Web Applications

Performance through WebAssembly

Ferdinand Enger

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Institute of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

2022

Web Applications

Performance through WebAssembly

Ferdinand Enger

Web Applications

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

A web application executes inside a browser where built-in APIs enable lightweight and portable apps using a single codebase.

The primary advantage of web apps is their broad reach. Their main drawback is restricting developers to languages native to the browser. Historically JavaScript has held a monopoly as the web programming language. While JavaScript is flexible and supports multiple programming paradigms. It is object-oriented at its core and uses weak typing with prototypes. The advantage of prototypes is powerful inheritance through chaining. The drawback is that every piece of primitive data has to reside inside an object wrapper.

This orientation inhibits fast maths, as there is no way to opt-out of inheritance and object wrappers. Substantial changes to JavaScript have not been practicable due to backward compatibility. Browser vendors have therefore agreed to introduce WebAssembly (WASM) as a new language to offload computation. WebAssembly is a statically typed language emphasising fast maths. This thesis investigates its performance.

Acknowledgements

The following persons deserve special mention:

- Prof. Tor Skeie for his congeniality and excellent spirits.

- Prof. Dag Langmyhr for lectures on programming and compilers.

- Prof. Carsten Griwodz for performance analysis in INF3190.

- Prof. Michael Welzl for the course on internet protocol evolution.

- Associate Prof. Safiqul Islam for presenting his work on WebRTC.

- Associate Prof. Solveig Bruvoll for her lectures on disassembly.

- Fellow students, family and friends who were supportive.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programming is the art of problem-solving through abstractions. Operating systems support programs by abstracting hardware, allowing programmers to use high-level languages. Together the operating system and hardware make up a platform.

The first decision in programming is deciding which platforms to support. Let us assume we are building an augmented reality app. This app must function on all tablets, desktops and smartphones. Our platforms are iOS, Android, macOS, Windows and Linux.

Each platform has a different format for packaging and will require a separate executable. Assuming our application will need access to the GPU. Each executable will likely require a platform-specific graphics library. We will therefore have to re-implement graphics on each platform.

In the proceeding, please make two assumptions regarding our application. First, assume that we will want to maximise code reuse. Secondly, consider the web as an independent platform. If we can implement our application using web technology. We would eliminate platform-specific concerns. No installation is required, and distribution does not require packaging executables. Our users merely need to access the URL. The main benefit of the web platform is its broad reach. A single website can serve a plethora of devices through responsive design.

Browsers offer a special kind of platform. Built-in libraries provide access to storage, database and graphics facilities. In addition, APIs manage input

devices such as cameras, microphones and touch sensors.

Browsers orchestrate hardware but do not offer an operating system per se. As a taxonomy, consider platforms with kernel access to be native platforms. Kernel access enables hardware support and shared libraries. Lack of kernel access restricts web apps to existing browser interfaces. In this sense, web apps come with batteries included but offer a fixed set of tools.

In summary, the strengths of the web platform include:

- Easy access via URLs with no installation.
- No third party has to sign releases.
- Standards enforce consistency.
- Using a single codebase across devices.

Whereas weaknesses constitute:

- Reliance on existing APIs that limit extensibility.
- Development is limited to formats native to the browser.
- No interaction with the operating system.
- Browsers impose overhead and restrict performance.

There is a trade-off between reach and capability. Web apps operate across platforms but offer a restricted feature set. Suppose we cannot rely on our users installing plugins or browser extensions. Then we can only use languages with universal support. Three such languages exist.

- HTML provides structure.
- CSS provides styling.
- JavaScript provides interactivity.

HTML and CSS are static with no execution context. Their outputs are tree structures. These structures are called the document and CSS object models. It is common to refer to the document object model as the DOM and the CSS object model as the CSSOM. The DOM and CSSOM need to merge when painting pixels on a screen. This stepwise process is called the critical render path. Please refer to figure 1.1 as an example of the render path in action.

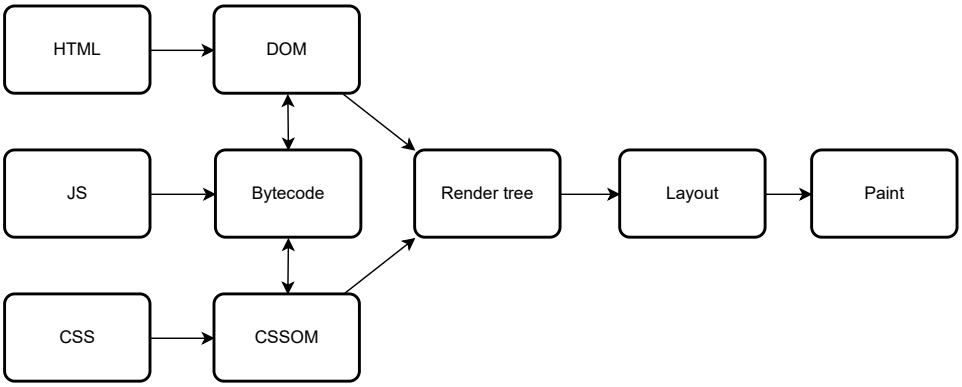

Figure 1.1: The critical render path

JavaScript provides the dynamic component in web apps. It combines event-driven, functional and object-oriented styles. JavaScript is always active because it executes inside an infinite loop on a single thread. This loop is called the event loop. The thread in which it is running is called the user interface thread. Every facet of rendering is accessible through JavaScript. The critical rendering path in figure 1.1 illustrates how JavaScript can manipulate the DOM and CSSOM. Because JavaScript can change the CSSOM and DOM on its own, it is as if JavaScript can write new HTML and CSS code independently.

All function calls inside the event loop need to complete quickly. Failure to observe timing constraints will cause the user interface to freeze while the thread is engaged. Synchronous code will execute immediately while asynchronous code passes onto a queue. Once synchronous code completes, asynchronous code will run by callbacks. All code runs on the same thread. This thread is the only thread with access to the DOM and CSSOM, which is why it is known as the user interface thread.

The advantage of a single thread is preventing race conditions. Note that strict single-threaded access to the DOM and CSSOM partly reflects the age of JavaScript. It was introduced in 1995, whereas multi-core processors would first appear in the next decade. Figure 1.2 shows the JavaScript execution context. Synchronous code runs via the stack, whereas asynchronous code passes through the message queue. This message queue also executes via the stack but will not run until the callback is ready. Note that function arguments are placed on the heap to support access in nested functions.

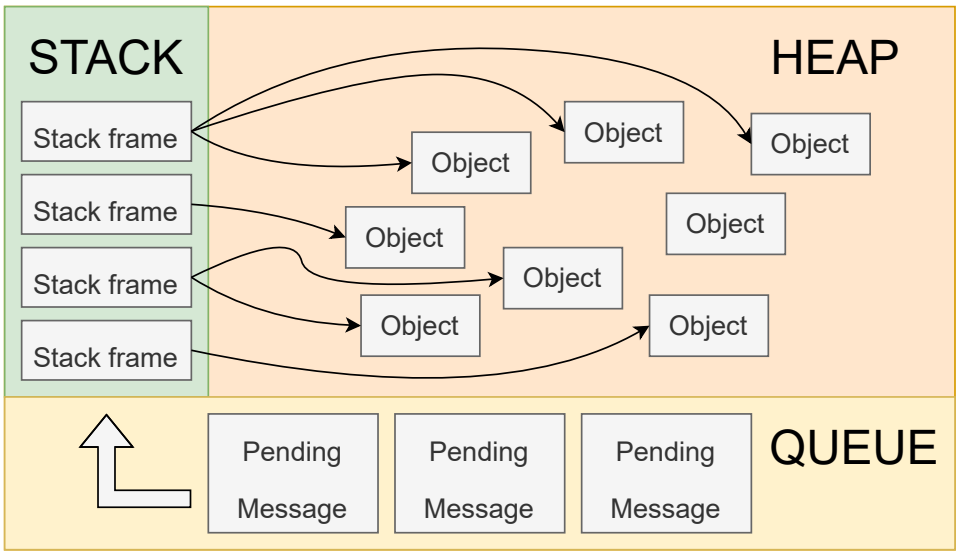

Figure 1.2: JavaScript runtime

JavaScript prevents blocking by handling events via its message queue. Each event generates a new message on the queue, and each message is like a delayed function call. The user interface thread can execute these calls whenever it is free. It grabs any pending messages as it becomes idle. Note that there are no explicit timing constraints.

Applications become reactive by event-driven programming. Programs need to register events and attach event handlers. These event handlers act as callback functions. Possible events include keyboard, mouse or touch inputs. This model also extends to accessing camera, microphone and location data. Each of these requires using JavaScript APIs.

Expanding APIs is the most common pattern to add new functionality to JavaScript. Vendors introduce new features with novel APIs. Such as service workers and local storage. Offline web apps use service workers to intercept requests and respond from local storage. Figure 1.3 illustrates how JavaScript offers a gateway to all inputs.

Nearly everything in JavaScript is an object. For instance, both functions and datatypes are objects. JavaScript relies on prototype inheritance to implement its object model. Prototypes are object blueprints. Every object must have a prototype attached, and these can be chained. JavaScript can define properties either at the object or the prototype chain. Adding a new

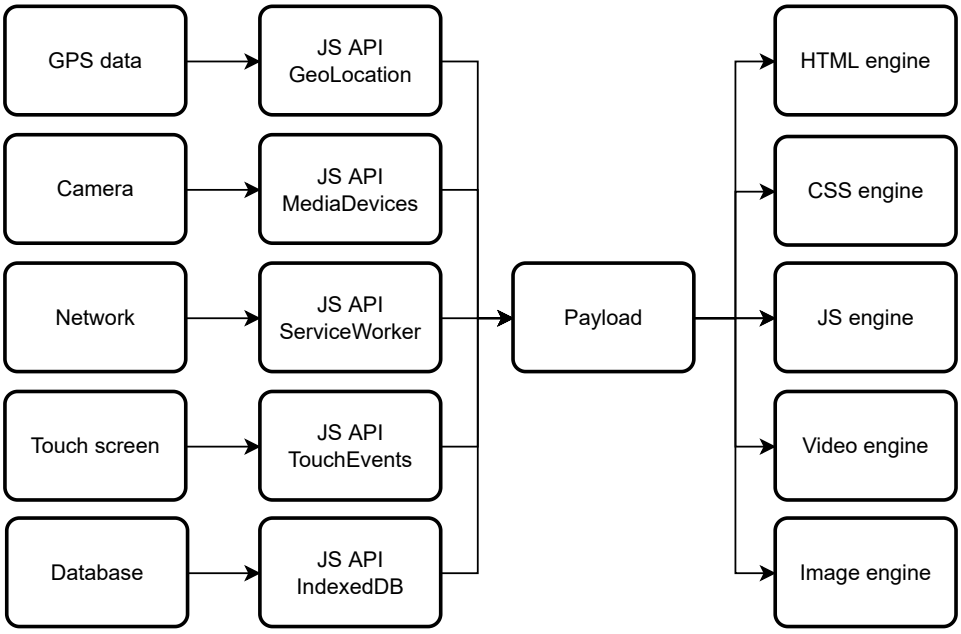

Figure 1.3: JavaScript APIs to access inputs

property to a prototype will impact all existing objects on the chain. JavaScript will climb the prototype chain on every lookup and return the first match. It combines weak and dynamic typing. Each line of code has a high potential impact as JavaScript engines perform type conversion without warning.

With this background in place, please reconsider the question of what constraints the web platform imposes. Consider our augmented reality app and its requirements concerning content and programming.

Content on screen can use existing fonts, images, audio and video formats. Custom elements can extend HTML, and CSS can target any markup. For 3d graphics, WebGL can render content via a canvas. Canvas also exists as a way of painting pixels directly on the screen. Arguably we should be able to render any content we desire with pixel precision if needed.

While content offers outstanding flexibility, our programming efforts will likely be more restrained. JavaScript has held a monopoly as the web programming language. While it is flexible, it is not performant. Compared to statically compiled languages, JavaScript is permissive. Research by Shull et al. [16] reflects how JavaScript becomes a performance bottleneck with increasing script sizes.

Inheritance using prototypes is suited to complex objects but inhibits primitive data. For example, consider the impact on numeric values:

- Storing numbers as objects adds overhead.
- Operations on numbers require checking prototypes.
- Prototype chaining is unwarranted for primitive values.

Intensive computation contradicts the JavaScript event loop model. Performing long-running calculations on its single thread will cause blocking. JavaScript can use workers for multi-threading. However, message passing must be via the queue or shared buffers. The former has no timing constraints, whereas the latter requires atomic operations. Furthermore, parallelisation may not be viable in all instances.

The restrictions mentioned above limit max performance. In contrast, a compiled language with static types restricts inputs. Compilers can use this additional structure to optimise execution. JavaScript just-in-time compilers cannot depend on this structure due to a lack of explicit typing. Assuming our application cannot execute quickly enough with JavaScript math operations. We would have to develop a native app instead. Fast maths at scale has not been viable in web apps.

In summary, JavaScript is impeded by:

- Strong object-orientation where everything is an object.
- Primitive values need object wrapping.
- Prototype inheritance has expensive lookups.
- Weak and dynamic types cause fluid object structure.
- It has a single user interface thread which blocks on long-running code. Thereby rendering the application unresponsive.

Altering the JavaScript programming model is challenging due to its broad adoption. Quirks have had to remain in place to ensure backward compatibility. In this sense, every peculiarity becomes a feature. An example is how a null value in JavaScript is an actual object.

Browser vendors have tried remedies via extensions. One example is Google Native Client, which introduced a sandbox to run native code.

Despite vendors' best efforts, previous attempts have failed because they lacked universal adoption. What was needed was a new web-native language. In 2015 browser vendors agreed on a new language called WebAssembly (WASM). It is a compiled language with static types. WebAssembly is used to augment JavaScript by offloading calculations. Significantly it can rely on universal support. By late 2017, Chrome, Firefox, Safari and Edge all shipped with WebAssembly. It is also part of Chromium.

As WebAssembly is a new language, its performance characteristics require research. The research question of this thesis is as follows:

Can WebAssembly improve web application performance?

WebAssembly will have to outperform alternatives to merit adoption. Performance is defined by:

- Consistency. Is performance stable without fluctuation?
- Peak throughput. What is the upper bound?
- Maturity. Does it function across engines?

The research question will be made concrete by two hypotheses.

1. WebAssembly offers better performance than JavaScript
2. WebAssembly offers parity to native performance

The thesis will proceed by presenting previous research in chapter 2. Chapters 3 and 4 introduce theoretical foundations and the design of tests. Chapter 5 contains results and discussions. Finally, the closing chapters contain conclusions and suggestions for future research.

Chapter 2

Previous research

This section provides a summary of research on JavaScript performance and WebAssembly. What follows are abbreviated conclusions from selected papers.

JavaScript has automatic memory management. Garbage collection requires tracking references on the heap. Cleanups require execution to pause for 10-30 milliseconds. These intermittent halts cause performance fluctuations. The engine manages cleanups, and their timing is not controllable. Degenbaev et al. [2] provide background on JavaScript garbage collection.

Ahn et al. [1] demonstrate how dynamic typing prevents efficient code generation. When changing the implementation of prototypes, they could reduce execution time by more than 30% in their tests.

Gal et al. [3] introduce just-in-time compilation. First, a monitor watches code to infer types. Subsequently, a compiler generates code using the inferred type information. Replacing bytecode with compiled code causes a 10x speedup.

Using just-in-time compilation requires swapping optimised code during execution. Techniques for on-stack-replacement are presented by Wang et al. [17]

Selakovic and Pradel [15] report performance measurements. JavaScript engine improvement is not stable. Engines will often improve in some aspects at the expense of others. Performance tends to increase on average

but shifts unevenly across releases. In addition, functionally equivalent APIs differ in performance and code generation is found to be sensitive. Slight changes in the source code can cause substantial impacts on performance, which indicates that JavaScript is difficult to optimise.

Just-in-time compilation has emerged as a best practice.
Taken together, these papers illustrate how:

- JavaScript performance increases with just-in-time compilation.
- Relying on an interpreter is slow.
- An interpreter is best used to generate the initial bytecode.
- A monitor watches execution to detect repeating segments.
- Compilation requires inferring types. Mimicking a type system.
- As compilation uses observed probabilities. Programmers can help by self-imposing type consistency.
- Replacing bytecode is known as optimisation. If objects change, compiled code has to bail out by de-optimisation.
- Scripts can get caught in optimisation/de-optimisation cycles with unpredictable performance.

WebAssembly is based upon work to impose static types. Its precursor was ASM.js. ASM.js functions by rewriting JavaScript to include type coercion. Removing ambiguity enables fast compilation by assured typing. ASM.js is a compiler-friendly subset of JavaScript. It exists as an LLVM backend which makes it a compiler target. Figure 2.1 illustrates an AMS.js pipeline built using LLVM. The most well-known pipeline is Emscripten, which compiles C/C++ into ASM.js format.

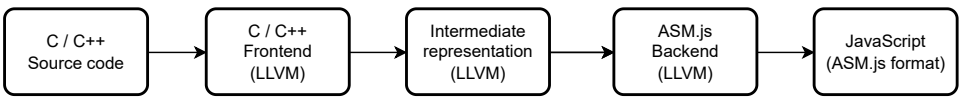

Figure 2.1: Emscripten LLVM pipeline for ASM.js

Emscripten provides a drop-in replacement for GCC via LLVM. Languages with static typing restrict input to suit ASM.js. The motivation behind ASM.js was to provide faster maths in JavaScript. [20] Its most notable success was porting a 3d graphics engine to run inside Firefox. The drawback

of ASM.js was relying on quirks in JavaScript. Basing a web standard on implicit type conversion would not be acceptable, which meant that ASM.js was merely a stop-gap solution.

WebAssembly (WASM) builds on ASM.js. It is a compiler target which outputs WebAssembly code instead of optimised JavaScript. WebAssembly instructions are virtual to support portability. Virtual instructions are mapped into native machine instructions by a compiler. Figure 2.2 illustrates how any source language can be trans-compiled into WebAssembly. WebAssembly functions as a universal intermediate format. A virtual machine provided by the local WebAssembly runtime will compile WebAssembly files into native machine instructions. The same WebAssembly file can therefore execute across different computer architectures.

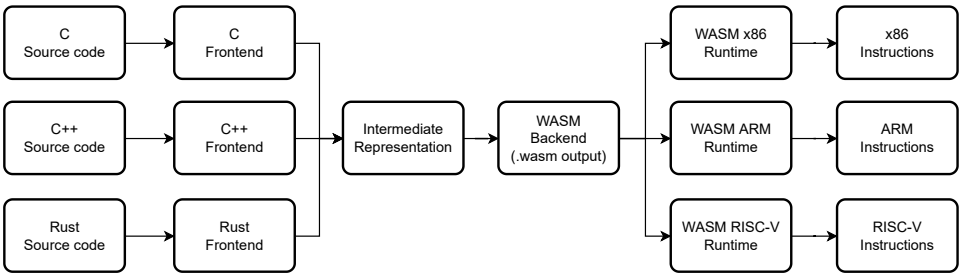

Figure 2.2: WASM compiler pipeline

Haas et al. [4] introduce WebAssembly. Authors sign as representatives of Google, Microsoft, Mozilla and Apple. Confirming adoption across their browsers and Chromium. WebAssembly has the following characteristics:

- Binary format. Executing via a stack.
- Static typing. With integers and floats.
- Virtual instruction set. Hardware independent.
- Streaming compilation. Processing code as it arrives.
- Memory is linear. Stored as an array.
- No registers exist. Everything is stack-based.

Figure 2.2 depicts how WebAssembly fits into an LLVM compiler chain. Due to LLVMs' three-piece structure, existing front-ends can target WebAssembly and convert Python, C, and Rust codebases into the WASM

file format.

WebAssembly can emulate any program by building new data types using its binary format. The format is universal, but its motivation is to offload demanding JavaScript. WebAssembly engines exist inside V8, SpiderMonkey and JavaScriptCore.

Existing research into WebAssembly has been concerned with safety, performance and adoption. Safety is the responsibility of the runtime. Modules separate code and are analogous to individual scripts in JavaScript. The runtime sandboxes each module individually. Access to resources or APIs requires passing handles into the module. Modules start from the lowest possible authority. Melicher et al. [11] provide background on the security model.

Stepwise reduction rules define execution. Abnormal behaviour causes the runtime to trap and shut down. Watt [18] provides proof for the reduction rules in WASM 1.0. While formal verification confirms semantics, correct language specification is insufficient for security. Each runtime will also have to implement all checks. Failure to conduct checks will fail to halt offending programs. Perenyi and Midtgaard [14] apply fuzzing techniques and report crashing bugs in both JavaScriptCore and SpiderMonkey. Johnson et al. [8] designed a verifier to analyse memory isolation in an x86 compiler before code execution. While sufficient checks are crucial to security, an attacker can also break the integrity of the runtime by targeting hardware. Naryan et al. [13] consider possible side-channel attacks and propose mitigations for WebAssembly.

Vulnerabilities also result from mismatching abstractions. Classic exploits in C code, such as stack and buffer overflows, resurface in WebAssembly binaries. C compilers have evolved to insert stack canaries and rewrite unsafe functions. However, converting C code using a WebAssembly backend will not rewrite vulnerable code, as the WebAssembly backend in LLVM is unaware of the source language. A code review must therefore take place before conversion into WebAssembly. Lehman et al. [9] provide additional details on these topics.

Early studies by Haas et al. [4] and Zakai [21] report performance as better than JavaScript and occasionally on par with native. WebAssembly

proves to be 30% faster than ASM.js in tests. Benchmarking suggests WebAssembly can execute within 2x of native code across 3d graphics, file compression and physics simulations. Furthermore, 7 out of 24 tests in PolybenchC run within 10% of native.

The previous benchmarks do not require system calls. To extend the scope of tests Jangda et al. [7] modify a Unix-compatible kernel to run inside Chrome and Firefox. Performance is close to 50% of native across 15 benchmarks from the SPEC-CPU test suite. Again results confirm WebAssembly to be around 30% faster than ASM.js. The authors develop a harness to analyse WebAssembly against native code. They report the following limitations:

- WebAssembly requires security checks. Every memory access is verified to fall within array bounds. The runtime also has to check function signatures before each call. Mandatory checks require more branches and conditional instructions.
- WebAssembly omits registers. Actual WebAssembly compilers use physical registers but display less sophisticated registry allocation than native. Because WebAssemly engines reside inside JavaScript engines, they also suffer shortfalls because of garbage collection. Registers are continuously reserved to aid garbage collection in JavaScript. As a result, inefficiencies in registry use cause more load/store operations.
- WebAssembly virtual instructions are less efficient than x86 instructions. WebAssembly executes 1.8x more instructions than native code.

The authors conclude that improved code generation is required to enhance performance.

WebAssembly compilers in V8 and SpiderMonkey use tiered compilation for fast startup. A baseline compiler performs initial code generation. Subsequently, an optimising compiler spends time refining the code. Yan et al. [19] report a slight speedup of 1.1x with tiered compilation. However, some tests only achieve parity, and some even show slight decreases. Isolating the optimising compiler provides the best results. Its execution time was 0.9x relative to compilers working in tandem. Swapping code requires time and results indicate that tiered compilers do not offer enough improvement to offset this cost. However, Hall and Ramachandran [5] report

that execution time for their benchmarks improved by 50% through updates to V8. This result indicates that compilers appeared to evolve quickly at that time.

The first research on adoption stems from 2019. Initial results from Musch et al. [12] reported that the most common use was crypto-jacking. However, a later study by Hilbig et al. [6] suggested crypto had fallen to less than 1% of WebAssembly binaries. With WebAssembly being a compiler target. One question is which source languages dominate. Hilbig et al. [6] find that two-thirds of binaries stem from C or C++ code. Rust is the runner-up with 15% of binaries. There is no clear contender for third place. The most popular toolchain is Emscripten, based on LLVM. Nearly 80% of binaries originate from LLVM using a WebAssembly backend. The study [6] found 8461 unique binaries as of 2021. Content processing and games were the most common use cases, with a share of 30% and 25%. The remaining use cases spanned a wide range and did not reveal precise results.

Helping researchers analyse performance requires instruction counting and call graph extraction. Lehman and Pradel [10] present a dynamic analysis tool for WebAssembly. It instruments binaries by inserting hooks. Every action on the stack is replicated and logged. The runtime overhead is close to 150x when monitoring every WebAssembly instruction. Full instrumentation of binaries tends to increase their size by nearly sevenfold. Using more selective instrumentation reduces these figures. When monitoring only function calls, the runtime overhead is closer to three times that of standard.

Chapter 3

Methodology and theory

The previous section introduced just-in-time compilation as a theoretical breakthrough in JavaScript performance.

In theory, JavaScript is:

- Interpreted with each line evaluated in real-time.
- Weakly and dynamically typed with implicit conversion.
- Object-oriented with prototype chaining. No primitive data types exist in JavaScript.

When using just-in-time compilation a JavaScript engine will try to mimic statically typed languages by opportunistic compilation. Engines chain several compilers to form a sequence. For example, SpiderMonkey and V8 use a two-tiered structure with one baseline and one optimising compiler. The baseline compiler provides fast startup, while the optimising compiler performs additional passes to refine code. Code swapping requires using on-stack replacement techniques to swap new code as it becomes ready. Swaps have an associated time cost, and improved execution speed must offset the time penalty of swapping code. JavaScriptCore also uses tiered compilation but chains up to 4 compilers through its pipeline.

Compilation functions best when objects are consistent. Code is invalidated when objects change shape, forcing engines to discard optimised code and return to interpreting. When objects change, optimisation cycles cause uneven performance. At worst such cycles can incur penalties that make compilation counterproductive. Figure 3.1 provides a high-level overview of a just-in-time compiler pipeline.

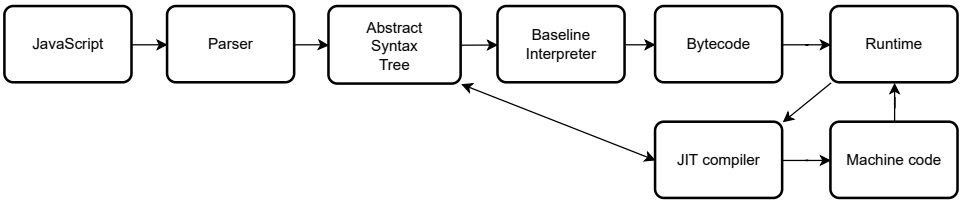

Figure 3.1: Tiered compilation pipeline

Would it be possible to address these shortcomings by trans-compiling a static language into JavaScript? Unfortunately, this approach offers no guarantee because type information vanishes during compilation. The exception is ASM.js which uses type coercion for numbers. This approach is only suitable for numbers and does not extend to complex objects. As a result, engines must try to impose implicit structure by inferring types through observing execution. Type inference is complicated because JavaScript objects are fluid and can change by altering the actual object or any link on its prototype chain. Therefore engines must continuously monitor object shape to ensure the compiler will bail out on changes.

While adhering to type consistency can assist just-in-time compilers, it does not impose explicit structure. The problem of inferring object shape remains. JavaScript uses prototype inheritance which is time and memory intensive. Accessing a property with class-based inheritance executes in O(1) while traversing prototypes is O(n). Adding a property on the prototype chain affects all objects beneath the prototype. With implicit type coercion, interpreters must set aside additional space to modify objects quickly. Engines also try to pre-compute results to increase throughput. Casual observation suggests that 1 MB of JavaScript can cause Safari and Chrome to reserve more than 50 to 100 MB of memory. Note that this is a measure of reserved capacity and not actual utilisation.

In summary, just-in-time compilation adds overhead both in time and space. Objects reside in boxed structures, with space to modify, which creates overhead from storage. In addition, compilers need to infer types by observing execution. Type inference requires scripts to warm up before they can be optimised, reducing code generation efficiency. Continuous monitoring also adds overhead because it requires maintaining counters.

Explicit types would remedy these shortcomings. However, imposing static typing in JavaScript was not thought practicable. Browser vendors decided a new format would be required and accepted WebAssembly as a new standard.

WebAssembly is statically typed and compiled.

A recap of its model as presented by Haas et al. [4]:

- Binary format which is portable across platforms.
- Hardware independent with a virtual instruction set.
- Virtual instructions are translated to native instructions by the runtime.
- Stack-based operations without registers.
- Native to the browser and implemented inside the JavaScript engine.
- Modules are suitable for sandboxing. Each module has its own execution context.

WebAssembly is a compiler target. Therefore, programmers should not write modules by hand. Most WebAssembly builds are by LLVM. While WebAssembly modules can run independently, this thesis does not consider independent runtimes. The focus is on using WebAssembly as a tool to offload computation in web apps. Figure 3.2 exemplifies WebAssembly module instantiation from JavaScript. JavaScript is responsible for starting WebAssembly modules when executing in browsers. JavaScript and WebAssembly functions are interoperable as WebAssembly modules can import and export functions.

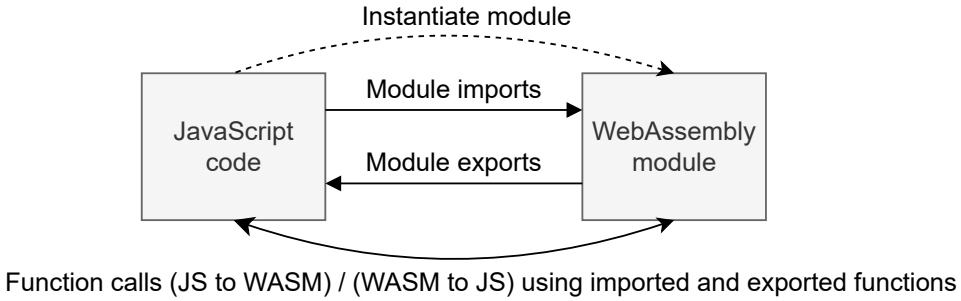

Figure 3.2: JavaScript and WebAssembly interaction

Requirements for WebAssembly 1.0. Hardware must support:

- 4 types of data. 32/64 bit integers and floats.
- 32-bit memory addresses, max four Gb of memory
- Common load/store and mathematical operations
- Standard 2 complements logic

WebAssembly provides "pancake" programming via stack operations. Operations run by small step reductions. Figure 3.3 and 3.4 shows WebAssembly calculating 1 + 5 + 6 = 12 in steps. Figure 3.3 shows the actual small step reductions. Figure 3.4 shows each step within a tree structure.

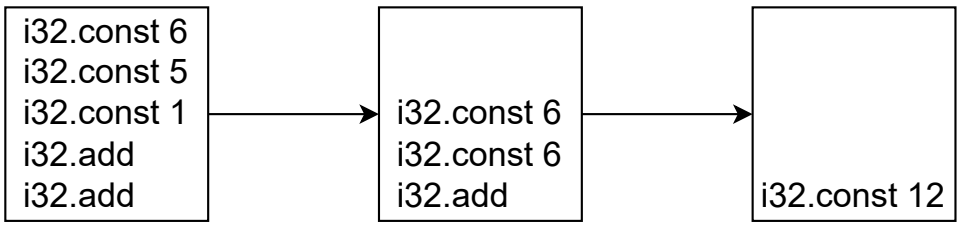

Figure 3.3: WebAssembly stepwise reduction

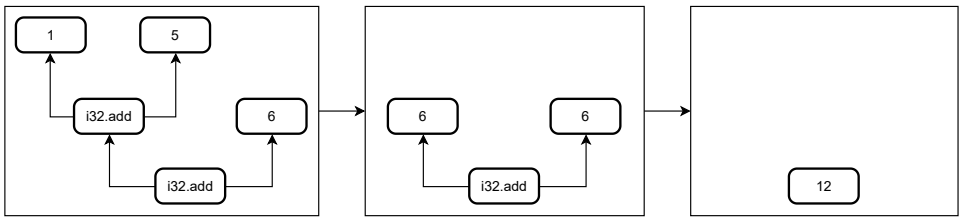

Figure 3.4: WebAssembly stepwise reduction within tree structure

WebAssembly executes on a virtual machine with a managed stack. The virtual machine manages both the stack and the stack pointer. Therefore, it is not possible to peek down the stack or directly manipulate the stack pointer during execution. This obfuscation is intentional to avoid exposing the stack or the stack pointer. The runtime also restricts random jumps. Execution is structured, and there is no go-to statement. This design intends to restrict control flow during execution.

Memory exists as an array without gaps. Every lookup into memory uses an array index. The same pattern holds for function calls, where each call uses an index in a lookup table. The runtime hides memory addresses, and

only indices are exposed. Each memory access is verified to fall within array bounds. Function calls must match against their signature. The return value of a function is verified to match its signature when exiting the function. These inspections result in continuously monitoring values on the stack. Figure 3.5 demonstrates memory and function lookups using indices.

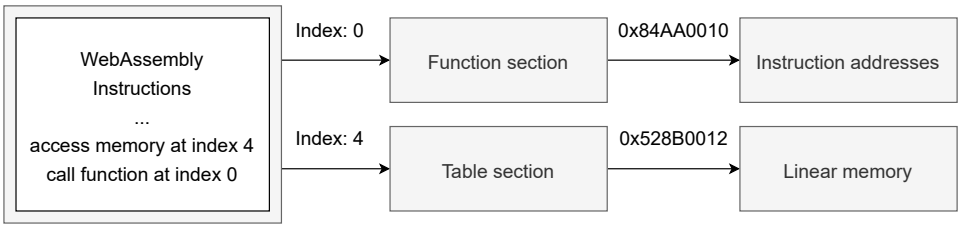

Figure 3.5: WebAssembly lookups by indices

Code execution requires running a module. Modules are self-contained and analogous to individual scripts in JavaScript. Modules have the following characteristics.

- Each module executes independently.
- No mandatory sections.
- Known sections are defined. Using them is optional.
- Known sections must follow an order.
- Custom sections can be defined arbitrarily.
- Modules can import and export functions and memory.
- Imports and exports can interact with JavaScript.

Figure 3.6 shows an example of module structure. This example includes every known section defined within the WebAssembly 1.0 specification. The key takeaway is that the module has imports and exports defined in separate sections. In addition, data shared across functions reside in a separate section under the global label. Finally, modules support single pass validation by sharing function signatures. The shared signatures are used as placeholders in code to enable parallel processing.

WebAssembly modules can import functions from JavaScript. This feature is key to performing input and output operations. These require

passing through JavaScript as it is the browser's gateway. Previous research [15] documented how large surface areas complicate JavaScript engine development. Improvements were uneven due to interdependencies. WebAssembly can potentially help resolve these issues as it offloads work into a separate engine, supporting de-coupling and isolation.

As WebAssembly 1.0 only supports numbers as datatypes. It is most suitable for calculation. WebAssembly can emulate any code by building new datatypes, such as UTF-8 encoding, but its format inherently favours numerical workloads.

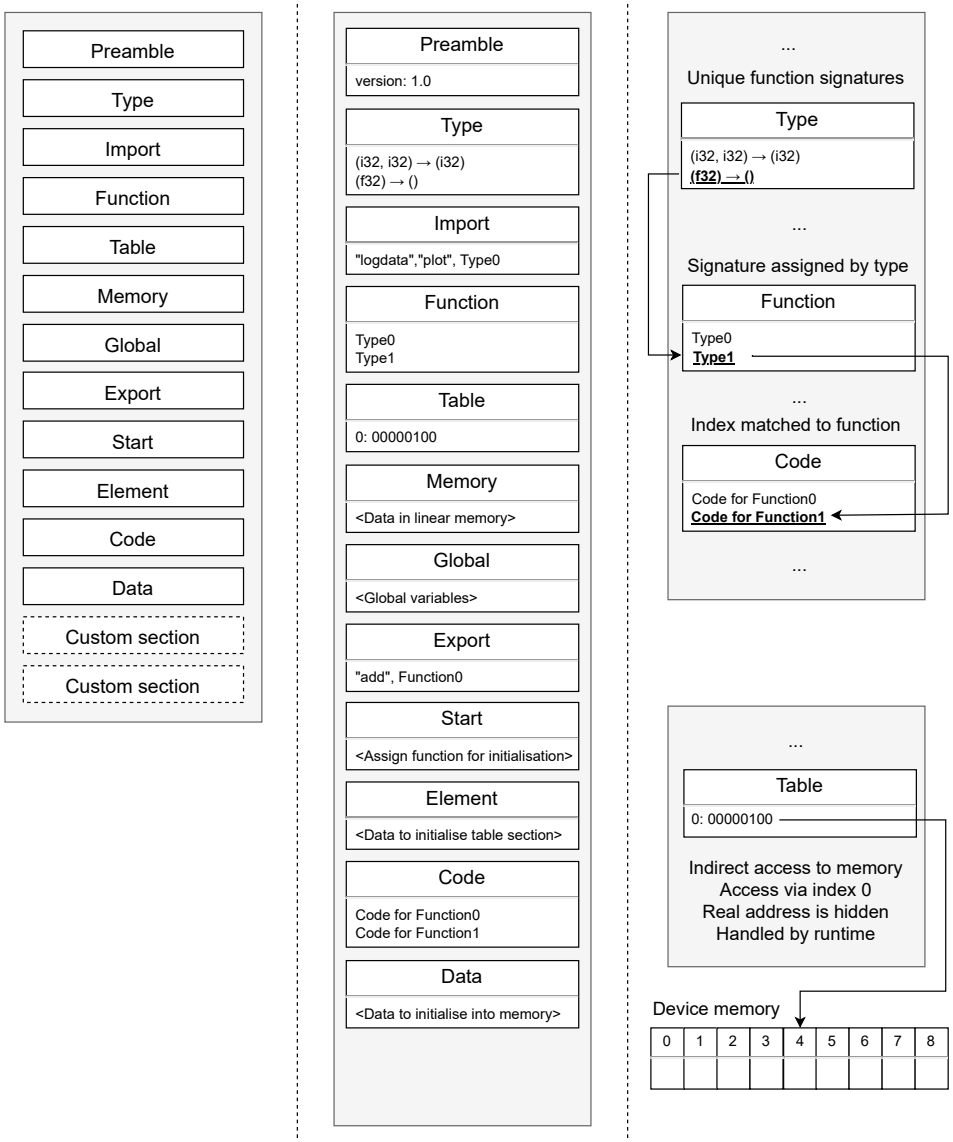

Figure 3.6: WebAssembly module structure

Browser engines differ in their implementation of WebAssembly. All engines use tiered compilation, but JavaScriptCore uses an interpreter first. JavaScriptCore executes with no compilation guarantee. A WebAssembly module could run using only interpretation throughout its lifetime. JavaScriptCore is unique in using an interpreter. Both V8 and SpiderMonkey move straight to compilation. V8 uses Liftoff as a baseline. With TurboFan as its optimising compiler. TurboFan runs in the background, and its refined output replaces that from Liftoff as it becomes ready. SpiderMonkey uses the same structure. JavaScriptCore uses two compilers behind an interpreter. First, the interpreter generates the initial bytecode. Subsequently, a monitor decides which functions will proceed onto the compiler pipeline. The first stage is a fast compiler (BBQ), and the latter is an optimising compiler (OMG). With no compilation guarantee, parts of WebAssembly code executed by JavaScriptCore may not proceed beyond the initial bytecode.

Reflecting on how WebAssembly can improve upon JavaScript compute performance:

- WebAssembly uses static types based on numbers. In contrast, numbers in JavaScript are objects stored in 64-bit binary format IEEE 754. JavaScript also offers BigInt as a datatype to contain numbers which exceed the 64-bit format.
- Compilation into WebAssembly is ahead of time.
- WebAssembly should offer faster memory management. It omits prototype inheritance and stores arguments on the stack. In contrast, JavaScript depends on the heap for objects, prototypes and function arguments. Garbage collection in JavaScript requires checking references and pausing execution.
- WebAssembly enables offloading calculations into a different runtime. Each WebAssembly module has its own execution context. JavaScript only has to call a WebAssembly function to free up its runtime. Long-running calculations on the user interface thread will otherwise freeze apps. While JavaScript supports multi-threading by workers, it has weak timing constraints.

Reflecting on whether WebAssembly can offer parity to native performance. Its hardware-independent format imposes restrictions. Comparing the WebAssembly execution model to that of native x86:

- No registers exist in WebAssembly. Everything is stack-based.
- Less expressive language with fewer specialised instructions.
- Mandatory checks for memory lookups and function calls. Alongside continuous monitoring of stack operations.

The implications of these restrictions are likely to be:

- Increased instruction count due to less specialised instructions.
- Checking requires branches. Further increasing instruction count.
- Less efficient addressing due to lack of registers. Likely to cause more load/store instructions and cache misses.

Recent versions of WebAssembly have added new instructions. Vector operations are critical to fast maths, and WebAssembly 2.0 supports fixed-width single instruction multiple data (SIMD). SIMD operations use a new datatype V128 which is a vector of 128 bits.

Fixed width refers to the constant input size. The vector is divided into lanes to pack multiple values. For instance, it can represent four 32-bit values or eight values of 16-bits. Whenever inputs do not require the full width, the compiler can leave parts of vectors empty. Figure 3.7 shows the addition of four values across two V128 vectors, i.e. one SIMD operation with four lanes.

The methodology of this thesis will be to benchmark WebAssembly. Recording data to evaluate two hypotheses.

1. WebAssembly offers better performance than JavaScript
2. WebAssembly offers parity to native performance

The approach is quantitative and explorative, as measurements require adjusting parameters and registering outcomes. Each test will use execution time as the deciding factor in hypothesis testing. In the first instance, the null hypothesis is that WebAssembly does not offer an improvement. Such that execution time is equal. In the second instance, the null is that WebAssembly and native have equal performance. In each case rejecting the null requires performance to differ significantly. These hypotheses are suitable for t-tests.

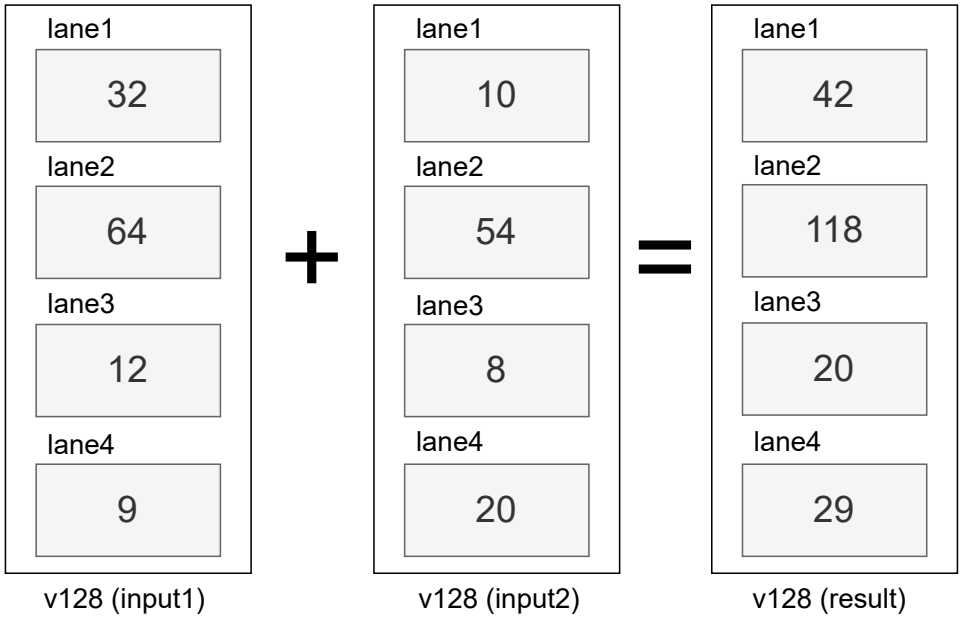

Figure 3.7: WebAssembly SIMD with four lanes in 2 vectors

Hypothesis testing will use objective metrics, but these measurements do not provide a causal analysis. A thorough study also requires an analysis pipeline intended to document causal links. This research will only consider the x86 instruction set, and all the tests will use benchmarks compiled into WebAssembly from C source code. Figure 3.8 presents the relevant WebAssembly pipeline. Proper analysis across this pipeline will require combining results from both static and dynamic code analysis.

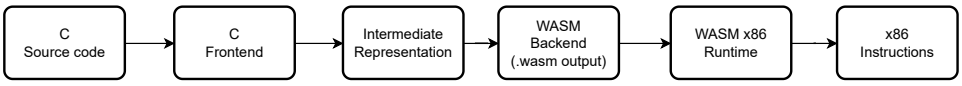

Figure 3.8: WebAssembly pipeline used for tests

WebAssembly is a virtual instruction set, and it is crucial to study how WebAssembly instructions translate into x86 instructions. Key questions relate to efficiency and execution speed. Because WebAssembly instructions are architecture-independent and less specific, WebAssembly modules will likely use more instructions than equivalent x86 binaries. However, if this is the case, WebAssembly runtimes may still compensate if they can select fast instructions. Instruction mix will therefore be an essential consideration.

Because WebAssembly only executes via stack operations. There is a particular interest in understanding how WebAssembly engines manage registry use. The WebAssembly runtime must independently adapt the WebAssembly instructions to the physical architecture as it sees fit. Recall that WebAssembly is a universal intermediate format. It makes no allowances for any specific computer architecture. Each WebAssembly runtime is entirely responsible for mapping a WebAssembly module to hardware. There are no compiler hints included in modules. Hence, it is necessary to perform taint analysis for registry use and consider both registry pressure and cache utilisation. Cache utilisation must also consider effects throughout the cache hierarchy across L1, L2 and L3 levels. This analysis is required to consider the relative cost of cache misses for WebAssembly modules against native x86 binaries.

The research relies on combining data across different sources.

- Static code analysis is possible by using debug flags in WebAssembly engines. Flags enable engines to output their generated x86 code.
- Tracing WebAssembly instructions requires instrumenting binaries. Instrumentation forces a WebAssembly module to record each instruction executed via its stack. The result is a complete trace with call graph extraction and instruction counts.
- Observing the translation of WebAssembly instructions into x86 instructions requires attaching a harness to the WebAssembly engine. This harness will register the instructions which exit the CPU pipeline. This harness also extends its data collection to record cache misses, CPU cycles and branch counts.

Figure 3.9 illustrates data collection across these sources. The final step combines the data using taint analysis and observing frequencies. Taint analysis involves considering the number of times a registry is accessed. Which merely requires aggregating the number of accesses per register across instruction traces. When comparing the results across all registers, this frequency analysis also yields a measure of relative spread in registry use. In addition to frequency analysis, the research also employs established statistical techniques, such as linear regression and studying correlations.

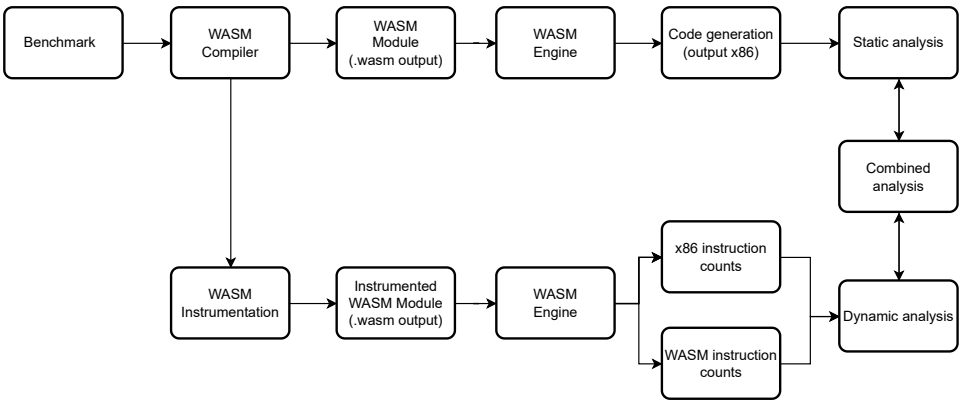

Figure 3.9: WebAssembly analysis pipeline

In order to obtain comparable data for native x86 binaries, a test harness collects data while benchmarks are executing via the command line. Both native and WebAssembly binaries have traces collected using the same tools to provide comparable metrics. Figure 3.10 depicts the pipeline for data collection with native binaries. Note that these techniques are also applicable to analysing JavaScript engine performance. For example, JavaScript engines support code dumps through debug flags to inspect generated code. At times these techniques were used to cross-check commonalities across WebAssembly and JavaScript engines. Some WebAssembly engines replicate parts of JavaScript engine structure.

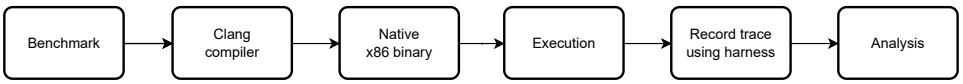

Figure 3.10: Native binary analysis pipeline

Although combining static and dynamic analysis provides a robust basis for inference. It is necessary to state that a considerable degree of judgement is involved in summarising conclusions from this research. While the analysis tries to identify the most critical factors and establish causality, there are often minor inconsistencies within a broad dataset. As a result, qualitative findings intend to complement the objective findings but do not offer unambiguous conclusions.

Chapter 4

Design of tests

As outlined in the introduction, the research question of this thesis is whether WebAssembly improves web application performance.

Performance measurements will consider multiple axes:

- Consistency. Is performance stable without fluctuation?
- Peak throughput. What is the upper bound on performance?
- Maturity. Does it function across engines?

Execution speed and its standard deviation provide measures for throughput and consistency. Speed is also related to the size of an application because a larger payload requires more time to transfer across the network. WebAssembly uses a binary format which should be compressible. Therefore the effects of compression on code size also need to be considered.

Maturity is concerned with whether code can execute as intended across all engines. Web applications lack universal access if some engines only offer partial support. Recall that universal access via a common codebase is the most significant advantage for web applications. Programmers must consider the worst-case scenario when developing their applications. Therefore maturity is concerned with the lowest common denominator. The least capable engine defines the lower bound, and measurements of maturity are concerned with the least consistent and lowest output recorded in the tests.

Arguably maturity is less important than consistency and peak throughput. Engine performance should converge across time because each engine is open source, and vendors can freely imitate their competitors. No patents

or trade secrets provide a unique advantage. Therefore engine performance should be a question of time spent in implementation. Accordingly, this thesis is most concerned with peak throughput. While this argument is an interesting conjecture, whether it holds is not vital to interpreting results as they stand.

The research question is made testable by two hypotheses.

1. WebAssembly offers better performance than JavaScript
2. WebAssembly offers parity to native performance

The scope is limited to WebAssembly as a tool to offload computation in web applications. Accordingly, tests only consider engines available in browsers, and all benchmarks are limited to numerical computing. No tests require kernel support. Such that there are no system calls or input/output operations. Tests will only consider the x86 instruction set.

Restricting tests to numeric data without system calls implies measuring WebAssembly in a best-case scenario. Metrics must provide a snapshot in time suitable for hypothesis testing. It is also desirable to investigate trends. For example, assuming WebAssembly is not more performant than JavaScript, or whether it is close to matching native performance. What is the rate of improvement? Is it likely to catch up?

The answers to these questions rest not just on analysing trends but also on trying to predict future development. Therefore it is also necessary to analyse the WebAssembly execution model. Uncovering factors that restrict performance and consider whether these are due to WebAssembly design decisions or dependent upon engine implementation.

In summary, tests need to cover:

- Hypothesis testing at a fixed point in time.
- Assessing the rate of improvement using longitudinal data.
- Studying the WebAssembly execution model and its implementation.

Tests need to use a range of numerical operations. The PolyBenchC suite provides a broad set of mathematical tests. It offers varied workloads across 30 benchmarks. These benchmarks are small scientific kernels performing well-defined tasks, such as calculating correlations, matrix opera-

tions, and triangulation. There are also more specific applications, such as edge detection filters and finding the shortest paths between each pair of nodes in a graph. Descriptions of each benchmark are available in table 7.1.

PolyBenchC benchmarks offer mixed runtimes, which make them suited to evaluate startup and code refinement in different conditions. Source code is in C and compiled into JavaScript and WebAssembly using Emscripten via LLVM. This pipeline represents the most common use of WebAssembly today. Around 80% of binaries follow this route. [6] Compilation for the native binaries will be via Clang.

Using Emscripten via LLVM requires selecting parameters for compilation. Support exists for different optimisation levels. Levels include O0, O1, O2 and O3. Higher levels perform additional passes over code, and size tends to drop with each level. The analysis will also consider trade-offs of size against performance.

Engines subject to tests comprise V8, SpiderMonkey and JavaScriptCore. Other browsers re-use these engines, e.g. through Chromium.

Each benchmark will be executed five times in each runtime. The standard deviation across the five runs provides a measure of consistency. Execution time will use the median across five runs. Tests generate 600 data points for JavaScript execution, 600 data points for WebAssembly execution and 120 data points for native execution. A total of 1320 observations.

Estimating trends requires repeating the tests using different engine versions. Therefore, the benchmarks will be re-run across different releases to compare results for the best-performing engine.

In considering WebAssembly implementation, the influence of tiered compilation is interesting. Just-in-time compilation provided a foundation to increase JavaScript performance. WebAssembly engines mimic this structure. Research on just-in-time compilers underscores how on-stack replacement is critical to offset the time-cost of refinement [17]. However, one WebAssembly study reports no substantial gain from using tiered compilation [19]. Tests should evaluate tiered compilation by isolating the baseline and optimising compilers. Whereas V8 and SpiderMonkey use tiered compilation. JavaScriptCore places an interpreter ahead of its tiered

compilers. Therefore, parts of WebAssembly modules executed by JavaScriptCore may not proceed to compilation and JavaScriptCore is not suited for these tests.

Studying WebAssembly execution will focus on how compilers map WebAssembly instructions into x86 instructions. The previous section on research methodology provided a high-level overview of the analysis pipeline. The following clarifies the pipeline implementation.

WebAssembly uses a virtual instruction set, so the instruction traces must span two levels. The first level is inside the WebAssembly module. The second level is the runtime. Merging records across both levels uncovers how WebAssembly instructions map into x86 instructions. It is also necessary to instrument native x86 binaries to collect traces from their execution. Comparing traces across native and WebAssembly will enable comparing code generation. This data is required to distinguish code efficiency and registry use across WebAssembly and native.

Apple Instruments enables recording traces for the x86 instructions. It supports collecting counts for instructions, cycles, branches, loads/stores, and cache entries. Native code is run via the command line, while WebAssembly modules run in Node.js. Node.js is prefered as it uses the V8 engine and offers better isolation. Background processes in browsers would otherwise impact measurements.

To ensure results are consistent. Tests have also run on a Linux virtual machine using the same hardware. Emulation was via Parallels Desktop 17 using Ubuntu 20.04. With data collection using Intel PinTools and Perf. Results from traces and instruction counting mirror Apple Instruments. Although the virtual machine adds some overhead, the results were in line with those recorded by Apple Instruments.

Apple Instruments is part of the Xcode developer tools. Apple Xcode also includes tools for code inspection. These tools provide insight into code generation. For example, enabling comparison of WebAssembly and native x86 instruction selection and register use.

Analysing execution also requires dynamic analysis inside WebAssembly binaries. The Wasabi framework provides a harness to record all

WebAssembly instructions. Wasabi instruments WebAssembly binaries by inserting hooks, calling out to JavaScript and recording data outside the WebAssembly module. This procedure yields a complete instruction count and can also provide call graph extraction for stack tracing.

The theory section outlined why WebAssembly may suffer an impediment compared to native code. Data collected will answer to:

- Whether WebAssembly requires more instructions than native code.
- WebAssembly instruction mix and its performance impact.

Even if WebAssembly requires additional instructions, individual instructions can be faster. Effects from the instruction mix can offset instruction counts when considering execution time for a module. Data must support analysing the relationship between WebAssembly instruction counts and execution time.

Lastly, the theory section explains how SIMD instructions have extended WebAssembly. Support exists in V8 and SpiderMonkey at this time. SIMD instructions supported by WebAssembly fit with sequence alignment. This workload is typical in bio-informatics. For instance, DNA sequences are strings of four letters A, T, C, and G. Sequence alignment can suggest whether species share DNA. Smith-Waterman is an algorithm in this space. It tries to find the best alignment across two sequences. It results in a scoring matrix with dimensions equal to sequence lengths. The best alignment is found by backtracking from the highest score. Figure 4.1 provides an example. Running the algorithm requires defining rules for scoring and obtaining actual sequences. A substitution matrix is required to calculate the score for each pair of values. These tests use a standard substitution matrix commonly used in bioinformatics.

The substitution matrix is of no particular importance to the research in this thesis. However, interested readers may care to know that the different scores reflect how A, T, C and G fit together in particular ways. There is a complementary nature between the base pairs in DNA. A fits with T, and C fits with G. This artefact is why DNA forms strong bonds with the distinct double helix shape. Double-stranded DNA consists of two polynucleotide chains with nitrogenous bases connected by hydrogen bonds. Each strand will mirror the other due to the antiparallel orientation of the sugar-phosphate backbones, which makes up the molecular structure of

DNA. The substitution matrix assigns a positive value if the bases are a match and a negative score if they are not.

The algorithm performs a search by trying each alignment and calculating the scores according to the substitution matrix. The previous discussion of the scoring matrix and sequence alignment discloses the time complexity of the Smith-Waterman algorithm. It has quadratic complexity both in time and space. The algorithm tests each position (N) in the first sequence against every position (M) in the second sequence, which amounts to O(N x M) complexity.

The key takeaway is that the algorithm will use SIMD instructions to perform multiple matrix operations. Essentially it is trying to match two long vectors by sliding them alongside each other and counting the scores. Each position's score depends on whether the base pair fits with biology, as reflected by the substitution matrix. It is also possible for the algorithm to insert gaps. This option is part of the substitution matrix, allowing for sequences of different lengths. The source code is in C. Compilation to native is via Clang. Smith-Waterman is compiled into WebAssembly using Emscripten. Modules are generated both with and without SIMD support.

Tests in summary:

1. Run PolyBenchC benchmarks compiled by Emscripten into JavaScript, WebAssembly, and Clang for native code. Evaluate WebAssembly performance versus JavaScript and native.
2. Compare WebAssembly performance across time. Consider the pace of improvement for the best engine.
3. Compare results using tiered, baseline and optimising compilers. Run tests across SpiderMonkey and V8. Report on the impact of tiered compilation.
4. Collect data on instruction counts, branching behaviour and CPU cycles. Evaluate how WebAssembly execution compares to native and consider whether it has an impediment due to its execution model.
5. Perform sequence alignment with and without SIMD instructions. Assess the impact of SIMD on execution speed.

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	5	3	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	6	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

Figure 4.1: Smith-Waterman alignment with two sequences

Chapter 5

Results and discussion

This section will present the data from tests and provide additional discussion. These results derive from the analysis pipeline outlined in the research methodology chapter. Before presenting the results, the following contains an inventory of hardware and software used throughout testing.

Data is collected using a MacBookPro 16 - 2019 model year:

- Intel Core i7 with six cores (Base 2.6 GHz - Turbo up to 4.5GHz)
- 16GB RAM (2666MHz DDR4 onboard)
- AMD Radeon Pro 5300M with 4GB of GDDR6 (UHD Graphics 630)
- Connected to auxiliary power
- macOS Monterey version 12.2

Compiler specification was:

- GCC (Native) Clang v13.1.6
- EMCC (Emscripten) v3.1.8 with LLVM v15.0.0

Browsers used in testing:

- Chrome version 100.0
- Firefox version 99.0.1
- Safari version 15.3

Each was a standard production release.

Compression is via gzip, using Apple gzip 353.100.22.

The first set of tests is to run PolyBenchC benchmarks compiled by Emscripten into JavaScript, WebAssembly, and Clang for native code. Then use the results to evaluate WebAssembly performance against JavaScript and native.

Table 5.1 summarises the results for JavaScript, WebAssembly and native across optimisation levels. All JavaScript engines recorded their best result at the O2 level, and JavaScriptCore was the fastest across all levels for JavaScript. Note that these tests run on an Intel x86 processor rather than one of Apple's own processors. The strong performance of JavaScriptCore is not due to hardware advantages. Each engine is afforded the same access level and privileges in the operating system. Therefore these results provide a like-for-like comparison across engines.

JavaScriptCore records an impressive performance executing JavaScript but delivers lacklustre results when executing WebAssembly. It even fails to complete all benchmarks in the PolyBenchC suite, as the benchmark ludcmp fails to complete within 3 minutes. This benchmark performs a lower-upper decomposition across a matrix followed by a forward substitution. Unfortunately, JavaScriptCore gets stuck in a loop during its execution and is unable to complete it. At worst other engines complete this benchmark in less than 20 seconds. Even with runtime for ludcmp set to 0 seconds in the JavaScriptCore dataset, it falls considerably behind V8 and SpiderMonkey across O1, O2 and O3 optimisation levels.

It would be interesting to consider why JavaScriptCore records such poor performance for WebAssembly. First, recall that JavaScriptCore uses an interpreter in front of its chained compilers. This design intends to provide fast startup, but does it function as intended? In answering this question, JavaScriptCore is locked to use its interpreter, while V8 and SpiderMonkey are locked to use their baseline compilers. Re-running tests with PolyBenchC indicate JavaScriptCore is slightly ahead in benchmarks which complete in less than half a second. However, the baseline compilers catch up and surpass the interpreter when moving beyond the half-second mark.

This result raises the question of whether using an interpreter is warranted for WebAssembly. Averages taken across benchmarks show JavaScriptCore interpretation to be around seven times slower than V8 using tiered compilation. This result is verified by cross-checking against Wasm3, which ex-

ecutes WebAssembly by interpretation inside a stand-alone runtime. Results indicate that an interpreter is typically seven to eight times slower than tiered compilation and around three to four times slower than baseline compilers. In summary, interpreter execution speed is slow, but its faster startup is significant. Compiler startup time puts interpreters ahead until the half-second mark, and these results indicate that an interpreter may be a sensible approach.

Closer inspection reveals that the conceptual differences between the interpreter in JavaScriptCore and the baseline compilers used in V8 and SpiderMonkey are not all that large. All engines perform similar single pass validation and code generation. V8 and SpiderMonkey use a fast single-pass compiler as their baseline. While JavaScriptCore uses an interpreter, which translates a WebAssembly module into an intermediate representation called Llint. Llint is a proprietary bytecode used as the first stage in JavaScriptCore. JavaScriptCore uses Llint bytecode as the first stage both when executing JavaScript and WebAssembly. V8 and SpiderMonkey also use rewrites via bytecode during code generation. Therefore there are commonalities across the JavaScriptCore interpreter and its competing single-pass compilers.

Certain peculiarities in the WebAssembly format suggest it is somewhat difficult to interpret. Recall that a WebAssembly module contains instructions from a virtual instruction set. The modules are in binary format, but this binary encodes bytecode instructions. WebAssembly bytecode is translated into actual machine instructions by the runtime. In this sense, WebAssembly modules are programs written in a universal bytecode, much like the approach used in Java for its virtual machine.

WebAssembly bytecode is somewhat unusual in its control flow. Recall that WebAssembly has no go-to or jump instructions. Instead, it has a structured control flow in which loops and if-else conditions are inline. Code blocks are nested into a hierarchy, and WebAssembly uses nesting depth to indicate control flow. The key takeaway is that whereas it is common for bytecode to jump by offsets from the current instruction pointer, for instance, by jumping four lines ahead. WebAssembly requires calculating a side table to know which addresses correspond to which block. This design makes it more challenging to execute modules line-by-line as required in interpretation. At first glance, this design appears to cause overhead by

having to calculate a control table. However, the engine can perform this work while validating the module, incurring low overhead. Validation is mandatory by WebAssembly specification, and validation can be fast. Recall that WebAssembly supports single pass validation by sharing function signatures and that shared signatures function as placeholders to enable parallel processing. The upshot is that achieving fast startups with an interpreter is possible.

The similarities between how an interpreter and a baseline compiler will validate and generate code in a single pass suggest we cannot conclude that the poor performance of JavaScriptCore is due to inherent flaws in its design. This design works well for JavaScript and has proven to yield class-leading performance. Instead, its poor performance appears to reflect a lack of effort in implementing the WebAssembly engine. Traces from JavaScriptCore indicate less efficient code generation and poor registry use. Interestingly JavaScriptCore records a better result than V8 and SpiderMonkey when running the least optimised WebAssembly modules compiled at the O0 level. JavaScriptCore can execute the least performant WebAssembly the fastest, but it lacks refinement because its implementation of WebAssembly is fairly rudimentary.

Execution speed (seconds)	Native	JavaScript	WebAssembly
Optimisation level - O0			
Native	254,34		
V8		554,99	272,91
SpiderMonkey		383,36	287,15
JavaScriptCore		259,78	262,07
Optimisation level - O1			
Native	116,00		
V8		209,03	144,30
SpiderMonkey		249,17	165,12
JavaScriptCore		180,76	346,01
Optimisation level - O2			
Native	102,99		
V8		214,43	122,59
SpiderMonkey		245,33	146,02
JavaScriptCore		169,57	204,24
Optimisation level - O3			
Native	102,74		
V8		214,44	117,24
SpiderMonkey		250,35	145,45
JavaScriptCore		175,51	204,01

Table 5.1: Execution speed (in seconds) across optimisation levels

	Native	JavaScript	WebAssembly
Optimisation level - O0			
Average size (kB)	16,93	336,73	27,10
Std deviation (kB)	1,80	4,73	1,45
Compressed size (kB)	2,84	61,50	12,93
Std deviation compressed size (kB)	0,25	0,82	0,98
Size reduction (in percent)	83,23 %	81,74 %	52,28 %
Best time (in seconds)	254,34	259,78	262,07
Optimisation level - O1			
Average size (kB)	14,67	196,97	23,20
Std deviation (kB)	1,52	1,54	0,41
Compressed size (kB)	2,71	42,97	10,47
Std deviation compressed size (kB)	0,18	0,19	0,51
Size reduction (in percent)	81,52 %	78,19 %	54,89 %
Best time (in seconds)	116,00	180,76	144,30
Optimisation level - O2			
Average size (kB)	17,07	62,83	21,70
Std deviation (kB)	1,72	0,70	0,53
Compressed size (kB)	2,98	20,57	9,91
Std deviation compressed size (kB)	0,27	0,50	0,10
Size reduction (in percent)	82,54 %	67,27 %	54,35 %
Best time (in seconds)	102,99	169,57	122,59
Optimisation level - O3			
Average size (kB)	17,20	58,80	21,00
Std deviation (kB)	1,63	0,66	0,37
Compressed size (kB)	3,02	19,03	9,54
Std deviation compressed size (kB)	0,33	0,18	0,12
Size reduction (in percent)	82,42 %	67,63 %	54,59 %
Best time (in seconds)	102,74	175,51	117,24

Table 5.2: Results across optimisation levels

Table 5.2 includes data on size and performance. There is no negative trade-off between speed and optimisation level for neither JavaScript nor WebAssembly. Both code size and execution speed decrease with higher optimisation levels. JavaScript is best at the 02 level, while the others perform best at the 03 level. The difference between the 02 and 03 levels is not significant for JavaScript.

With regard to native binaries, these tend to increase slightly in size with

greater optimisation, while performance improves for each optimisation level. The relationship between greater optimisation and performance is therefore positive for native binaries while slightly negative with respect to size. JavaScript and WebAssembly unambiguously benefit from optimisation by simultaneously decreasing the size and increasing performance.

As mentioned previously in chapter 4, the size of binaries is relevant because a reduced payload spends less time in transit across a network. Tests should consider if each format is compressible. Results show native, JavaScript, and WebAssembly compress around 80%, 65% and 55% on average. While WebAssembly is the least compressible, it is more compact than JavaScript. Compressed WebAssembly modules require nearly half the size of equivalent JavaScript code. However, they require twice the size of native binaries on average. This result is as expected because WebAssembly has a less specific instruction set, requiring more instructions to complete the same operations.

The previous chapter on tests asserted that PolyBenchC offers a varied set of execution times. Table 5.3 records the median execution time for each benchmark. Results reported for JavaScript and WebAssembly use data from the best engine. Figure 5.1 project the results from table 5.3 onto a diagram. These results confirm that PolyBenchC offers a broad range of execution times. Please note that eight benchmarks execute so fast that they are hardly visible in the diagram.

Execution time (seconds)	Native	WebAssembly	JavaScript
2mm	2,013	2,767	3,547
3mm	3,570	4,608	5,415
atax	0,007	0,008	0,013
bicg	0,011	0,012	0,013
cholesky	1,434	1,598	1,691
doitgen	0,520	0,561	0,699
gemm	0,523	1,294	2,082
gemver	0,026	0,030	0,062
gesummv	0,005	0,006	0,005
mvt	0,023	0,024	0,027
symm	2,788	3,261	3,480
syr2k	4,214	4,485	4,957
syrk	0,649	1,208	1,544
trisolv	0,003	0,003	0,007
trmm	2,186	2,594	2,635
durbin	0,004	0,005	0,015
gramschmidt	9,868	10,623	10,418
lu	5,634	6,786	8,069
ludcmp	5,625	6,826	8,536
correlation	5,693	6,396	6,382
covariance	5,653	6,328	6,444
floyd-warshall	15,375	22,162	48,999
nussinov	4,773	6,642	8,729
deriche	0,253	0,283	0,266
adi	8,126	9,270	8,891
fdtd-2d	1,705	2,767	4,323
jacobi-1d	0,001	0,002	0,013
jacobi-2d	1,573	3,056	5,711
seidel-2d	18,127	20,523	19,408
heat-3d	2,359	4,398	7,185

Table 5.3: Execution time benchmarks (best median across engines)

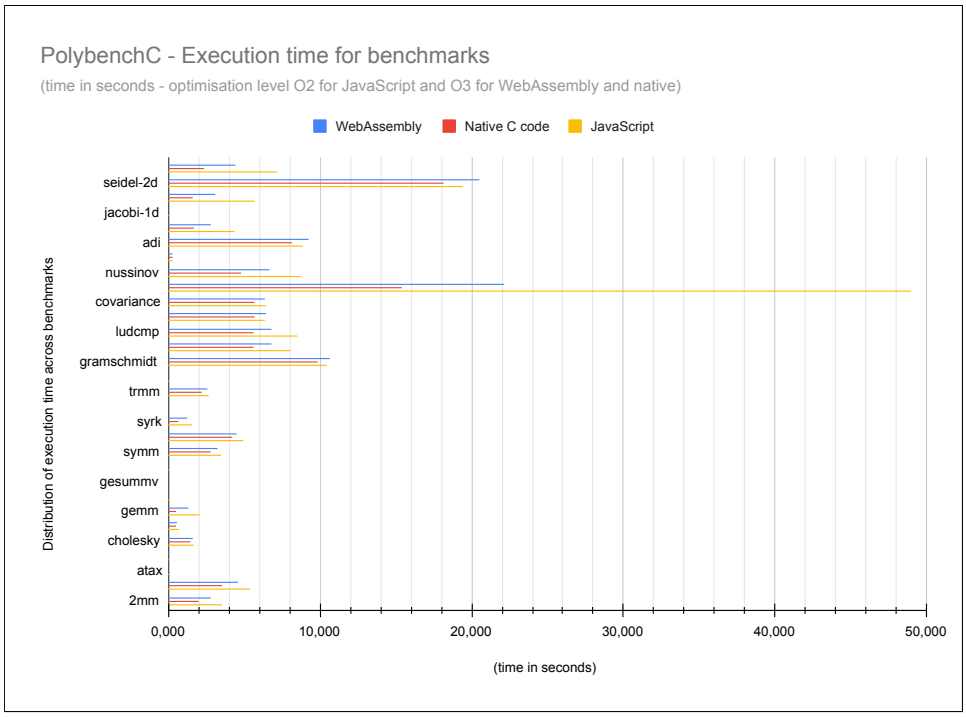

Figure 5.1: PolyBenchC execution time for benchmarks

Table 5.4 shows data for the best performance across native, WebAssembly and JavaScript. The following will discuss consistency, maturity and peak performance.

	Execution time (seconds)	Relative to native	Warm std. dev. (%)	Cold std. dev. (%)
JavaScript - O2				
V8	214,43	208,71 %	2,84 %	2,89 %
SpiderMonkey	245,33	238,79 %	4,08 %	4,09 %
JavaScriptCore	169,57	165,04 %	19,59 %	19,99 %
WASM - O3				
V8	117,24	114,11 %	3,33 %	22,81 %
SpiderMonkey	145,45	141,57 %	4,49 %	4,87 %
JavaScriptCore	204,01	198,57 %	1,95 %	1,96 %
Native - O3	102,74	100,00 %	1,89 %	1,92 %

Table 5.4: Statistics for engines across PolyBenchC benchmarks

Readers will note that the table includes cold and warm measures for standard deviation. Two measures are needed to show the effects of caching. Each measure uses data taken from five runs. The cold standard deviation includes the first run, whereas the warm measure omits the first run and samples the five subsequent runs. The warm standard deviation reflects code caching.

There are no significant effects from caching for JavaScript. Cold and warm standard deviations are very close. V8 and SpiderMonkey both display low standard deviations. These results reflect consistent throughput in their JavaScript pipelines. V8 and SpiderMonkey both use compilation with two tiers.

On the other hand, JavaScriptCore displays pronounced variation. This result is an artefact of its structure. Recall that JavaScriptCore uses four compilers in executing JavaScript, two more than V8 and SpiderMonkey. The additional compilers cause greater variability when scripts run for a short time. The cold start standard deviation is reduced from 19,99% to less than 7% when omitting scripts that run for less than one second.

As mentioned in the chapter on theory, JavaScript provides automatic

garbage collection. Cleanups require pausing execution while checking references and removing old objects. Such intermittent pauses could cause performance fluctuations. Therefore tests were repeated with a configurable timer to assess the impact of garbage collection. There was only a slight indication that garbage collection caused performance issues in JavaScript. Tests without garbage collection were in line with ordinary execution, and differences in execution time were not statistically significant. For these tests, there was no clear indication that garbage collection led to significant fluctuation in throughput.

Moving on to WebAssembly, the results show no substantial effect from caching for JavaScriptCore or SpiderMonkey. However, V8 shows aggressive caching and records a difference between its cold and warm standard deviations close to 20%. The cold-start standard deviation is more than 22%, while just above 3% when warm. This outcome is because V8 caches code and uses refined compiler output after the first run.

Looking at consistency, both JavaScript and WebAssembly are comparable when making allowances for caching. Native execution is the most consistent. JavaScript and WebAssembly show no clear winner for second place.

Maturity is concerned with the most variable and least performant engine. JavaScript engines all show acceptable consistency while their performance is mixed. For example, SpiderMonkey lags JavaScriptCore by 45% and V8 by 15% in speed.

Considering maturity for WebAssembly, consistency in all engines is on par with JavaScript when modules are warm. However, the lacklustre performance of JavaScriptCore suggests that WebAssembly is not yet production ready. For example, when the benchmarks from PolyBenchC run in JavaScriptCore, they require nearly 20% more time to execute in WebAssembly than in JavaScript. In addition, JavaScriptCore is not able to complete all the benchmarks, as it gets stuck in a loop executing a single benchmark. These facts indicate that WebAssembly falls behind JavaScript and native binaries in terms of maturity.

While maturity is a weak spot for WebAssembly, it shows encouraging results for performance. The peak performance for WebAssembly exceeds the best result for JavaScript by 30%, and it is only 15% behind native.

Furthermore, the trend is positive when looking at WebAssembly over time. Previous studies reported 7 of 24 benchmarks within 1.1x of native in 2017 [4] and 13 of 24 within 1.1x in 2019 [7]. Current results show 16 of 30 within 1.1x and 19 out of 30 benchmarks within 1.15x in V8. Note that the figures are not directly comparable across 2017, 2019 and 2022. It appears that previous studies have had some issues compiling all benchmarks and therefore included only 24 out of 30 tests. This study has been able to compile all benchmarks from PolyBenchC. The latest sample therefore includes six additional benchmarks. It is not clear why other researchers have not included all benchmarks.

In summary:

- WebAssembly is faster than JavaScript by 30%.
- WebAssembly is slower than native by 15%.
- WebAssembly has a relatively unrefined implementation in JavaScriptCore. Its performance lags V8 and SpiderMonkey.
- WebAssembly has shrunk the gap to native over time.
- JavaScript and native are more mature than WebAssembly.
- Garbage collection in JavaScript did not affect results significantly.
- No negative relationship exists between size and performance for JavaScript or WebAssembly. Size decreases while execution speed increases with more optimisation.
- Compressed WebAssembly is nearly half the size of JavaScript. At the same time, it is more than twice the size of native binaries.

Returning to the hypotheses:

1. WebAssembly offers better performance than JavaScript
2. WebAssembly offers parity to native performance

Conclusions:
Based on benchmarks from PolyBenchC. WebAssembly offers a better execution speed than JavaScript. Data rejects the null hypothesis of equal performance, and this difference is significant at more than a 1% significance level.

Data does not indicate that WebAssembly can offer parity to native performance. Data rejects the null hypothesis of equal execution speed. The difference is significant at more than a 1% significance level.

The next set of tests is to compare WebAssembly performance across time. These tests consider the pace of improvement for the best engine.

The previous results show that V8 offers the best performance for WebAssembly. The section on previous research mentioned the results reported by Hall and Ramachandran [5] for V8. They discovered that execution time for their benchmarks improved by 50% through updates to V8. It is interesting to consider if V8 has been able to continue this positive trend. Tests will focus on results from Node.js. It offers a separate runtime built on the V8 engine and will permit running WebAssembly modules via the command line.

Node.js is prefered for performance testing as it reduces overhead from background processes in browsers. PolyBenchC benchmarks have run in Node.js with V8 versions spanning nearly two years. Results are also presented for Chrome to indicate its overhead as it uses the same V8 engine. Data show that Node.js outperforms Chrome by close to 4%.

(Engine, version - Time stated in seconds)	Aggregate time
Native	102,74
Node.js - V8 version 10.1 (released February 2022)	113,04
Chrome - V8 version 10.0 (released February 2022)	117,24
Node.js - V8 version 8.4 (released May 2020)	124,30

Table 5.5: Improvement in execution speed

Table 5.5 summarises the findings, while figure 5.2 exhibits the same data in a graph. Node.js using V8 version 10 has improved over version 8.4 by more than 9%. Code analysis reveals that V8 version 10 can execute benchmarks using fewer x86 instructions. Figure 5.3 shows the eight benchmarks from PolyBenchC that run for longer than 5 seconds. The decrease in execution time is almost entirely due to improvements in the Floyd-Warshall benchmark. Floyd-Warshall is an algorithm to find the shortest path in a directed graph with edge weights. This algorithm is based on nested for-loops, and V8 version 10 improves instruction selections for x86 when running the intensive loops. The improvement in Floyd-Warshall from V8

version 10 to 8.4 is close to 25%. The number of x86 instructions is reduced by more than 5%, but the most critical factor is reducing the number of branches. Adjustments in instruction selection cut the number of branches by more than half. Inspection reveals a performance increase that stems from a better instruction mix. The number of micro-operations (uops) per cycle is in fact lower while each cycle still achieves more work.

Conclusion:
Performance is still increasing for WebAssembly. However, while the trend is positive, it is modest compared to the 50% reported by Hall and Ramachandran [5]. Returns from code refinement appear to be diminishing, and future improvements will likely proceed at a slower rate.

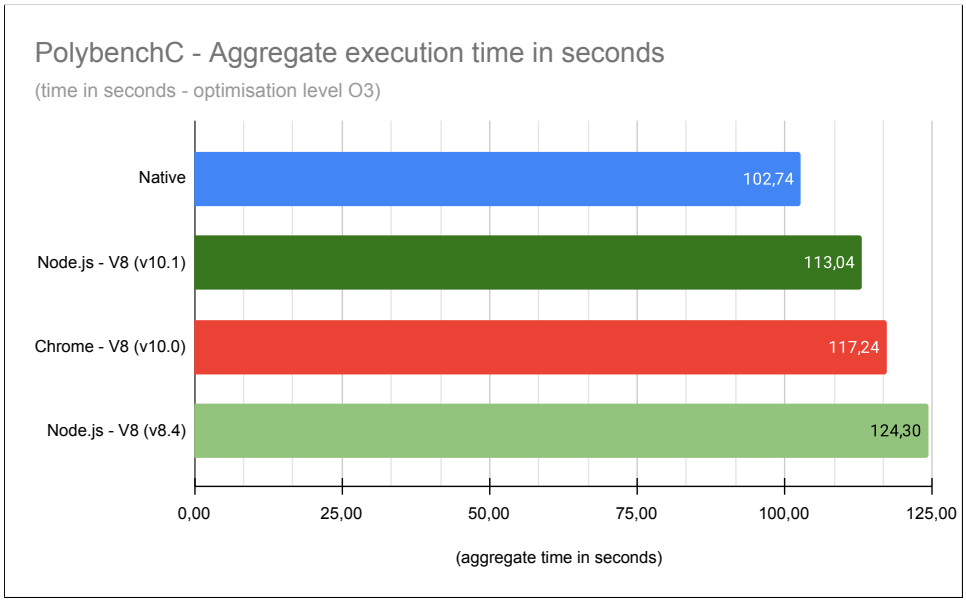

Figure 5.2: PolyBenchC aggreate execution time

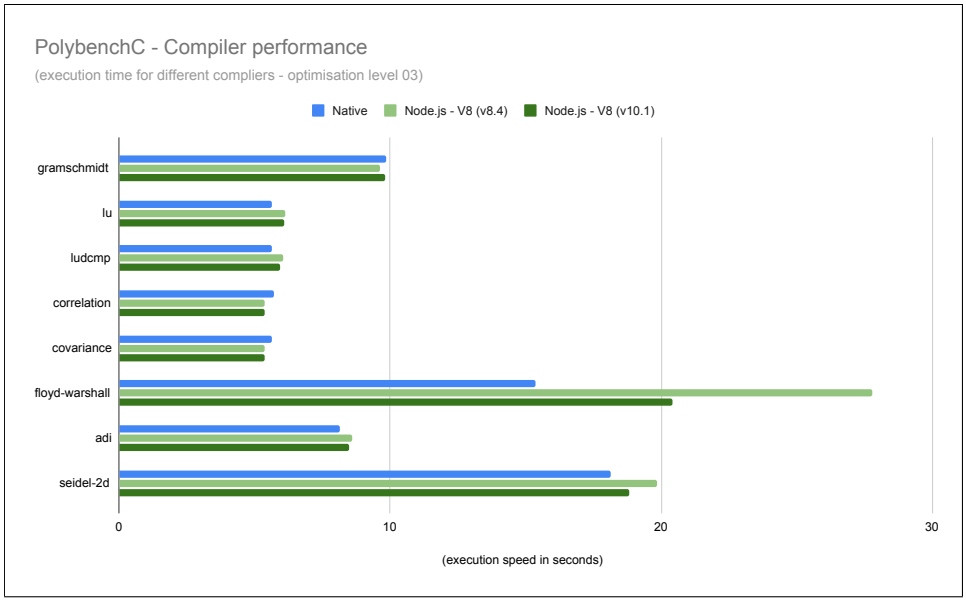

Figure 5.3: PolyBenchC with longer running benchmarks

An issue related to compiler performance is the impact of tiered compilation. Recall that tiered compilation caused an inflexion point in JavaScript performance. The question is whether this success can transfer to WebAssembly. Compiler writers have repurposed the engines from JavaScript in their WebAssembly engines, adopting techniques using sea-of-nodes with static single assignment form. However, as cited in the research section, Yan et al. [19] report only a slight speedup of 1.1x with tiered compilation. Some tests only achieve parity, while some even show slight decreases.

Furthermore, these authors found the best results when isolating the optimising compiler, suggesting that tiered compilers do not offer enough improvement to offset the cost of swapping code. The next set of tests will consider the impact of tiered compilation. Recall that JavaScriptCore is not part of these tests as it offers no compilation guarantee. WebAssembly modules in JavaScriptCore can be run entirely by the interpreter. Whereas V8 and SpiderMonkey will eagerly compile modules.

Compare results using tiered, baseline and optimising compilers. Run tests across SpiderMonkey and V8. Report on the impact of tiered compilation.

Tests run with WebAssembly modules compiled at the O3 level. Figures 5.4, 5.5 and 5.6 show the results. Tiered compilation is advantageous in V8. The aggregate time is 117 seconds with tiered compilation. Locking to baseline adds 56 seconds, while locking to optimising is 4.7 seconds slower than tiered compilation. Figures for V8 put the baseline compiler at 1.49x and the optimising compiler at 1.04x relative to tiered compilation.

Results from SpiderMonkey are unexpected. SpiderMonkey is fastest when locked to the optimising compiler. It records the baseline compiler at 1.56x and the optimising compiler at 0.93x relative to tiered compilation.

Figures 5.5 and 5.6 show that both SpiderMonkey and V8 optimising compilers outperform their baseline compilers. However, SpiderMonkey struggles to swap optimised code fast enough, while V8 benefits from better code generation and faster on-stack replacement.

Note that Yan et al. [19] found V8 and SpiderMonkey to be fastest when

locked to their optimising compilers. Their results show 0.88x in V8 and 0.9x in SpiderMonkey relative to tiered compilation. Their results from SpiderMonkey are in line with 0.93x reported in this thesis. On the other hand, V8 appears to have evolved through better code swapping, and now its best results are obtained with tiered compilation.

Conclusion:
Using tiered compilation is beneficial in V8. However, its benefits in SpiderMonkey are questionable. V8 has evolved to benefit from tiered compilation, and SpiderMonkey can hopefully follow a similar trajectory.

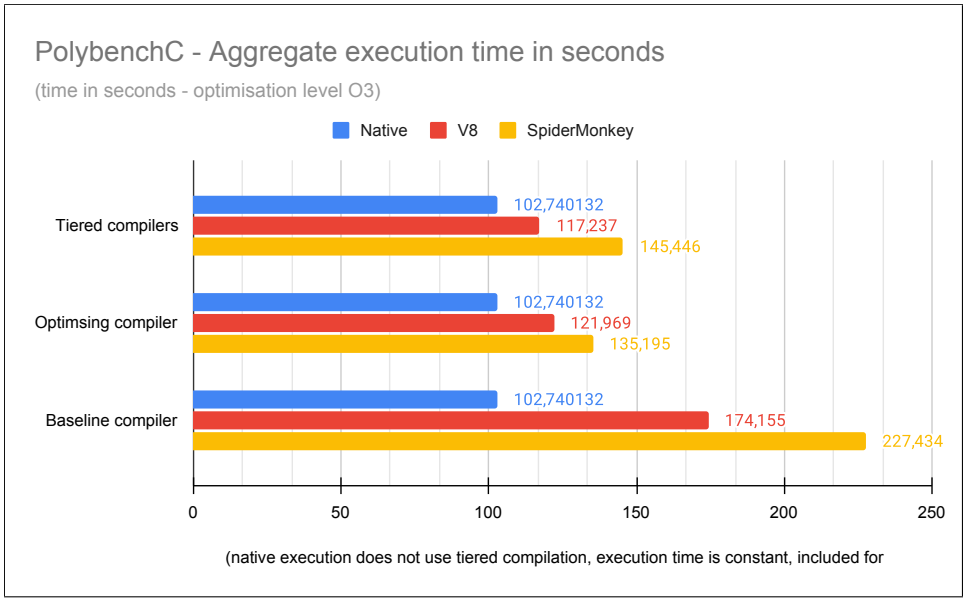

Figure 5.4: PolyBenchC with baseline, optimising and tiered compilation

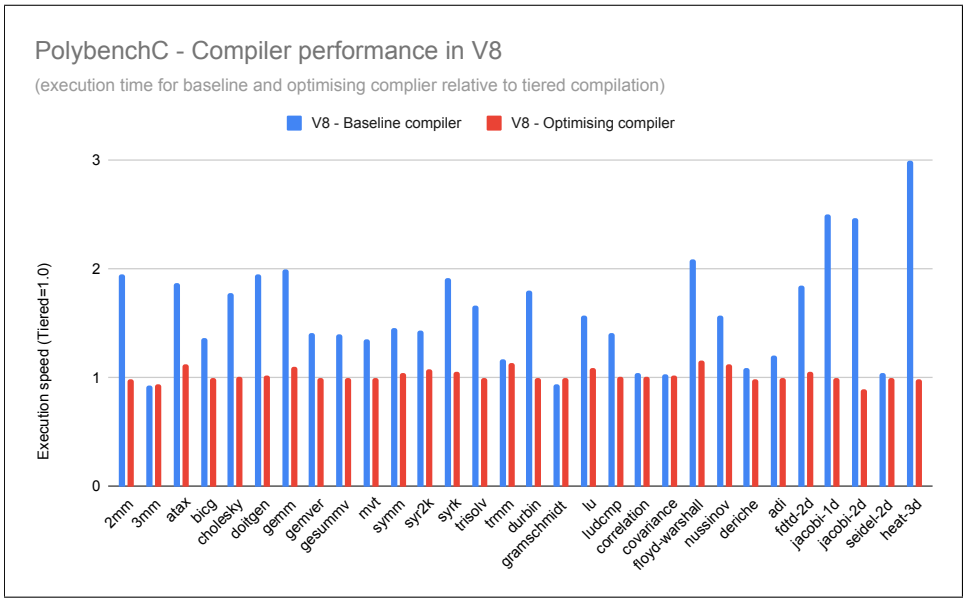

Figure 5.5: PolyBenchC execution in V8

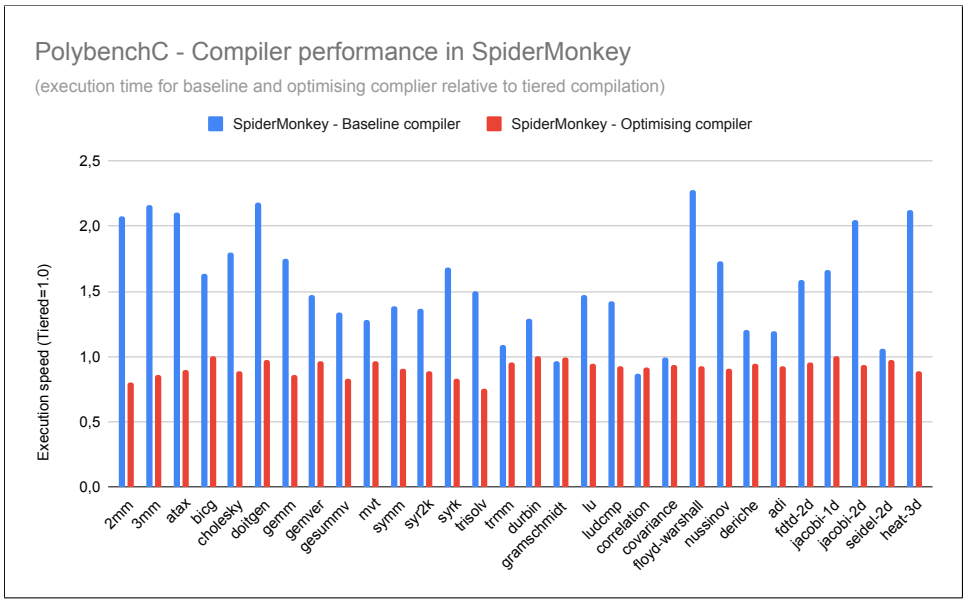

Figure 5.6: PolyBenchC execution in SpiderMonkey

Thus far, the results indicate a positive trend in WebAssembly performance and that WebAssembly can be competitive with native across several benchmarks. What is interesting in this regard is providing a deeper understanding of the factors which make WebAssembly competitive. While in theory, WebAssembly suffers from a less specific instruction set and stack-based execution. In practice, it proves to be competitive. How come WebAssembly can occasionally overcome inherent deficits and compete with native binaries? The subsequent sections will be concerned with identifying causal links. Tests will proceed as outlined in chapter 4.

Collect data on instruction counts, branching behaviour and CPU cycles. Evaluate how WebAssembly execution compares to native and consider whether it has an impediment due to its execution model.

As previously explained, WebAssembly instructions are different from x86 instructions. Evaluating x86 instructions used to implement WebAssembly modules against native x86 binaries requires an analysis pipeline with three steps:

1. Instrument WebAssembly binaries for dynamic analysis. These traces record all WebAssembly instructions.
2. Instrument native binaries for dynamic analysis, recording all x86 instructions.
3. Collect traces for WebAssembly modules while executing in Node.js, recording all x86 instructions.

Together these enable the mapping of WebAssembly instructions into x86 instructions. When combined with data from Apple Instruments, it also provides a set of traces recording cache misses, branches and CPU cycles.

The first question is whether WebAssembly modules require more instructions than native binaries. Figure 5.7 details the results. WebAssembly requires 6x as many instructions as native binaries on average. At best, it executes 3.5x the number of instructions, while at worst, it requires 15x as many instructions as native.

WebAssembly has a smaller instruction set. In these benchmarks, the WebAssembly modules use 56 to 60 individual instructions, whereas native x86 binaries use more than twice these figures. Numbers vary between 123 to 139 instructions for native code. Note that x86 instructions are often

related. Such as memory read or write instructions at 8, 16 or 32 bits.

When considering the smaller WebAssembly instruction set, it is interesting to know if its instructions are equal in execution time. Processors use instruction-level parallelism and pipelining. A fast instruction mix can offset the disadvantages of using more instructions. If WebAssembly instructions require equal time, the relationship between instruction count and execution time is linear. Figure 5.8 plots a regression line across benchmarks. The instruction count explains a mere 43% of the variance. Hence the conclusion is that time required to execute different WebAssembly instructions is variable.

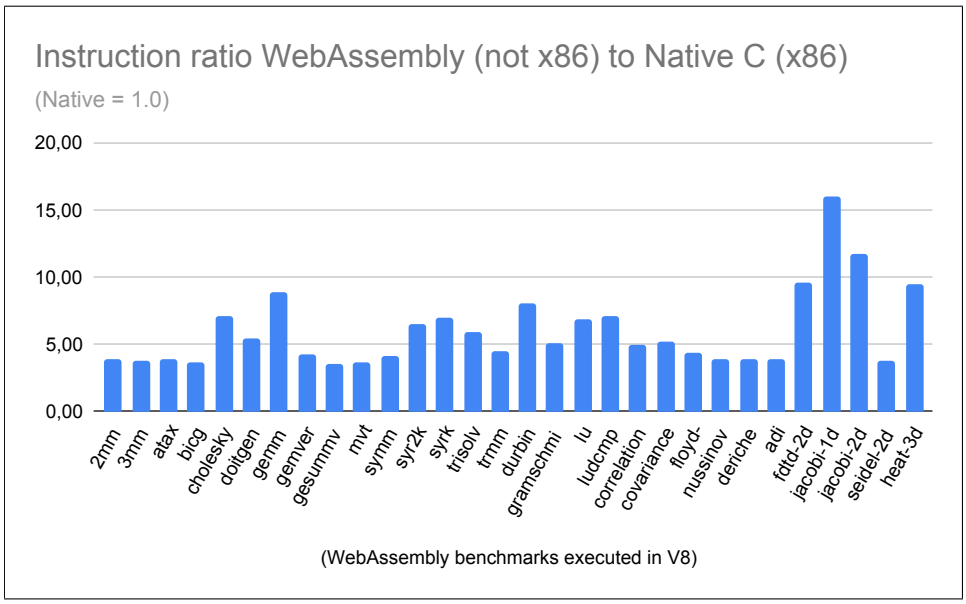

Figure 5.7: Instruction ratio WebAssembly (not x86) to C (x86)

When WebAssembly modules execute in Node.js, the runtime will compile WebAssembly instructions into x86 instructions. Figure 5.9 exhibits the ratio of executed WebAssembly x86 instructions to native x86 instructions. Jacobi-1d is an outlier in the data, and a mean value replaces it. With this adjustment, WebAssembly modules execute around four times the number of x86 instructions relative to native binaries.

Multiple WebAssembly instructions can reduce to a single x86 instruction. On average, two WebAssembly instructions execute per x86 instruction. However, this number varies across benchmarks, as seen in figure 5.10.

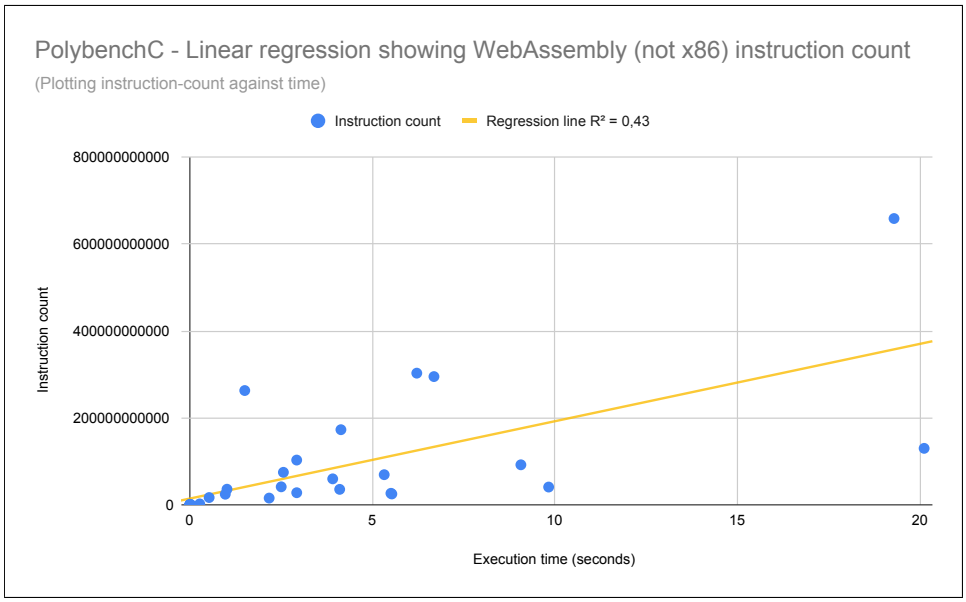

Figure 5.8: Linear regresssion WASM (not x86) to instruction count

While compacting WebAssembly instructions into fewer x86 instructions is generally positive for speed. Compacting WebAssembly instructions is not consistently decisive for speed. Benchmarks such as Atax and Mvt score poorly with 0.67 and 0.72. Indicating that they increase, rather than reduce, x86 instruction count when executing their modules. Even with this expansion, these benchmarks were close to native speed. These figures demonstrate how instruction counts are of limited value in themselves. Effects from instruction mix can offset instruction count. This finding reinforces how the time required to execute WebAssembly instructions is variable.

The chapter on theory explained how WebAssembly would presumably be disadvantaged by:

1. Less specialised instructions, which increase instruction counts.
2. Checks at runtime, which requires branches.
3. Less efficient addressing due to lack of registers, forcing more load/store instructions and cache misses.

In considering these three factors, the subsequent analysis will consider the eight benchmarks in PolyBenchC that run for more than 5 seconds.

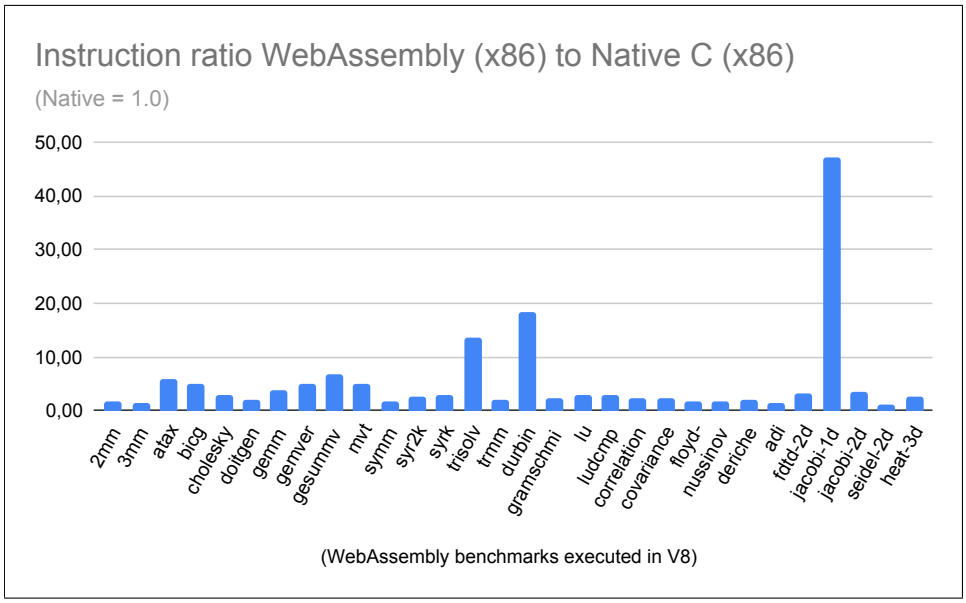

Figure 5.9: Instruction ratio WebAssembly (x86) to C (x86)

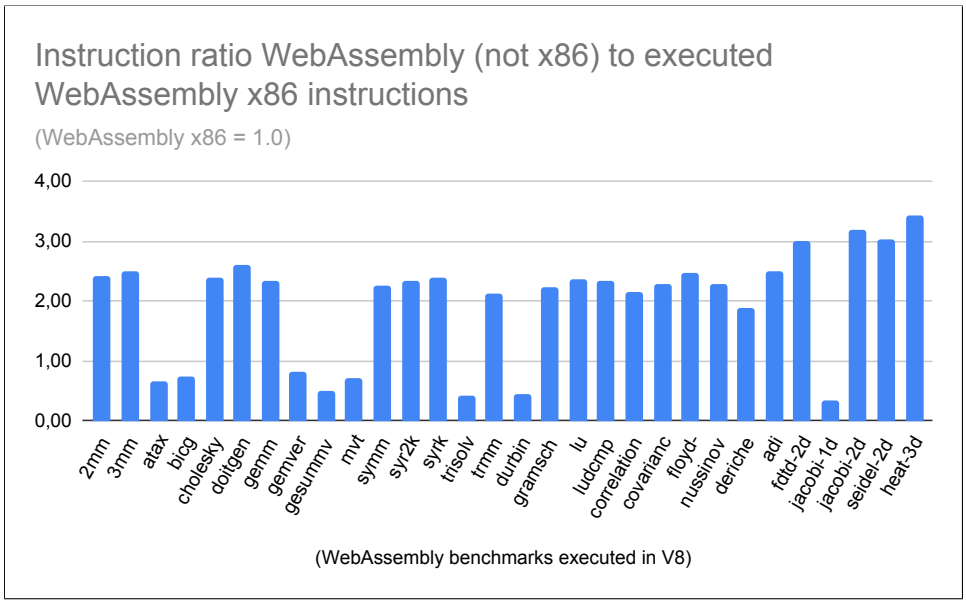

Figure 5.10: Instruction ratio WebAssembly not x86 to x86

Figures 5.11 and 5.12 confirm that WebAssembly requires more instructions and branches. Previous results indicated that instruction counts were not highly correlated to speed. This finding also holds for branch counts.

Figure 5.13 shows the time ratio for WebAssembly x86 to native x86. It also includes the WebAssembly x86 branch count relative to native x86. Figure 5.14 shows the same data with instruction counts. Both show counts can be twice that of native x86, while the time ratio is still close to 1. The key observation is that WebAssembly can execute twice the number of instructions or twice the number of branches and still complete at near-native speed.

Figure 5.15 shows the time ratio with load/store operations included. Again there is no clear pattern. Finally, table 5.6 shows the correlation of execution speed to each factor. Correlations are all close to 0.65. The summary is that no single factor holds the key. There is a positive relationship, but different factors contribute differently at different times.

Correlation of execution speed to	
Instruction count	0.66
Branch count	0.67
Loads count	0.63
Stores count	0.64

Table 5.6: Correlation of execution speed against counts

Code inspection has enabled additional analysis. High-frequency instruction traces enable timing studies. Findings are as follows:

- As already mentioned. Instruction mix can offset the impact of a higher instruction count. Timing studies support this.
- Hardware support mitigates the impact of some security checks, especially for memory lookups.
- Cache utilisation drives performance. Comparing load misses at L1, L2 and L3 levels. Native has 9,32% misses for all instructions, while WebAssembly has 4,37%. WebAssembly still records more cache misses in absolute terms, but its relative share is less than native. The average cache miss for WebAssembly was less costly than native code. WebAssembly spent fewer cycles in waiting, and the WebAssembly execution maintained a higher instruction throughput.

These factors conspire to put WebAssembly ahead in a few benchmarks.

Conclusions:

- WebAssembly provides an abstraction over hardware and requires more instructions than native.
- Mandatory checks and monitoring require more branch instructions. WebAssembly is confirmed to have an impediment in this respect.
- Hardware support can partially offset the impact of checks.
- Instruction counts are of limited value in explaining execution speed. Effects from instruction mix can offset instruction count.
- When WebAssembly performs well, it benefits from caching and fast instruction mix.

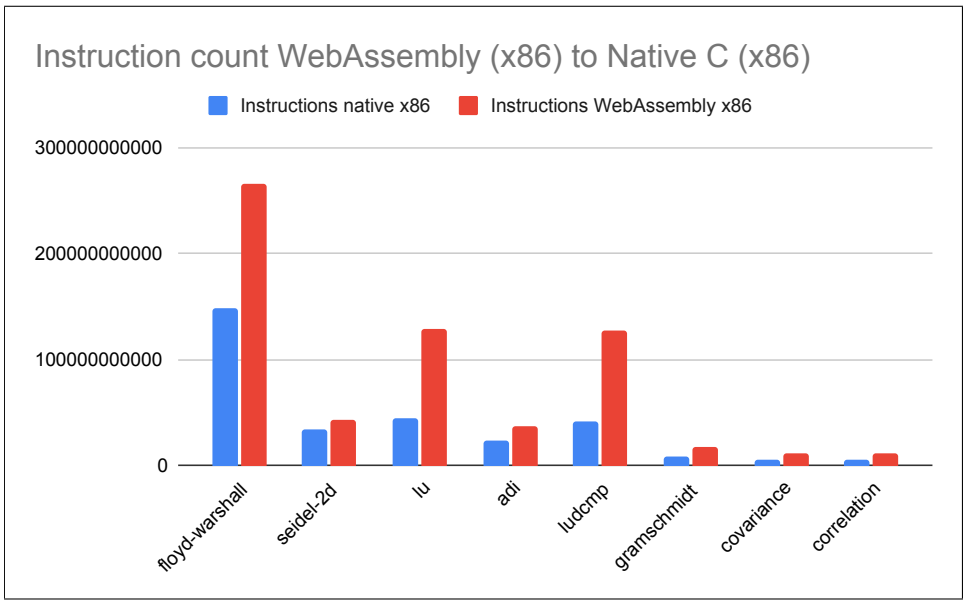

Figure 5.11: Instruction count WebAssembly (x86) to C (x86)

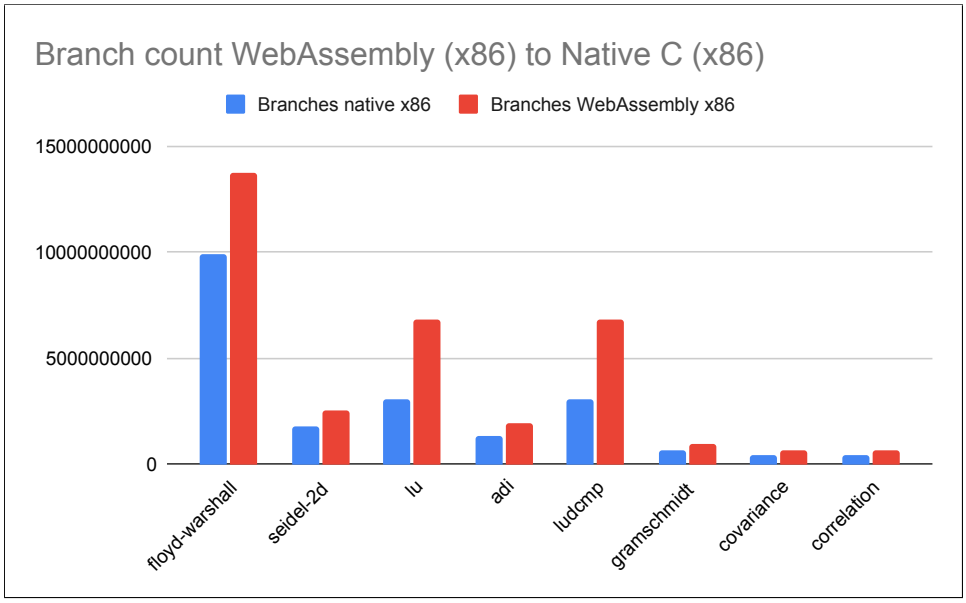

Figure 5.12: Branch count WebAssembly (x86) to C (x86)

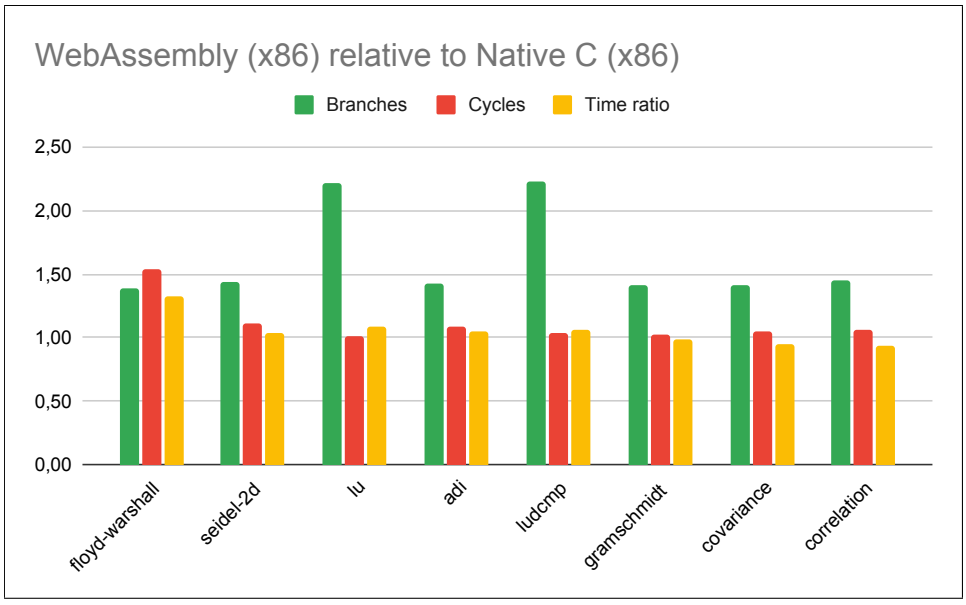

Figure 5.13: Branches WebAssembly (x86) relative to C (x86)

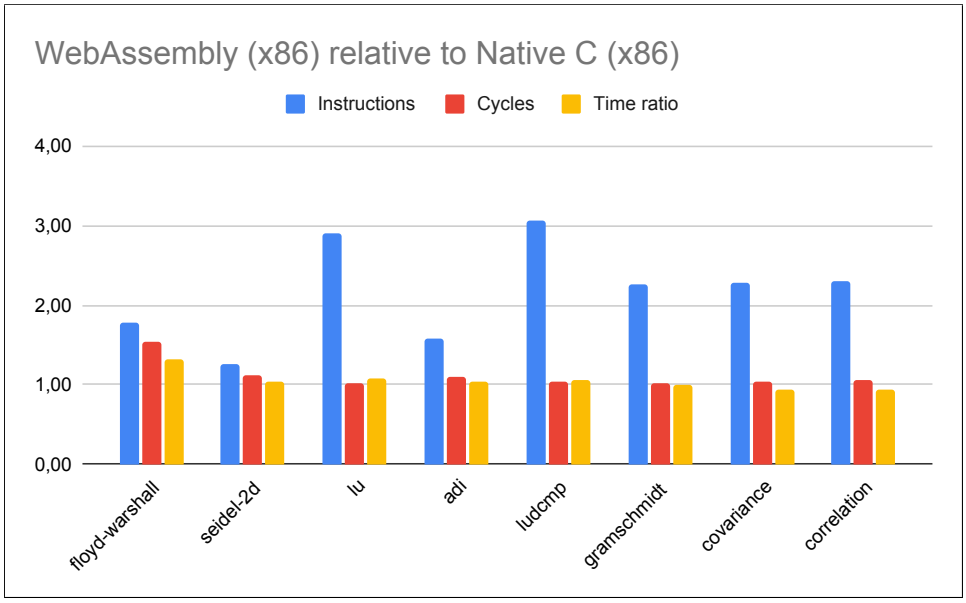

Figure 5.14: Instructions WebAssembly (x86) relative to C (x86)

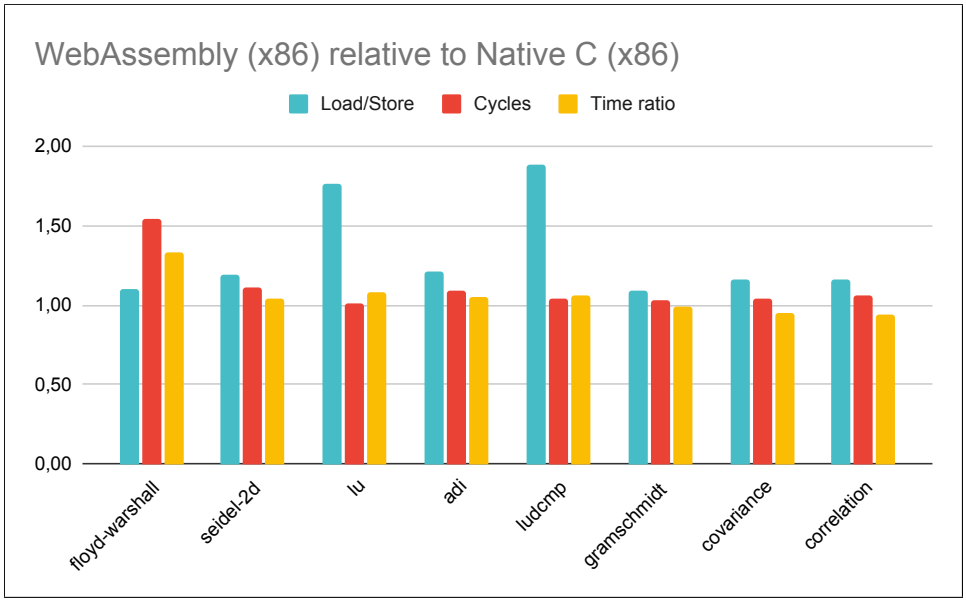

Figure 5.15: Load and store WebAssembly (x86) relative to C (x86)

These conclusions round off the presentation of results based on the WebAssembly 1.0 specification. The most important finding is that peak performance in WebAssembly is 30% better than that of JavaScript. JavaScriptCore provided the best result for benchmarks running in JavaScript, whereas V8 proved to be the most performant engine for WebAssembly. Comparing the same benchmarks across WebAssembly and JavaScript, V8 improved 30% over JavaScriptCore, while it improved by 45% and 52% over itself and SpiderMonkey. However, one may suggest that these results are not that relevant. What is important is the relative difference between WebAssembly and JavaScript inside each engine. A web application must perform well inside the user's prefered browser. We cannot assume users will change browsers to access our app. Hence the relevant measure considers switching from JavaScript into WebAssembly inside each engine.

Switching a codebase from JavaScript to WebAssembly will increase performance by 45% in V8 and 41% in SpiderMonkey. However, in JavaScriptCore, the performance will decrease by 20%. This example serves to illustrate that web applications need to provide a fallback to JavaScript. Hence WebAssembly modules are not currently positioned to replace JavaScript outright but to offer a potential performance uplift.

Detractors will point out that a speedup of less than 2x does not constitute a paradigm shift for web applications. The evolution of CPUs has tended to offer better single-threaded performance regardless of format. One may argue that workloads achievable with JavaScript will increase as the rising tide lifts all boats. As a point of comparison. CPUs announced by Intel and AMD in 2022 offer 15 to 30% improvement in single-threaded performance over their previous generation. Why should a performance increase of 30% over JavaScript convince developers to offer WebAssembly binaries? Especially if they also have to maintain JavaScript as a fallback in either event.

A counter-argument would be that judging WebAssembly performance based on its 1.0 specification is misleading. WebAssembly is designed to be extendible and has gained features unavailable in JavaScript. For example, chapter 3 explains how WebAssembly 2.0 added support for fixed-width SIMD instructions. SIMD exemplifies how WebAssembly can provide substantial speedup to unlock workloads which are not feasible in JavaScript.

The final set of tests run sequence alignment with and without SIMD

instructions. The objective is to assess the impact of SIMD on execution speed.

Tests use Chrome 102 with V8 10.2 and Firefox 102 with the corresponding SpiderMonkey release. Compilation for WebAssembly and native uses the 03 optimisation level. As with previous tests, the median and standard deviation use data collected from five runs. Stated standard deviations are warm and therefore reflect code caching.

Chapter 4 introduced sequence alignment used to find matching parts of DNA. Smith-Waterman is an algorithm that performs an exhaustive search by testing all possible alignments for two sequences. It does this by sliding two arrays alongside each other. DNA sequences are strings built using the letters A, T, C, or G. Gaps are also permissible to allow strings of different lengths. The algorithm starts from the first position in the first array and slides the second array along the first to test all possible alignments.

Each sequence uses one byte per position. As previously explained, the Smith-Waterman algorithm has quadratic time and space complexity. In these tests, the sequences will have equal lengths. Hence the number of combinations to tests is NxN. Results reside in an NxN matrix, and the algorithm backtracks from the highest score to find the best alignment.

The non-SIMD version of the Smith-Waterman algorithm runs by two nested loops. The outer loop iterates across the first sequence, while the inner loop calculates the alignment scores using the second sequence. The SIMD implementation also runs using loops, but fixed width 128-bit SIMD instructions enable parallel processing of 16 elements at a time.

The algorithm benefits from SIMD operations while fetching pairs, calculating their score and writing back to the score matrix. With these three steps, an informal heuristic can indicate a measure of the speedup resulting from SIMD operations. For example, if instructions require the same execution time, handling 15 additional values across three operations could provide a speedup of 3 x 15 = 45.

The previous paragraphs omit some details of the algorithm that are not crucial to asses WebAssembly performance. For instance, in these tests, the non-SIMD version of the algorithm has been rewritten to do less bookkeeping for backtracing the best alignment. These adjustments help reduce the

number of memory accesses.

Initial results use sequence lengths of 3500, 7000, 10 500 and 14 000 characters. Table 5.7 demonstrates how alignment without SIMD becomes impractical. For example, at a length of 14 000, the non-SIMD algorithm in V8 requires almost 5 seconds. While it takes less than 0,11 seconds when using V8 with SIMD.

Table 5.8 comprises data on the speedup for native, V8 and SpiderMonkey. Notice how both native and V8 converge towards the heuristic for speedup, which was 45. SpiderMonkey also converges but at a slower pace. This effect is partly because tiered compilation is less efficient in SpiderMonkey, as explained previously in this thesis.

Sequence length (time in seconds)	3500	7000	10 500	14 000
Native - SIMD	0,007	0,022	0,051	0,085
Native	0,247	0,953	2,147	3,794
V8 - SIMD	0,011	0,030	0,071	0,109
V8	0,304	1,192	2,680	4,827
SpiderMonkey - SIMD	0,012	0,033	0,074	0,128
SpiderMonkey	0,259	1,020	2,293	4,130

Table 5.7: Execution speed (in seconds) for sequence alignment

Sequence length (speedup)	3500	7000	10 500	14 000
Native	36,63	43,80	41,89	44,64
V8	27,64	39,73	37,75	44,28
SpiderMonkey	21,60	30,91	30,98	32,27

Table 5.8: Speedup for SIMD to non-SIMD sequence alignment

Figure 5.16 illustrates the initial results. Note that time on the y-axis uses a logarithmic scale. In summary, results show that native is slightly ahead of V8 and SpiderMonkey. When executing benchmarks with SIMD, V8 has the advantage over SpiderMonkey as it is around 10% faster on average.

Results from longer sequences only consider tests with SIMD as non-SIMD becomes impractical. Figure 5.17 shows sequence lengths up to 448 000 characters. The y-axis shows time in seconds, and data is available in table 5.9. The key takeaway is that V8 and native are indiscernible in the graph.

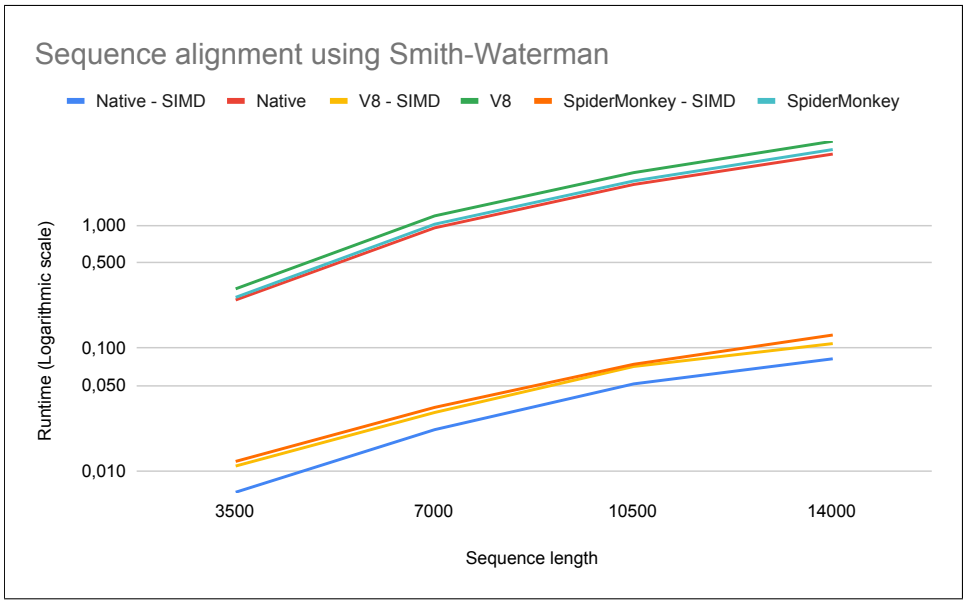

Figure 5.16: Sequence alignment, both SIMD and non-SIMD

V8 can keep up with native and even records a faster result at a sequence length of 224 000. SpiderMonkey is competitive but almost 20% behind V8 on average.

Sequence length	28 000	56 000	112 000	224 000	448 000
Native - SIMD	0,244	0,791	3,014	11,602	46,827
Chrome - SIMD	0,271	0,798	2,980	11,393	46,875
Firefox - SIMD	0,332	0,963	3,471	13,479	55,270

Table 5.9: Execution speed (in seconds) for sequence alignment

With V8 and native recording nearly identical performance, an obvious question is whether they execute using the same instructions. Traces reveal that they execute code in different ways. The native binaries use the SSE2 instruction set, which incorporates extensions to the original Streaming SIMD Extensions (SSE) and performs fixed-width 128-bit operations.

SSE2 instructions can use the entire width of 128 bits per operation. In contrast, V8 prefers 64-bit operations as it compiles the WebAssembly SIMD instructions into x86 instructions. V8 executes almost twice the number of x86 instructions as native x86 binaries. The exact instruction count ratio is close to 1.9x. The familiar pattern for WebAssembly execution repeats for WebAssembly binaries with SIMD instructions. Recall that the WebAssembly compilers have tended to prefer fast instructions with high

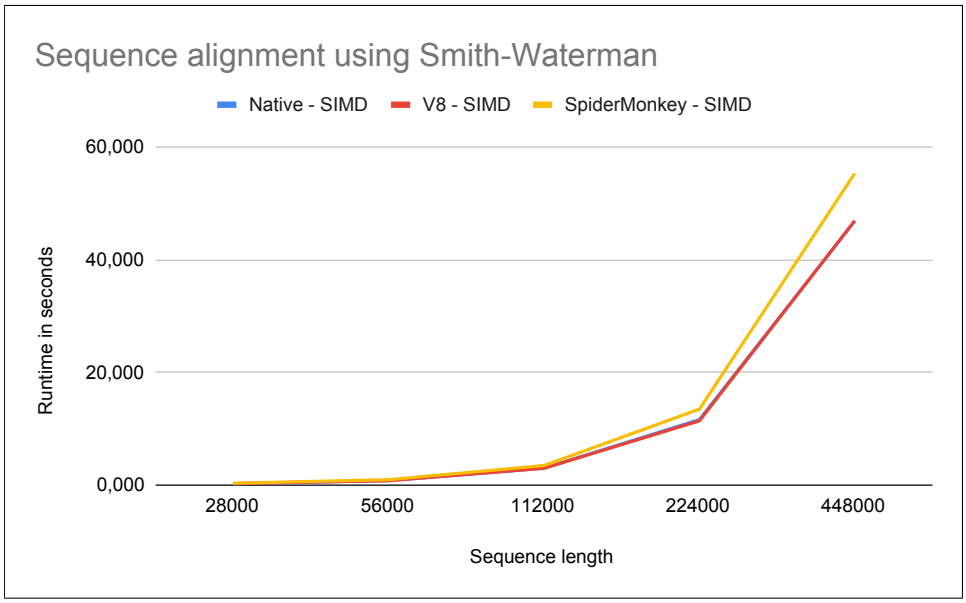

Figure 5.17: Sequence alignment SIMD

throughput. WebAssembly makes up for doubling the instruction count by using fast operations with 64-bit width. In the sequence alignments, the number of instructions is almost double, but the throughput rises accordingly. Such that WebAssembly can match native performance.

This section concludes with a summary of SIMD performance along the same axis previously used in this thesis. These were consistency, maturity and peak performance. The standard deviation provides a measure of consistency. When using SIMD, the warm standard deviation in V8 and SpiderMonkey was less than 3% on average, while the average standard deviation was closer to 5% for non-SIMD. The comparable figure for native binaries was close to 2%. The upshot is that SIMD execution was consistent.

Maturity was satisfactory for SpiderMonkey and V8. Testing with SIMD did not uncover significant issues.

Peak performance is strong in both V8 and SpiderMonkey. V8 is on par with native for sequences longer than 40 000 characters. However, due to time spent in initial validation and compilation, V8 requires slightly more time to reach peak throughput than native. Therefore it is behind native up to 40 000 characters. For its part SpiderMonkey falls some 18% behind V8 when sequences exceed 40 000 characters.

While these results are encouraging for WebAssembly, it remains the case that native x86 binaries can benefit from a more extensive set of vector instructions. WebAssembly does not offer parity in this respect. However, these results indicate that when restricted to 128-bit vector instructions, WebAssembly can keep up with native binaries. Importantly the speedup from vector operations is highly significant, and WebAssembly can offer exciting opportunities for fast computation in browsers. It is proven to offer a substantial performance increase relative to JavaScript. The conclusion is that WebAssembly can execute workloads which have hitherto been impossible in web applications.

Chapter 6

Conclusions

The research question of this thesis is whether WebAssembly can improve web application performance.

The question was made testable by two hypotheses.

1. WebAssembly offers better performance than JavaScript
2. WebAssembly offers parity to native performance

Data has been obtained by compiling benchmarks into JavaScript, WebAssembly and native. Based on workloads in PolyBenchC, results indicate that WebAssembly can outperform JavaScript. WebAssembly improved execution time by 30% over equivalent JavaScript. The first hypothesis was accepted.

When comparing WebAssembly to native, PolyBenchC benchmarks show native binaries are 15% faster than equivalent WebAssembly. Data rejected the second hypothesis. While WebAssembly could not match native binaries, it kept close to native performance in two-thirds of benchmarks. A few examples also put WebAssembly on par with native performance. Analysis suggested WebAssembly can rival native performance when effects from caching and instruction mix are favourable. Tests also reveal a positive trend where WebAssembly compilers have increased performance through better code generation. However, the returns from compiler refinement appear to have subsided in recent times.

An unfortunate finding is that not all engines recorded good results for

WebAssembly. This finding is important because the least capable engine determines the performance bottleneck. WebAssembly cannot improve over JavaScript in all engines. Converting PolyBenchC benchmarks from JavaScript into WebAssembly will increase performance by 45% in V8 and 41% in SpiderMonkey, but performance in JavaScriptCore will decrease by 20%. These results indicate web applications need to provide a fallback to JavaScript. WebAssembly is not yet positioned to replace JavaScript but instead to offer a potential performance uplift.

Irregular performance across engines is a cause for concern. Success for WebAssembly is contingent on universal support with high performance to drive adoption. At the same time, it is important not to pass judgement prematurely. WebAssembly is still a young language with many proposed extensions. One of the most important features added to WebAssembly 2.0 was the support for SIMD instructions. This thesis has documented how SIMD instructions can speed up workloads. Examples from sequence alignment demonstrate speedup close to 45x. While support is not universal across engines. Results from V8 and SpiderMonkey suggest WebAssembly SIMD instructions raise the bar regarding computing power. WebAssembly can unlock new use cases and dramatically improve computation in web applications.

WebAssembly is some way from competing with native code, not least because it cannot access shared libraries and system calls. Native code can also benefit from more advanced SIMD instructions. Nonetheless, WebAssembly can improve web application performance. The main conclusion from this thesis is that workloads previously not possible in web apps can execute in WebAssembly.

Chapter 7

Future research

This thesis has only considered WebAssembly as a tool to offload computing in web apps. However, WebAssembly is not limited to use in browsers as it can execute in standalone runtimes. Such runtimes exist in different niches, such as embedded and cloud computing. Research is needed to document how implementation decisions vary across runtimes.

WebAssembly portability can simplify software distribution. For instance, Python modules use C code, as in Cython and Numpy, which requires per platform compilation. WebAssembly makes do with a single binary and supports streaming compilation. As a result, applications can be split into modules and fetched on demand. Future research should investigate whether WebAssembly can scale to provide modular distribution for large applications. For example, to document how swarms of modules scale.

Future versions of WebAssembly will likely support multi-threading, and such extensions require analysis of atomic operations and security features.

WebAssembly supports a single memory. Future research is needed to consider the impact of multiple memories. Potential problems include race-free access as memory grows or shrinks.

Proposals exist to include branch hinting. Hints can aid compilers in code generation, but research is needed to document best practices. It is also interesting to consider whether WebAssembly modules can benefit from including platform-specific compiler hints. For instance, to tell the runtime to use a particular set of instructions based on its architecture.

Different proposals exist to expand WebAssembly in future versions. In theory, version numbers will indicate compatibility. But mechanisms for feature detection will likely aid programmers. Research is needed to suggest standards for feature detection.

Appendix

2mm	2 Matrix Multiplications (alpha * A * B * C + beta * D)
3mm	3 Matrix Multiplications ((A*B)*(C*D))
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
deriche	Edge detection filter
doitgen	Multi-resolution analysis kernel
durbin	Toeplitz system solver
floyd-warshall	Shortest paths between each pair of nodes in a graph
fdtd-2d	2-D Finite Different Time Domain Kernel
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
head-3d	Heat equation over 3D data domain
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition followed by Forward Substitution
mvt	Matrix Vector Product and Transpose
nussinov	Dynamic programming algorithm for sequence alignment
seidel	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k update
syrk	Symmetric rank-k update
trisolv	Triangular solver
trmm	Triangular matrix-multiply

Table 7.1: Benchmarks in PolyBenchC version 4.2.1

Bibliography

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

[2] Ulan Degenbaev et al. 'Cross-Component Garbage Collection'. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018). DOI: 10.1145/3276521. URL: https://doi.org/10.1145/3276521.

[3] Andreas Gal et al. 'Trace-Based Just-in-Time Type Specialization for Dynamic Languages'. In: 44.6 (2009), pp. 465–478. ISSN: 0362-1340. DOI: 10.1145/1543135.1542528. URL: https://doi.org/10.1145/1543135.1542528.

[4] Andreas Haas et al. 'Bringing the Web up to Speed with WebAssembly'. In: *SIGPLAN Not.* 52.6 (2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: https://doi.org/10.1145/3140587.3062363.

[5] Adam Hall and Umakishore Ramachandran. 'An Execution Model for Serverless Functions at the Edge'. In: IoTDI '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 225–236. ISBN: 9781450362832. DOI: 10.1145/3302505.3310084. URL: https://doi.org/10.1145/3302505.3310084.

[6] Aaron Hilbig, Daniel Lehmann and Michael Pradel. 'An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases'. In: *Proceedings of the Web Conference 2021*. WWW '21. Ljubljana, Slovenia: Association for Computing Machinery, 2021, pp. 2696–2708. ISBN: 9781450383127. DOI: 10.1145/3442381.3450138. URL: https://doi.org/10.1145/3442381.3450138.

[7] Abhinav Jangda et al. 'Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code'. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association,

2019, pp. 107–120. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/jangda.

[8] Evan Johnson et al. 'SFI safety for native-compiled Wasm'. In: 2021. DOI: 10.14722/ndss.2021.24078.

[9] Daniel Lehmann, Johannes Kinder and Michael Pradel. 'Everything Old is New Again: Binary Security of WebAssembly'. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 217–234. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

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

[11] Darya Melicher et al. 'A Capability-Based Module System for Authority Control'. In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Ed. by Peter Müller. Vol. 74. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 20:1–20:27. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.20. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7270.

[12] Marius Musch et al. 'New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild'. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Roberto Perdisci et al. Cham: Springer International Publishing, 2019, pp. 23–42. ISBN: 978-3-030-22038-9.

[13] Shravan Narayan et al. 'Swivel: Hardening WebAssembly against Spectre'. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021, pp. 1433–1450. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

[14] Árpád Perényi and Jan Midtgaard. 'Stack-Driven Program Generation of WebAssembly'. In: *Programming Languages and Systems*. Ed. by Bruno C. d. S. Oliveira. Cham: Springer International Publishing, 2020, pp. 209–230. ISBN: 978-3-030-64437-6.

[15] Marija Selakovic and Michael Pradel. 'Performance Issues and Optimizations in JavaScript: An Empirical Study'. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016, pp. 61–72. DOI: 10.1145/2884781.2884829.

[16] Thomas Shull et al. 'NoMap: Speeding-Up JavaScript Using Hardware Transactional Memory'. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 412–425. DOI: 10.1109/HPCA.2019.00054.

[17] Kunshan Wang et al. 'Hop, Skip, and Jump: Practical On-Stack Replacement for a Cross-Platform Language-Neutral VM'. In: *SIGPLAN Not.* 53.3 (2018), pp. 1–16. ISSN: 0362-1340. DOI: 10.1145/3296975.3186412. URL: https://doi.org/10.1145/3296975.3186412.

[18] Conrad Watt. 'Mechanising and Verifying the WebAssembly Specification'. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 53–65. ISBN: 9781450355865. DOI: 10.1145/3167082. URL: https://doi.org/10.1145/3167082.

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

[20] Alon Zakai. 'Emscripten: An LLVM-to-JavaScript Compiler'. In: OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 301–312. ISBN: 9781450309424. DOI: 10.1145/2048147.2048224. URL: https://doi.org/10.1145/2048147.2048224.

[21] Alon Zakai. 'Fast Physics on the Web Using C++, JavaScript, and Emscripten'. In: *Computing in Science Engineering* 20.1 (2018), pp. 11–19. DOI: 10.1109/MCSE.2018.110150345.