper and ours is the width and depth. The authors have elected to focus on a very specific optimization problem at a low level in JavaScript, and further narrows the scope as they implement a potential improvement for a certain type of instruction. Meanwhile, our work captures a broader scope at a more shallow level, instead focusing on analyzing reasons for discrepancies in performance of two solutions to the same problem over multiple different problem domains. The contexts and purpose are different as well, as their work is meant to analyze- and strengthen JavaScript and JIT-compilers, while ours mainly study strengths and weaknesses in a contender to JavaScript, evaluating its potential as a solution to keeping web pages responsive in the more complex web landscape. The results are not directly comparable to ours, as we did not gather information

about javascript execution on a low level. However, their results can be used as one of the many possible reasons that JavaScript has worse algorithmic performance compared to WebAssembly, as WebAssembly does not have the additional runtime overhead that comes with deoptimization checks.

6.5 Future work

During this work, there were a few concepts- and solutions that were not properly explored, that could potentially address the same problem. This section will briefly present a few of these.

As it currently stands, WebAssembly has no direct support for code instrumentation, and requires a context switch to JavaScript in order to register timestamps for events. Furthermore, this makes the process of instrumenting certain parts of the code less precise, as the instrumentation has to be created at a higher level. An existing proposal to address this is the introduction of instrumentation and tracing functionality in the language [47], which could be an interesting development and its impact and proper usage could be a subject for further studies.

A threat to the validity of our results was that there was a difference in overall execution time for WebAssembly that is larger than the difference in JavaScript when the tests were done on different computers. We did not have time to further investigate this but it points out that the execution time of WebAssembly in the browser is more heavily affected by the computer specs compared to JavaScript. This is an area that needs further investigation in order to fully understand under what conditions WebAssembly should be used.

As a way of avoiding data passing the entire web application can be built using WebAssembly. Frameworks for developing such a web application using Rust as the targeting language are Yew [48] and Seed [49]. It should be noted that using WebAssembly for the entire web application is not yet the solution as it is still slower calling DOM APIs from WebAssembly compared to JavaScript. With the Blazor framework [50], the entire .NET application can be build by using C# and WebAssembly. In this thesis we decided to not pursue the making of an application entirely developed in WebAssembly, but to fully grasp the potential of WebAssembly such an application could be compared to another identical application developed in JavaScript.

The newly added feature of SIMD-operations in WebAssembly was not tested in this thesis. But from the research it can be argued that algorithms implemented with the goal of utilizing this feature will execute faster in WebAssembly compared to JavaScript. But to see the extent of improvement that this feature will bring, algorithms that can utilize the feature should be tested.

Data exchange is a significant issue for certain algorithms, as the data from this thesis shows. There is certain work ongoing to improve WebAssembly and JavaScript interoperability. As previously mentioned, externref was briefly analyzed and tested with poor results. It is unknown if the bad performance was due to missing optimizations, having to switch to the JavaScript context more often, or some combination of both. As support for this feature becomes available in wasm-opt and wasm-pack, a better evaluation of the concept would be interesting. Furthermore, support for interface types is one current proposal to the language, aimed to further improve the interoperability between the languages [51]. Research

into how these concepts could be used to remove the need of costly data exchanges would be very interesting. Although WebAssembly currently has a specific use case in certain domains which do not require large data exchange overheads, and a strict design goal of being minimalist, portable and deterministic in its execution, the appeal of the language could only be strengthened by making it more useful in common algorithms.

Chapter 7
Conclusion

Using WebAssembly can drastically improve execution time performance in web applications under the correct circumstances. The test result shows that WebAssembly performs the algorithm phase of most algorithms faster compared to JavaScript. But passing "complex" data between JavaScript and WebAssembly is complicated and can be detrimental since it relies on JSON serialization and deserialization which is time consuming. This means that for processes which pass trivial data such as integers or fixed-size typed arrays, where little to no data exchange is needed, WebAssembly outperforms JavaScript with regards to execution time. This is because of the difference in memory layout between the two contexts; JavaScript objects may or may not structure fields or array indices in well-defined linear memory which is expected in WebAssembly.

The machine learning benchmark shows that WebAssembly outperforms JavaScript at the classification task. This can mainly be accredited to the deserialization step being less computationally demanding than the iterative euclidean distance calculation of the algorithm, together with the trivial classification output which has no serialization cost.

The canvas rendering benchmark shows that WebAssembly performs better at the image generation task. Since the image data can be shared directly by the two contexts, no deserialization or serialization is required between frame generation.

The QR-code generation benchmark shows that WebAssembly executes faster than JavaScript, even though it relies on string input and output. Although the data passed is non-trivial, it is fairly limited in size and as such, has a very small impact on performance.

Our benchmarks in the grouping, sorting and mapping tasks show that the algorithmic part of JavaScript execution stays competitive, while WebAssembly has a large overhead for passing data between the contexts. As such, common built-in functions for iteratively applying functions to, or organizing arbitrary data in JavaScript are preferable to their counterpart in WebAssembly.

The bin packing benchmark shows that WebAssembly clearly executes the algorithmic part faster but the serialization step is very expensive. The used algorithm is too naive and as a result, too fast. As such, large program inputs were used, resulting in a large serialization

overhead. We speculate that if we had enough time to create a more sophisticated solution, WebAssembly would come out on top regardless of the serialization overhead.

In conclusion, routines that fall into one of the following descriptions are expected to perform better in WebAssembly than JavaScript.

- function input and output can be passed without conversion between the contexts
- function input and output requires conversion, but the conversion time is small in relation to the algorithm time
- Data can be shared without conversion between contexts through memory buffers

References

[1] Izabella Bielecka. *Ikea Sålde för 40 miljarder - nära 500 miljoner Kunder Online*. 2021. URL: https://www.ehandel.se/ikea-salde-for-40-miljarder-nara-500-miljoner-kunder-online (visited on June 5, 2022).

[2] *Usage statistics of JavaScript as client-side programming language on websites*. URL: https://w3techs.com/technologies/details/cp-javascript/ (visited on June 9, 2022).

[3] V. Adve et al. "LLVA: a low-level virtual instruction set architecture". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 2003, pp. 205–216. DOI: 10.1109/MICRO.2003.1253196.

[4] S Sravani. *How javascript was created?* 2018. URL: https://www.tutorialspoint.com/How-JavaScript-was-created (visited on June 9, 2022).

[5] Amit Khonde. *JavaScript internals - ignition and turbofan*. 2021. URL: https://dev.to/amitkhonde/javascript-internals-ignition-and-turbofan-48ef (visited on June 9, 2022).

[6] Anthony Heddings. *What is just-in-time (JIT) compilation?* 2020. URL: https://www.howtogeek.com/devops/what-is-just-in-time-jit-compilation/ (visited on June 9, 2022).

[7] Joao De Macedo et al. "On the runtime and energy performance of WebAssembly: Is webassembly superior to JavaScript yet?" In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)* (2021). DOI: 10.1109/asew52652.2021.00056. (Visited on June 9, 2022).

[8] Richard Artoul. *Javascript Hidden Classes and Inline Caching in V8*. URL: https://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html (visited on June 9, 2022).

[9] Jiho Choi, Thomas Shull, and Josep Torrellas. "Reusable Inline Caching for JavaScript Performance". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, 889–901. ISBN: 9781450367127. DOI: 10.1145/3314221.3314587. URL: https://doi.org/10.1145/3314221.3314587 (visited on June 9, 2022).

[10] Wonsun Ahn et al. "Improving JavaScript Performance by Deconstructing the Type System". In: *SIGPLAN Not.* 49.6 (2014), 496–507. ISSN: 0362-1340. DOI: 10.1145/2666356.2594332. URL: https://doi.org/10.1145/2666356.2594332 (visited on June 9, 2022).

[11] Gabriel Southern and Jose Renau. "Overhead of deoptimization checks in the V8 javascript engine". In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 2016, pp. 1–10. DOI: 10.1109/IISWC.2016.7581268. (Visited on June 9, 2022).

[12] Peter 'the garbo' Marshall. *Trash talk: The orinoco garbage collector*. 2019. URL: https://v8.dev/blog/trash-talk (visited on June 9, 2022).

[13] mkuts12. *Memory management in JavaScript*. 2019. URL: https://medium.com/walkme-engineering/memory-management-in-javascript-2d193c78d125 (visited on June 9, 2022).

[14] *Introduction*. URL: https://webassembly.github.io/spec/core/intro/introduction.html#scope (visited on June 5, 2022).

[15] Abhinav Jangda et al. "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 107–120. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/jangda (visited on June 5, 2022).

[16] *WebAssembly compilation pipeline*. URL: https://v8.dev/docs/wasm-compilation-pipeline (visited on June 5, 2022).

[17] Weihang Wang. "Empowering Web Applications with WebAssembly: Are We There Yet?" In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, pp. 1301–1305. DOI: 10.1109/ASE51524.2021.9678831. (Visited on June 9, 2022).

[18] Yutian Yan et al. "Understanding the Performance of Webassembly Applications". In: *Proceedings of the 21st ACM Internet Measurement Conference*. IMC '21. Virtual Event: Association for Computing Machinery, 2021, 533–549. ISBN: 9781450391290. DOI: 10.1145/3487552.3487827. URL: https://doi.org/10.1145/3487552.3487827 (visited on June 5, 2022).

[19] Alon Zakai. *Why webassembly is faster than asm.js – mozilla hacks - the web developer blog*. 2017. URL: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/ (visited on June 5, 2022).

[20] *Understanding the JS API*. URL: https://webassembly.org/getting-started/js-api/ (visited on June 5, 2022).

[21] WebAssembly. *WebAssembly ES-Module Proposal*. URL: https://github.com/WebAssembly/esm-integration/tree/main/proposals/esm-integration (visited on June 5, 2022).

[22] Niko Mäkitalo et al. "Bringing WebAssembly up to Speed with Dynamic Linking". In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 1727–1735. ISBN: 9781450381048. DOI: 10.1145/3412841.3442045. URL: https://doi.org/10.1145/3412841.3442045 (visited on June 5, 2022).

[23] *WebAssembly Runtime structure*. URL: https://webassembly.github.io/spec/core/exec/runtime.html (visited on June 5, 2022).

[24] David Loshin. *High-Performance Business Intelligence*. 2013. URL: https://www.sciencedirect.com/topics/computer-science/data-parallelism (visited on June 5, 2022).

[25] *WebAssembly Core Specification*. https://webassembly.github.io/spec/core/download/WebAssembly.pd W3C, Dec. 5, 2019. URL: https://www.w3.org/TR/wasm-core-1/ (visited on June 5, 2022).

[26] Angela Pohl et al. "An Evaluation of Current SIMD Programming Models for C++". In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '16. Barcelona, Spain: Association for Computing Machinery, 2016. ISBN: 9781450340601. DOI: 10.1145/2870650.2870653. URL: https://doi.org/10.1145/2870650.2870653 (visited on June 5, 2022).

[27] *Understanding the JS API*. URL: https://webassembly.org/getting-started/js-api/ (visited on June 5, 2022).

[28] WebAssembly. *GC/overview*. 2021. URL: https://github.com/WebAssembly/gc/blob/main/proposals/gc/Overview.md (visited on June 5, 2022).

[29] WebAssembly. *STRINGREF*. URL: https://github.com/WebAssembly/stringref/blob/main/proposals/stringref/Overview.md (visited on June 5, 2022).

[30] *Crate Serde*. URL: https://docs.serde.rs/serde/ (visited on June 5, 2022).

[31] Qiang Yang et al. "Federated Machine Learning: Concept and Applications". In: *ACM Trans. Intell. Syst. Technol.* 10.2 (2019). ISSN: 2157-6904. DOI: 10.1145/3298981. URL: https://doi.org/10.1145/3298981 (visited on June 5, 2022).

[32] Daniel Lehmann and Michael Pradel. "Wasabi: A Framework for Dynamically Analyzing WebAssembly". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, 1045–1058. ISBN: 9781450362405. DOI: 10.1145/3297858.3304068. URL: https://doi.org/10.1145/3297858.3304068 (visited on June 9, 2022).

[33] Koushik Sen et al. "Jalangi: A Tool Framework for Concolic Testing, Selective Record-Replay, and Dynamic Analysis of JavaScript". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, 615–618. ISBN: 9781450322379. DOI: 10.1145/2491411.2494598. URL: https://doi.org/10.1145/2491411.2494598 (visited on June 9, 2022).

[34] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. "SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers". In: *2021 IEEE European Symposium on Security and Privacy (EuroSP)*. 2021, pp. 472–486. DOI: 10.1109/EuroSP51992.2021.00039. (Visited on June 9, 2022).

[35] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002. (Visited on June 9, 2022).

[36] Allen D Malony and Daniel A Reed. "Models for performance perturbation analysis". In: *ACM SIGPLAN Notices* 26.12 (1991), pp. 15–25. (Visited on June 9, 2022).

[37] Benjamin Lannon. *Bundling a rust library to webassembly with Webpack and Wasm-Pack*. 2020. URL: https://lannonbr.com/blog/2020-02-17-wasm-pack-webpack-plugin (visited on June 9, 2022).

[38] *The wasm-bindgen guide*. URL: https://rustwasm.github.io/docs/wasm-bindgen/ (visited on June 5, 2022).

[39] *What is the K-nearest neighbors algorithm?* URL: https://www.ibm.com/topics/knn (visited on June 5, 2022).

[40] Cliff L Biffle. *Making really tiny WebAssembly graphics demos*. 2019. URL: http://cliffle.com/blog/bare-metal-wasm/ (visited on June 5, 2022).

[41] *Project nayuki*. 2021. URL: https://www.nayuki.io/page/qr-code-generator-library (visited on June 5, 2022).

[42] James MacAulay. *A 3D bin packing algorithm in Rust*. URL: https://gist.github.com/jamesmacaulay/471759553c2530a041fdfd6d78b9e836 (visited on June 5, 2022).

[43] Lin Clark. "Calls between JavaScript and WebAssembly are finally fast". In: *Mozilla Hacks – the Web developer blog* (2018). URL: https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast-\%F0\%9F\%8E\%89/ (visited on June 9, 2022).

[44] Bernd Malle et al. *The Need for Speed of AI Applications: Performance Comparison of Native vs. Browser-based Algorithm Implementations*. 2018. arXiv: 1802.03707 [cs.AI]. (Visited on June 5, 2022).

[45] Micha Reiser and Luc Bläser. "Accelerate JavaScript Applications by Cross-Compiling to WebAssembly". In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, 10–17. ISBN: 9781450355193. DOI: 10.1145/3141871.3141873. URL: https://doi.org/10.1145/3141871.3141873 (visited on June 5, 2022).

[46] Alberto Parravicini and Rene Mueller. "The Cost of Speculation: Revisiting Overheads in the V8 JavaScript Engine". In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 2021, pp. 13–23. DOI: 10.1109/IISWC53511.2021.00013. (Visited on June 5, 2022).

[47] *Instrument-tracing in WebAssembly*. URL: https://github.com/WebAssembly/instrument-tracing/blob/main/proposals/instrument-tracing/Overview.md (visited on June 5, 2022).

[48] *What is Yew?: Yew*. URL: https://yew.rs/ (visited on June 5, 2022).

[49] *Seed framework*. 2020. URL: https://seed-rs.org/.

[50] *Host and deploy ASP.NET Core Blazor webassembly*. 2022. URL: https://docs.microsoft.com/en-us/aspnet/core/blazor/host-and-deploy/webassembly?view=aspnetcore-6.0 (visited on June 5, 2022).

[51] Lin Clark. *WebAssembly interface types: Interoperate with all the things!* 2019. URL: https://hacks.mozilla.org/2019/08/webassembly-interface-types/ (visited on June 5, 2022).

Appendices

Appendix A
Interview guide

When interviewing product owners and developers, they were first asked to demo their product. This was done in order to provide insights into the general use of their product and to get a scope of where there could be potential bottlenecks in performance, with regards to execution time. Later followed questions that were specific to each participant's role. Questions to product owners and engineering managers;

- What are future goals for your product(s)? (To product owners)
- What are future goals for your team(s) and IKEA? (To engineering managers)
- Is there something that prohibits you from reaching them?
- Do you know if there are any bottlenecks in your product that can slow down the web page for users?

For the developers the focus of the questions were more on the actual usage of algorithms, since they are the ones closest to the code implementation. Questions to web developers

- What are common algorithms that are used in the product you are developing?
- Is there any algorithm that creates a bottleneck?
- Does the product handle a lot of data?
- If the data is handled on the server-side, is it sensitive information?