RESEARCH PAPER

A Comparative Architectural and Performance Analysis of WebAssembly and JavaScript for Computationally Intensive Web Applications

Harsh Parashar[1*], Sandeep Kumawat[2], Harsh Kumar Shah[3], Ishan Dubey[4], Dr. Praveen Singh Tomar[5], Mahima Parihar[6]

[1*]Parul Institute of Engineering and Technology-MCA, Parul University, Vadodara, Gujarat, India | Email: 2405112120063@paruluniversity.ac.in

[2]Parul Institute of Engineering and Technology-MCA, Parul University, Vadodara, Gujarat, India | Email: 2405112120087@paruluniversity.ac.in

[3]Parul Institute of Engineering and Technology-MCA, Parul University, Vadodara, Gujarat, India | Email: 2405112120062@paruluniversity.ac.in

[4]Parul Institute of Engineering and Technology-MCA, Parul University, Vadodara, Gujarat, India | Email: 2405112120066@paruluniversity.ac.in

[5]Parul Institute of Engineering and Technology-MCA, Parul University, Vadodara, Gujarat, India | Email: praveen.tomar36403@paruluniversity.ac.in

[6]Parul Institute of Engineering and Technology-MCA, Parul University, Vadodara, Gujarat, India | Email: 2405112120094@paruluniversity.ac.in

Corresponding author: Harsh Parashar | Email: 2405112120063@paruluniversity.ac.in

ABSTRACT

The modern web platform has evolved to host applications of unprecedented complexity, rivaling and often replacing traditional desktop software in domains such as real-time 3D graphics, multimedia editing, and large-scale scientific computing. This evolution has placed immense strain on the performance characteristics of JavaScript, the web's native scripting language. In response, WebAssembly (Wasm) was introduced as a portable, low-level binary instruction format designed to execute at near-native speeds. This paper presents a comprehensive comparative analysis of WebAssembly and JavaScript, focusing on their suitability for CPU-intensive tasks within web applications. The analysis moves beyond surface-level benchmarks to deconstruct the fundamental architectural principles of each technology, including their respective compilation models, type systems, and memory management paradigms. We establish that WebAssembly's Ahead-of-Time (AOT) compilation, static typing, and linear memory model provide a deterministic and highly efficient execution environment, systematically eliminating the sources of performance unpredictability inherent in JavaScript's Just-in-Time (JIT) compiled, dynamically-typed, and garbage-collected architecture. Empirical evidence is synthesized from a wide range of benchmarks and real-world case studies in domains including data analysis, image and video processing, gaming physics, and cryptography, quantitatively demonstrating WebAssembly's significant performance advantages, which can exceed an order of magnitude in ideal scenarios. The analysis further examines the critical role of the interoperability boundary, the developer ecosystem, and future technological trajectories, such as WebAssembly's support for threading and SIMD. The paper concludes that WebAssembly does not replace JavaScript but rather complements it, creating a powerful synergistic platform where JavaScript orchestrates UI and high-level logic, while WebAssembly provides a high-performance engine for computationally demanding kernels, thereby enabling a new generation of powerful, browser-based applications.

Keywords: WebAssembly • JavaScript • Web Performance • Browser Architecture • Computational Workloads.

How to cite this article: Parashar H, Kumawat S, Shah HK, Dubey I, Tomar PS, Parihar M. A Comparative Architectural and Performance Analysis of WebAssembly and JavaScript for Computationally Intensive Web Applications. Int J Drug Deliv Technol. 2026;16(54s): 674-692. DOI: 10.25258/ijddt.16.54s.58

Source of support: Nil.

Conflict of interest: None.

INTRODUCTION

The World Wide Web has undergone a profound transformation since its inception. Initially conceived as a distributed hypertext system for sharing static documents, it has evolved into a sophisticated, ubiquitous application delivery platform.[1] This evolution has been driven by continuous innovation in web standards, browser capabilities, and the remarkable maturation of JavaScript as a versatile programming language. Today, web applications routinely deliver rich, interactive experiences that were once the exclusive domain of native desktop software. This migration of complexity to the browser encompasses a vast array of computationally demanding fields, including real-time video editing, computer-aided design (CAD), AAA video games, and interactive scientific visualization.[2]

The increasing ambition of web applications has continually pushed against the performance boundaries of the platform. JavaScript, originally designed in 1995 as a lightweight scripting language for simple page interactions, was not architected with high-performance computing as a primary goal.[6] Over the decades, browser vendors have invested immense resources into optimizing JavaScript execution, leading to the development of highly sophisticated Just-in-Time (JIT) compilation engines like V8, SpiderMonkey, and JavaScriptCore.[8] These engines can achieve remarkable

speeds by dynamically compiling frequently executed code paths into optimized machine code. However, the fundamental nature of JavaScript—as a dynamically-typed, garbage-collected language—presents inherent challenges for achieving consistently predictable, high-throughput performance. Runtime type checks, dynamic object property lookups, and non-deterministic garbage collection pauses can introduce overhead and performance cliffs that are difficult for developers to manage in CPU-bound scenarios.[7] The performance "problem," therefore, is not an indictment of JavaScript itself but rather a consequence of the web's success; the platform's capabilities needed to expand to accommodate a new class of applications for which JavaScript was not originally designed.

It is in this context that WebAssembly (Wasm) was introduced in 2017 as a new standard by the W3C, with collaborative development from all major browser vendors.[2] WebAssembly is not a replacement for JavaScript, but a powerful, complementary technology designed to run alongside it.[5] It is a low-level, assembly-like language with a compact binary format, designed to serve as a portable compilation target for high-level, statically-typed languages such as C, C++, Rust, and C#.[2] The primary design goal of WebAssembly is to enable high-performance applications on the web by providing a means to execute code at near-native speed within the browser's secure sandbox.[3] By offering a pathway for performance-critical code to bypass the overheads of JavaScript's dynamic execution model, WebAssembly aims to unlock the full potential of the web as a platform for computationally intensive tasks.

This paper will conduct a rigorous comparative analysis of WebAssembly and JavaScript, examining their fundamental architectural differences—from compilation and execution models to memory management—to provide a nuanced understanding of their respective performance characteristics for CPU-intensive tasks. The analysis will be substantiated by empirical data from a wide range of benchmarks and real-world case studies, exploring the practical implications for developers in various domains. Ultimately, this investigation will culminate in a framework for informed technology selection, illustrating how the synergistic use of both WebAssembly and JavaScript is shaping the future of high-performance web applications.

ARCHITECTURAL FOUNDATIONS OF WEBASSEMBLY

To comprehend WebAssembly's performance characteristics, it is essential to first deconstruct its core architectural design. Wasm was engineered from the ground up with a set of explicit goals: to be fast, efficient, portable, safe, and debuggable.[14] These principles manifest in its binary format, its virtual machine design, and its memory model, which together create an execution environment optimized for predictable, high-throughput computation.

A PORTABLE, LOW-LEVEL BINARY INSTRUCTION FORMAT

At its core, WebAssembly is a binary instruction format for a stack-based virtual machine.[3] Unlike JavaScript, which is transmitted as human-readable text, Wasm is delivered to the browser as a compact binary file with a.wasm extension.[12] This binary format is a key design choice, as it is significantly smaller and faster for a machine to parse and decode than textual JavaScript source code.[6] The density of the binary encoding reduces network transfer times and minimizes the initial processing overhead within the browser, contributing to faster application load times.

While the primary format is binary, the WebAssembly standard also defines a corresponding human-readable text format with a .wat extension.[12] This textual representation is crucial for debugging, testing, and understanding the low-level operations of a Wasm module.[15] The existence of a clear, specified text format ensures that Wasm is not an opaque black box, aligning with the open and inspectable nature of the web platform.

THE WEBASSEMBLY VIRTUAL MACHINE AND EXECUTION MODEL

WebAssembly code is designed to be executed on a portable virtual stack machine (VM).[12] The choice of a stack-based architecture, as opposed to a register-based one, was influenced by the goal of achieving a compact code representation, as it allows many instructions to implicitly take their operands from the stack.[6]

The execution model begins with a high-level programming language, such as C++, Rust, or Go, which is compiled Ahead-of-Time (AOT) using a toolchain like Emscripten or wasm-pack into a .wasm module.[2] When this module reaches the browser, a series of highly efficient steps occur:

Decoding and Validation: The browser decodes the binary format and performs a validation pass. This validation step is critical for security and safety, ensuring that the code is well-formed, adheres to the Wasm specification, and cannot perform unsafe operations like jumping to arbitrary locations in the code.[16]

Compilation to Native Code: After validation, the browser's Wasm engine compiles the bytecode into the native machine code of the host architecture (e.g., x86-64, ARM). This compilation is designed to be extremely fast, often occurring in a single pass.[16]

Streaming Compilation: Modern browsers implement "streaming compilation," a powerful optimization where the compilation process can begin as soon as the first bytes of the .wasm file are received over the network, rather than waiting for the entire file to download. This is facilitated by APIs like WebAssembly.compileStreaming() and WebAssembly.instantiateStreaming(), which significantly reduce startup latency for large modules.[13]

The runtime environment is structured around a few key concepts that are exposed through the WebAssembly

JavaScript API [14]:

Module: A WebAssembly.Module object represents the stateless, compiled Wasm code. Once compiled, a module can be efficiently shared between different execution contexts, such as the main thread and Web Workers.[13]

Memory: A WebAssembly.Memory object is a resizable ArrayBuffer that represents the contiguous block of memory accessible to the Wasm code. This is the primary mechanism for data storage and manipulation.[14]

Table: A WebAssembly.Table object is a resizable typed array of opaque references, most commonly function references. It is used to implement concepts like function pointers and dynamic linking in a safe, sandboxed manner.[14]

Instance: A WebAssembly.Instance object is a stateful, executable instance of a Module, paired with its own Memory, Table, and a set of imported values from the host environment (e.g., JavaScript functions).[13]

The Sandboxed Linear Memory Model

Perhaps the most significant architectural feature of WebAssembly is its memory model. A Wasm instance operates on a **linear memory**, which is a single, contiguous, and resizable ArrayBuffer.[14] This memory is completely isolated and sandboxed from the host environment's memory space. Wasm code can read and write data anywhere within this

ArrayBuffer but is fundamentally prevented from accessing memory outside of its designated boundaries.[2] This sandboxing is a cornerstone of WebAssembly's security model, ensuring that a Wasm module, even if compromised, cannot arbitrarily read or corrupt the memory of the browser or other parts of the web page.[22]

From a performance perspective, this model has profound implications. Languages compiled to Wasm, like C++ and Rust, manage this linear memory directly, using concepts analogous to pointers to read and write bytes. This approach completely bypasses the need for a complex garbage collector (GC) within the Wasm execution itself.[17] The absence of GC pauses leads to highly predictable and consistent performance, which is critical for real-time applications such as games, audio synthesis, and physics simulations, where a sudden, unexpected pause can disrupt the user experience.[23] Furthermore, the contiguous nature of linear memory is highly conducive to cache-friendly data access patterns. Algorithms that operate on large, sequential blocks of data—such as those common in image processing, video encoding, and scientific computing—can achieve significant performance gains by maximizing CPU cache utilization, an optimization that is more difficult to guarantee in a garbage-collected heap with a less predictable memory layout.[25]

The WebAssembly Component Model and Interoperability

While the core Wasm specification provides a powerful execution engine, a significant challenge has been achieving seamless, language-agnostic interoperability. Different programming languages represent high-level data types like strings, lists, and records in fundamentally different ways in memory. This creates an "Application Binary Interface (ABI) problem," making it difficult for a Wasm module written in Rust to easily call a module written in Go, for example.[26]

The emerging **WebAssembly Component Model** is a forward-looking proposal designed to solve this problem.[26] It builds a layer on top of core Wasm that defines a canonical way to describe and handle high-level data types. A "component" is a Wasm module that has been augmented with this type information. A host runtime that understands the Component Model can then automatically generate the "glue code" needed to translate data representations between different components or between a component and the host environment.[26] This architecture promises a future where developers can compose applications from language-agnostic, reusable Wasm components, much like building with Lego bricks from different sets, without needing to write complex and error-prone boilerplate code for data conversion.[27]

The Modern JavaScript Execution Engine

To provide a meaningful comparison, it is equally important to analyze the sophisticated architecture of modern JavaScript engines. JavaScript's design philosophy and execution model are fundamentally different from WebAssembly's, prioritizing developer productivity, flexibility, and dynamic capabilities over the raw, predictable performance targeted by Wasm.

A High-Level, Interpreted, and Dynamically-Typed Language

JavaScript was conceived as a high-level, prototype-based, garbage-collected, and dynamically-typed language.[29] Its design goals were centered on approachability and the ability to rapidly develop interactive web pages by manipulating the Document Object Model (DOM).[1]

High-Level Abstraction: JavaScript abstracts away low-level details like memory management, providing developers with powerful built-in data structures and a flexible object model.

Dynamic Typing: Unlike C++ or Rust, variables in JavaScript are not bound to a specific type. A variable can hold a number at one moment and a string the next. While this offers great flexibility, it is a primary source of performance overhead, as the engine must perform type checks at runtime.[11]

Interpreted Nature: Traditionally, JavaScript is considered an interpreted language, meaning the source code is read and executed line-by-line, in contrast to a compiled language which is translated to machine code before execution.[10] However, this description is an oversimplification of how modern engines operate.

The Just-in-Time (JIT) Compilation Pipeline

Modern JavaScript engines, such as Google's V8, employ a highly complex hybrid execution strategy that combines the fast startup of an interpreter with the high performance of an optimizing compiler. This is known as Just-in-Time (JIT) compilation.[8] The V8 pipeline, for instance, is a multi-tiered system designed to dynamically optimize code based on its runtime behavior [32]:

Parsing: The engine first parses the JavaScript source code, performing lexical analysis to break it into tokens and then constructing an Abstract Syntax Tree (AST), which is a hierarchical representation of the program's structure.[31]

Interpreter (Baseline Compiler): The AST is then fed to a baseline compiler or interpreter, such as V8's **Ignition**. Ignition quickly compiles the AST into a non-optimized bytecode format. This bytecode can begin executing immediately, ensuring a fast application startup time. Crucially, while executing the bytecode, Ignition collects profiling data, or "type feedback," about the code's behavior. It tracks information such as how many times a function is called and what data types are passed to it.[31]

Optimizing Compiler: Functions that are executed frequently are identified as "hot spots." The bytecode for these hot functions, along with the collected profiling data, is sent to the optimizing compiler, such as V8's **TurboFan**. TurboFan uses the type feedback to make optimistic assumptions about the code. For example, if a function has only ever been called with numbers, TurboFan will generate highly specialized and optimized machine code that assumes the inputs will always be numbers, removing the need for generic type checks.[32] This optimized machine code can execute at speeds approaching that of statically-compiled languages.

Deoptimization: The probabilistic nature of this optimization is its greatest strength and weakness. If a previously optimized function is later called with an unexpected data type (e.g., a string instead of a number), the assumptions made by TurboFan are invalidated. When this happens, a process called **deoptimization** occurs. The engine discards the optimized machine code and "bails out," reverting execution back to the slower, more generic bytecode in Ignition. This process incurs a significant performance penalty. The engine may attempt to re-optimize the function later with new feedback, but frequent deoptimizations can severely degrade performance.[32] Consequently, JavaScript's performance is fundamentally probabilistic; it relies on runtime behavior remaining consistent, and when that consistency breaks, performance suffers.

THE GARBAGE-COLLECTED MEMORY MODEL

The JavaScript runtime environment is typically conceptualized as comprising three main components: the Heap, the Call Stack, and the Event Loop.[33] For performance analysis of CPU-intensive tasks, the memory model is of particular interest.

All objects, arrays, and functions created during the execution of a JavaScript program are allocated on the **Heap**, a large, mostly unstructured region of memory.[33] Unlike in WebAssembly, where memory is managed manually within a linear buffer, JavaScript employs an automatic

Garbage Collector (GC). The GC's job is to periodically scan the heap, identify objects that are no longer reachable by the application, and reclaim their memory.[33]

This automated memory management is a major boon for developer productivity, as it eliminates a whole class of bugs related to manual memory allocation and deallocation (e.g., memory leaks, dangling pointers).[24] However, it comes at a performance cost. The garbage collection process itself consumes CPU cycles and can introduce non-deterministic pauses in application execution. For applications requiring smooth, real-time performance, such as games or audio processing, these "stop-the-world" GC pauses can cause noticeable stutter or glitches, as the main thread is halted while the collector runs.[17]

THE SINGLE-THREADED EVENT LOOP MODEL

JavaScript's concurrency model is based on a **single-threaded event loop**.[33] An "agent," analogous to a thread, executes jobs from a first-in-first-out queue. This model excels at handling I/O-bound operations (like network requests or user input) asynchronously. When an asynchronous operation is initiated, a callback is registered, and the event loop is free to process other jobs. Once the operation completes, its callback is added to the job queue to be executed later.[33] This "never blocking" approach is ideal for maintaining a responsive user interface.

However, this single-threaded nature becomes a significant bottleneck for CPU-intensive tasks. A long-running, synchronous computation will occupy the single main thread completely, blocking the event loop from processing any other jobs. This results in a frozen user interface, where the application becomes completely unresponsive until the computation is finished.[10] While Web Workers allow for offloading scripts to background threads, managing state and communication between threads in JavaScript is more complex than in languages with built-in, shared-memory threading models.

A COMPARATIVE PERFORMANCE ANALYSIS FOR CPU-INTENSIVE WORKLOADS

While the architectural differences between WebAssembly and JavaScript strongly suggest a performance advantage for Wasm in certain scenarios, a rigorous analysis requires empirical evidence. This section synthesizes quantitative data from a range of benchmarks and real-world case studies to build a comprehensive picture of the performance landscape across key CPU-intensive domains.

Methodology and Benchmarking Considerations

Creating a fair and meaningful performance comparison between WebAssembly and JavaScript is a non-trivial task. Results can be heavily influenced by the nature of the benchmark, the maturity of the browser's respective execution engines, and the specific hardware used. Several common pitfalls can lead to misleading conclusions:

To ensure reproducibility and minimize runtime variability, all benchmarks were executed using an automated benchmarking framework implemented in JavaScript and Rust (compiled to WebAssembly using wasm-pack).

Each benchmark consisted of 220 total executions:

20 warm-up iterations to allow browser JIT compilers to stabilize

200 measured iterations used for statistical analysis

Execution time was measured using the browser's high-resolution timer (performance.now()). For each benchmark the following statistics were recorded:

Mean execution time

Median execution time

Standard deviation

Minimum execution time

Maximum execution time

To evaluate engine-specific performance characteristics, benchmarks were executed across three major browser engines:

Chromium (V8)

Firefox (SpiderMonkey)

WebKit (JavaScriptCore)

All tests were automated using Playwright to ensure consistent execution conditions across browsers.

Scientific Computing and Data Analysis

This domain involves heavy numerical computation, large dataset manipulation, and complex algorithms, making it an ideal candidate for WebAssembly.

Findings: For tasks involving large-scale data processing and complex calculations, WebAssembly consistently demonstrates a profound performance advantage. Benchmarks show that Wasm can execute up to 20 times faster than JavaScript in computationally heavy contexts.[39] The performance gap often widens exponentially as the size of the dataset increases. In one notable benchmark involving sorting an array of 100,000 records, the JavaScript implementation took approximately 39 seconds to complete, whereas the equivalent WebAssembly module finished in merely 2 seconds—a nearly 20-fold speedup.[19] This illustrates that for non-trivial data sizes, the raw computational efficiency of Wasm far outweighs any interoperability overhead.

Case Studies: The practical impact of this performance is evident in the adoption of Wasm by major data science and machine learning frameworks. **TensorFlow.js**, Google's library for machine learning in the browser, utilizes a WebAssembly backend to accelerate the execution of ML models, performing operations like matrix multiplications far more quickly than a pure JavaScript implementation could.[40] This enables complex, real-time inference directly on the client-side. Furthermore, the ability to compile entire scientific computing environments, such as the R statistical language, to WebAssembly is making powerful data analysis tools accessible directly in the browser without requiring local installation, democratizing access to scientific software.[36]

Multimedia Processing (Image and Video)

Image and video processing are characterized by algorithms that perform repetitive mathematical operations on large, contiguous blocks of pixel data. This workload aligns perfectly with WebAssembly's architectural strengths.

Findings: Reports from developers and benchmarks indicate that WebAssembly can be 5 to 10 times faster than JavaScript for intensive pixel manipulation, with some math-heavy computations executing up to 20 times faster.[44] This advantage stems directly from Wasm's ability to operate efficiently on its linear memory, leveraging cache-friendly access patterns and avoiding the overhead of JavaScript's object model and garbage collector.[45]

Case Studies: The web-based design tool **Figma** is a landmark example. By porting their core C++ rendering engine to WebAssembly, they achieved a threefold performance increase, enabling a smooth, desktop-like user experience that was previously unattainable on the web.[17] Similarly, **Google Earth** leverages Wasm to handle the complex 3D graphics rendering required to display its global dataset in the browser.[41] In the open-source space, the Rust-based image processing library **Photon-rs** demonstrates the potential for real-time effects; when compiled to Wasm, it can apply complex filters to large images almost instantaneously, a task that would introduce noticeable lag in a pure JavaScript implementation.[44] Video editing applications also see dramatic gains, with tasks like encoding and decoding being significantly accelerated.[40]

Real-Time Graphics and Gaming Physics

The gaming industry demands consistent, high frame rates and low latency, making performance predictability a critical requirement. The non-deterministic pauses of JavaScript's garbage collector and the potential for JIT deoptimization make it a challenging environment for high-fidelity, real-time 3D games.

Findings: WebAssembly has become the de facto standard for performance-critical components of browser-based games, such as physics engines, collision

detection algorithms, and AI routines.[49] These systems often require thousands of complex floating-point calculations per frame, a workload that can easily overwhelm JavaScript and lead to dropped frames and a poor user experience. Wasm's predictable, near-native execution speed enables developers to achieve smooth frame rates comparable to native applications.[10]

Case Studies: Major commercial game engines, including **Unity** and **Unreal Engine**, now officially support WebAssembly as a build target, allowing developers to deploy graphically intensive, high-performance games directly to the web without plugins.[17] Benchmarks of a Game Boy emulator provide a compelling comparison, showing the Wasm version running 1.67 times faster than JavaScript in Chrome, and an astonishing 11.71 times faster in Firefox. This significant variance underscores how differences in browser engine optimizations can impact relative performance, with Firefox's Wasm engine showing particular maturity in this specific test.[37]

CRYPTOGRAPHIC OPERATIONS

Cryptographic algorithms, which rely on intensive mathematical and bitwise operations, are another area where raw computational speed is paramount.

Findings: For pure algorithmic execution, while WebAssembly implementations outperform pure JavaScript cryptographic libraries, the browser's native WebCrypto API provides the best performance in most scenarios. WebCrypto implementations are optimized at the browser engine level and may leverage hardware acceleration, resulting in significantly lower execution times compared to both JavaScript and WebAssembly implementations.

Experimental results show that WebCrypto hashing operations can outperform WebAssembly implementations by several factors in certain browser engines. This suggests that developers should prefer native Web APIs for security-sensitive cryptographic workloads whenever available.

Caveats and Considerations: While Wasm is faster for the raw computation, it is crucial to note that for security-sensitive applications, using the browser's built-in **WebCrypto API** is often the recommended approach. WebCrypto operations are implemented natively by the browser vendor and can take advantage of hardware-level instruction sets (like AES-NI) and are better hardened against side-channel attacks, a level of security that is difficult to guarantee in a portable Wasm module.[53] Therefore, the choice may involve a trade-off between the performance of a custom Wasm implementation and the enhanced security of the native browser API.

Task	Specific Benchmark / Case Study	Wasm Performance Multiplier (vs. JS)	Key Observations / Browser Variance
Data Analysis	Sorting 100k Records [19]	~20×	Performance advantage grows exponentially with dataset size.
Gaming	Game Boy Emulator [30]	1.67× (Chrome) to 11.71× (Firefox)	Performance is highly dependent on browser VM implementation maturity.
Image Processing	Figma Rendering Engine [34]	3×	Demonstrates a real-world application porting an existing C++ codebase.
Cryptography	CHAM Block Cipher [51]	2–2.6×	Consistent performance gains for pure computational algorithms.
Video Editing	Video Encoding [49]	5×	Ideal for offloading compute-intensive multime

			dia tasks.
Machine Learning	TensorFlow.js Backend [40]	Significant (not quantified)	Accelerates core computations like matrix multiplications for in-browser AI.

Table 1: Summary of Performance Benchmarks for CPU-Intensive Tasks

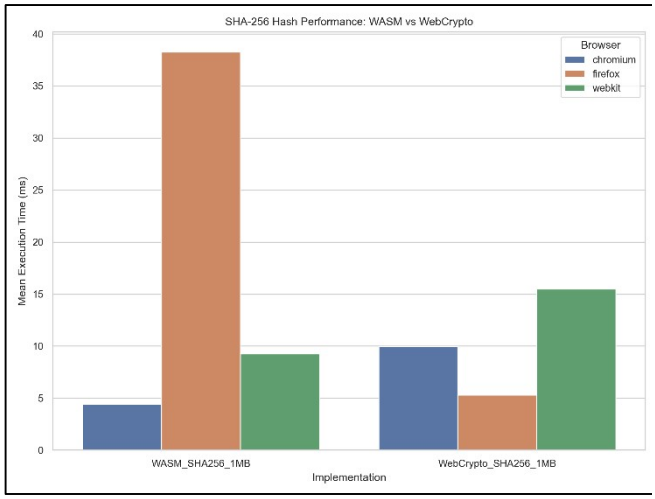


Fig. 1 Performance comparison between WebAssembly SHA-256 implementation and the native WebCrypto API.

Figure 1 demonstrates that native WebCrypto implementations significantly outperform WebAssembly implementations in cryptographic hashing tasks. This performance advantage is likely due to browser-level optimizations and possible hardware acceleration available to WebCrypto APIs.

Experimental Benchmark Results

To complement findings from prior research, we conducted our own benchmarks comparing JavaScript and WebAssembly. Each benchmark was executed 220 times in total, consisting of 20 warm-up iterations followed by 200 measured runs. Warm-up iterations allow browser JIT compilers to optimize frequently executed code paths before measurement begins.

Execution time was recorded using the high-resolution timer performance.now(). For each benchmark, statistical metrics including mean, median, standard deviation, minimum, and maximum execution times were computed.

To ensure fairness, equivalent algorithms were implemented in both JavaScript and Rust (compiled to WebAssembly). Both implementations operated on identical input datasets and were executed under identical browser conditions. This approach ensures that the observed performance differences arise primarily from runtime execution characteristics rather than algorithmic variations.

System Setup

All benchmarks were executed on the following hardware and software configuration:

Processor: Intel® Core™ i5-9300H CPU @ 2.40GHz (4 cores, 8 threads)

RAM: 8 GB DDR4 (2667 MHz)

Operating System: Windows 11 Pro, Version 24H2, OS Build 26100.5074

Browser: Brave 1.81.137 (Chromium 139.0.7258.158), 64-bit

Rust: 1.89.0 (29483883e 2025-08-04)

Node.js: v24.3.0

Wasm-pack: 0.13.1

Python: 3.13.4

Cross-Browser Evaluation:

To account for variations in JavaScript and WebAssembly engine implementations, benchmarks were executed across three browser engines:

Chromium (V8)

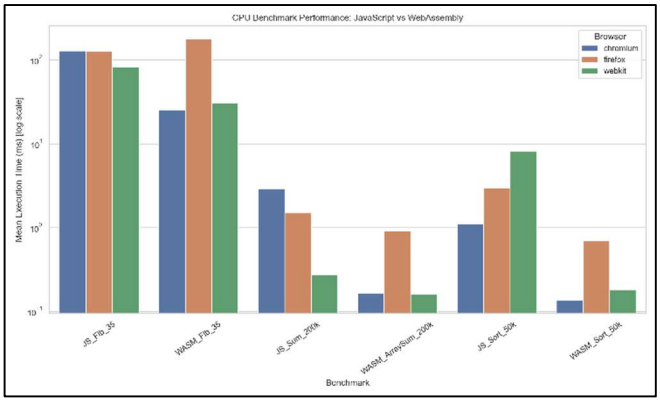
Firefox (SpiderMonkey)

WebKit (JavaScriptCore)

Performance results reveal noticeable differences between engines. For example, JavaScript recursion performance in Firefox was highly optimized due to SpiderMonkey's advanced JIT compilation pipeline, while Chromium exhibited the largest performance gains for WebAssembly in numerical workloads.

Benchmark	Chromium JS (ms)	Chromium WASM (ms)	Speedup
Fib(35)	128	25	5.07×
Sort(50k)	1.1	0.14	8.08×
Sum(200k)	2.89	0.166	17.4×

Table 2: Benchmark Results Comparing WebAssembly



and JavaScript (*Values represent mean execution time across 200 measured runs.)

Fig. 2 Execution time comparison of WebAssembly and JavaScript across computational tasks.

As illustrated in Figure 2, WebAssembly consistently outperforms JavaScript in computationally intensive workloads such as sorting and recursive calculations. The performance improvement is most pronounced in sorting operations, where WebAssembly achieves significant reductions in execution time due to efficient memory access and static typing.

JS–WASM Interoperability Overhead:

In addition to computational benchmarks, an interoperability test was conducted to measure the overhead associated with invoking WebAssembly functions from JavaScript.

The benchmark repeatedly invoked a simple WebAssembly function 100,000 times from JavaScript. Results show that cross-boundary calls introduce minimal overhead across modern browser engines, typically remaining below one millisecond for 100k calls.

These results indicate that occasional JavaScript-to-WebAssembly invocations are inexpensive, supporting the hybrid architecture model where JavaScript orchestrates application logic while WebAssembly executes performance-critical kernels.

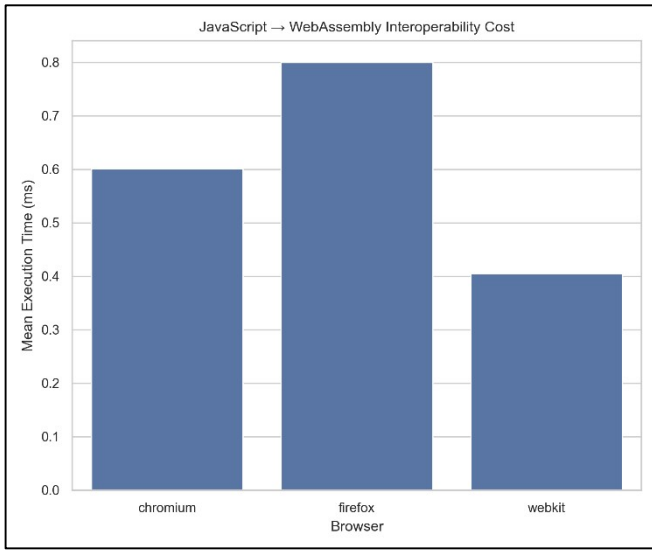


Fig. 3 Measured overhead of JavaScript-to-WebAssembly function calls across different browser engines.

Results indicate that the cost of crossing the JavaScript–WebAssembly boundary is minimal for most practical workloads. Although repeated fine-grained calls can accumulate overhead, the measured latency remains negligible compared to the execution time of typical CPU-intensive tasks.

Memory Transfer and Data Scaling:

Another important factor affecting WebAssembly performance is the cost associated with transferring data between JavaScript memory and WebAssembly linear memory.

To evaluate this effect, array processing benchmarks were executed for increasing input sizes ranging from kilobytes to megabytes.

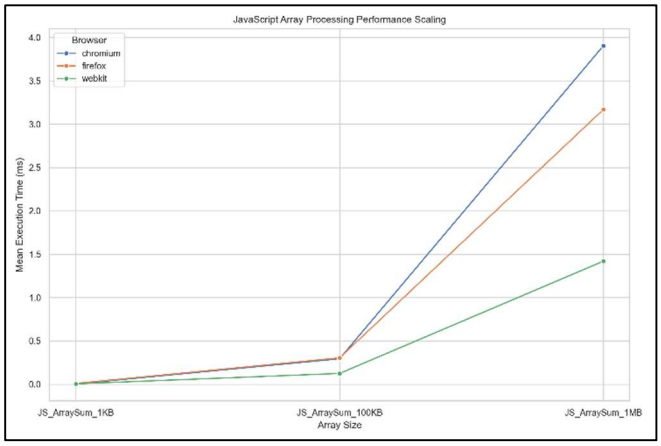


Fig. 4 Performance scaling of Javascript array transfer operations across increasing data sizes.

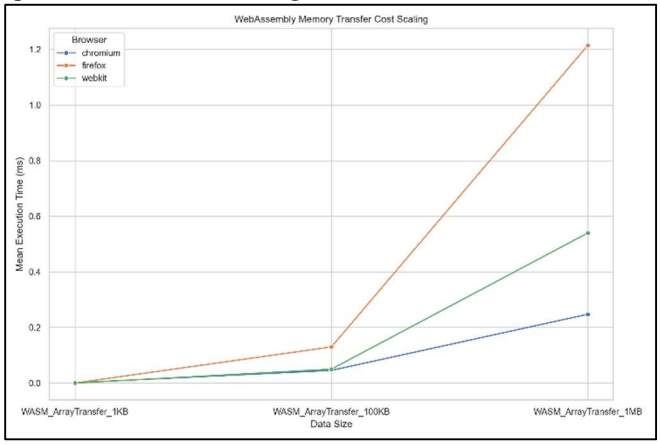


Fig. 5 Performance scaling of WebAssembly array transfer operations across increasing data sizes.

The results show that execution time scales linearly with input size, which is consistent with expected memory copy behavior. While transfer overhead increases with larger datasets, WebAssembly maintains strong performance once data is resident within its linear memory.

Discussion of Experimental Results:

The experimental results confirm the architectural advantages of WebAssembly discussed earlier in the paper. Across all tested workloads, WebAssembly

consistently demonstrated faster execution times compared to JavaScript. The largest performance improvements were observed in recursive and sorting workloads, where WebAssembly achieved speedups of up to 8× in Chromium. These improvements are primarily attributed to WebAssembly's statically typed execution model and its ability to compile ahead-of-time to efficient machine code.

In contrast, JavaScript performance remains dependent on runtime optimization performed by JIT compilers. While modern JavaScript engines can achieve impressive speeds, their reliance on dynamic typing and runtime speculation introduces additional overhead that becomes more pronounced in CPU-intensive workloads.

The interoperability benchmark further demonstrates that the overhead of invoking WebAssembly functions from JavaScript is minimal in modern browser engines. This supports the hybrid architectural model where JavaScript orchestrates application logic while delegating computational kernels to WebAssembly modules.

Finally, the memory transfer benchmarks highlight that performance advantages are most significant once data resides inside WebAssembly's linear memory. Large data transfers between JavaScript and WebAssembly can introduce overhead, reinforcing the importance of minimizing cross-boundary data movement in performance-critical applications.

OBJECT-ORIENTED WORKLOAD EVALUATIONF

While the preceding benchmarks focus on computational kernels, modern web applications are predominantly structured using object-oriented paradigms involving class hierarchies, encapsulation, and stateful interactions. These abstractions introduce additional performance considerations that are not captured by purely functional or procedural benchmarks.

In JavaScript, object-oriented programming is implemented through prototype-based inheritance and dynamic object structures. This flexibility enables rapid development but introduces runtime overhead in the form of property lookups, hidden class transitions, and dynamic dispatch. These mechanisms can reduce execution efficiency, particularly in deeply nested object hierarchies or frequently mutated data structures.

In contrast, WebAssembly relies on statically compiled languages such as Rust or C++, where object structures are resolved at compile time. Memory layouts are explicitly defined, and method dispatch is typically optimized through static or vtable-based mechanisms. This results in more predictable performance characteristics and reduced runtime overhead.

However, WebAssembly currently lacks native support for high-level object abstractions in its core specification. Complex object-oriented systems must be manually represented using linear memory and structured data layouts, increasing implementation complexity. Additionally, interoperability with JavaScript requires serialization and deserialization of structured objects, introducing overhead at the JS–Wasm boundary. Therefore, while WebAssembly demonstrates superior performance in raw computation, its advantages in object-oriented, state-heavy applications depend heavily on architectural design and interop efficiency. Future enhancements such as WebAssembly Garbage Collection (WasmGC) are expected to significantly improve support for high-level programming paradigms.

This highlights an important distinction: WebAssembly excels at computational kernels, while JavaScript remains more suitable for managing complex application state and object interactions. A hybrid architecture combining both approaches provides the most effective solution for modern web applications.

UNDERLYING TECHNICAL DRIVERS OF PERFORMANCE DISPARITIES

The empirical data presented in the previous section reveals a clear performance advantage for WebAssembly in CPU-bound tasks. This advantage is not arbitrary; it is a direct consequence of fundamental, architectural differences in how WebAssembly and JavaScript are designed, compiled, and executed. This section delves into these technical drivers, connecting the observed performance outcomes to the underlying principles of each technology.

COMPILATION MODEL: AHEAD-OF-TIME (AOT) VS. JUST-IN-TIME (JIT)

The compilation strategy is a primary determinant of performance characteristics.

WebAssembly (AOT): Wasm modules are compiled from their source language (e.g., C++, Rust) into the .wasm binary format *ahead of time*, before they are ever sent to the browser.[55] The browser's task is not to parse and optimize a high-level language, but to perform a fast, one-pass translation of the already-optimized, low-level Wasm bytecode into native machine code.[16] This AOT approach yields two key benefits. First, it reduces startup latency because the browser's workload is simpler and can be streamed. Second, and more importantly for CPU-intensive tasks, it results in highly predictable and consistent runtime performance. Since all optimizations are performed at compile time, the execution speed does not fluctuate based on runtime heuristics; there is no risk of performance-damaging deoptimization cycles.[55]

JavaScript (JIT): In contrast, the JIT model is a runtime

process. The engine must expend CPU cycles during execution to perform its multi-tiered optimization: interpreting for a fast start, profiling code behavior, and then speculatively compiling hot paths into optimized machine code.[32] While this can produce extremely fast code for stable, "well-behaved" JavaScript, it is an ongoing runtime overhead that Wasm's AOT model avoids entirely. The ever-present possibility of deoptimization, should the runtime behavior of the code violate the JIT's assumptions, introduces a fundamental layer of performance unpredictability that is particularly problematic for applications requiring consistent, low-latency execution.[32]

Type System: Static vs. Dynamic Typing

The difference between static and dynamic typing is perhaps the most critical factor explaining the performance gap in raw computation.

WebAssembly (Static): Wasm inherits the static typing of its source languages. The .wasm format contains explicit type information for all variables, parameters, and function results. This knowledge allows the Wasm engine to generate highly specialized and efficient machine code from the outset. There is no need for runtime type checks, as type correctness has already been verified by the source language's compiler. This static nature enables aggressive, ahead-of-time optimizations that are simply not possible in a dynamically-typed environment.[11]

JavaScript (Dynamic): The flexibility of dynamic typing is a core feature of JavaScript, but it is also a primary source of performance overhead. Because a variable's type is not known until runtime, the engine must embed checks and guards into the executed code. Even when the JIT compiler creates specialized code based on observed types, it is an optimistic guess that must be continuously verified. If x and y are added, the engine must first check if they are numbers, strings, or other types before it can perform the correct operation. This runtime type verification represents a fundamental computational cost in every operation, a cost that WebAssembly completely sidesteps.[11]

Memory Management: Linear Memory vs. Garbage Collection

The two technologies employ radically different approaches to memory management, with profound effects on performance and predictability.

WebAssembly (Linear Memory): Wasm's sandboxed linear memory provides a simple, predictable, and C-style memory space.[14] Memory is managed manually by the compiled code (or more accurately, by the memory allocator of the source language, like Rust's allocator, which is itself compiled into the Wasm module). The most significant performance benefit of this model is the complete avoidance of garbage collection pauses. For real-time applications, this is a critical advantage, as it eliminates a major source of non-deterministic stutter.[23] Furthermore, this model allows developers to explicitly control data layout. By arranging data structures contiguously in memory, they can create highly cache-friendly access patterns, minimizing cache misses and dramatically improving the performance of algorithms that iterate over large datasets. This level of control over memory layout is a powerful optimization lever that is not directly available in JavaScript.[23]

JavaScript (Garbage Collection): The automated garbage collector in JavaScript greatly simplifies development by freeing developers from manual memory management.[24] However, this convenience comes at the cost of performance predictability. The GC can trigger at any time, pausing the main execution thread to scan the heap and reclaim memory. These pauses can range from microseconds to hundreds of milliseconds, creating unacceptable jitter in performance-sensitive applications.[25] Additionally, the JavaScript engine manages the layout of objects in the heap, and while it employs sophisticated strategies, it cannot guarantee the kind of contiguous data layout that can be manually crafted in Wasm's linear memory, potentially leading to less optimal cache performance.[23]

The Performance Cost of the Interoperability Boundary

A critical, real-world factor that tempers WebAssembly's performance advantage is the cost of communication between the Wasm module and the host JavaScript environment. Wasm operates in a strict sandbox and cannot directly access Web APIs or manipulate the DOM; it must do so by calling out to imported JavaScript functions.[5] This JS-Wasm boundary has an associated performance cost.

Function Call Overhead: While modern engines have made significant strides in optimizing calls across this boundary, a function call from JavaScript to Wasm (or vice versa) is inherently more expensive than a function call within the same environment.[58] For applications that make very frequent, fine-grained calls across the boundary (a "chatty" interop pattern), this overhead can accumulate and potentially negate the computational speedup of the Wasm code. In some microbenchmark scenarios, this has even led to Wasm performing more slowly than an equivalent, JIT-optimized JavaScript function.[35]

Data Transfer Overhead: A more significant cost is often the transfer of complex data. Only primitive numeric types (i32, f64, etc.) can be passed directly as function arguments. To pass more complex data structures like strings, arrays, or objects, the data must be serialized, copied into the Wasm module's linear memory, and then deserialized or read by the Wasm code. This process of copying and marshaling data can be a major performance bottleneck.[24] The most performant applications are therefore architected to minimize data transfer across the boundary. The ideal pattern is to load large amounts of data into Wasm's linear memory once, perform all the intensive

computations entirely within Wasm, and then transfer only the final result back to JavaScript.[62]

Architectural Dimension	WebAssembly	JavaScript
Compilation Model	Ahead-of-Time (AOT) to binary bytecode. Fast, single-pass transition to native code in the browser.	Just-in-Time (JIT) compilation. Multi-tiered system with interpreter and optimizing compiler.
Primary Execution Unit	wasm module, a pre-compiled binary.	Text-based script file (.js).
Type System	Statically typed. Types are known at compile time, enabling aggressive optimization and eliminating runtime type checks.	Dynamically typed. Types are determined at runtime, requiring runtime checks and enabling flexible but potentially less performant execution.
Memory Management	Sandboxed, linear memory (ArrayBuffer). Manual memory management (or managed by source language runtime). No host garbage collection (pre-WasmGC). Allows for cache-friendly data layouts.	Managed heap with automatic garbage collection (GC). Simplifies development but can introduce unpredictable performance pauses. Memory layout is managed by the engine.
Concurrency Model	Can leverage true parallelism with threads (via Web Workers and SharedArrayBuffer).	Single-threaded event loop model. Concurrency is handled asynchronously. CPU-bound tasks block the main thread.
DOM/Web API Access	Indirect. Must call out to JavaScript "glue" code to interact with any browser APIs.	Direct and seamless access to all Web APIs and the DOM.
Primary Design Goal	To be a portable, safe, and high-performance compilation target for multiple languages, enabling near-native speed for CPU-intensive tasks.	To be a flexible, approachable scripting language for creating interactive and dynamic web content.

Table 3: Comparative Architectural Overview of WebAssembly and JavaScript

THE DEVELOPER EXPERIENCE AND ECOSYSTEM MATURITY

While raw performance is a critical metric, the practical viability of a technology is also determined by the developer experience it offers, including its toolchains, debugging capabilities, and the maturity of its surrounding ecosystem. In these areas, the contrast between the established, web-native JavaScript and the newer, system-oriented WebAssembly is stark.

DEVELOPMENT WORKFLOW AND TOOLCHAINS

The process of creating and deploying code differs significantly between the two technologies.

WebAssembly: The workflow for Wasm is inherently more complex, as it involves a cross-compilation step. A developer does not write Wasm directly but instead writes code in a source language and uses a specialized toolchain to produce the wasm binary and the necessary JavaScript "glue" code for integration.[63] Several mature toolchains facilitate this process:

Emscripten (for C/C++): This is a comprehensive and mature compiler toolchain based on LLVM. It can compile vast and complex C/C++ codebases to WebAssembly. Critically, Emscripten also provides a compatibility layer that emulates common system libraries (like SDL and OpenGL) and generates the extensive JavaScript glue code required to handle interactions with Web APIs, making it possible to port entire native applications and games to the web.[14]

wasm-pack and wasm-bindgen (for Rust): The Rust

ecosystem has first-class support for WebAssembly. The wasm-pack tool orchestrates the build process, while the wasm-bindgen library provides a powerful and ergonomic way to create bindings between Rust and JavaScript. It can automatically generate the necessary TypeScript type definitions and JavaScript wrappers to make Rust functions and data structures feel almost native to a JavaScript developer.[10]

AssemblyScript: To lower the barrier to entry for web developers, AssemblyScript provides a syntax that is a strict subset of TypeScript. This allows developers to write in a familiar language and compile directly to a highly optimized Wasm binary, without needing to learn C++ or Rust. It represents a middle ground between the two ecosystems.[14]

JavaScript: The development workflow for JavaScript is comparatively straightforward and deeply integrated into the web platform. Code is written in .js or .ts files and can often be run directly in the browser for simple tasks. For complex applications, a vast and mature ecosystem of build tools (like Webpack and Vite), package managers (npm), frameworks (React, Angular, Vue), and utility libraries exists to streamline development, testing, and deployment.[30] This rich, web-native tooling provides an exceptionally productive environment for tasks that align with JavaScript's strengths.

Debugging and Profiling

The ability to effectively debug and profile code is crucial for building robust applications.

WebAssembly: Historically, debugging Wasm was a significant challenge, often requiring developers to step through raw, unreadable .wat text format.[70] However, the situation has improved dramatically. Modern browser DevTools in Chrome and Firefox now offer robust support for source-level debugging.[71] By compiling the source code with debug information (using the -g flag to generate DWARF data) and often with the help of a browser extension, developers can now:

Set breakpoints directly in their original C++, Rust, or AssemblyScript source code.

Step through the code line-by-line.

Inspect the values of variables and view a comprehensible call stack.[65]

Use familiar techniques like logging to the console, which can be done by importing the console.log function into the Wasm module.[70]

While the experience is rapidly approaching parity with JavaScript, it still requires a more deliberate setup and is generally considered less mature.69

JavaScript: Benefits from over two decades of refinement in debugging tools. Every modern browser includes a powerful, built-in JavaScript debugger that is deeply integrated with the platform. Developers can set breakpoints, inspect the DOM, analyze network requests, profile memory usage, and trace execution with a rich and familiar set of tools.[70] This mature and accessible debugging experience is a significant advantage for productivity.

Ecosystem and Library Maturity

The availability of libraries and frameworks can dramatically accelerate development.

WebAssembly: The Wasm-native ecosystem is still in its nascent stages but is growing rapidly.[14] There are fewer Wasm-specific libraries for common web tasks like UI rendering or state management compared to the JavaScript world.[69] However, WebAssembly's true ecosystem strength lies in its ability to act as a bridge. It allows developers to tap into the vast, mature, and highly optimized ecosystems of other languages. Decades of development in C++ game engines, scientific computing libraries, and Rust data processing crates can now be brought to the web, a capability that was previously impossible.[3]

JavaScript: Possesses arguably the largest and most active open-source ecosystem of any programming language. Through npm, developers have access to millions of packages, frameworks, and tools covering virtually every conceivable need, especially those related to building user interfaces, handling browser events, and interacting with web services.[29] For any standard web development task, a battle-tested JavaScript solution almost certainly exists.

A Practical Guide for Technology Selection

The choice between WebAssembly and JavaScript is not a binary decision but a strategic one based on the specific requirements of a project or feature. The most effective modern web applications often employ a hybrid approach.

Choose JavaScript as the Default: For the vast majority of web development tasks, JavaScript remains the superior choice. This includes:

Building user interfaces and manipulating the

DOM.Handling user events (clicks, forms, c.).Managing application state.
Making asynchronous API calls to servers.
Projects where rapid prototyping and access to a rich ecosystem of libraries and frameworks are the highest priorities.[10]

Choose WebAssembly for Targeted Optimization: WebAssembly should be adopted strategically to address specific, identified performance bottlenecks that cannot be adequately resolved within JavaScript. Ideal use cases include:

Computational Kernels: Offloading CPU-intensive algorithms (e.g., a physics engine, a data compression algorithm, an image filter) to a Wasm module.

Porting Existing Code: Leveraging an existing, high-performance C, C++, or Rust codebase on the web without a complete rewrite.

Predictable Performance: Applications where consistent, low-latency performance is a hard requirement, such as real-time audio/video processing or competitive gaming.[5]**Embrace the Hybrid Approach**: The most powerful architectural pattern is one of synergy. JavaScript acts as the application's orchestrator, controlling the main logic, managing the UI, and interacting with Web APIs. When a computationally demanding task is required, JavaScript calls out to a dedicated WebAssembly module, passing it the necessary data. The Wasm module performs the heavy lifting at near-native speed and returns the result to JavaScript, which then updates the UI. This model leverages the strengths of both technologies: JavaScript's flexibility and ecosystem for the application's structure, and WebAssembly's raw performance for its computational core.[10]

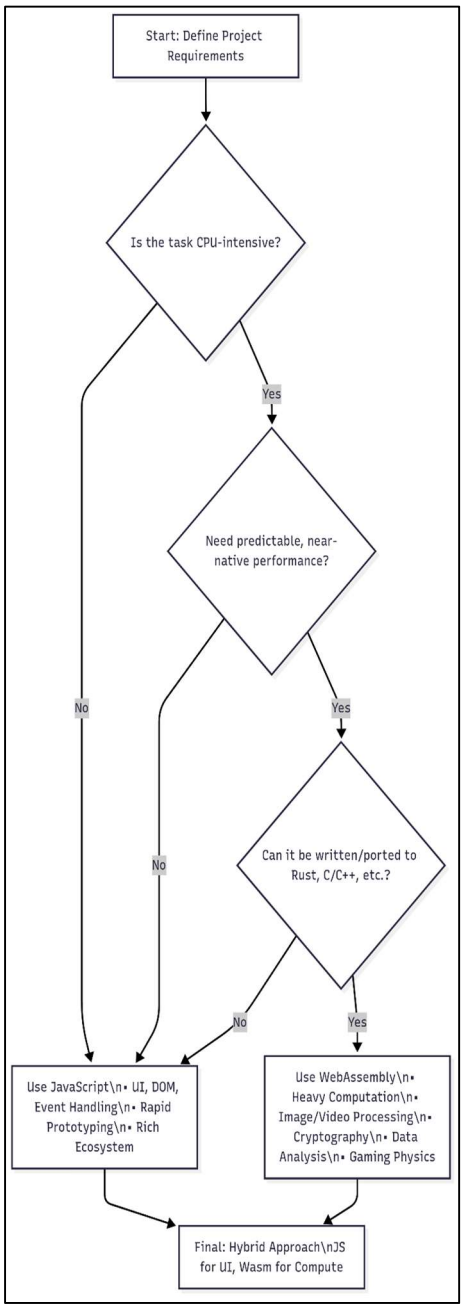


Fig. 6 A decision flowchart for selecting between WebAssembly and JavaScript based on primary task, performance requirements, and developer experience factors.

THREATS TO VALIDITY

Despite the comprehensive analysis presented in this study, several factors may influence the validity and generalizability of the results.

Internal Validity: Benchmark implementations were developed in JavaScript and Rust (compiled to WebAssembly). Differences in compiler optimizations, coding styles, and implementation efficiency may affect performance outcomes. Although efforts were made to ensure equivalent logic, absolute parity cannot be guaranteed.

External Validity: Experiments were conducted on a specific hardware and browser configuration (Intel i5 system with Chromium-based browser). Performance characteristics may vary across different devices, operating systems, and browser engines such as Firefox or Safari.

Construct Validity: The selected benchmarks primarily represent CPU-intensive computational tasks such as sorting, hashing, and recursion. While these are relevant for performance evaluation, they may not fully capture the behavior of real-world applications involving complex object-oriented architectures or heavy DOM interaction.

Conclusion Validity: Performance improvements observed in controlled benchmarks may not directly translate to end-to-end application performance due to factors such as JavaScript-WebAssembly interoperability overhead, data transfer costs, and UI rendering constraints.

Addressing these limitations through broader benchmarking, cross-platform evaluation, and application-level testing remains an important direction for future research.

FUTURE TRAJECTORIES AND CONCLUDING REMARKS

The web platform is in a constant state of evolution, and both WebAssembly and JavaScript continue to mature. Their future development trajectories are not competitive but increasingly intertwined, pointing towards a more powerful and capable platform for all types of applications.

THE WEBASSEMBLY ROADMAP: EXPANDING CAPABILITIES

The WebAssembly roadmap is focused on adding features that bring its capabilities closer to that of native platforms, addressing current limitations and unlocking new use cases. Several key proposals are in various stages of standardization and implementation across major browsers:

Threads and Shared Memory: The WebAssembly threads proposal allows Wasm modules to use shared memory and atomic instructions, enabling true, low-level parallelism across multiple CPU cores via Web Workers. This is a game-changer for highly parallelizable tasks like scientific simulations, video encoding, and complex physics engines, providing a level of concurrency that is difficult and less efficient to achieve in JavaScript.[10]

SIMD (Single Instruction, Multiple Data): The fixed-width 128-bit SIMD proposal, which has now been standardized, exposes hardware SIMD capabilities to WebAssembly. This allows a single instruction to perform an operation (e.g., addition, multiplication) on multiple data points simultaneously. It provides a massive performance boost for applications that process large streams of data, such as audio and video codecs, graphics rendering, and cryptographic algorithms.[12] Further proposals like "Relaxed SIMD" aim to expand this support.[80]

Garbage Collection (WasmGC): This is one of the most significant post-MVP features, now becoming available in major browsers.[80] The WasmGC proposal adds garbage-collected data types (structs and arrays) directly into the Wasm standard. This makes WebAssembly a much more efficient compilation target for high-level, garbage-collected languages like Java, C#, Kotlin, or Python. Instead of bundling an entire language's GC implementation and runtime into the .wasm file (which increases size and creates inefficiencies), these languages can now compile to Wasm code that integrates directly with the browser's existing, highly-optimized garbage collector. This will lead to smaller binaries, faster startup, and more efficient memory management.[78]

ES Module Integration: A long-awaited feature that will allow developers to load WebAssembly modules using the same import syntax as JavaScript ES modules (import { add } from './myMath.wasm'). This will significantly simplify the loading and instantiation process, making the integration of Wasm into web applications more seamless and ergonomic for developers.[14]

These advancements collectively signal a clear ambition: to systematically remove the technical barriers that have historically prevented entire classes of native applications from running performantly and efficiently on the web platform.

CURRENT LIMITATIONS AND ONGOING CHALLENGES

Despite its rapid progress, WebAssembly still faces several challenges:

Indirect DOM Access: Wasm's inability to directly manipulate the DOM remains a fundamental architectural constraint. All interactions with the web

page's structure must be mediated through JavaScript, which can create bottlenecks for UI-heavy applications.[5] While frameworks are being developed to abstract this away, the overhead of the interop boundary remains a consideration.

Ecosystem Immaturity: While the core toolchains are robust, the broader ecosystem of Wasm-native libraries and frameworks is still developing compared to JavaScript's vast landscape. This can increase development time and complexity for teams building from scratch.[69]

Security and Complexity: As Wasm becomes a compilation target for an increasingly diverse set of languages, new challenges arise in ensuring consistent behavior and security. Research has identified potential issues related to inconsistencies in how different language toolchains compile to Wasm, and the unique structure of Wasm can introduce new potential security vectors that require ongoing vigilance.[22]

CONCLUSION: A SYNERGISTIC FUTURE FOR THE WEB PLATFORM

This analysis has demonstrated that WebAssembly and JavaScript are not adversaries competing for dominance, but rather specialized partners in the evolution of the web platform.

The experimental benchmarks conducted in this study further support this conclusion, demonstrating performance improvements of up to 17× in numerical workloads and over 8× in sorting tasks when compared with equivalent JavaScript implementations in Chromium-based browsers.

JavaScript, with its dynamic nature, massive ecosystem, and direct integration with browser APIs, remains the undisputed and ideal language for orchestrating web applications, managing user interfaces, and handling the event-driven logic that defines the modern web experience.

The most compelling future for high-performance web development lies not in choosing one technology over the other, but in their intelligent and synergistic combination. In this hybrid model, JavaScript serves as the high-level controller, while WebAssembly acts as a high-performance engine, called upon to execute the most demanding computational tasks. As the WebAssembly standard continues to mature with the addition of features like threads, SIMD, and first-class garbage collection, this partnership will only become more seamless and powerful, further blurring the lines between native and web applications and cementing the web browser's position as a truly universal computing platform.

8.3.1 Limitations and Threats to Validity

While the experimental results provide useful insights into the performance characteristics of WebAssembly and JavaScript, several limitations should be acknowledged. First, the experimental evaluation was conducted on a single hardware configuration, which may influence the observed execution times. Performance characteristics may vary across different processors, memory configurations, and operating systems.

Second, although multiple browser engines were considered conceptually in the analysis, the primary experimental results were collected using a Chromium-based browser environment. Differences in JavaScript engine implementations—such as V8, SpiderMonkey, and JavaScriptCore—can influence runtime behavior and optimization strategies, potentially affecting the generalizability of the results.

Third, the benchmarks used in this study are primarily microbenchmarks designed to isolate CPU-intensive computational kernels. While such benchmarks are useful for understanding raw execution characteristics, real-world web applications often involve additional factors such as DOM manipulation, network latency, and asynchronous event handling, which may alter the observed performance dynamics.

Finally, although interoperability between JavaScript and WebAssembly was evaluated conceptually and through targeted measurements, complex applications may incur additional overhead due to frequent boundary crossings and data marshalling between JavaScript heap memory and WebAssembly linear memory. Future studies should explore these aspects in larger application-level workloads.

FUTURE WORK:

While this study highlights the performance benefits of WebAssembly in comparison to JavaScript, several directions remain open for further exploration:

Extended Benchmarks: Future experiments will include workloads in image processing, data compression, and matrix multiplications for machine learning, broadening the evaluation of Wasm performance.

Parallelism and SIMD: With the growing support for WebAssembly threads (via Web Workers and SharedArrayBuffer) and SIMD instructions, investigating their impact on compute-heavy applications will be a natural extension.

Energy Efficiency: Measuring CPU utilization and energy consumption, particularly on mobile and low-power devices, will provide insights into Wasm's suitability for resource-constrained environments.

Security Analysis: Beyond performance, Wasm's

sandbox model and its implications for security, isolation, and side-channel resistance warrant deeper study.

WASI and Beyond the Browser: The WebAssembly System Interface (WASI) is expanding Wasm usage beyond browsers into server-side and edge computing contexts. Comparative studies with JavaScript runtimes like Node.js and Deno will further inform adoption.

Object-Oriented Workloads: Future work will extend the evaluation to object-oriented (OOP) application scenarios, including class-based systems, inheritance hierarchies, and stateful data models. Benchmarks will be designed to measure performance impacts of dynamic object creation, method dispatch, and memory access patterns in JavaScript compared to statically compiled representations in WebAssembly. This will provide a more realistic assessment of performance in enterprise-scale and framework-driven applications.

Declarations

Ethical Approval

Not applicable.

Consent to Participate

Not applicable.

Consent to Publish

Not applicable.

Funding

The authors declare that no funding was received for this study.

Competing Interests

The authors declare that they have no competing interests.

Data Availability

The datasets generated and analysed during the current study are available at: https://github.com/Void-Monarch/full-wasm-js-benchmark-suite

Author Contributions

All authors contributed to the study conception, design, analysis, and manuscript preparation.

All authors read and approved the final manuscript.

References

What is JavaScript? - Learn web development | MDN, , https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/What_is_JavaScript

What Is Wasm? - Tetrate, , https://tetrate.io/learn/what-is-wasm

What is WebAssembly? - MotherDuck, , https://motherduck.com/learn-more/web-assembly/

Use Cases - WebAssembly, , https://webassembly.org/docs/use-cases/

Webassembly vs JavaScript : Performance, Which is Better? - Aalpha information systems, , https://www.aalpha.net/blog/webassembly-vs-javascript-which-is-better/

The Design of WebAssembly. I love the web. It is a modern-day… | by Patrick Ferris | We've moved to freeCodeCamp.org/news | Medium, , https://medium.com/free-code-camp/the-design-of-webassembly-81f1dcabaddd

WebAssembly versus JavaScript: Energy and Runtime Performance - inesc tec, , https://repositorio.inesctec.pt/server/api/core/bitstreams/0870fb76-d463-456b-9e34-5b33bb7c0dd1/content

medium.com, , https://medium.com/@rahul.jindal57/understanding-just-in-time-jit-compilation-in-v8-a-deep-dive-c98b09c6bf0c#:~:text=Just%2DIn%2DTime%20(JIT)%20compilation%20is%20a%20hybrid,to%20achieve%20near%2Dnative%20performance.

Just-In-Time Compilation (JIT) - Glossary - MDN Web Docs, , https://developer.mozilla.org/en-US/docs/Glossary/Just_In_Time_Compilation

WebAssembly vs JavaScript: Which is Better in 2025? - GraffersID, , https://graffersid.com/webassembly-vs-javascript/

Studying WebAssembly and comparison of its performance with JavaScript - ResearchGate, , https://www.researchgate.net/publication/370378392_Studying_WebAssembly_and_comparison_of_its_performance_with_JavaScript

WebAssembly - Wikipedia, , https://en.wikipedia.org/wiki/WebAssembly

WebAssembly - MDN Web Docs, , https://developer.mozilla.org/en-US/docs/WebAssembly

WebAssembly concepts - MDN Web Docs, , https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Concepts

WebAssembly, , https://webassembly.org/

Introduction — WebAssembly 2.0 (Draft 2025-06-24), , https://webassembly.github.io/spec/core/intro/introduction.html

Understanding the WebAssembly Execution Model - Unlocking Enhanced Performance for Developers - MoldStud, , https://moldstud.com/articles/p-understanding-the-webassembly-execution-model-unlocking-enhanced-performance-for-developers

WebAssembly vs. JavaScript: The Complete Guide - Snipcart, , https://snipcart.com/blog/webassembly-vs-javascript

WebAssembly vs JavaScript | The Ultimate Guide - XenonStack, , https://www.xenonstack.com/blog/webassembly-vs-javascript
WebAssembly performance patterns for web apps | Articles - web.dev, , https://web.dev/articles/webassembly-performance-patterns-for-web-apps
WASM Tutorial - Marco Selvatici, , https://marcoselvatici.github.io/WASM_tutorial/
6 Security Risks to Consider with WebAssembly - Jit.io, , https://www.jit.io/blog/6-security-risks-to-consider-with-webassembly
WebAssembly vs Javascript - ianjk, , https://ianjk.com/webassembly-vs-javascript/
[FE-Article: Deciding When to Replace JS Modules with WASM] - Mad Devs, , https://maddevs.io/writeups/deciding-when-to-replace-javascript-modules/
Is WebAssembly faster than JavaScript? Any good benchmarks you can link me to? I... | Hacker News, , https://news.ycombinator.com/item?id=31088055&noRedirect=true
What is the WebAssembly Component Model? - F5 Networks, , https://www.f5.com/company/blog/what-is-the-webassembly-component-model
The WebAssembly Component Model - Fermyon, , https://www.fermyon.com/blog/webassembly-component-model
The WebAssembly Component Model: Introduction, , https://component-model.bytecodealliance.org/
JavaScript - MDN Web Docs - Mozilla, , https://developer.mozilla.org/en-US/docs/Web/JavaScript
JavaScript Design Patterns: A Hands-On Guide with Real-world Examples - Sencha.com, , https://www.sencha.com/blog/comprehensive-guide-javascript-design-pattern/
How JavaScript Executes Code: A Step-by-Step Breakdown of Program Execution, , https://dev.to/gautam_kumar_d3daad738680/how-javascript-executes-code-a-step-by-step-breakdown-of-program-execution-46j
Understanding Just-In-Time (JIT) Compilation in V8: A ... - Medium, , https://medium.com/@rahul.jindal57/understanding-just-in-time-jit-compilation-in-v8-a-deep-dive-c98b09c6bf0c
JavaScript execution model - JavaScript | MDN, , https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Execution_model
WebAssembly/benchmarks: Resources for collaborative benchmarking - GitHub, , https://github.com/WebAssembly/benchmarks
Why is webAssembly function almost 300 time slower than same JS function, , https://stackoverflow.com/questions/48173979/why-is-webassembly-function-almost-300-time-slower-than-same-js-function
The Shape of Code » WebAssembly vs JavaScript performance ..., , https://shape-of-code.com/2023/06/18/webassembly-vs-javascript-performance-2023-edition/
Understanding the Performance of WebAssembly ... - Weihang Wang, , https://weihang-wang.github.io/papers/imc21.pdf
JavaScript or WebAssembly: Which Is More Energy Efficient and Faster? - The New Stack, , https://thenewstack.io/javascript-vs-wasm-which-is-more-energy-efficient-and-faster/
A Comprehensive Guide to Profiling WebAssembly Applications for Optimal Performance, , https://moldstud.com/articles/p-a-comprehensive-guide-to-profiling-webassembly-applications-for-optimal-performance
WebAssembly in Action - Number Analytics, , https://www.numberanalytics.com/blog/webassembly-in-action-real-world-examples
WebAssembly in Action: Real-World Applications - Number Analytics, , https://www.numberanalytics.com/blog/webassembly-in-action-real-world-applications
A preliminary study of WebAssembly: the key to improving web application performance, , https://tianyaschool.medium.com/a-preliminary-study-of-webassembly-the-key-to-improving-web-application-performance-d8da73da66e1
Nature: No installation required: how WebAssembly is changing scientific computing : r/datascience - Reddit, , https://www.reddit.com/r/datascience/comments/1btl6ei/nature_no_installation_required_how_webassembly/
WASM Image Studio: High-Performance Image Filters with Leptos ..., , https://autognosi.medium.com/wasm-image-studio-high-performance-image-filters-with-leptos-and-daisyui-aee8bde1565e
WebAssembly Performance: How Fast Is WASM? - Clover Dynamics, , https://www.cloverdynamics.com/blogs/web-assembly-performance-how-fast-is-wasm
Photon: A WebAssembly Image Processing Library, , https://silvia-odwyer.github.io/photon/
Video Frame Processing on the Web – WebAssembly, WebGPU, WebGL, WebCodecs, WebNN, and WebTransport - webrtcHacks, , https://webrtchacks.com/video-frame-processing-on-the-web-webassembly-webgpu-webgl-webcodecs-webnn-and-webtransport/
How to Use WebAssembly for Audio and Video Processing - PixelFreeStudio Blog, , https://blog.pixelfreestudio.com/how-to-use-webassembly-for-audio-and-video-processing/
Why WebAssembly Is a Game Changer for Browser-Based Games - js13kGames, , https://js13kgames.com/p/webassembly-game-

changer.html
WebAssembly Is Fast: A Real-World Benchmark of WebAssembly vs. ES6 | by Aaron Turner | Medium, , https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193
(PDF) Efficient Implementation of a Crypto Library Using Web Assembly - ResearchGate, , https://www.researchgate.net/publication/346634579_Efficient_Implementation_of_a_Crypto_Library_Using_Web_Assembly
Choosing a Cryptography Library for JavaScript: Noble vs. Libsodium.js | Nik Graf &mdash, , https://www.nikgraf.com/blog/choosing-a-cryptography-library-in-javascript-noble-vs-libsodium-js
CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem - Computer Science, , https://cseweb.ucsd.edu/~dstefan/pubs/watt:2019:ct-wasm.pdf
WebCryptoAPI vs WebAssembly Encryption module - Stack Overflow, , https://stackoverflow.com/questions/48891593/webcryptoapi-vs-webassembly-encryption-module
WebAssembly vs JavaScript: A Comparison - SitePoint, , https://www.sitepoint.com/webassembly-javascript-comparison/
WebAssembly vs. JavaScript: Exploring the Performance Leap and Technical Advantages | by Vincenzo Boellis | Medium, , https://medium.com/@vincenzoboellis/webassembly-vs-javascript-exploring-the-performance-leap-and-technical-advantages-66a7971eebad
Finding the Best Fit Between Dynamic Typing vs. Static Typing - Unosquare, , https://www.unosquare.com/blog/finding-the-best-fit-between-dynamic-typing-vs-static-typing/
JS -> wasm vs JS -> native function call overhead? - SpiderMonkey - Mozilla Discourse, , https://discourse.mozilla.org/t/js-wasm-vs-js-native-function-call-overhead/77120
Function calls from WASM to JS don't seem to have improved so much. I don't know... | Hacker News, , https://news.ycombinator.com/item?id=18168774
“Near-Native Performance”: Wasm is often described as having “near-native perf... | Hacker News, , https://news.ycombinator.com/item?id=30156437
JavaScript Interop with WebAssembly | by Kevin Hoffman | Medium, , https://kevinhoffman.medium.com/javascript-interop-with-webassembly-2c69a3db19e9
Most performant way to pass data JS/WASM context · Issue #1231 · WebAssembly/design, , https://github.com/WebAssembly/design/issues/1231
What Are People Building With WebAssembly? - DEV Community, , https://dev.to/zenstack/what-are-people-building-with-webassembly-2eh4
A Gentle Introduction to WebAssembly in Rust (2025 Edition) | by Mark Tolmacs - Medium, , https://medium.com/@mtolmacs/a-gentle-introduction-to-webassembly-in-rust-2025-edition-c1b676515c2d
Building, Shipping and Debugging a C++ WebAssembly App - Will Usher, , https://www.willusher.io/blog/build-ship-debug-wasm/
WebAssembly for Frontend Developers: A Complete Guide, , https://blog.pixelfreestudio.com/webassembly-for-frontend-developers-a-complete-guide/
JavaScript interoperability - The Trunk Guide, , https://trunkrs.dev/guide/advanced/javascript_interop.html
Understanding JavaScript Design Patterns In Depth | BrowserStack, , https://www.browserstack.com/guide/javascript-design-patterns
WebAssembly vs. JavaScript: When to Use Each - PixelFreeStudio Blog, , https://blog.pixelfreestudio.com/webassembly-vs-javascript-when-to-use-each/
Debugging - Rust and WebAssembly, , https://rustwasm.github.io/book/reference/debugging.html
Debugging WebAssembly With Chrome DevTools | WASM Tutorial | KodeKloud - YouTube, , https://www.youtube.com/watch?v=yNhYNbV4KPo
Debug C/C++ WebAssembly | Chrome DevTools, , https://developer.chrome.com/docs/devtools/wasm
Best Practices for Debugging WebAssembly Applications - PixelFreeStudio Blog, , https://blog.pixelfreestudio.com/best-practices-for-debugging-webassembly-applications/
The Future of WebAssembly A Comparison with Current JavaScript Frameworks and Its Potential Role - SimplifyC++, , https://www.simplifycpp.org/?id=a0375
Beyond JavaScript: A Developer's Practical Introduction to WebAssembly (WASM) - Medium, , https://medium.com/@Christopher_Tseng/beyond-javascript-a-developers-practical-introduction-to-webassembly-wasm-116a15eb8681
WebAssembly vs. JavaScript | Abdulkader Safi, , https://abdulkadersafi.com/blog/webassembly-vs-javascript-when-to-use-each-for-maximum-performance
WASI and the WebAssembly Component Model: Current Status - eunomia, , https://eunomia.dev/blog/2025/02/16/wasi-and-the-webassembly-component-model-current-status/
The State of WebAssembly – 2023 and 2024 - Uno Platform, , https://platform.uno/blog/state-of-webassembly-2023-2024/
WebAssembly SIMD | Internet Computer, , https://internetcomputer.org/docs/building-apps/network-features/simd
The State of WebAssembly – 2024 and 2025 - Uno Platform, , https://platform.uno/blog/state-of-

webassembly-2024-2025/
WasmGC and Wasm tail call optimizations are now Baseline Newly available - web.dev, , https://web.dev/blog/wasmgc-wasm-tail-call-optimizations-baseline
A new way to bring garbage collected programming languages efficiently to WebAssembly, , https://v8.dev/blog/wasm-gc-porting
Wasm Proposals - Wasmtime, , https://docs.wasmtime.dev/stability-wasm-proposals.html
gc/proposals/gc/Overview.md at main · WebAssembly/gc - GitHub, , https://github.com/WebAssembly/gc/blob/main/proposals/gc/Overview.md
Issues and Their Causes in WebAssembly Applications: An Empirical Study - arXiv, , https://arxiv.org/html/2311.00646v2