Document Information

Analyzed document	An_Evaluation_on_the_Performance_of_Code_Generated_with_WebAssembly_Compilers.pdf (D108864937)
Submitted	6/14/2021 2:35:00 PM
Submitted by	
Submitter email	D19125652@mytudublin.ie
Similarity	2%
Analysis address	455507.dit@analysis.urkund.com

Sources included in the report

SA	**Examensarbete+-+Aevan+Dino%2C+Seth+%C3%96berg+%5BGrupp+23%5D.pdf** Document Examensarbete+-+Aevan+Dino%2C+Seth+%C3%96berg+%5BGrupp+23%5D.pdf (D105869396)	1
SA	**thesis.pdf** Document thesis.pdf (D48195363)	2
W	URL: https://www.software-lab.org/publications/usenixSec2020-WebAssembly.pdf Fetched: 1/4/2021 10:23:48 AM	1
SA	**Examens_Arbete_Magnus_Medin_dv18mmn.pdf** Document Examens_Arbete_Magnus_Medin_dv18mmn.pdf (D108840609)	1
SA	**wbt_a_webassembly_benchmarking_tool_for_measuring_algorithm_performance.pdf** Document wbt_a_webassembly_benchmarking_tool_for_measuring_algorithm_performance.pdf (D107312600)	3
W	URL: https://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=1583&context=creativecomponents Fetched: 8/26/2020 11:52:33 PM	1
W	URL: https://hal.archives-ouvertes.fr/hal-02158925/document Fetched: 6/6/2020 4:07:03 PM	1
W	URL: https://www.lennard-golsch.de/article/webassembly.pdf Fetched: 2/15/2021 10:15:49 AM	2
SA	**MinhLe_HiepPhung_WebAssembly_thesisv2-20201030.docx** Document MinhLe_HiepPhung_WebAssembly_thesisv2-20201030.docx (D87333559)	1

Entire Document

An Evaluation on the Performance of Code Generated with WebAssembly Compilers Raymond Phelan A dissertation submitted in partial ful lment of the requirements of Dublin Institute of Technology for the degree of M.Sc. in Computing (TU060) Date: June 14, 2021

Declaration I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Stream), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work. This dissertation was prepared according to the regulations for postgraduate study of the Dublin Institute of Technology and has not been submitted in whole or part for an award in any other Institute or University. The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research. Signed: Date: June 14, 2021 I

Abstract WebAssembly is a new technology that is revolutionizing the web. Essentially it is a low-level binary instruction set that can be run on browsers, servers or stand-alone environments. Many programming languages either currently have, or are working on, compilers that will compile the language into WebAssembly. This means that applications written in languages like C++ or Rust can now be run on the web, directly in a browser or other environment. However, as we will highlight in this research, the quality of code generated by the di erent WebAssembly compilers varies and causes performance issues. This research paper aims to evaluate the code generated by a number of existing WebAssembly compilers in order to determine whether or not there is a signi cant di erence in their performances regarding execution times. Keywords: WebAssembly, WebAssembly Compilers, Benchmarking Performance, WASM, WAT, AssemblyScript, C/C++, Rust II

Acknowledgments I would like to thank my supervisor, Paul Kelly, for his on going support and advise throughout this dissertation. I also express my sincere gratitude to all the lecturers in TU Dublin that I have had the pleasure of attending their classrooms. I would also like to say a very special thank you to, my mother June Phelan, my father Raymond Patrick Phelan, my wife Diana Phelan and my children, Lucas Perico Phelan, Tawana Akinlolu and Tamiya June Phelan, for their constant encouragement and support. Finally, I would like to thank my classmates that I have had the pleasure of getting to know along this journey. III

List of Acronyms WASI WebAssembly System Interface WAT WebAssembly Text Format AS AssemblyScript VM Virtual Machine EVM Ethereum Virtual Machine CSV Comma Separated Values CPU Computer Processing Unit XI

Chapter 1 Introduction 1.1 Background WebAssembly, also known as WASM, is a new technology revolutionizing the web. It is a low-level binary instruction set, similar to pure assembly language, designed to run at near-native speeds otherwise only achieved with C/C++ code. It can be run on the client-side in web-browsers, on the server-side with NodeJS, and even as a stand-alone environment beyond the web, known as WASI (WebAssembly System Interface), (\bytecodealliance/wasmtime," 2021; \Standardizing WASI," n.d.; \WASI |," n.d.). WebAssembly was rst announced on December 5th, 2019, by the World Wide Web Consortium (W3C). For many years, JavaScript has been the only option for providing interactive applications in websites (Musch et al., 2019), although there have been many attempts to remedy this, for example Adobe Flash and Microsoft Active- X. However, these needed to be installed as browser extensions which prevented them from large scale acceptance. WebAssembly on the other hand, requires no extensions to be installed and is already supported by all of the major web-browsers. Programs written in statically typed languages, such as C, C++, and Rust can now be compiled into WebAssembly. As JavaScript is a dynamically typed language, it cannot be directly compiled into WebAssembly. TypeScript however, a subset of the JavaScript language, can be compiled into WebAssembly. Many other program- 1

CHAPTER 1. INTRODUCTION ming languages are currently working towards enabling compilation to WebAssembly (appcypher, 2021). This opens the door for many new possibilities as developers from these languages can now develop for the web, where previously this was not possible. As WebAssembly is essentially binary code, this cannot be easily understood by human readers. There is however a textual representation of WebAssembly, known as WAT (WebAssembly Text Format), (\Understanding WebAssembly text format - WebAssembly | MDN," n.d.). There are tools available for converting pure We- bAssembly binary code into WAT format and vice-versa, such as the WABT tool (\WebAssembly/wabt," 2021). C and C++ can be compiled to WebAssembly using the Emscripten compiler (\Main | Emscripten 2.0.21 documentation," n.d.) and Rust can be compiled using the wasm-pack compiler (\wasm-pack," n.d.). AssemblyScript (\AssemblyScript," n.d.) is a variant of TypeScript which can compile directly to WebAssembly. These compilers will be the focus of this research project. 1.2 Research Project/problem As more and more WebAssembly compilers emerge, we can expect to see a large variety of applications developed in multiple programming languages and compiled into WebAssembly. However, these compilers do not necessarily produce the same binary code for a given program written in di erent programming languages, as we will demonstrate later in this paper. In fact, even for a very simple program, such as a function to calculate the Fibonacci sequence, the di erent compilers are producing very di erent WebAssembly binary format code. These di erences can be visually veri ed by inspecting the WAT format of the We- bAssembly function. Since the WebAssembly compilers are producing vastly di erent code between them, even if they are performing the same function, this may lead to questions such as - which compiler generated the correct code? Do the compiled We- bAssembly modules all have the same performances, or do they vary in e ciency in terms of execution times and le sizes? Initially, one might have thought that once compiled into WebAssembly, there 2

CHAPTER 1. INTRODUCTION would be little or no di erence in the overall performance between the WebAssem- bly modules compiled with di erence compilers. However, if they have signi cantly di erent performances, even though they are all running as WebAssembly les, this means that it is important to understand which programming language and respective compiler developers should choose before developing WebAssembly applications. These concerns will be the purpose and focus of this research paper. We propose the following research question: Is there a signi cant di erence in the performance of WebAssembly mod- ules, in terms of execution times, for the same program written in di erent programming languages and compiled into WebAssembly using their respec- tive compilers? 1.3 Research Objectives The main goal of this research is to determine whether or not there is a signi cant di erence in the performance of WebAssembly modules generated from di erent com- pilers. As we will discuss later in the literature review, at the time of writing this paper, no other research was found that answered our proposed research question. In order to answer the question, we identi ed the following research objectives: • Set up a VM (Virtual Machine) environment for running our experiments • Identify programming languages that can be compiled into WebAssembly • Set up the respective WebAssembly compilers for each of the selected program- ming languages • Identify and code relevant functions and algorithms that can be written in the selected programming languages and that can be properly compiled into We- bAssembly, maintaining the same syntax as much as possible between the lan- guages, and considering the current limitations of WebAssembly and compilers 3

CHAPTER 1. INTRODUCTION • Create a benchmark runner for executing the WebAssembly modules from each of the selected programming languages and gather their execution times for post- analysis • Perform statistical analysis on the gathered execution times in order to answer the research question, by either accepting or rejecting the null or alternate hy- potheses. 1.4 Research Methodologies The type of research we will perform is considered a primary research, since the data that will be used in our experiments does not exist currently in the reviewed litera- ture. The data will be generated by executing the WebAssembly modules in a single environment and gathering their execution times. In terms of the research objective and whether it implies quantitative or qualita- tive research, we will be performing our statistical analysis based on numerical data representing the execution times of the WebAssembly modules. Therefore our research can be considered to be a quantitative research. The form of research we are conducting can be considered to be an empirical research, since we will be making direct observations on the data collected through our scienti c experiment. We will also be de ning the null and alternative hypotheses based on our research question. Finally, the reasoning involved in our research can be considered to be deductive reasoning, since we will be answering the research question by testing the hypotheses through a process of evaluating the statistical data obtained during our experiments. We will then draw conclusions to our research question based on the outcome of testing the null and alternative hypotheses. 4

CHAPTER 1. INTRODUCTION 1.5 Scope and Limitations The aim of this research paper is to compare the performance of WebAssembly mod- ules generated by di erent compilers for di erent programming languages. Our choice of programming languages was in uenced by the current available support for each language. We ultimately chose C, C++, Rust and AssemblyScript for our experi- ments as they provided enough documentation and support for implementing their respective compilers. Long et al. (2021) also came to the same conclusion regarding WebAssembly compilers with su cient support for di erent programming languages. There are of course other WebAssembly compilers currently available for compiling other programming languages into WebAssembly, however, they will remain out of scope for this research, but will be suggested as part of future work in this area of research in the concluding chapter. We also found limited support and resources for implementing benchmarking func- tions in the languages that we selected, considering that they all needed to maintain the same syntax, and take into account the current limitations of WebAssembly and the compilers. Therefore chose benchmarking functions that could be easily written in all of the selected programming languages. Another important aspect of this research paper worth highlighting is that we did not necessarily try to nd the most e cient implementation for each benchmarking function. Once we were able to replicate the same function in all the selected pro- gramming languages, then we would be able to properly compare their performance, regardless of whether the functions were the most e cient implementation or not. Fur- thermore, this research is limited to execution

Curiginal

of WebAssembly modules in a NodeJS environment, rather than the full spectrum of possible environments, such as web-browsers and IoT devices, which we will also suggest as part of future work in the concluding chapter. 5

CHAPTER 1. INTRODUCTION 1.6 Document Outline This research project will be divided into logical chapters. Chapter 1 provides some background knowledge on WebAssembly in general. Here we explain the research problem we have identi ed and present the research question. We also describe the research objectives and methodologies used. Finally we provide an insight into the scope and limitations of this research. Chapter 2 will involve a review of the current literature based on relevant scienti c publications in order to highlight the research gaps and importance of our research question. Chapter 3 describes the experiment design and set up used to gather the required data for our evalutation. Chapter 4 will provide an in-dept evaluation and discussion on the results gathered through our statistical tests. Finally, chapter 5 will provide a conclusion to this research, describing the impact of our contributions and outlining future work. 6

Chapter 2 Review of existing literature Although WebAssembly is still relatively new, it is rapidly becoming a highly re-searched topic in the last 18 months. Researchers have been evaluating the possible bene ts and use cases of WebAssembly, while developers have been enjoying freedom of choice in the programming languages they use to develop their applications, using a WebAssembly compiler to achieve the near native speeds promised by WebAssembly. In this section, we will review currently available literature that carried out some benchmarking on the performance of WebAssembly, or that proposed WebAssembly as solution to improve some existing applications, compared with the traditional ap- proaches. In doing so, we will reveal the gap in the literature that has led us to our research question, highlighting that not enough research and evaluation has been done on the performance of code generated by the various WebAssembly compilers. We highlight the research gap that choosing only one WebAssembly compiler to evaluate the performance of WebAssembly is insu cient. Sandhu et al. (2018) compared the performance of JavaScript and WebAssembly against C for sparse matrix-vector multiplication. They discover a slowdown of 2.2x to 5.8x with JavaScript versus C, but WebAssembly performing similar or better at times when compared with C in the browser. They used the Emscripten compiler to generate the WebAssembly modules from C code for their experiments. However, only the one WebAssembly compiler was used, therefore it is unknown whether WebAssembly code generated from other compilers might have produced di erent results. 7

CHAPTER 2. REVIEW OF EXISTING LITERATURE Sandhu et al. (2020) compared WebAssembly to native C code, again using sparse matrix computations. However, they developed the WebAssembly implementations by hand rather then using a compiler. This time they discover that WebAssembly executed in the Chrome browser has performance issues due to memory addressing in the x86 instruction set. Ultimately, this leaves a gap in the research, as perhaps it would be important to understand if various WebAssembly compilers might have produced more e cient WebAssembly code than the hand-written implementation. Jangda et al. (2019) use the PolybenchC Benchmark Suite (\PolyBench/C { Home-page of Louis-Noel Pouchet," n.d.) to compare WebAssembly-compiled Unix applica- tions versus native speeds inside the browser. Their ndings revealed that WebAssem- bly ran up to 55% slower. However, the WebAssembly modules that were used in their experiments were generated from the Emscripten compiler only, so once again we do not know if WebAssembly from other compilers might result in di erent performances. Haas et al. (2017) also compared the performance of WebAssembly using the Poly- benchC Benchmark Suite on browser JavaScript engines, such as Mozilla SpiderMon- key, Google V8 and Microsoft Chakra. The generation of WebAssembly modules was done with the assistance of the OCaml Programming Language (\OCaml - The core language," n.d.), rather than using any compiler. Once again, there was no evaluation of the WebAssembly code in comparison to what other compilers would generate by implementing di erent programming languages. Herrera et al. (2018) evaluate the performance of JavaScript and WebAssembly compared to native C code. They used ve of the benchmarks identi ed in the Ostrich Benchmark Suite (Khan et al., 2015) for numerical computing, and compiled them into WebAssembly modules using Emscripten. Their comparisons were done between web browsers, IoT devices and NodeJS. Their ndings once again show that WebAssembly comes close to native C performance, but is faster than JavaScript. However, these experiments also were limited by only evaluating the WebAssembly generated from one compiler, rather than multiple compilers. Reiser and Blaser (2017) presented their own WebAssembly compiler, Speedy.js, which compiles JavaScript/TypeScript to WebAssembly, although their compiler uses 8

CHAPTER 2. REVIEW OF EXISTING LITERATURE the Emscripten runtime library. They use a number of computing algorithms, such as Fibonacci, Prime Number and Merge Sort, to evaluate the performance of TypeScript compared to the WebAssembly implementation generated by their proposed solution. Indeed they nd WebAssembly performance to be faster than plain TypeScript, how- ever only the WebAssembly from their compiler was evaluated, rather than including WebAssembly generated by other compilers. Protzenko et al. (2019) also present their own compiler, which compiles Low*, a low-level subset of F* programming language, into WebAssembly. They develop an alternative to the JavaScript encryption library, LibSignal (\signalapp/libsignal- protocol-javascript," 2021), which is used in many modern services such as What- sApp, Skype and Signal for managing end-to-end encryption. They report no speed up between WebAssembly and the JavaScript library, mainly due to overhead be- tween encoding and decoding between WebAssembly and JavaScript. They suggest that future JavaScript libraries should be designed to be more WebAssembly friendly. However, only WebAssembly modules generated from their own compiler were used in this evaluation, therefore it is unknown if other WebAssembly compilers might have produced more e cient WebAssembly code. Watt et al. (2019) propose a similar solution to cryptography using WebAssembly, rather than JavaScript. However, they use a di erent approach, proposing CT-WASM as an extension to WebAssembly, or as a new programming language that is similar to TypeScript, which includes its own compiler for generating pure WebAssembly modules. However, the performance of their generated WebAssembly modules is not evaluated against WebAssembly modules generated from other compilers. Mendki (2020) proposed a WebAssembly Serverless Edge Computing as an alterna- tive to the native container-based application. They developed a compute and memory intensive application with le I/O and image classi cation using Rust and compiled it into WebAssembly. They found that WebAssembly had a lower footprint than the container-based solution while also showing faster start-up times, but the runtimes were slower than the native container application. However, only WebAssembly com- piled from Rust was evaluated in their proposal, therefore it is unknown how well 9

CHAPTER 2. REVIEW OF EXISTING LITERATURE WebAssembly compiled from other languages might have performed. A WebAssembly solution to the stateless containers used by existing serverless platforms was published in (Shillaker & Pietzuch, 2020), providing isolated memory and CPU (Computer Processing Unit), with state sharing capabilities. Their solution used the LLVM compiler (\The LLVM Compiler Infrastructure Project," n.d.) to translate applications into WebAssembly from multiple programming languages such as, C, C++, Python, JavaScript and TypeScript. Their evaluation reported a 2x performance speed-up with 10x less memory consumption using their proposal against dockerised containers. However, they also only used the one compiler for the generation of WebAssembly modules in their evaluations. Jeong et al. (2019) proposed an edge computing framework for o oading JavaScript and WebAssembly state between mobile devices, edge computing devices and cloud services. This system saves the state of JavaScript workers, objects and WebAssembly functions, and enables the transfer and restoration of these between devices, resulting in an 8.4x speedup compared with only o oading pure JavaScript code. All the We- bAssembly les used in their experiments were generated from C++ code using the Emscripten compiler. Once again, this highlights the gap of knowing how WebAssem- bly from other compilers would have performed in these experiments. WebAssembly was proposed in (Long et al., 2021) as a lightweight alternative for serverless Function as a Service environments, that could enable ne grained resource consumption at runtime, as WebAssembly functions can be started and stopped on demand. They concluded that WebAssembly VMs outperform Docker containers and have a smaller footprint. However, only the Emscripten WebAssembly compiler was used to generate the WebAssembly bytecode for their evaluations. Hall and Ramachandran (2019) also proposed WebAssembly as an alternative to traditional Dockerised containers for serverless functions in edge computing. They found that WebAssembly can provide many of the same isolation requirements, such as memory safety and security, while eliminating the cold start time penalty that traditional serverless containers incur. Once again, the WebAssembly modules used in their experiments were all generated from C++ code compiled into WebAssembly 10

CHAPTER 2. REVIEW OF EXISTING LITERATURE using the Emscripten compiler. Koren (2021) presented a proof-of-concept for a standalone WebAssembly devel- opment environment, capable of running on the edge and in the cloud. It features a built-in web-server and a browser-based Integrated Development Environment (IDE) as an alternative to running containerized microservices on low powered IoT devices with limited capabilities. Their evaluations were also done using WebAssembly mod- ules generated from a single compiler, this time, the AssemblyScript compiler. Tiwary et al. (2020) proposed an alternative to container-based serverless comput- ing, addressing their cold-start problems and complicated architectures, while provid- ing stateful memory and multi-tenancy isolation. Their proposal suggests WebAssem- bly to be executed directly on ring 0 with the serverless functions placed as close to the data as possible, providing CPU memory and lesystem isolation that is required for multi-tenancy in cloud computing. Their proposal was developed in Rust and compiled into WebAssembly using the Rust WebAssembly compiler, therefore once again, performance of WebAssembly is based on the code generated from a single WebAssembly compiler. (Zheng et al., 2020) evaluate the performance of Ethereum blockchain smart con- tracts using WebAssembly. They implement 12 benchmarks in Rust and compile to WebAssembly. They also use the emerging compiler SOLL (\second-state/SOLL," 2021) that generates WebAssembly bytecode from Solidity (Team, n.d.). Their exper- iments compare the performance of the WebAssembly driven blockchain against the EVM (Ethereum Virtual Machine) implementation, nding that WebAssembly driven blockchain does not perform as well as the EVM. However, they did not use the full range of available WebAssembly compilers by implementing their benchmarks in other programming languages, which perhaps might have produced di erent results for the evaluation of WebAssembly. FAUST is a domain speci c functional programming language used in audio pro- cessing and sound synthesis for applications such as synthesizers, musical instruments and sound e ects used in concerts, artistic productions, education and research. Letz et al. (2018) propose an alternative by compiling FAUST into WebAssembly and perform- 11

CHAPTER 2. REVIEW OF EXISTING LITERATURE ing benchmarks to compare the performance of WebAssembly with native versions. They found that WebAssembly ran up to 66% slower than native C++ versions. Inter- estingly, they question the code quality that will eventually be generated by multiple WebAssembly compilers. However, their experiments only involved the WebAssembly modules generated by the Emscripten compiler. Taheri (2018) discuss how computer vision on the web su ers from performance is- sues due to limitations in JavaScript. They developed a WebAssembly implementation of the OpenCV computer-vision library, which was initially implemented in C++. The library was compiled into WebAssembly using Emscripten and the performance was evaluated against the original implementation. Their ndings were that WebAssembly did indeed run at close to native speeds, however only one WebAssembly compiler was used to generate the WASM bytecode. This pattern of only using one WebAssembly compiler to benchmark the perfor- mance of WebAssembly code repeats itself through all the literature we reviewed. Murphy et al. (2020) only used the Clang compiler to evaluate WebAssembly for serverless computing. Cabrera Arteaga et al. (2020) proposed a toolchain for the su- peroptimization of WebAssembly binary code to improve execution time and overall le size. However, their experiments are limited to just C code using the Emscripten and LLVM compilers. In this section we discussed some of the available literature that evaluated the performance of WebAssembly generated from programming languages such as C, C++, Rust and AssemblyScript using their respective compilers. However, none of these evaluations compared the code generated from these compilers against each other. As we highlight in this paper, each WebAssembly compiler generates di erent code even for simple programs, which produce signi cant di erences in performance times. 12

Chapter 3 Experiment design and methodology The entire source code and results of our experiments are publicly available on a GitHub repository and can be located at: https://github.com/rayphelan/MastersProject2021 After installing the WebAssembly compilers for each of the selected languages and their required dependencies, this experiment can be fully reproduced by cloning the Git repository on a local VM and following the steps provided on the Github repository and in this paper. 3.0.1 Hypotheses As an example, if we write a simple Fibonacci algorithm in C/C++, AssemblyScript and Rust, maintaining identical syntax between the languages, will they all have the same performance when compiled into WebAssembly and run on the same environ- ment? In order to answer our research question using the scienti c methods mentioned in chapter 1, we propose the following hypotheses: 13

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY H0 - Null Hpyothesis: There is no signi cant di erence in the performance of WebAssembly mod- ules, in terms of execution times, for the same program written in di erent programming languages and compiled into WebAssembly using their respec- tive compilers. H1 - Alternate Hpyothesis: There is a signi cant di erence in the performance of WebAssembly mod- ules, in terms of execution times, for the same program written in di erent programming languages and compiled into WebAssembly using their respec- tive compilers. An important assumption of our experiments is that each of the algorithms across the selected programming languages is provided with the same input parameters and that the output of these algorithms is also the same for each language. This ensures that the results of the benchmarks are comparable, as highlighted in (Khan et al., 2015). For this reason, we have hardcoded the input parameters into each of the algorithms to ensure that each version of the algorithm in each language will have the same workload to operate on. Furthermore, the objective of our experiments is not to benchmark the printing capabilities of each compiled WebAssembly algorithm, therefore only one single output is printed at the end of each algorithm execution for the purpose of visually con rming that the algorithm executed correctly. 3.1 Experiment Set up In the following sections will provide details on the VM set up, Software Installa- tion, Programming Languages and Algorithms used to perform our experiments. We also provide details of how the benchmarks were executed and how the results were gathered. 14

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY 3.1.1 VM and Host Environment The VM was installed on top of a Windows desktop computer. The details of both are listed below. Table 3.1 show the details of the VM used in the experiments. Operating System Ubuntu 20.10 (64-bit) RAM 12075 MB Table 3.1: VM Table 3.2 show the details of the environment used for hosting the VM. Operating System Windows 10 Enterprise (64-bit) RAM 32 GB Processor Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz 2.30 GHz Table 3.2: Host Environment 3.1.2 Software and Compilers The following section provides details on the software that was required. This includes details on the programming languages and the WebAssembly compilers used in the experiments. Emscripten Emscripten (\Main | Emscripten 2.0.21 documentation," n.d.) is the compiler used to generate the WebAssembly les from C/C++ code. Installing Emscripten has a number of prerequisites, such as GIT (\Git," n.d.), NodeJS (Version 15.14.0) (Node.js, n.d.) and Python3 (\Download Python," n.d.) and the GCC Compiler for C and C++ (\GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)," n.d.). 15

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY Rust Rust was downloaded and installed from (\Rust Programming Language," n.d.) and the documentation for setting up the Rust to WebAssemby compiler, wasm-pack (\wasm-pack," n.d.), was found at (\Introduction - Rust and WebAssembly," n.d.). AssemblyScript Documentation for setting up the AssemblyScript WebAssembly compiler can be found at (\AssemblyScript," n.d.). RStudio RStudio (\RStudio | Open source & professional software for data science teams," n.d.) was used to perform the statistical analysis of our results. 3.2 Algorithms The algorithms used in our experiments were rst sourced on the internet as C/C++ programs. Then the equivalent of these algorithms were written in AssemblyScript and Rust programming languages. We created a directory for each of the selected programming languages. Inside each of these directories, we created a directory for each of the selected algorithms. The algorithms were coded in their respective folder for their respective programming language. The algorithms were then compiled to We- bAssembly using their respective compilers. The selection of algorithms used includes searching, sorting and numerical computing algorithms. In our experiments, some of the algorithms were given an array of 100,000 elements as their input. A JavaScript program was created which generated 3 versions of an array with 100,000 elements. Each element is a unique oating point number with 7 decimal places. The rst version is a sorted array with 100,000 elements. The second version is the same array in reserve order. The third version is the array with the elements in random positions. The respective array for each experiment was copied 16

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY directly into the algorithms as their input parameters and the same array was provided to all algorithms which required an array as the input parameter. The selection of algorithms used in this research were in uenced by other research papers that previously used these algorithms. The NQueens algorithm has been used in papers such as (Herrera et al., 2018) and (Khan et al., 2015). Recursive functions, such as the Fibonacci algorithm, are used in the following benchmarking paper (van Eekelen et al., 2019). Reiser and Blaser (2017) used a number of algorithms for their benchmarks, including the Prime Numbers, Merge Sort and Fibonacci algorithms. The Bubble Sort, Merge Sort, Fibonacci, NQueens, Prime Number were also used in previous versions of the Ostrich Benchmarking Suite (\Sable/Ostrich2," n.d.). 3.2.1 Numerical Computing Algorithms The numerical computing algorithms used in our experiments are NQueens, Primary Number and Fibonacci Number algorithms. NQueens The 8 Queens Puzzle was originally published in 1848 (\Eight queens puzzle," 2021). The objective is to place 8 Chess Queens on an 8x8 chess board in such a manor that no Queens are in direct threat from each other. Given the complexity of this puzzle when used with larger board sizes, it has been used many times as a software algorithm for benchmarking, known as the NQueens back-tracking algorithm. The NQueens algorithm is part the Ostrich Benchmark Suite (\McLab," n.d.) developed at the Sable Lab at McGill University. It has been a popular research topic in computer science research for many years (Alhassan, 2019; Ayub et al., 2018; Guldal et al., 2016). In our experiments, we implemented two versions of the NQueens algorithm. The rst using a board of 24x24, the second using a board of 27x27. The reason we chose the 24x24 size was that it provided enough complexity between all the selected programming languages in order to return a benchmarked time other than micro- seconds. The reason we did not go beyond the 27x27 size is because AssemblyScript 17

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY was already taking considerably longer than the other programming languages the larger the board size became. The algorithm was implemented in all of the selected programming languages keeping the same code structure as much as possible. The output of the algorithms was the printed solution of the board. The C/C++ version of this algorithm was sourced at (\C Program for N Queen Problem | Backtracking-3," 2011). The NQueens algorithm has a complexity of O(n n). Prime Number An important topic in computer science and cryptography is Number Theory, where the Prime Number calculation plays an important role (Elhakeem Abd Elnaby & El- Baz, 2021). A Prime Number is a number that has only two factors, 1 and the number itself. In this experiment, the Prime Numbers algorithm will calculate every prime number from 2 until 100,000. The output will print the 100,000th prime number. The C/C++ version of this algorithm was sourced at (\Write a Program to Find the nth Prime Number," n.d.) and the code for the other selected languages was written from scratch. The Prime Numbers algorithm has a complexity of O(p (n)). Fibonacci The Fibonacci sequence is another well known computer algorithm used for bench- marking and was recently used in (Mendki, 2020) for benchmarking WebAssembly performance. It takes a number as the input parameter, and adds the previous two numbers together to produce the next Fibonacci number sequence of the given in- put. For our experiment, we used the recursive method to calculate the Fibonacci sequence. The algorithm will return the 45th Fibonacci sequence number. The reason that we limited our experiment to calculating only the 45th number and not higher, is that it already provided enough complexity in the selected programming languages to perform our evaluations, with the execution times benchmarking between 5 and 10 seconds between the languages. The Fibonacci algorithm has a complexity of O(log n). 18

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY 3.2.2 Sorting Algorithms Sorting plays a crucial role in many algorithms (Abhay et al., 2019). The sorting algorithms used in this experiment are Bubble Sort, Heap Sort, Merge Sort, Selection Sort and Shell Sort. Each algorithm is given the same array of 100,000 elements as the input. Each element consists of a unique oating point number between 0 and 99,999 with 7 decimal places. The input array is sorted in reverse order, providing the algorithm with the worst case scenario, and the output of the algorithms will produce the same array in sorted order. Bubble Sort The Bubble Sort algorithm compares two adjacent elements and swaps them around if they are not in the required order. The C/C++ version of this algorithm was sourced at (\Bubble Sort (With Code)," n.d.) and the algorithms were written in the other languages from scratch. The Bubble Sort algorithm has a complexity of O(n2). Heap Sort The Heap Sort algorithm is generally slower than other sorting algorithms and there- fore not commonly used, however we wanted to include this ine cient sorting al- gorithm in our experiments as we believe it still provides interesting insights. The C/C++ version of this algorithm was sourced at (\Heap Sort (With Code)," n.d.) and the algorithms were written in the other languages from scratch. The Heap Sort algorithm has a complexity of O(n log n). Merge Sort The Merge Sort algorithm follows the principle of Divide and Conquer. It works by dividing the problem into multiple sub-problems, solving each of the smaller sub- problems, and merging the results back together to generate the output. The C/C++ versions of this algorithm was sourced at (\Merge Sort (With Code)," n.d.) and the algorithms were written in the other languages from scratch. The Merge Sort algorithm 19

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY has a complexity of O(n log n). Selection Sort The Selection Sort algorithm selects the smallest element from an array and places it at the beginning. It repeats this process until the results represent a sorted array. The C/C++ version of this algorithm was sourced at (\Selection Sort (With Code)," n.d.) and the algorithms were written in the other languages from scratch. The Selection Sort algorithm has a complexity of O(n2). Shell Sort The Shell Sort algorithm sorts elements at di erent intervals, starting from the ele- ments that are furthest apart from each other and repeating the process until all the elements are in a sorted order. The C/C++ version of this algorithm was sourced at (\Shell Sort (With Code)," n.d.) and the algorithms were written in the other languages from scratch. The Shell Sort algorithm has a complexity of O(n2). 3.2.3 Searching Algorithms The most commonly known searching algorithms are the Binary Search and Linear Search algorithms (Jacob et al., 2018), therefore we have implemented these algorithms in our experiments. The input for these algorithms consist of an array of 100,000 elements of unique oating point numbers between 0 and 99,999 with 7 decimal places. The algorithm will run a loop from 0 to 99,999 searching for the number of each iteration within the input array. The output will be the index of the array where the searched element was found on the last iteration. Binary Search The Binary Search algorithm assumes that the given array will in a sorted order. Given a sorted array, the algorithm divides the array in half by selecting the middle element of the array. It determines if the number being searched for is greater or less 20

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY than the value of the middle element. This means that one half of the array can be ignored as the number being searched for must be in the other half. This process is repeated recursively until the number being searched for is found. The C/C++ code for this algorithm was sourced at (\Binary Search (With Code)," n.d.) and the algorithms were written in the other languages from scratch. The Binary Search has a complexity of O(log n). Linear Search The Linear Search algorithm does not require a sorted array. In this experiment, the Linear Search algorithm was given an array of 100,000 elements of unique oating point numbers between 0 and 99,999 with 7 decimal places and the array has been given a random ordering. The algorithm performs a simple search on every element in the array, starting from the rst element and nishing when the number being searched for is found, which could go all the way to the last element. The C/C++ code for this algorithm was sourced at (\Linear Search (With Code)," n.d.) and the algorithms were written in the other languages from scratch. The Linear Search has a complexity of O(n). 3.3 Compiling to WebAssembly This section will provide details on how each WebAssembly compiler was used. Each of the compilers come with optimization level options. Some optimizations can be made which will reduce the overall code size. Other optimizations can be made which increase performance in execution time. In our experiments, we chose to optimize for execution time speed rather than code size. 3.3.1 C/C++ to WebAssembly Emscripten is used to compile C and C++ into WebAssembly. It takes in a C pro- gram as an argument and generates a WebAsssembly le and a JavaScript glue le. Emscripten allows for di erent levels of optimization, ranging from no optimization 21

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY to highly optimized. For our experiments, we chose the optimization level -O2 as this provided a combination of well optimized code with fast compile times. The optimization levels that Emscripten provides are listed in table 3.3. Optimization Level Description -O0 No optimization -O1 Low optimization, shorter compile time -O2 Well optimized build -O3 Highly optimized, longer compile time -Os Highly optimized, reduced code size, longer compile time Table 3.3: Emscripten Optimization Levels Emscripten also takes an arguement to modularize the code. This means that the compiler will generate the JavaScript glue le in such a way that allows it to be imported as a JavaScript module. The compiler puts all of the required JavaScript into a factory function which can be called on to create an instance of the WebAssembly module. The following code in listing 3.1 shows how the WebAssembly le and the JavaScript glue le were generated using Emscripten for the BinarySearch algorithm. Listing 3.1: Compiling BinarySearch from C to WebAssembly 1 emcc binarySearch . c -o binarySearch . j s -O2 2 -s MODULARIZE -s "EXPORTED FUNCTIONS=[' binarySearch '] " 3.3.2 AssemblyScript to WebAssembly AssemblyScript was designed speci cally to target WebAssembly while o ering a fa- miliar syntax for TypeScript and JavaScript developers. It uses Node Package Man- ager (NPM) (\npm," n.d.) as the installer. We created a new NPM application for each algorithm. Instructions for setting up new NPM applications are detailed on our Github repository. The AssemblyScript compiler also comes with options for opti- mization. For our experiments we used the optimization level 3, for speed rather than 22

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY code size, like we did with the Emscripten optimization levels. However, these opti- mization levels are actually applied by default, but can be altered in the con guration les of the compiler. The following code in listing 3.2 show how the WebAssembly and JavaScript les were generated using AssemblyScript. Listing 3.2: AssemblyScript Optimization 1 npm run asbuild 3.3.3 Rust to WebAssembly To compile Rust to WebAssembly, the wasm-pack compiler was used. Instructions for creating new wasm-pack applications can be found on our Github repository. We cre- ated a new wasm-pack application for each of our selected algorithms. Once inside the directory of the wasm-pack application created for each algorithm, the WebAssembly modules can be compiled. We passed an option to the compiler to note that we were targetting the NodeJS environment. This customizes the JavaScript glue le so it can be easily run in NodeJS. Wasm-pack allows us to specify whether or not the build should be optimized for pro- duction, using the {release keyword. However if no keyword is provided, the {release pro le is automatically used. This was the optimization levels used in our experi- ments. The following code in listing 3.3 shows how the WebAssembly and JavaScript les were generated for Rust. Listing 3.3: Rust to WebAssembly 1 wasm-pack build --target nodejs 3.4 Benchmarking and Gathering Results WebAssembly modules are most commonly loaded via its JavaScript API (Hall & Ra- machandran, 2019). The WebAssembly compilers used in our experiments generate the boilerplate JavaScript glue which can then be be converted into a JavaScript mod- ule, allowing it to be conveniently imported into other JavaScript functions. Inspired 23

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY by the Ostrich Benchmark Suite (\McLab," n.d.) which created a python runner script for executing the benchmarks and recording the results, we created our own JavaScript runner which runs our experiments in NodeJS with the desired number of iterations and records the nal results onto a CSV (Comma Separated Values) le for post processing. Each compiler generates the WASM le and the JavaScript glue for loading and interacting with the WASM le. For each algorithm in each language, we created a new JavaScript runner which loads the JavaScript Glue le. The JavaScript runner was manually invoked using NodeJS command line for each algorithm in each programming language. Once the WebAssembly module has been loaded, the JavaScript runner executes a loop which runs the main function in the corresponding WebAssembly module. The number of iterations in the loop depends on how many times we want to perform the experiment. After each iteration, the execution time in nanoseconds is measured and converted into seconds and milliseconds before being saved to an array of results. After the nal iteration, the array is saved into a CSV le. Note that the time for loading the WebAssembly module is not being measured, since we are only interested in the execution time. The following code in list3.4 represents the JavaScript runner for the Fibonacci algorithm in C. Listing 3.4: JavaScript runner for Fibonacci in C WebAssembly 1 const wasmModule = require ('./ fibonacci . js ') ; 2 const f s = require (' fs ') ; 3 const r e s u l t s = [ ] ; 4 const i t e r a t i o n s = 120; 5 wasmModule ( ) . then ((instance) =< f 6 for (n = 1; n >= i t e r a t i o n s ; n++) f 7 8 // Start Timer 9 const start = process . hrtime ( ) ; 10 11 // Run WASM 12 const wasm = instance . f i bonac c i ( ) ; 13 14 // End Timer 15 const d i f f = process . hrtime (start) ; 24

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY 16 17 const r e s u l t = (d i f f [0] * 1e9 + d i f f [1])/1000000000; 18

32%	MATCHING BLOCK 1/13	SA	Examensarbete+-+Aevan+Dino%2C+Seth+%C3%96berg+ ... (D105869396)

r e s u l t s . push (r e s u l t) ; 19 console . log (wasm) ; 20 g 21 const csv = r e s u l t s . join ('nn ') ; 22 23 // Write File 24 f s . writeFile (' r e s u l t s .

csv ' , csv , function (err) f 25 i f (err) return console . log (err) ; 26 console . log (' Filesaved ') ; 27 g) ; 28 g) ; Figure 3.1 shows a sequence diagram of the JavaScript Runner le we created. Figure 3.1: Experiment JavaScript Runner Herrera et al. (2018) suggested running each experiment 30 times, (Mendki, 2020) ran their experiments 100 times, while the

JetStream2 Benchmark (\JetStream 2 In- Depth Analysis," n.d.) suggested running the experiments 120 times. Initially we only ran our experiments 30 times each. However, upon visual inspection of the distribution 25

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY of the results using histograms, we found that our data was not normalised. We then proceeded to run the experiments 120 times each. However, even after running the experiments 120 times each, the results mainly did not have a normal distribution. This would in uence our choice in statistical tools during our evaluation of the results. When all of the experiments had been completed, we created Excel les for each language (C, C++, AssemblyScript, Rust). Each column in the spreadsheet represents the algorithm being used in the experiment. Each row represents the execution time of the algorithm for the number of iterations performed. The Excel les were then saved as CSV les for further processing in RStudio to perform the statistical analysis. Figure 3.2 shows an example of how the results were stored in Excel once the experiments have been nished. Figure 3.2: Gathering Results 3.5 Analysis of Results RStudio was used to perform the statistical analysis of the results. RStudio provides a convenient way of running statistical tests for normality and signi cance. It also provides functionality to plot graphs for visual interpretation of the results. Using the previously generated CSV les with the execution times of the algorithms in each language, these were imported into RStudio. The required RStudio libraries were also included. Figure 3.3 shows how the libraries were included and how the CSV les were imported. The entire source code used in RStudio is included in the Appendix section of this paper and on our Github repository. 26

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY Figure 3.3: Importing Data into RStudio 3.5.1 Testing for Normal Distribution It is important that we understand the distribution of data in our results, as this will determine how we test for signi cant di erences in the execution times, either with parametric tests for a normal distribution, or non-parametric tests for a non-normal distribution. We ran the Shapiro Wilk \Shapiro{Wilk test," 2021 test on each programming language result le for each algorithm. If the results from the test return a value of 0.05 or more, then it means our data has a normal distribution, otherwise it has a non-normal distribution. In RStudio, running this test is quite a simple process. The following listing 3.5 shows and example of the R commands used to perform the Shapiro Wilk test on each algorithm for the AssemblyScript results. This process was repeated for each of the other programming languages, C, C++ and Rust. Listing 3.5: Shapiro Wilk Test in RStudio 1 shapiro . t e s t (AssemblyScript$BinarySearch) 2 shapiro . t e s t (AssemblyScript$BubbleSort) 3 shapiro . t e s t (AssemblyScript$Fibonacci) 4 shapiro . t e s t (AssemblyScript$HeapSort) 5 shapiro . t e s t (AssemblyScript$LinearSearch) 6 shapiro . t e s t (AssemblyScript$MergeSort) 7 shapiro . t e s t (AssemblyScript$NQueen24) 8 shapiro . t e s t (AssemblyScript$NQueen27) 9 shapiro . t e s t (AssemblyScript$PrimeNumber) 10 shapiro . t e s t (AssemblyScript$ SelectionSort) 11 shapiro . t e s t (AssemblyScript$ ShellSort) The results of the Shapiro Wilk tests for normality showing the p-value con rms that our results were not normally distributed. The results can be seen in table 27

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY 3.4. For the purpose of facilitating the visualisation of the data, we will abbreviate AssemblyScript to AS. AS C C++ Rust BinarySearch 9.081e-08 2.251e-09 4.471e-08 0.4335 BubbleSort 0.000131 2.95e-07 5.32e-09 8.002e-07 Fibonacci 1.921e-06 1.394e-08 7.82e-09 2.501e-07 HeapSort 2.93e-06 5.966e-06 4.656e-06 3.685e-07 LinearSearch 1.97e-09 9.529e-06 8.256e-05 > 2.2e-16 MergeSort 1.878e-06 7.426e-06 2.564e-09 > 2.2e-16 NQueen24 0.0005585 0.0002786 > 2.2e-16 0.01424 NQueen27 0.0009603 1.927e-13 2.8e-11 8.382e-12 PrimeNumber 1.986e-06 8.051e-09 1.183e-12 2.399e-08 SelectionSort 1.995e-13 2.365e-06 1.451e-06 4.591e-08 ShellSort 1.713e-07 9.023e-12 1.832e-06 0.04334 Table 3.4: Shapiro Wilk Test for Normality The Shapiro Wilk test showed that our data was not normally distributed. The Central Limit Theorem \Central Limit Theorem," n.d. states that if we have su - ciently large data samples, then the distribution will be approximately normal and parametric tests can be used. However, if the sample size is not su ciently large and the distribution of data is not normal, then non-parametric tests can be used to determine the signi cant di erence between the samples. At this stage, we had to choose between running the experiments again a greater number of times until we achieved a normal distribution, so we could use parametric tests on our results, or keeping them as they are and using non-parametric tests on the results. Given the time constraints on completing this paper, we decided to use the results we had already obtained and apply non-parametric tests. As a visual aid, we also generated some graphs to con rm the tests. We created visual graphs in both RStudio and Tableau Public \Tableau Public," n.d. The follow- 28

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY ing gure 3.4 created in Tableau Public shows the distribution of the results of the Fibonacci algorithm in the selected languages. The entire collection of graphs can be found in the Appendix of this paper. Figure 3.4: Fibonacci Histograms The following graphs created in RStudio, shown in gure 3.5 represent the distri- bution of the results for each algorithm in AssemblyScript. The entire collection of these graphs can be found in the Appendix section of this paper. 29

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY Figure 3.5: AssemblyScript Distribtion (a) BinarySearch (b) BubbleSort (c) Fibonacci (d) HeapSort (e) LinearSearch (f) MergeSort (g) NQueen24 (h) NQueen27 (i) PrimaryNumber (j) SelectionSort (k) ShellSort 3.5.2 Testing for Signi cant Di erence In order to accept or reject the null hypothesis, we need to determine whether there is a signi cant di erence in the execution times of the algorithms when compared to 30

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY each other. We grouped the languages into pairs and ran a non-parametric test on them to obtain the p-value. Having a p-value of 0.05 or less means that we can reject the null hypothesis as there is signi cant di erence between the two samples. The non-parametric test we chose is the Mann-Whitney U Test, which is used to compare exactly two independent samples and the data is not normally distributed. We ran the test on each algorithm for AssemblyScript vs C, AssemblyScript vs C++, AssemblyScript vs Rust, C vs C++, C vs Rust and C++ vs Rust. The following code in listing 3.6 shows how the Mann-Whitney U Test was run on the BinarySearch algorithm for AssembyScript vs C, AssemblyScript vs C++ and AssemblyScript vs Rust. The entire RStudio code can be found in the Appendix of this paper and on our Github repository. Listing 3.6: Mann-Whitney U Test in RStudio 1 wilcox . t e s t (AssemblyScript$BinarySearch ,C$BinarySearch) 2 wilcox . t e s t (AssemblyScript$BinarySearch ,CPP$BinarySearch) 3 wilcox . t e s t (AssemblyScript$BinarySearch , Rust$BinarySearch) The values represented in following the results are the p-value. In order to facilitate the visualisataion of the results, we have divided the results into two tables. The rst table 3.5 shows the results of AssemblyScript vs C, AssemblyScript vs C++ and AssemblyScript vs Rust. The second table 3.6 shows the results of C vs C++, C vs Rust and C++ vs Rust. 31

CHAPTER 3. EXPERIMENT DESIGN AND METHODOLOGY AS vs C AS vs C++ AS vs Rust BinarySearch > 2.2e-16 > 2.2e-16 > 2.2e-16 BubbleSort > 2.2e-16 > 2.2e-16 > 2.2e-16 Fibonacci > 2.2e-16 > 2.2e-16 > 2.2e-16 HeapSort > 2.2e-16 > 2.2e-16 > 2.2e-16 LinearSearch > 2.2e-16 > 2.2e-16 > 2.2e-16 MergeSort > 2.2e-16 > 2.2e-16 > 2.2e-16 NQueen24 > 2.2e-16 > 2.2e-16 > 2.2e-16 NQueen27 > 2.2e-16 > 2.2e-16 > 2.2e-16 PrimeNumber > 2.2e-16 > 2.2e-16 > 2.2e-16 SelectionSort > 2.2e-16 > 2.2e-16 > 2.2e-16 ShellSort > 2.2e-16 > 2.2e-16 8.265e-06 Table 3.5: Mann-Whitney U Test C vs C++ C vs Rust C++ vs Rust BinarySearch 0.1421 2.427e-09 7.219e-12 BubbleSort 3.225e-13 > 2.2e-16 > 2.2e-16 Fibonacci 0.07379 > 2.2e-16 4.606e-16 HeapSort 0.6791 1.119e-13 2.901e-14 LinearSearch 0.001119 > 2.2e-16 > 2.2e-16 MergeSort 0.6215 > 2.2e-16 > 2.2e-16 NQueen24 > 2.2e-16 0.01026 > 2.2e-16 NQueen27 > 2.2e-16 > 2.2e-16 > 2.2e-16 PrimeNumber 2.354e-06 > 2.2e-16 > 2.2e-16 SelectionSort 0.001779 > 2.2e-16 > 2.2e-16 ShellSort 0.4074 > 2.2e-16 > 2.2e-16 Table 3.6: Mann-Whitney U Test 32

Chapter 4 Results, evaluation and discussion The main objective of our experiments was to answer the research question by either accepting or rejecting the null hypothesis. Research Question: Is there a signi cant di erence in the performance of WebAssembly mod- ules, in terms of execution times, for the same program written in di erent programming languages and compiled into WebAssembly using their respec- tive compilers? H0 - Null Hpyothesis: There is no signi cant di erence in the performance of WebAssembly mod- ules, in terms of execution times, for the same program written in di erent programming languages and compiled into WebAssembly using their respec- tive compilers. H1 - Alternate Hpyothesis: There is a signi cant di erence in the performance of WebAssembly mod- ules, in terms of execution times, for the same program written in di erent programming languages and compiled into WebAssembly using their respec- tive compilers. 33

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION 4.1 Statistical Tests We ran statistical tests on our paired data groups to obtain the p-values, which de- termines whether or not there is a signi cant di erence between grouped pairs. A p-value of 0.05 or less indicates that there is indeed a signi cant di erence, therefore we must reject the null hypothesis and accept the alternate hypothesis. This can be summarized by the following formula: p 0:05 = H1 On the other hand, having a p-value of higher than 0.05 indicates that there is no signi cant di erence between the pair, and we must accept the null hypothesis and reject the alternate hypothesis. This can be summarized by the following formula: p<0:05 = H0 The statistical tests clearly showed signi cant di erences in almost all of the exper- iments. One exception was while comparing C to C++, which on some experiments had a p-value greater than 0.05. However, this is not surprising, since both C and C++ are compiled to WebAssembly using the same compiler and the code is almost identical. Figure 4.1 highlights the results where the p-values were greater than 0.05, meaning there were no signi cant di erence in those execution times. The results also show that for all other language pairs, there were signi cant di erences in the execution times. Figure 4.1: P-Values 34

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION 4.2 Visual Representations In this section, we will provide a detailed evaluation of the results for each algorithm. In order to provide a visual representation of the execution times, Tableau Public was used to produce line-charts of the execution time of each language compiled into WebAssembly for each algorithm. The visualizations clearly show that there is a signi cant di erence in execution time between the algorithms. For visual inspection of the WebAssembly code, we used the Wabt toolkit (\WebAssembly/wabt," 2021) to transform the WASM les into the human readable WAT format. 4.2.1 Numerical Computing Algorithms The Numerical Computing algorithms used in our experiments are pure mathematical functions and did not require the input array of 100,000 elements. Fibonacci The gure 4.2 shows the execution times for the Fibonacci algorithm. C and C++ are consistently running with similar execution times. If fact the p-values for the Fibonacci algorithm for C and C++ showed that there was no signi cant di erence in their execution times. However, Rust appears to run faster than C and C++, while it can be visually observed that AssemblyScript performed the slowest of all the other languages. 35

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION Figure 4.2: Fibonacci In all of our selected programming languages, the code structure for the Fibonacci sequence have the same syntax. This can be con rmed in the following listings. As mentioned previously, we implemented the Fibonacci algorithm using the recursive method. The algorithms will produce an output with the 45th Fibonacci sequence number. The following listing 4.1 shows the Fibonacci algorithm in AssemblyScript. Listing 4.1: Fibonacci in AssemblyScript 1 function f i b (n : i32) : i32 f 2 return n >= 1 ? 1 : f i b (n-1) + f i b (n-2); 3 g 4 export function f ib on ac ci ( ) : i32 f 5 const n : i32 = 45; 6 const r e s u l t : i32 = f i b (n) ; 7 return r e s u l t ; 8 g The following listing 4.2 shows the Fibonacci algorithm in C. Listing 4.2: Fibonacci in C 1 int f i b (n) f 36

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION 2 i f (n >= 1) return 1; 3 return f i b (n - 1) + f i b (n - 2) ; 4 g 5 int f i b o n a c ci () f 6 int n = 45; 7 int r e s u l t = f i b (n) ; 8 return r e s u l t ; 9 g The following listing 4.3 shows the Fibonacci algorithm in C++. Listing 4.3: Fibonacci in C++ 1 extern "C" f 2 int f i b (int n) f 3 i f (n >= 1) return 1; 4 return f i b (n - 1) + f i b (n - 2) ; 5 g 6 int f i b o n a c ci () f 7 int n = 45; 8 int r e s u l t = f i b (n) ; 9 return r e s u l t ; 10 g 11 g The following listing 4.4 shows the Fibonacci algorithm in Rust. One important di erence with Rust is that we need to include the a library so it can interact with JavaScript. Listing 4.4: Fibonacci in Rust 1 use wasm bindgen : : prelude : : * ; 2 #[wasm bindgen] 3 extern "C" f 4 #[wasm bindgen (js namespace = console)] 5 fn log (s : u64) ; 6 g 7 fn fi b ona c ci (n : i32) -< u64 f 8 i f n >= 1 f return 1 g 9 return fi bo na cc i (n-1) + f i bo na cc i (n-2); 10 g 11 #[wasm bindgen (s t a r t)] 12 pub fn main js () f 13 log (fi bo na cc i (4 5)) ; 14 g 37

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION Given that the code syntax for the Fibonacci algorithm in each of the selected languages is as consistent as possible by implementing the algorithm using the recur- sive method, an interesting observation is that they produced considerably di erent WebAssembly binaries when we performed the visual inspection of the WAT les. The following listing 4.5 is the WAT le representation of the compiled Fibonacci algorithm in AssemblyScript. Listing 4.5: Fibonacci WAT format for AssemblyScript 1 (module 2 (type $none =< i32 (func (r e s u l t i32))) 3 (type $i32 =< i32 (func (param i32) (r e s u l t i32))) 4 (memory $0 0) 5 (export " f ib on ac ci " (func $assembly/ index / f ib on ac ci)) 6 (export "memory" (memory $0)) 7 (func $assembly/ index / f i b (param $0 i32) (r e s u l t i32) 8 l o c a l . get $0 9 i32 . const 1 10 i32 . l e s 11 i f (r e s u l t i32) 12 i32 . const 1 13 e l s e 14 l o c a l . get $0 15 i32 . const 1 16 i32 . sub 17 c a l l $assembly/ index / f i b 18 l o c a l . get $0 19 i32 . const 2 20 i32 . sub 21 c a l l $assembly/ index / f i b 22 i32 . add 23 end 24) 25 (func $assembly/ index / f i b o n a c c i (r e s u l t i32) 26 i32 . const 45 27 c a l l $assembly/ index / f i b 28) 29) The following code in listing 4.6 is the WAT le representation of the compiled Fibonacci algorithm in C and C++. The WebAssembly le that was generated for C++ was identical to the one generated for C, again this was to be expected as they 38

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION are both compiled using the same compiler. However, an interesting observation here is that there is considerably more code in the C and C++ version of the compiled WebAssembly when compared to AssemblyScript for the exact same algorithm. Listing 4.6: Fibonacci WAT format for C and C++ 1 (module 2 (type (; 0 ;) (func (r e s u l t i32))) 3 (type (; 1 ;) (func (param i32) (r e s u l t i32))) 4 (type (; 2 ;) (func)) 5 (type (; 3 ;) (func (param i32))) 6 (func (; 0 ;) (type 2) 7 nop) 8 (func (; 1 ;) (type 1) (param i32) (r e s u l t i32) 9 (l o c a l i32 i32) 10 i32 . const 1 11 l o c a l . set 1 12 l o c a l . get 0 13 i32 . const 2 14 i32 . ge s 15 i f (r e s u l t i32) ; ; l a b e l = @1 16 i32 . const 0 17 l o c a l . set 1 18 loop ; ; l a b e l = @2 19 l o c a l . get 0 20 i32 . const 1 21 i32 . sub 22 c a l l 1 23 l o c a l . get 1 24 i32 . add 25 l o c a l . set 1 26 l o c a l . get 0 27 i32 . const 3 28 i32 . gt s 29 l o c a l . set 2 30 l o c a l . get 0 31 i32 . const 2 32 i32 . sub 33 l o c a l . set 0 34 l o c a l . get 2 35 b r i f 0 (; @2 ;) 36 end 37 l o c a l . get 1 38 i32 . const 1 39 i32 . add 39

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION 40 e l s e 41 l o c a l . get 1 42 end) 43 (func (; 2 ;) (type 0) (r e s u l t i32) 44 i32 . const 45 45 c a l l 1) 46 (func (; 3 ;) (type 0) (r e s u l t i32) 47 global . get 0) 48 (func (; 4 ;) (type 3) (param i32) 49 l o c a l . get 0 50 global . set 0) 51 (func (; 5 ;) (type 1) (param i32) (r e s u l t i32) 52 global . get 0 53 l o c a l . get 0 54 i32 . sub 55 i32 . const -16 56 i32 . and 57 l o c a l . tee 0 58 global . set 0 59 l o c a l . get 0) 60 (func (; 6 ;) (type 0) (r e s u l t i32) 61 i32 . const 1024) 62 (table (; 0 ;) 1 1 fu ncre f) 63 (memory (; 0 ;) 256 256) 64 (global (; 0 ;) (mut i32) (i32 . const 5243920)) 65 (export "memory" (memory 0)) 66 (export " wasm call ctors " (func 0)) 67 (export " f i b o n a c c i " (func 2)) 68 (export " e r r n o l o c a t i o n " (func 6)) 69 (export " stackSave " (func 3)) 70 (export " stackRestore " (func 4)) 71 (export " stackAlloc " (func 5)) 72 (export " i n d i r e c t f u n c t i o n t a b l e " (table 0))) The following listing 4.7 is the WAT le representation of the compiled Fibonacci algorithm in Rust. Listing 4.7: Fibonacci WAT format for Rust 1 (module 2 (type (; 0 ;) (func)) 3 (type (; 1 ;) (func (param i32 i32))) 4 (type (; 2 ;) (func (param i32 i32 i32))) 5 (type (; 3 ;) (func (param i32) (r e s u l t i32))) 6 (type (; 4 ;) (func (param i64) (r e s u l t i64))) 7 (import " wbindgen placeholder " " wbg log e2b7116aabd69db1 " (func (; 0 ;) (type 1))) 40

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION 8 (func (; 1 ;) (type 4) (param i64) (r e s u l t i64) 9 (l o c a l i64) 10 i64 . const 1 11 l o c a l . set 1 12 l o c a l . get 0 13 i64 . const 2 14 i64 . ge u 15 i f (r e s u l t i64) ; ; l a b e l = @1 16 i64 . const 0 17 l o c a l . set 1 18 loop ; ; l a b e l = @2 19 l o c a l . get 0 20 i64 . const -1 21 i64 . add 22 c a l l 1 23 l o c a l . get 1 24 i64 . add 25 l o c a l . set 1 26 l o c a l . get 0 27 i64 . const -2 28 i64 . add 29 l o c a l . tee 0 30 i64 . const 1 31 i64 . gt u 32 b r i f 0 (; @2 ;) 33 end 34 l o c a l . get 1 35 i64 . const 1 36 i64 . add 37 e l s e 38 l o c a l . get 1 39 end) 40 (func (; 2 ;) (type 2) (param i32 i32 i32) 41 (l o c a l i64) 42 l o c a l . get 0 43 l o c a l . get 1 44 i64 . extend i32 u 45 l o c a l . get 2 46 i64 . extend i32 u 47 i64 . const 32 48 i64 . shl 49 i64 . or 50 c a l l 1 51 l o c a l . tee 3 52 i64 . store32 41

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION 53 l o c a l . get 0 54 l o c a l . get 3 55 i64 . const 32 56 i64 . shr u 57 i64 . store32 o f f s e t =4) 58 (func (; 3 ;) (type 0) 59 (l o c a l i64) 60 i64 . const 40 61 c a l l 1 62 l o c a l . tee 0 63 i32 . wrap i64 64 l o c a l . get 0 65 i64 . const 32 66 i64 . shr u 67 i32 . wrap i64 68 c a l l 0) 69 (func (; 4 ;) (type 3) (param i32) (r e s u l t i32) 70 l o c a l . get 0 71 global . get 0 72 i32 . add 73 global . set 0 74 global . get 0) 75 (memory (; 0 ;) 17) 76 (global (; 0 ;) (mut i32) (i32 . const 1048576)) 77 (export "memory" (memory 0)) 78 (export " f i b o n a c c i " (func 2)) 79 (export " main js " (func 3)) 80 (export " wbindgen add to stack pointer " (func 4)) 81 (export " wbindgen start " (func 3))) Based on our experiment and analysis of the Fibonacci algorithm, we can clearly see a signi cant di erence in the performance of the WebAssembly modules from di erent programming languages, with the exception of C and C++ that produced identical WebAssembly les and had a p-value greater than 0.05. We can also see a clear di erence of the WebAssembly les when converted into WAT format. In order to not unnecessarily bloat the contents of this paper, the source code and WAT les for the remaining algorithms will not be included directly in the paper. However, they may be viewed on our Github repository. 42

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION PrimeNumber The visualization in gure 4.3 shows that C and C++ perform with similar execution times, although their p-values were less than 0.05. We can also see that Rust performs slightly slower than C and C++. Both however generally performed the PrimeNumber algorithm in less than 1 second every time. On the other hand, we can clearly see that AssemblyScript took longer than all the other languages, taking between 4 to 5 seconds to complete. Figure 4.3: PrimeNumber We con rmed that the syntax for the PrimaryNumber algorithm is the same for each of the selected languages. However, the WebAssembly les generated were con- siderably di erent. We observed that the AssemblyScript produced 279 lines of code when the WAT extension le was inspected, the C and C++ versions produced only 36 lines of code, while the Rust version produced 7005 lines of code. 43

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION NQueen24 and NQueen27 The source code for the NQueens algorithms for each of the selected languages were kept syntactically the same as much as possible in our experiments. Upon inspection of the WAT les for each of the selected languages, we observed that AssemblyScript generated 3090 lines, C and C++ generated 3218 lines, while Rust generated 7622 lines. The execution times of the NQueen24 and NQueen27 algorithms can be viewed in gure 4.4 and gure 4.5. Figure 4.4: NQueen24 44

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION Figure 4.5: NQueen27 From these visualizations we can see that C, C++ and Rust performed with similar execution times, generally with less than 1 second every time. However, there was a clear di erence in the execution times for AssemblyScript, which took between 3 and 5 seconds per execution. 4.2.2 Searching Algorithms The source code for the Searching algorithms were kept syntactically the same as much as possible. An array of 100,000 elements was hard-coded into each of the algorithms. This caused the WebAssembly compilers generated huge WASM and WAT les for each language. 45

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION BinarySearch For the BinarySearch algorithm, all languages had visually similar execution times, but it is still clear that there are di erences between their performances. The statistical tests for C and C++ returned a value greater than 0.05, meaning that there was no signi cant di erence between their execution times. However, Rust appears to have the fastest execution times over C and C++, while AssemblyScript has once again the slowest execution times. There were also signi cant di erences in the inspection of the WAT les for the BinarySearch algorithm when converted into WebAssembly. The execution times for the BinarySearch algorithm can be view in gure 4.6. Figure 4.6: BinarySearch 46

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION LinearSearch The LinearSearch algorithm statistical tests showed that all languages had signi cant di erences in their execution times, although visually, C and C++ appear to have similar performances. Once again Rust had the fastest execution times, generally running at around 4 seconds each time. C and C++ executed at about 5 to 6 seconds each time, while AssemblyScript had the slowest execution times, performing between 12 and 16 seconds for each iteration. The gure 4.7 shows the execution times for the LinearSearch algorithm in the selected languages. Figure 4.7: LinearSearch 4.2.3 Sorting Algorithms From the following visualizations for the sorting algorithms, it can be determined that C and C++ consistently had the fastest execution times over the other languages, while Rust and AssemblyScript had a mixed result, with sometimes one performing better than the other. 47

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION HeapSort According to the statistical tests we ran on the HeapSort algorithm, C and C++ had a p-value of greater than 0.05, meaning that there was no signi cant di erence between their execution times. This can be visually veri ed in gure 4.8. An interesting obser- vation is that Rust appears to start slowly and then speed up during the experiment. This experiment was repeated many times and the same behavior from Rust was al- ways observed. The results show that C and C++ had the fastest times, followed by Rust, and AssemblyScript had the slowest times. Figure 4.8: HeapSort 48

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION MergeSort The statistical tests performed on the MergeSort algorithm also returned a p-value greater than 0.05 for C and C++, meaning there was no signi cant di erence between their execution times. This is visible on gure 4.9. Here we can see that once again Rust starts o slower than begins to speed up. Another interesting observation is that AssemblyScript actually runs faster than Rust for this algorithm, while C and C++ have once again the fastest execution times. Figure 4.9: MergeSort 49

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION ShellSort The p-values for the ShellSort algorithm for C and C++ again came back with values greater than 0.05, meaning that there was no signi cant di erence between their ex- ecution times. This can be visually con rmed on gure 4.10. It can also be observed that Rust once again starts o slower, then speeds up towards the end of the execu- tion. C and C++ have the fastest execution times, while Rust and AssemblyScript have visually similar execution times. Figure 4.10: ShellSort 50

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION SelectionSort The SelectionSort visualization shows that C and C++ have visually similar perfor- mances, however the statistical tests returned a p-value of less than 0.05. From this visualization we can see that C and C++ generally took around 4 seconds to execute, while Rust took around 8 seconds. The performance of AssemblyScript however was much slower, taking between 30 and 40 seconds for each iteration. Figure 4.11: SelectionSort 51

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION BubbleSort The BubbleSort algorithm revealed great di erences between the performance of As- semblyScript and the other languages. While C and C++ performed slightly faster than Rust, with execution times of around 7 to 8 seconds, and Rust with execution times of around 9 to 10 seconds, AssemblyScript had taken between 70 and 80 seconds to perform each iteration. Figure 4.12: BubbleSort 4.3 Discussion WebAssembly is generally considered to be still in its infancy stages, and many fea- tures are still being developed. This imposes some restrictions on what compilers can actually compile into WebAssembly at this time. Overall, the performance of AssemblyScript was considerably slower than Rust, C and C++. The performance 52

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION times of C and C++ were mostly faster than all of the other languages. However, there were times that Rust performed faster than C/C++ and there was one occasion where AssemblyScript performed faster than Rust. Some of the statistical test results showed that there were signi cant di erence between C and C++, however we believe that given a much higher number of iterations in each experiment, that the statisti- cal tests would eventually show no signi cant di erence between all of the algorithms implemented in C and C++. Although not within the scope of this research, there was another interesting ob- servation made regarding the compilers. The C, C++ and AssemblyScript modules usually compiled with reasonable speeds, normally ranging between 1 and 5 seconds for each algorithm. However, for the algorithms with the arrays of 100,000 elements, the Rust compiler would take around 30 minutes to compile. The research work done in this paper highlights an important aspect of WebAssem- bly compilers, in that they do in fact su er from signi cantly di erent performance times, even if the source code is syntactically identical in the various di erent pro- gramming languages. The idea behind WebAssembly is to allow multiple languages, such as C and Rust, to be used on the web. However, since the compilers will ul- timately produce di erent WebAssembly les with varying performances, developers should give careful consideration into which language and compiler they use to gener- ate WebAssembly modules. Given more time to complete this research paper, we would have included more WebAssembly compilers in our experiments and evaluations. Also, the number of iterations for each algorithm would ideally be increased in order to produce a normal distribution of data and to use parametric tests to

Curiginal

either accept of reject the hypothesis. Another aspect that could be improved is the choice of algorithms for each language. We believe that the Ostrich Benchmark Suite would be a suitable candidate. Ideally these benchmarks would be written in a variety of programming languages, such as Haskell and Prolog, covering a variety of programming paradigms such as procedural, object-oriented, function and logical. We believe the analysis of the same algorithms written across a large spectrum of programming languages and paradigms converted 53

CHAPTER 4. RESULTS, EVALUATION AND DISCUSSION into WebAssembly would provide interesting and useful insights. One nal observation that we will make in our evaluation is the comparison of the total time in milliseconds that each language took to execute each of the algorithms for every iteration. This resulted in C++ being the fastest overall, followed C and Rust, with AssemblyScript being the slowest overall. Also worth mentioning is that the Em- scripten compiler is actually capable of achieving even higher optimization levels than the level -O2 that we used in our experiments, at the cost of longer compile times. However, even so, C/C++ compiled into WebAssembly with Emscripten provided the best overall performance against the other languages and compilers we evaluated. This reveals that currently, although WebAssembly promises speeds of near native performance otherwise only achieved with C/C++, that WebAssembly compiled from C/C++ still provide the best performances over other languages compiled into We- bAssembly. This can be viewed in gure 4.13. Figure 4.13: Overall Performance in milliseconds 54

Chapter 5 Conclusion 5.1 Research Overview WebAssembly is becoming more and more popular in research and web-development as it promises improved performance to compute-intensive operations for web-based applications. In this paper we critically evaluate the current literature that bench- marks WebAssembly performance compared with traditional implementations. In our opinion, there is no doubt that WebAssembly o ers an exciting and e cient alterna- tive to many of the limitations found in web-based and cloud-based applications of today. However, we raise the concern that the performance of WebAssembly is greatly impacted by the choice of compiler used, meaning that certain programming lan- guages will still have signi cant di erences in their performance when compiled to WebAssembly. At the time of writing this paper, we did not nd any other research paper that compares the code generated by WebAssembly compilers and evaluates their performance against each other. We also highlight that many research papers have proposed WebAssembly alter- natives to native applications, performing benchmarks to evaluate the performance of WebAssembly. However, these evaluations only ever used one compiler for the genera- tion of WebAssembly. Based on the results of our experiments, we show that di erent WebAssembly compilers produce di erent results in terms of performance. There- 55

CHAPTER 5. CONCLUSION fore we believe that evaluating the performance of WebAssembly generated from only one compiler is not su cient when performing evaluations between WebAssembly and native speeds. 5.2 Problem De nition The objective of this research was to evaluate the performance of WebAssembly code generated by di erent compilers, highlighting a potential issue with the performance of the generated code. For example, if we write even a simple loop in C, C++, Rust and AssemblyScript, one might initially assume that the WebAssembly code produced by the compilers would be similar and would run without signi cant di erences in performance. However, we have highlighted that this is not the case, demonstrating that each of the compilers produced signi cantly di erent results in terms of execution speed. 5.3 Design/Experimentation, Evaluation & Results For our experiments, we used a selection of 11 di erent programs written in C, C++, Rust and AssemblyScript. These programs ranged from simple Fibonacci, Prime Num- ber, Searching and Sorting functions to more demanding numerical computing func- tions such as the N-Queens problem. We compiled each program into WebAssembly using the respective compiler for each programming language. We used NodeJS to run the WebAssembly compiled functions and measure their execution times. Each function was run a total of 120 times. Upon inspection of the data produced by these experiments, we determined that the data did not have a normal distribution. This in turn in uenced the choice of statistical test that we had to use to determine whether or not there were signi cant di erences in the performance of the compiled WebAssembly modules. Our results showed that in fact there are signi cant di erences in the performance of WebAssembly modules generated by the di erent compilers, not only in terms of 56

CHAPTER 5. CONCLUSION execution times, but upon visual inspection of the human-readable WAT format of the WebAssembly code, we can clearly see signi cant di erences even for simple programs compiled to WebAssembly. We also revealed that, although WebAssembly promises performance of near native speed otherwise only achieved with C/C++, currently WebAssembly compiled from C/C++ still o er the best performance over the other programming languages used in our research. 5.4 Contributions and impact In this research we have identi ed a potential problem with WebAssembly compilers, in that they produce di erent byte code for even simple programs, and have signi cant di erences in their performance, depending on which compiler was used to generate the WebAssembly module. We show that even though WebAssembly promises per- formances of near native speeds otherwise only obtainable with C/C++ code, that when C/C++ code is compiled into WebAssembly, it still out-performes all the other languages in our experiments. We believe that this provides valuable insights for de- velopers as they choose which programming language to develop in and subsequently compile into WebAssembly. We also provided a critical evaluation of the literature currently available for We- bAssembly benchmarking and for WebAssembly solutions being proposed as alterna- tives to existing approaches in applications. We revealed a gap in the current literature and highlighted that for evaluating the performance of WebAssembly, using only one compiler is not su cient, as di erent WebAssembly compilers may produce di er- ent results, which in turn may determine the success or failure of the WebAssembly evaluations currently being researched. 57

CHAPTER 5. CONCLUSION 5.5 Future Work & recommendations In chapter 1 of this paper we identi ed the scope and limitations of our research. These limitations create opportunities for future work in this area. We recommend that more research be done on the comparison of code generated by WebAssembly compilers to identify the di erences between their produced code and determine the reasons why these di erences exist. Another area of interest could be to compare the le size of the compiled WebAssembly modules from di erent compilers while compiling the same programming logic. The list of programming languages that currently compile to WebAssembly can be found at (appcypher, 2021). Future research should be done to evaluate the per- formance of all available WebAssembly compilers. Furthermore, as more and more compilers emerge, programming languages with di erent programming paradigms will be able to compile to WebAssembly, for example Haskell, a functional programming language and ProLog, a logic programming language. We believe that analysing the WebAssembly byte code and performance of functional, logical, procedural and object-oriented programming languages will be of interest to researchers and application de- velopers. The Ostrich Benchmark Suite was published in (Khan et al., 2015). They identi- ed important patterns for numerical computation and ran experiments to compare JavaScript performance against native C code for sequential and parallel operations. Therefore another possibility for future work would be to implement the numerical computing benchmarks identi ed in the Ostrich Benchmark Suite in multiple pro- gramming languages, including languages from di erent programming paradigms such as procedural, functional, logical and object-oriented, and evaluating the performance of these benchmarks in WebAssembly from multiple compilers. Finally, since our experiments were only executed on a NodeJS environment, future work might also involve running such experiments on all web-browsers and IoT devices. 58

Bibliography Abhay, G., Abhishek, S., & Namita, G. (2019). A variant of Bucket Sort Shell Sort vs Insertion Sort. 2019 10th International Conference on Computing, Communi- cation and Networking Technologies, ICCCNT 2019, 6{10. https://doi.org/ 10.1109/ICCCNT45670.2019.8944607 Alhassan, A. (2019). Build and conquer: Solving N queens problem using iterative compression. Proceedings of the International Conference on Computer, Con- trol, Electrical, and Electronics Engineering 2019, ICCCEEE 2019. https: //doi.org/10.1109/ICCCEEE46830.2019.9070976 appcypher. (2021, June 6). Appcypher/awesome-wasm-langs [original-date: 2017-12- 15T11:24:02Z]. Retrieved June 7, 2021, from https://github.com/appcypher/ awesome-wasm-langs AssemblyScript. (n.d.). Retrieved May 24, 2021, from https://www.assemblyscript .org/ Ayub, M. A., Kalpoma, K. A., Proma, H. T., Kabir, S. M., & Chowdhury, R. I. H. (2018). Exhaustive study of essential constraint satisfaction problem techniques based on N-Queens problem. 20th International Conference of Computer and Information Technology, ICCIT 2017, 2018-Janua, 1{6. https://doi.org/10 .1109/ICCITECHN.2017.8281850 Binary search (with code). (n.d.). Retrieved May 24, 2021, from https://www.programiz .com/dsa/binary-search Bubble sort (with code). (n.d.). Retrieved May 24, 2021, from https://www.programiz .com/dsa/bubble-sort 59

BIBLIOGRAPHY Bytecodealliance/wasmtime [original-date: 2017-08-29T14:01:55Z]. (2021, June 8). Byte- code Alliance. Retrieved June 8, 2021, from https://github.com/bytecodealliance/ wasmtime/blob/884a6500e91a4b8113d2b0c9497cb62cefe87111/docs/WASI -intro.md C program for n queen problem | backtracking-3 [GeeksforGeeks] [Section: C Pro- grams]. (2011, July 21). Retrieved May 24, 2021, from https://www.geeksforgeeks .org/c-program-for-n-queen-problem-backtracking-3/ Cabrera Arteaga, J., Donde, S., Gu, J., Floros, O., Satabin, L., Baudry, B., & Mon- perrus, M. (2020). Superoptimization of WebAssembly bytecode. ACM Inter- national Conference Proceeding Series, 36{40. https://doi.org/10.1145/ 3397537.3397567 Central limit theorem. (n.d.). Retrieved May 26, 2021, from https://sphweb.bumc .bu.edu/otlt/MPH-Modules/BS/BS704 Probability/BS704 Probability12 .html Download python [Python.org]. (n.d.). Retrieved May 24, 2021, from https://www .python.org/downloads/ Eight queens puzzle [Page Version ID: 1022941393]. (2021, May 13). In Wikipedia. Retrieved May 24, 2021, from https://en .wikipedia .org/w/index .php ?title=Eight queens puzzle&oldid=1022941393 Elhakeem Abd Elnaby, A., & El-Baz, A. H. (2021). A new explicit algorithmic method for generating the prime numbers in order. Egyptian Informatics Journal, 22(1), 101{104. https://doi.org/10.1016/j.eij.2020.05.002 GCC, the GNU compiler collection - GNU project - free software foundation (FSF). (n.d.). Retrieved May 24, 2021, from https://gcc.gnu.org/ Git. (n.d.). Retrieved May 24, 2021, from https://git-scm.com/ Guldal, S., Baugh, V., & Allehaibi, S. (2016). N-Queens solving algorithm by sets and backtracking. Conference Proceedings - IEEE SOUTHEASTCON, 2016-July. https://doi.org/10.1109/SECON.2016.7506688

Curiginal

84%	MATCHING BLOCK 2/13	SA	thesis.pdf (D48195363)

Haas, A., Rossberg, A., Schu , D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. F. (2017). Bringing the web up to speed with 60 BIBLIOGRAPHY WebAssembly. ACM SIGPLAN Notices, 52(6), 185{200.

https://doi.org/ 10.1145/3062341.3062363

Hall, A., &

Ramachandran,

U. (2019).

84%	MATCHING BLOCK 3/13	W	https://www.software-lab.org/publications/usen ...

An execution model for serverless functions at the edge. IoTDI 2019 - Proceedings of the 2019 Internet of Things Design and Implementation, 225{236.

https://doi.org/10.1145/3302505.3310084 Heap sort (with code). (n.d.). Retrieved May 24, 2021, from https://www.programiz .com/dsa/heap-sort Herrera, D., Chen, H., Lavoie, E., & Hendren, L. (2018).

84%	MATCHING BLOCK 4/13	SA	Examens_Arbete_Magnus_Medin_dv18mmn.pdf (D108840609)

WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices.

Introduction - rust and WebAssembly. (n.d.). Retrieved May 24, 2021, from https: //rustwasm.github.io/docs/book/ Jacob, A. E., Ashodariya, N., & Dhongade, A. (2018). Hybrid search algorithm: Com- bined linear and binary search algorithm. 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing, ICECDS 2017, 1543{1547. https://doi.org/10.1109/ICECDS.2017.8389704 Jangda,

61%	MATCHING BLOCK 5/13	SA	wbt_a_webassembly_benchmarking_tool_for_measur ... (D107312600)

A., Powers, B., Berger, E. D., & Guha, A. (2019). Not so fast: Analyzing the performance of webassembly vs. native code. 2019 USENIX Annual Tech- nical Conference (USENIX ATC 19), 107{120. https://www .usenix .org/ conference/atc19/presentation/jangda

Jeong, H. J., Shin, C. H., Shin, K. Y., Lee, H. J., & Moon, S. M. (2019). Seamless O oading of Web App Computations from Mobile Device to Edge Clouds via HTML5 Web Worker Migration. SoCC 2019 - Proceedings of the ACM Symposium on Cloud Computing, 38{49. https://doi.org/10.1145/3357223 .3362735 JetStream 2 in-depth analysis. (n.d.). Retrieved May 25, 2021, from https://browserbench .org/JetStream/in-depth.html

64%	MATCHING BLOCK 6/13	W	https://lib.dr.iastate.edu/cgi/viewcontent.cgi ...

Khan, F., Foley-Bourgon, V., Kathrotia, S., Lavoie, E., & Hendren, L. (2015). Using JavaScript and WebCL for numerical computations: A comparative study of 61 BIBLIOGRAPHY native and web technologies.

ACM SIGPLAN Notices, 50(2), 91{102.

https: //doi.org/10.1145/2661088.2661090 Koren, I. (2021). A Standalone WebAssembly Development Environment for the Inter- net of Things, 353{360. https://doi.org/10.1007/978-3-030-74296-6 27 Letz, S., Orlarey, Y., & Fober, D. (2018).

78%	MATCHING BLOCK 7/13	W	https://hal.archives-ouvertes.fr/hal-02158925/ ...

FAUST Domain Speci c Audio DSP Lan- guage Compiled to WebAssembly. The Web Conference 2018 -

Companion of the World Wide Web Conference, WWW 2018, 701{709. https://doi.org/ 10.1145/3184558.3185970 Linear search (with code). (n.d.). Retrieved May 24, 2021, from https://www.programiz .com/dsa/linear-search The LLVM compiler infrastructure project. (n.d.). Retrieved June 11, 2021, from https://llvm.org/ Long, J., Tai, H. Y., Hsieh, S. T., & Yuan, M. J. (2021). A Lightweight Design for Serverless Function as a Service. IEEE Software, 38(1), 75{80. https://doi .org/10.1109/MS.2020.3028991 Main | emscripten 2.0.21 documentation. (n.d.). Retrieved May 24, 2021, from https: //emscripten.org/ McLab. (n.d.). Retrieved May 24, 2021, from http://www.sable.mcgill.ca/mclab/ projects/ostrich/ Mendki, P. (2020). Evaluating webassembly enabled serverless approach for edge com- puting. Proceedings - 2020 IEEE Cloud Summit, Cloud Summit 2020, 161{166. https://doi.org/10.1109/IEEECloudSummit48914.2020.00031 Merge sort (with code). (n.d.). Retrieved May 24, 2021, from https://www.programiz .com/dsa/merge-sort Murphy, S., Persaud, L., Martini, W., & Bosshard, B. (2020). On the use of web assem- bly in a serverless context. Lecture Notes in Business Information Processing, 396 LNBIP, 141{145. https://doi.org/10.1007/978-3-030-58858-8 15 Musch,

75%	MATCHING BLOCK 8/13	SA	wbt_a_webassembly_benchmarking_tool_for_measur ... (D107312600)

M., Wressnegger, C., Johns, M., & Rieck, K. (2019). New kid on the web: A study on the prevalence of webassembly in the wild.

Lecture Notes in Computer Science (including subseries Lecture Notes in Arti cial Intelligence and Lecture 62

BIBLIOGRAPHY Notes in Bioinformatics), 11543 LNCS, 23{42. https://doi.org/10.1007/ 978-3-030-22038-9 2 Node.js. (n.d.). Node.js [Node.js]. Retrieved May 24, 2021, from https://nodejs .org/en/ Npm. (n.d.). Retrieved May 29, 2021, from https://www.npmjs.com/ OCaml - the core language. (n.d.). Retrieved June 11, 2021, from https://ocaml .org/manual/coreexamples.html PolyBench/c { homepage of louis-noel pouchet. (n.d.). Retrieved June 11, 2021, from http://web.cs.ucla.edu/ ~ pouchet/software/polybench/ Protzenko, J., Beurdouche, B., Merigoux, D., & Bhargavan, K. (2019). Formally veri- ed cryptographic web applications in webassembly. Proceedings - IEEE Sym- posium on Security and Privacy, 2019-May, 1256{1274.

https://doi.org/10 .1109/SP.2019.00064

Reiser, M., &

69%	MATCHING BLOCK 9/13	SA	wbt_a_webassembly_benchmarking_tool_for_measur ... (D107312600)

Blaser, L. (2017). Accelerate javascript applications by cross-compiling towebassembly. VMIL 2017 - Proceedings of the 9th ACM SIGPLAN Interna- tional Workshop on Virtual Machines and Intermediate Languages,

co-located with SPLASH 2017, 10{17. https://doi.org/10.1145/3141871.3141873 RStudio | open source & professional software for data science teams. (n.d.). Re- trieved May 25, 2021, from https://rstudio.com/ Rust programming language. (n.d.). Retrieved May 24, 2021, from https://www.rust -lang.org/ Sable/ostrich2 [GitHub]. (n.d.). Retrieved May 24, 2021, from https://github.com/ Sable/Ostrich2

89%	MATCHING BLOCK 10/13	SA	thesis.pdf (D48195363)

Sandhu, P., Herrera, D., & Hendren, L. (2018). Sparse matrices on the web - Charac- terizing the performance and optimal format selection of sparse matrix-vector multiplication in JavaScript and WebAssembly.

ACM International Conference Proceeding Series. https://doi.org/10.1145/3237009.3237020 Sandhu, P., Verbrugge, C., & Hendren, L. (2020). A fully structure-driven performance analysis of sparse matrix-vector multiplication. ICPE 2020 - Proceedings of the 63

BIBLIOGRAPHY ACM/SPEC International Conference on Performance Engineering, 108{119. https://doi.org/10.1145/3358960.3379131 Second-state/SOLL [original-date: 2019-08-22T08:58:13Z]. (2021, June 9). Second State. Retrieved June 12, 2021, from https://github.com/second-state/SOLL Selection sort (with code). (n.d.). Retrieved May 24, 2021, from https://www.programiz .com/dsa/selection-sort Shapiro{wilk test [Page Version ID: 1012297944]. (2021, March 15). In Wikipedia. Retrieved May 25, 2021, from https://en .wikipedia .org/w/index .php ? title=Shapiro%E2%80%93Wilk test&oldid=1012297944 Shell sort (with code). (n.d.). Retrieved May 24, 2021, from https://www.programiz .com/dsa/shell-sort Shillaker, S., & Pietzuch, P. (2020). FAASM: Lightweight isolation for e cient state- ful serverless computing. Proceedings of the 2020 USENIX Annual Technical Conference, ATC 2020, 419{433. Signalapp/libsignal-protocol-javascript [original-date: 2016-06-07T01:59:24Z]. (2021, June 5). Signal. Retrieved June 7, 2021, from https://github.com/signalapp/ libsignal-protocol-javascript

88%	MATCHING BLOCK 11/13	W	https://www.lennard-golsch.de/article/webassem ...

Standardizing WASI: A system interface to run WebAssembly outside the web { mozilla hacks -

the web developer blog [Mozilla hacks { the web developer blog]. (n.d.). Retrieved June 8, 2021, from

100%	MATCHING BLOCK 12/13	W	https://www.lennard-golsch.de/article/webassem ...

https : / / hacks .mozilla .org / 2019 / 03 / standardizing-wasi-a-webassembly-system-interface

Tableau public [Tableau public]. (n.d.). Retrieved May 25, 2021, from https://public .tableau.com/en-us/s/ Taheri, S. (2018). OpenCV . js : Computer Vision Processing for the Open Web Plat- form. Team, S. (n.d.). Solidity programming language [Solidity programming language]. Re- trieved June 12, 2021, from https://soliditylang.org// Tiwary, M., Mishra, P., Jain, S., & Puthal, D. (2020). Data Aware Web-Assembly Function Placement. The Web Conference 2020 - Companion of the World 64

BIBLIOGRAPHY Wide Web Conference, WWW 2020, 4{5. https://doi.org/10.1145/3366424 .3382670 Understanding WebAssembly text format - WebAssembly | MDN. (n.d.). Retrieved June 8, 2021, from

100%	MATCHING BLOCK 13/13	SA	MinhLe_HiepPhung_WebAssembly_thesisv2-20201030 ... (D87333559)

https://developer.mozilla.org/en-US/docs/WebAssembly/ Understanding the text format

van Eekelen, M., Frumin, D., Geuvers, H., Gondelman, L., Krebbers, R., Schoold- erman, M., Smetsers, S., Verbeek, F., Viguier, B., & Wiedijk, F. (2019). A benchmark for C program veri cation. arXiv. WASI |. (n.d.). Retrieved June 8, 2021, from https://wasi.dev/ Wasm-pack. (n.d.). Retrieved May 24, 2021, from https://rustwasm.github.io/ wasm-pack/installer/ Watt, C., Renner, J., Popescu, N., Cauligi, S., & Stefan, D. (2019). CT-wasm: type- driven secure cryptography for the web ecosystem. Proceedings of the ACM on Programming Languages, 3(POPL), 1{29. https://doi .org/10 .1145/ 3290390 WebAssembly/wabt [original-date: 2015-09-14T18:14:23Z]. (2021, May 30). WebAssem- bly. Retrieved May 31, 2021, from https://github.com/WebAssembly/wabt Write a program to nd the nth prime number. (n.d.). Retrieved May 24, 2021, from https://www.csinfo360.com/2020/01/write-program-to-find-nth-prime -number.html Zheng, S., Wang, H., Wu, L., Huang, G., & Liu, X. (2020). VM matters: A comparison of WASM vms and evms in the performance of blockchain smart contracts. CoRR, abs/2012.01032. https://arxiv.org/abs/2012.01032 65

Appendix A Source Code The entire code and results for these experiments can be found at: https://github.com/rayphelan/MastersProject2021 Listing A.1 contains the entire RStudio code used for performing statistical tests. Listing A.1: Statistical Tests in RStudio 1 library (" dplyr ") 2 library ("ggpubr") 3 4 ## ## 5 # Load Languages 6 AssemblyScript >- read . csv ('C: /TuDublin/Masters Project/Git Clone/MastersProject2021/ 7 FinalResultsPerLanguage/CSV/AssemblyScript . csv ') 8 9 C >- read . csv ('C: /TuDublin/Masters Project/Git Clone/MastersProject2021/ 10 FinalResultsPerLanguage/CSV/C. csv ') 11 12 CPP >- read . csv ('C: /TuDublin/Masters Project/Git Clone/MastersProject2021/ 13 FinalResultsPerLanguage/CSV/CPP. csv ') 14 15 Rust >- read . csv ('C: /TuDublin/Masters Project/Git Clone/MastersProject2021/ 16 FinalResultsPerLanguage/CSV/Rust . csv ') 17 18 19 ## ## 20 # Density Plot - AssemblyScript 66

APPENDIX A. SOURCE CODE 21 ggdensity (AssemblyScript$BinarySearch) 22 ggdensity (AssemblyScript$BubbleSort) 23 ggdensity (AssemblyScript$Fibonacci) 24 ggdensity (AssemblyScript$HeapSort) 25 ggdensity (AssemblyScript$LinearSearch) 26 ggdensity (AssemblyScript$MergeSort) 27 ggdensity (AssemblyScript$NQueen24) 28 ggdensity (AssemblyScript$NQueen27) 29 ggdensity (AssemblyScript$PrimeNumber) 30 ggdensity (AssemblyScript$ SelectionSort) 31 ggdensity (AssemblyScript$ ShellSort) 32 33 # Normality Plot - AssemblyScript 34 ggqqplot (

AssemblyScript$BinarySearch) 35 ggqqplot (AssemblyScript$BubbleSort) 36 ggqqplot (AssemblyScript$Fibonacci) 37 ggqqplot (AssemblyScript$HeapSort) 38 ggqqplot (AssemblyScript$LinearSearch) 39 ggqqplot (AssemblyScript$MergeSort) 40 ggqqplot (AssemblyScript$NQueen24) 41 ggqqplot (AssemblyScript$NQueen27) 42 ggqqplot (AssemblyScript$PrimeNumber) 43 ggqqplot (AssemblyScript$ SelectionSort) 44 ggqqplot (AssemblyScript$ ShellSort) 45 46 # Histograms - AssemblyScript 47 hist (AssemblyScript$BinarySearch) 48 hist (AssemblyScript$BubbleSort) 49 hist (AssemblyScript$Fibonacci) 50 hist (AssemblyScript$HeapSort) 51 hist (AssemblyScript$LinearSearch) 52 hist (AssemblyScript$MergeSort) 53 hist (AssemblyScript$NQueen24) 54 hist (AssemblyScript$NQueen27) 55 hist (AssemblyScript$PrimeNumber) 56 hist (AssemblyScript$ SelectionSort) 57 hist (AssemblyScript$ ShellSort) 58 59 # Shapiro-Wilk normality t e s t - AssemblyScript 60 shapiro . t e s t (AssemblyScript$BinarySearch) # W = 0.89333 , p-value = 9.081e-08 61 shapiro . t e s t (AssemblyScript$BubbleSort) # W = 0.947 , p-value = 0.000131 62 shapiro . t e s t (AssemblyScript$Fibonacci) # W = 0.9183 , p-value = 1.921e-06 63 shapiro . t e s t (AssemblyScript$HeapSort) # W = 0.92146 , p-value = 2.93e-06 64 shapiro . t e s t (AssemblyScript$LinearSearch) # W = 0.85618 , p-value = 1.97e-09 65 shapiro . t e s t (AssemblyScript$MergeSort) # W = 0.91813 , p-value = 1.878e-06 67

APPENDIX A. SOURCE CODE 66 shapiro . t e s t (AssemblyScript$NQueen24) # W = 0.95549 , p-value = 0.0005585 67 shapiro . t e s t (AssemblyScript$NQueen27) # W = 0.9585 , p-value = 0.0009603 68 shapiro . t e s t (AssemblyScript$PrimeNumber) # W = 0.91855 , p-value = 1.986e-06 69 shapiro . t e s t (AssemblyScript$ SelectionSort) # W = 0.73501 , p-value = 1.995e-13 70 shapiro . t e s t (AssemblyScript$ ShellSort) # W = 0.89884 , p-value = 1.713e-07 71 72 # independent 2-group Mann-Whitney U Tests 73 74 # BinarySearch 75 wilcox . t e s t (AssemblyScript$BinarySearch ,C$BinarySearch) # W = 12582 , p-value > 2.2e-16 76 wilcox . t e s t (AssemblyScript$BinarySearch ,CPP$BinarySearch) # W = 12408 , p-value > 2.2e-16 77 wilcox . t e s t (AssemblyScript$BinarySearch , Rust$BinarySearch) # W = 13720 , p-value > 2.2e-16 78 79 # BubbleSort 80 wilcox . t e s t (AssemblyScript$BubbleSort ,C$BubbleSort) # W = 14400 , p-value > 2.2e-16 81 wilcox . t e s t (AssemblyScript$BubbleSort ,CPP$BubbleSort) # W = 14400 , p-value > 2.2e-16 82 wilcox . t e s t (AssemblyScript$BubbleSort , Rust$BubbleSort) # W = 14400 , p-value > 2.2e-16 83 84 # Fibonacci 85 wilcox . t e s t (AssemblyScript$Fibonacci ,C$Fibonacci) # W = 14400 , p-value > 2.2e-16 86 wilcox . t e s t (AssemblyScript$Fibonacci ,CPP$Fibonacci) # W = 14400 , p-value > 2.2e-16 87 wilcox . t e s t (AssemblyScript$Fibonacci , Rust$Fibonacci) # W = 14400 , p-value > 2.2e-16 88 89 # HeapSort 90 wilcox . t e s t (AssemblyScript$HeapSort ,C$HeapSort) # W = 14400 , p-value > 2.2e-16 91 wilcox . t e s t (AssemblyScript$HeapSort ,CPP$HeapSort) # W = 14400 , p-value > 2.2e-16 92 wilcox . t e s t (AssemblyScript$HeapSort , Rust$HeapSort) # W = 14400 , p-value > 2.2e-16 93 94 # LinearSearch 95 wilcox . t e s t (AssemblyScript$LinearSearch ,C$LinearSearch) # W = 14400 , p-value > 2.2e-16 96 wilcox . t e s t (AssemblyScript$LinearSearch ,CPP$LinearSearch) # W = 14400 , p-value > 2.2e-16 97 wilcox . t e s t (AssemblyScript$LinearSearch , Rust$LinearSearch) # W = 14400 , p-value > 2.2e-16 98 99 # MergeSort 100 wilcox . t e s t (AssemblyScript$MergeSort ,C$MergeSort) # W = 14400 , p-value > 2.2e-16 101 wilcox . t e s t (AssemblyScript$MergeSort ,CPP$MergeSort) # W = 14400 , p-value > 2.2e-16 102 wilcox . t e s t (AssemblyScript$MergeSort , Rust$MergeSort) # W = 362 , p-value > 2.2e-16 103 104 # NQueen24 105 wilcox . t e s t (AssemblyScript$NQueen24 ,C$NQueen24) # W = 14400 , p-value > 2.2e-16 106 wilcox . t e s t (AssemblyScript$NQueen24 ,CPP$NQueen24) # W = 14400 , p-value > 2.2e-16 107 wilcox . t e s t (AssemblyScript$NQueen24 , Rust$NQueen24) # W = 14400 , p-value > 2.2e-16 108 109 # NQueen27 110 wilcox . t e s t (AssemblyScript$NQueen27 ,C$NQueen27) # W = 14400 , p-value > 2.2e-16 68

APPENDIX A. SOURCE CODE 111 wilcox . t e s t (AssemblyScript$NQueen27 ,CPP$NQueen27) # W = 14400 , p-value > 2.2e-16 112 wilcox . t e s t (AssemblyScript$NQueen27 , Rust$NQueen27) # W = 14400 , p-value > 2.2e-16 113 114 # PrimeNumber 115 wilcox . t e s t (AssemblyScript$PrimeNumber ,C$PrimeNumber) # W = 14400 , p-value > 2.2e-16 116 wilcox . t e s t (AssemblyScript$PrimeNumber ,CPP$PrimeNumber) # W = 14400 , p-value > 2.2e-16 117 wilcox . t e s t (AssemblyScript$PrimeNumber , Rust$PrimeNumber) # W = 14400 , p-value > 2.2e-16 118 119 # SelectionSort 120 wilcox . t e s t (AssemblyScript$SelectionSort ,C$ SelectionSort) # W = 14400 , p-value > 2.2e-16 121 wilcox . t e s t (AssemblyScript$SelectionSort ,CPP$ SelectionSort) # W = 14400 , p-value > 2.2e-16 122 wilcox . t e s t (AssemblyScript$SelectionSort , Rust$ SelectionSort) # W = 14400 , p-value > 2.2e-16 123 124 # ShellSort 125 wilcox . t e s t (AssemblyScript$ShellSort ,C$ ShellSort) # W = 14325 , p-value > 2.2e-16 126 wilcox . t e s t (AssemblyScript$ShellSort ,CPP$ ShellSort) # W = 14400 , p-value > 2.2e-16 127 wilcox . t e s t (AssemblyScript$ShellSort , Rust$ ShellSort) # W = 9598 , p-value = 8.265e-06 128 129 130 ## ## 131 # Density Plot - C 132 ggdensity (C$BinarySearch) 133 ggdensity (C$BubbleSort) 134 ggdensity (C$Fibonacci) 135 ggdensity (C$HeapSort) 136 ggdensity (C$LinearSearch) 137 ggdensity (C$MergeSort) 138 ggdensity (C$NQueen24)

139 ggdensity (C$NQueen27) 140 ggdensity (C$PrimeNumber) 141 ggdensity (C$ SelectionSort) 142 ggdensity (C$ ShellSort) 143 144 # Normality Plot 1 - C 145 ggqqplot (C$BinarySearch) 146 ggqqplot (C$BubbleSort) 147 ggqqplot (C$Fibonacci) 148 ggqqplot (C$HeapSort) 149 ggqqplot (C$LinearSearch) 150 ggqqplot (C$MergeSort) 151 ggqqplot (C$NQueen24) 152 ggqqplot (C$NQueen27) 153 ggqqplot (C$PrimeNumber) 154 ggqqplot (C$ SelectionSort) 155 ggqqplot (C$ ShellSort) 69

APPENDIX A. SOURCE CODE 156 157 # Histograms - C 158 hist (C$BinarySearch) 159 hist (C$BubbleSort) 160 hist (C$Fibonacci) 161 hist (C$HeapSort) 162 hist (C$LinearSearch) 163 hist (C$MergeSort) 164 hist (C$NQueen24) 165 hist (C$NQueen27) 166 hist (C$PrimeNumber) 167 hist (C$ SelectionSort) 168 hist (C$ ShellSort) 169 170 # Shapiro-Wilk normality t e s t - C 171 shapiro . t e s t (C$BinarySearch) # W = 0.85759 , p-value = 2.251e-09 172 shapiro . t e s t (C$BubbleSort) # W = 0.90343 , p-value = 2.95e-07 173 shapiro . t e s t (C$Fibonacci) # W = 0.87602 , p-value = 1.394e-08 174 shapiro . t e s t (C$HeapSort) # W = 0.92663 , p-value = 5.966e-06 175 shapiro . t e s t (C$LinearSearch) # W = 0.92993 , p-value = 9.529e-06 176 shapiro . t e s t (C$MergeSort) # W = 0.92818 , p-value = 7.426e-06 177 shapiro . t e s t (C$NQueen24) # W = 0.95149 , p-value = 0.0002786 178 shapiro . t e s t (C$NQueen27) # W = 0.73445 , p-value = 1.927e-13 179 shapiro . t e s t (C$PrimeNumber) # W = 0.87064 , p-value = 8.051e-09 180 shapiro . t e s t (C$ SelectionSort) # W = 0.91986 , p-value = 2.365e-06 181 shapiro . t e s t (C$ ShellSort) # W = 0.79129 , p-value = 9.023e-12 182 183 184 # independent 2-group Mann-Whitney U Tests 185 186 # BinarySearch 187 wilcox . t e s t (C$BinarySearch , AssemblyScript$BinarySearch) # W = 1818 , p-value > 2.2e-16 188 wilcox . t e s t (C$BinarySearch ,CPP$BinarySearch) # W = 6410 , p-value = 0.1421 189 wilcox . t e s t (C$BinarySearch , Rust$BinarySearch) # W = 10409 , p-value = 2.427e-09 190 191 # BubbleSort 192 wilcox . t e s t (C$BubbleSort , AssemblyScript$BubbleSort) # W = 0 , p-value > 2.2e-16 193 wilcox . t e s t (C$BubbleSort ,CPP$BubbleSort) # W = 11118 , p-value = 3.225e-13 194 wilcox . t e s t (C$BubbleSort , Rust$BubbleSort) # W = 626 , p-value > 2.2e-16 195 196 # Fibonacci 197 wilcox . t e s t (C$Fibonacci , AssemblyScript$Fibonacci) # W = 0 , p-value > 2.2e-16 198 wilcox . t e s t (C$Fibonacci ,CPP$Fibonacci) # W = 8162 , p-value = 0.07379 199 wilcox . t e s t (C$Fibonacci , Rust$Fibonacci) # W = 12296 , p-value > 2.2e-16 200 70

APPENDIX A. SOURCE CODE 201 # HeapSort 202 wilcox . t e s t (C$HeapSort , AssemblyScript$HeapSort) # W = 0 , p-value > 2.2e-16 203 wilcox . t e s t (C$HeapSort ,CPP$HeapSort) # W = 7423 , p-value = 0.6791 204 wilcox . t e s t (C$HeapSort , Rust$HeapSort) # W = 3206 , p-value = 1.119e-13 205 206 # LinearSearch 207 wilcox . t e s t (C$LinearSearch , AssemblyScript$LinearSearch) # W = 0 , p-value > 2.2e-16 208 wilcox . t e s t (C$LinearSearch ,CPP$LinearSearch) # W = 8953 , p-value = 0.001119 209 wilcox . t e s t (C$LinearSearch , Rust$LinearSearch) # W = 14280 , p-value > 2.2e-16 210 211 # MergeSort 212 wilcox . t e s t (C$MergeSort , AssemblyScript$MergeSort) # W = 0 , p-value > 2.2e-16 213 wilcox . t e s t (C$MergeSort ,CPP$MergeSort) # W = 6934 , p-value = 0.6215 214 wilcox . t e s t (C$MergeSort , Rust$MergeSort) # W = 0 , p-value > 2.2e-16 215 216 # NQueen24 217 wilcox . t e s t (C$NQueen24 , AssemblyScript$NQueen24) # W = 0 , p-value > 2.2e-16 218 wilcox . t e s t (C$NQueen24 ,CPP$NQueen24) # W = 14090 , p-value > 2.2e-16 219 wilcox . t e s t (C$NQueen24 , Rust$NQueen24) # W = 5819 , p-value = 0.01026 220 221 # NQueen27 222 wilcox . t e s t (C$NQueen27 , AssemblyScript$NQueen27) # W = 0 , p-value > 2.2e-16 223 wilcox . t e s t (C$NQueen27 ,CPP$NQueen27) # W = 12789 , p-value > 2.2e-16 224 wilcox . t e s t (C$NQueen27 , Rust$NQueen27) # W = 2604 , p-value > 2.2e-16 225 226 # PrimeNumber 227 wilcox . t e s t (C$PrimeNumber , AssemblyScript$PrimeNumber) # W = 0 , p-value > 2.2e-16 228 wilcox . t e s t (C$PrimeNumber ,CPP$PrimeNumber) # W = 4661 , p-value = 2.354e-06 229 wilcox . t e s t (C$PrimeNumber , Rust$PrimeNumber) # W = 0 , p-value > 2.2e-16 230 231 # SelectionSort 232 wilcox . t e s t (C$SelectionSort , AssemblyScript$ SelectionSort) # W = 0 , p-value > 2.2e-16 233 wilcox . t e s t (C$SelectionSort ,CPP$ SelectionSort) # W = 8881 , p-value = 0.001779 234 wilcox . t e s t (C$SelectionSort , Rust$ SelectionSort) # W = 0 , p-value > 2.2e-16 235 236 # ShellSort 237 wilcox . t e s t (C$ShellSort , AssemblyScript$ ShellSort) # W = 75 , p-value > 2.2e-16 238 wilcox . t e s t (C$ShellSort ,CPP$ ShellSort) # W = 6754 , p-value = 0.4074 239 wilcox . t e s t (C$ShellSort , Rust$ ShellSort) # W = 529 , p-value > 2.2e-16 240 241 242 ## 243 # Density Plot - CPP 244 ggdensity (CPP$BinarySearch) 245 ggdensity (CPP$BubbleSort) 71

APPENDIX A. SOURCE CODE 246 ggdensity (CPP$Fibonacci) 247 ggdensity (CPP$HeapSort) 248 ggdensity (CPP$LinearSearch) 249 ggdensity (CPP$MergeSort) 250 ggdensity (CPP$NQueen24) 251 ggdensity (CPP$NQueen27) 252 ggdensity (CPP$PrimeNumber) 253 ggdensity (CPP$ SelectionSort) 254 ggdensity (CPP$ ShellSort) 255 256 # Normality Plot 1 - CPP 257 ggqqplot (CPP$BinarySearch) 258 ggqqplot (CPP$BubbleSort) 259 ggqqplot (CPP$Fibonacci) 260 ggqqplot (CPP$HeapSort) 261 ggqqplot (CPP$LinearSearch) 262 ggqqplot (CPP$MergeSort) 263 ggqqplot (CPP$NQueen24) 264 ggqqplot (CPP$NQueen27) 265 ggqqplot (CPP$PrimeNumber) 266 ggqqplot (CPP$ SelectionSort) 267 ggqqplot (CPP$ ShellSort) 268 269 # Histograms - CPP 270 hist (CPP$BinarySearch) 271 hist (CPP$BubbleSort) 272 hist (CPP$Fibonacci) 273 hist (CPP$HeapSort) 274 hist (CPP$LinearSearch) 275 hist (CPP$MergeSort) 276 hist

(CPP$NQueen24) 277 hist (CPP$NQueen27) 278 hist (CPP$PrimeNumber) 279 hist (CPP$ SelectionSort) 280 hist (CPP$ ShellSort) 281 282 # Shapiro-Wilk normality t e s t - CPP 283 shapiro . t e s t (CPP$BinarySearch) # W = 0.88697 , p-value = 4.471e-08 284 shapiro . t e s t (CPP$BubbleSort) # W = 0.86648 , p-value = 5.32e-09 285 shapiro . t e s t (CPP$Fibonacci) # W = 0.87035 , p-value = 7.82e-09 286 shapiro . t e s t (CPP$HeapSort) # W = 0.92485 , p-value = 4.656e-06 287 shapiro . t e s t (CPP$LinearSearch) # W = 0.94416 , p-value = 8.256e-05 288 shapiro . t e s t (CPP$MergeSort) # W = 0.85896 , p-value = 2.564e-09 289 shapiro . t e s t (CPP$NQueen24) # W = 0.56648 , p-value > 2.2e-16 290 shapiro . t e s t (CPP$NQueen27) # W = 0.80628 , p-value = 2.8e-11 72

APPENDIX A. SOURCE CODE 291 shapiro . t e s t (CPP$PrimeNumber) # W = 0.76245 , p-value = 1.183e-12 292 shapiro . t e s t (CPP$ SelectionSort) # W = 0.91616 , p-value = 1.451e-06 293 shapiro . t e s t (CPP$ ShellSort) # W = 0.91794 , p-value = 1.832e-06 294 295 # independent 2-group Mann-Whitney U Tests 296 297 # BinarySearch 298 wilcox . t e s t (CPP$BinarySearch , AssemblyScript$BinarySearch) # W = 1992 , p-value > 2.2e-16 299 wilcox . t e s t (CPP$BinarySearch ,C$BinarySearch) # W = 7990 , p-value = 0.1421 300 wilcox . t e s t (CPP$BinarySearch , Rust$BinarySearch) # W = 10886 , p-value = 7.219e-12 301 302 # BubbleSort 303 wilcox . t e s t (CPP$BubbleSort , AssemblyScript$BubbleSort) # W = 0 , p-value > 2.2e-16 304 wilcox . t e s t (CPP$BubbleSort ,C$BubbleSort) # W = 3282 , p-value = 3.225e-13 305 wilcox . t e s t (CPP$BubbleSort , Rust$BubbleSort) # W = 73 , p-value > 2.2e-16 306 307 # Fibonacci 308 wilcox . t e s t (CPP$Fibonacci , AssemblyScript$Fibonacci) # W = 0 , p-value > 2.2e-16 309 wilcox . t e s t (CPP$Fibonacci ,C$Fibonacci) # W = 6238 , p-value = 0.07379 310 wilcox . t e s t (CPP$Fibonacci , Rust$Fibonacci) # W = 11568 , p-value = 4.606e-16 311 312 # HeapSort 313 wilcox . t e s t (CPP$HeapSort , AssemblyScript$HeapSort) # W = 0 , p-value > 2.2e-16 314 wilcox . t e s t (CPP$HeapSort ,C$HeapSort) # W = 6977 , p-value = 0.6791 315 wilcox . t e s t (CPP$HeapSort , Rust$HeapSort) # W = 3111 , p-value = 2.901e-14 316 317 # LinearSearch 318 wilcox . t e s t (CPP$LinearSearch , AssemblyScript$LinearSearch) # W = 0 , p-value > 2.2e-16 319 wilcox . t e s t (CPP$LinearSearch ,C$LinearSearch) # W = 5447 , p-value = 0.001119 320 wilcox . t e s t (CPP$LinearSearch , Rust$LinearSearch) # W = 14280 , p-value > 2.2e-16 321 322 # MergeSort 323 wilcox . t e s t (CPP$MergeSort , AssemblyScript$MergeSort) # W = 0 , p-value > 2.2e-16 324 wilcox . t e s t (CPP$MergeSort ,C$MergeSort) # W = 7466 , p-value = 0.6215 325 wilcox . t e s t (CPP$MergeSort , Rust$MergeSort) # W = 0 , p-value > 2.2e-16 326 327 # NQueen24 328 wilcox . t e s t (CPP$NQueen24 , AssemblyScript$NQueen24) # W = 0 , p-value > 2.2e-16 329 wilcox . te s t (CPP$NQueen24 ,C$NQueen24) # W = 310 , p-value > 2.2e-16 330 wilcox . te st (CPP$NQueen24 , Rust$NQueen24) # W = 781 , p-value > 2.2e-16 331 332 # NQueen27 333 wilcox . te st (CPP$NQueen27 , AssemblyScript$NQueen27) # W = 0 , p-value > 2.2e-16 334 wilcox . te st (CPP$NQueen27 ,C$NQueen27) # W = 1611 , p-value > 2.2e-16 335 wilcox . te st (CPP$NQueen27 , Rust$NQueen27) # W = 872 , p-value > 2.2e-16 73

APPENDIX A. SOURCE CODE 336 337 # PrimeNumber 338 wilcox . te st (CPP$PrimeNumber , AssemblyScript$PrimeNumber) # W = 0 , p-value > 2.2e-16 339 wilcox . te st (CPP$PrimeNumber ,C$PrimeNumber) # W = 9739 , p-value = 2.354e-06 340 wilcox . te st (CPP$PrimeNumber , Rust$PrimeNumber) # W = 0 , p-value > 2.2e-16 341 342 # SelectionSort 343 wilcox . te st (CPP$SelectionSort , AssemblyScript$ SelectionSort) # W = 0 , p-value > 2.2e-16 344 wilcox . te st (CPP$SelectionSort ,C$ SelectionSort) # W = 5519 , p-value = 0.001779 345 wilcox . te st (CPP$SelectionSort , Rust$ SelectionSort) # W = 0 , p-value > 2.2e-16 346 347 # ShellSort 348 wilcox . te st (CPP$ShellSort , AssemblyScript$ ShellSort) # W = 0 , p-value > 2.2e-16 349 wilcox . te st (CPP$ShellSort ,C$ ShellSort) # W = 7646 , p-value = 0.4074 350 wilcox . te st (CPP$ShellSort , Rust$ ShellSort) # W = 565 , p-value > 2.2e-16 351 352 353 ## 354 # Density Plot - Rust 355 ggdensity (Rust$BinarySearch) 356 ggdensity (Rust$BubbleSort) 357 ggdensity (Rust$Fibonacci) 358 ggdensity (Rust$HeapSort) 359 ggdensity (Rust$LinearSearch) 360 ggdensity (Rust$MergeSort) 361 ggdensity (Rust$NQueen24) 362 ggdensity (Rust$NQueen27) 363 ggdensity (Rust$PrimeNumber) 364 ggdensity (Rust$ SelectionSort) 365 ggdensity (Rust$ ShellSort) 366 367 # Normality Plot 1 - Rust 368 ggqqplot (Rust$BinarySearch) 369 ggqqplot (Rust$BubbleSort) 370 ggqqplot (Rust$Fibonacci) 371 ggqqplot (Rust$HeapSort) 372 ggqqplot (Rust$LinearSearch) 373 ggqqplot (Rust$MergeSort) 374 ggqqplot (Rust$NQueen24) 375 ggqqplot (Rust$NQueen27) 376 ggqqplot (Rust$PrimeNumber) 377 ggqqplot (Rust$ SelectionSort) 378 ggqqplot (Rust$ ShellSort) 379 380 # Histograms - Rust 74

APPENDIX A. SOURCE CODE 381 hist (Rust$BinarySearch) 382 hist (Rust$BubbleSort) 383 hist (Rust$Fibonacci) 384 hist (Rust$HeapSort) 385 hist (Rust$LinearSearch) 386 hist (Rust$MergeSort) 387 hist (Rust$NQueen24) 388 hist (Rust$NQueen27) 389 hist (Rust$PrimeNumber) 390 hist (Rust$ SelectionSort) 391 hist (Rust$ ShellSort) 392 393 # Shapiro-Wilk normality t e s t - Rust 394 shapiro . te st (Rust$BinarySearch) # W = 0.9888 , p-value = 0.4335 395 shapiro . te st (Rust$BubbleSort) # W = 0.91152 , p-value = 8.002e-07 396 shapiro . te st (Rust$Fibonacci) # W = 0.90205 , p-value = 2.501e-07 397 shapiro . te st (Rust$HeapSort) # W = 0.90527 , p-value = 3.685e-07 398 shapiro . te st (Rust$LinearSearch) # W = 0.45708 , p-value > 2.2e-16 399 shapiro . te st (Rust$MergeSort) # W = 0.56386 , p-value > 2.2e-16 400 shapiro . te st (Rust$NQueen24) # W = 0.97237 , p-value = 0.01424 401 shapiro . te st (Rust$NQueen27) # W = 0.79029 , p-value = 8.382e-12 402 shapiro . te st (Rust$PrimeNumber) # W = 0.88119 , p-value = 2.399e-08 403 shapiro . te st (Rust$ SelectionSort) # W = 0.88721 , p-value = 4.591e-08 404 shapiro . te st (Rust$ ShellSort) # W = 0.97768 , p-value = 0.04334 405 406 407 # independent 2-group Mann-Whitney U Tests 408 409 # BinarySearch 410 wilcox . te st (Rust$BinarySearch , AssemblyScript$BinarySearch) # W = 680 , p-value > 2.2e-16 411 wilcox . te st (Rust$BinarySearch ,C$BinarySearch) # W = 3991 , p-value = 2.427e-09 412 wilcox . te st (Rust$BinarySearch ,CPP$BinarySearch) # W = 3514 , p-value = 7.219e-12 413 414 # BubbleSort 415 wilcox . te st (Rust$BubbleSort , AssemblyScript$BubbleSort) # W = 0 , p-value > 2.2e-16 416 wilcox . te st (Rust$BubbleSort ,C$BubbleSort) # W = 13774 , p-value > 2.2e-16 417 wilcox . te st (Rust$BubbleSort ,CPP$BubbleSort) # W = 14327 , p-value > 2.2e-16 418 419 # Fibonacci 420 wilcox . te st (Rust$Fibonacci , AssemblyScript$Fibonacci) # W = 0 , p-value > 2.2e-16 421 wilcox . te st (Rust$Fibonacci ,C$Fibonacci) # W = 2104 , p-value > 2.2e-16 422 wilcox . te st (Rust$Fibonacci ,CPP$Fibonacci) # W = 2832 , p-value = 4.606e-16 423 424 # HeapSort 425 wilcox . te st (Rust$HeapSort , AssemblyScript$HeapSort) # W = 0 , p-value > 2.2e-16 75

APPENDIX A. SOURCE CODE 426 wilcox . te st (Rust$HeapSort ,C$HeapSort) # W = 11194 , p-value = 1.119e-13 427 wilcox . te st (Rust$HeapSort ,CPP$HeapSort) # W = 11289 , p-value = 2.901e-14 428 429 # LinearSearch 430 wilcox . te st (Rust$LinearSearch , AssemblyScript$LinearSearch) # W = 0 , p-value > 2.2e-16 431 wilcox . te st (Rust$LinearSearch ,C$LinearSearch) # W = 120 , p-value > 2.2e-16 432 wilcox . te st (Rust$LinearSearch ,CPP$LinearSearch) # W = 120 , p-value > 2.2e-16 433 434 # MergeSort 435 wilcox . te st (Rust$MergeSort , AssemblyScript$MergeSort) # W = 14038 , p-value > 2.2e-16 436 wilcox . te st (Rust$MergeSort ,C$MergeSort) # W = 14400 , p-value > 2.2e-16 437 wilcox . te st (Rust$MergeSort ,CPP$MergeSort) # W = 14400 , p-value > 2.2e-16 438 439 # NQueen24 440 wilcox . te st (Rust$NQueen24 , AssemblyScript$NQueen24) # W = 0 , p-value > 2.2e-16 441 wilcox . te st (Rust$NQueen24 ,C$NQueen24) # W = 8581 , p-value = 0.01026 442 wilcox . te st (Rust$NQueen24 ,CPP$NQueen24) # W = 13619 , p-value > 2.2e-16 443 444 # NQueen27 445 wilcox . te st (Rust$NQueen27 , AssemblyScript$NQueen27) # W = 0 , p-value > 2.2e-16 446 wilcox . te st (Rust$NQueen27 ,C$NQueen27) # W = 11796 , p-value > 2.2e-16 447 wilcox . te st (Rust$NQueen27 ,CPP$NQueen27) # W = 13528 , p-value > 2.2e-16 448 449 # PrimeNumber 450 wilcox . te st (Rust$PrimeNumber , AssemblyScript$PrimeNumber) # W = 0 , p-value > 2.2e-16 451 wilcox . te st (Rust$PrimeNumber ,C$PrimeNumber) # W = 14400 , p-value > 2.2e-16 452 wilcox . te st (Rust$PrimeNumber ,CPP$PrimeNumber) # W = 14400 , p-value > 2.2e-16 453 454 # SelectionSort 455 wilcox . te st (Rust$SelectionSort , AssemblyScript$ SelectionSort) # W = 0 , p-value > 2.2e-16 456 wilcox . te st (Rust$SelectionSort ,C$ SelectionSort) # W = 14400 , p-value > 2.2e-16 457 wilcox . te st (Rust$SelectionSort ,CPP$ SelectionSort) # W = 14400 , p-value > 2.2e-16 458 459 # ShellSort 460 wilcox . te st (Rust$ShellSort , AssemblyScript$ ShellSort) # W = 4802 , p-value = 8.265e-06 461 wilcox . te st (Rust$ShellSort ,C$ ShellSort) # W = 13871 , p-value > 2.2e-16 462 wilcox . te st (Rust$ShellSort ,CPP$ ShellSort) # W = 13835 , p-value > 2.2e-16 463 464 465
466
467
76

Appendix B Distribution of Data The following gures represent the distribtion of the data obtained through statistical tests. Figure B.1 shows the distribution of data for the Fibonacci sequence algorithm. Figure B.1: Fibonacci 77

APPENDIX B. DISTRIBUTION OF DATA Figure B.2 shows the distribution of data for the NQueen24 algorithm. Figure B.2: NQueen24 Figure B.3 shows the distribution of data for the NQueen27 algorithm. Figure B.3: NQueen27 78

APPENDIX B. DISTRIBUTION OF DATA Figure B.4 shows the distribution of data for the PrimeNumber algorithm. Figure B.4: PrimeNumber Figure B.5 shows the distribution of data for the BinarySearch algorithm. Figure B.5: BinarySearch 79

APPENDIX B. DISTRIBUTION OF DATA Figure B.6 shows the distribution of data for the LinearSearch algorithm. Figure B.6: LinearSearch Figure B.7 shows the distribution of data for the BubbleSort algorithm. Figure B.7: BubbleSort 80

APPENDIX B. DISTRIBUTION OF DATA Figure B.8 shows the distribution of data for the HeapSort algorithm. Figure B.8: HeapSort Figure B.9 shows the distribution of data for the MergeSort algorithm. Figure B.9: MergeSort 81

APPENDIX B. DISTRIBUTION OF DATA Figure B.10 shows the distribution of data for the ShellSort algorithm. Figure B.10: ShellSort Figure B.11 shows the distribution of data for the SelectionSort algorithm. Figure B.11: SelectionSort 82

APPENDIX B. DISTRIBUTION OF DATA Figure B.12 represents the distribution of data for all the algorithms used in our experiments compiled from AssemblyScript. Figure B.12: AssemblyScript Distribtion (a) BinarySearch (b) BubbleSort (c) Fibonacci (d) HeapSort (e) LinearSearch (f) MergeSort (g) NQueen24 (h) NQueen27 (i) PrimaryNumber (j) SelectionSort (k) ShellSort 83

APPENDIX B. DISTRIBUTION OF DATA Figure 3.5 represents the distribution of data for all the algorithms used in our experiments compiled from C. Figure B.13: C Distribtion (a) BinarySearch (b) BubbleSort (c) Fibonacci (d) HeapSort (e) LinearSearch (f) MergeSort (g) NQueen24 (h) NQueen27 (i) PrimaryNumber (j) SelectionSort (k) ShellSort 84

APPENDIX B. DISTRIBUTION OF DATA Figure B.14 represents the distribution of data for all the algorithms used in our experiments compiled from C++. Figure B.14: C++ Distribtion (a) BinarySearch (b) BubbleSort (c) Fibonacci (d) HeapSort (e) LinearSearch (f) MergeSort (g) NQueen24 (h) NQueen27 (i) PrimaryNumber (j) SelectionSort (k) ShellSort 85

APPENDIX B. DISTRIBUTION OF DATA Figure B.15 represents the distribution of data for all the algorithms used in our experiments compiled from Rust. Figure B.15: Rust Distribtion (a) BinarySearch (b) BubbleSort (c) Fibonacci (d) HeapSort (e) LinearSearch (f) MergeSort (g) NQueen24 (h) NQueen27 (i) PrimaryNumber (j) SelectionSort (k) ShellSort 86

Hit and source - focused comparison, Side by Side

Submitted text	As student entered the text in the submitted document.
Matching text	As the text appears in the source.

1/13	SUBMITTED TEXT	60 WORDS	32%	MATCHING TEXT	60 WORDS

Submitted text: r e s u l t s . push (r e s u l t) ; 19 console . log (wasm) ; 20 g 21 const csv = r e s u l t s . join ('nn ') ; 22 23 // Write File 24 f s . writeFile (' r e s u l t s .

Matching text: r e s u l t s a r e s t r u c t u r e d s u c h t h a t t h e r e s u l t s

SA Examensarbete+-+Aevan+Dino%2C+Seth+%C3%96berg+%5BGrupp+23%5D.pdf (D105869396)

2/13	SUBMITTED TEXT	37 WORDS	84%	MATCHING TEXT	37 WORDS

Submitted text: Haas, A., Rossberg, A., Schu , D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. F. (2017). Bringing the web up to speed with 60 BIBLIOGRAPHY WebAssembly. ACM SIGPLAN Notices, 52(6), 185{200.

Matching text: Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. (2017). " Bringing the web up to speed with WebAssembly". ACM SIGPLAN Notices.

SA thesis.pdf (D48195363)

3/13	SUBMITTED TEXT	22 WORDS	84%	MATCHING TEXT	22 WORDS

Submitted text: An execution model for serverless functions at the edge. IoTDI 2019 - Proceedings of the 2019 Internet of Things Design and Implementation, 225{236.

Matching text: An Execution Model for Serverless Functions at the Edge. Proceedings of International Conference on Internet of Things Design and Implementation (

W https://www.software-lab.org/publications/usenixSec2020-WebAssembly.pdf

4/13	SUBMITTED TEXT	11 WORDS	84%	MATCHING TEXT	11 WORDS

Submitted text: WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices.

Matching text: WebAssembly and JavaScript Challenge: Numerical program performance using modern browser tech- nologies and devices".

SA Examens_Arbete_Magnus_Medin_dv18mmn.pdf (D108840609)

Curiginal

5/13	SUBMITTED TEXT	32 WORDS	61% MATCHING TEXT	32 WORDS

A., Powers, B., Berger, E. D., & Guha, A. (2019). Not so fast: Analyzing the performance of webassembly vs. native code. 2019 USENIX Annual Tech- nical Conference (USENIX ATC 19), 107{120. https://www .usenix .org/ conference/atc19/presentation/jangda

A. Jangda, B. Powers, E. D. Berger, and A. Not so fast: Analyzing the performance of webassembly vs. native code," USENIX Annual Technical Conference (USENIX ATC 19). Renton, WA: USENIX Association, Jul. 2019, pp. 107{120. [Online]. https: //www.usenix.org/conference/atc19/presentation/jangda [

SA wbt_a_webassembly_benchmarking_tool_for_measuring_algorithm_performance.pdf (D107312600)

6/13	SUBMITTED TEXT	27 WORDS	64% MATCHING TEXT	27 WORDS

Khan, F., Foley-Bourgon, V., Kathrotia, S., Lavoie, E., & Hendren, L. (2015). Using JavaScript and WebCL for numerical computations: A comparative study of 61 BIBLIOGRAPHY native and web technologies.

Khan, V. Foley-Bourgon, S. Kathrotia, E. Lavoie, and L. Using javascript and webcl for numerical computations: A comparative study of native and web technologies,"

W https://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=1583&context=creativecomponents

7/13	SUBMITTED TEXT	16 WORDS	78% MATCHING TEXT	16 WORDS

FAUST Domain Speci c Audio DSP Lan- guage Compiled to WebAssembly. The Web Conference 2018 -

FAUST Domain Specific Audio DSP Lan- guage Compiled to WebAssembly. The Web Conference, 2018,

W https://hal.archives-ouvertes.fr/hal-02158925/document

8/13	SUBMITTED TEXT	24 WORDS	75% MATCHING TEXT	24 WORDS

M., Wressnegger, C., Johns, M., & Rieck, K. (2019). New kid on the web: A study on the prevalence of webassembly in the wild.

M. Musch, C. Wressnegger, M. and K. New kid on the web: A study on the prevalence of webassembly in the wild,"

SA wbt_a_webassembly_benchmarking_tool_for_measuring_algorithm_performance.pdf (D107312600)

Ouriginal

9/13	SUBMITTED TEXT 26 WORDS	69% MATCHING TEXT 26 WORDS
	Blaser, L. (2017). Accelerate javascript applications by cross-compiling towebassembly. VMIL 2017 - Proceedings of the 9th ACM SIGPLAN Interna- tional Workshop on Virtual Machines and Intermediate Languages,	aser, \Accelerate javascript applications by cross-compiling to webassembly," in Proceedings of the 9th ACM SIGPLAN International Workshop on virtual machines and intermediate languages,

SA wbt_a_webassembly_benchmarking_tool_for_measuring_algorithm_performance.pdf (D107312600)

10/13	SUBMITTED TEXT 29 WORDS	89% MATCHING TEXT 29 WORDS
	Sandhu, P., Herrera, D., & Hendren, L. (2018). Sparse matrices on the web - Charac- terizing the performance and optimal format selection of sparse matrix-vector multiplication in JavaScript and WebAssembly.	Sandhu, P., Herrera, D., & Hendren, L. (2018). "Sparse Matrices on the Web: Characterizing the Performance and Optimal Format Selection of Sparse Matrix-vector Multiplication in JavaScript and WebAssembly".

SA thesis.pdf (D48195363)

11/13	SUBMITTED TEXT 15 WORDS	88% MATCHING TEXT 15 WORDS
	Standardizing WASI: A system interface to run WebAssembly outside the web { mozilla hacks -	Standardizing WASI: A system interface to run WebAssembly outside the web. https://hacks.

W https://www.lennard-golsch.de/article/webassembly.pdf

12/13	SUBMITTED TEXT 12 WORDS	100% MATCHING TEXT 12 WORDS
	https : / / hacks .mozilla .org / 2019 / 03 / standardizing-wasi-a-webassembly-system-interface	https://hacks.mozilla.org/2019/03/ standardizing-wasi-a-webassembly-system-interface/,

W https://www.lennard-golsch.de/article/webassembly.pdf

13/13	SUBMITTED TEXT 4 WORDS	100% MATCHING TEXT 4 WORDS
	https://developer.mozilla.org/en-US/docs/WebAssembly/ Understanding the text format	https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format [

SA MinhLe_HiepPhung_WebAssembly_thesisv2-20201030.docx (D87333559)