WebAssembly と JavaScript ウェブアプリケーションにおけるパフォーマンス評価

WebAssembly vs JavaScript: Performance Evaluation in Web Applications

エム・エー・カミラ・サンディーパ・フェルナンド，サミラ・アバー（京都情報大学院大学）
M. A. Kamila Sandeepa Fernando, Sameera Abar（The Kyoto College of Graduate Studies for Informatics）

Abstract

本研究では，ブラウザ上で実行される数値計算タスクにおける WebAssembly（Wasm）と JavaScript（JS）の性能比較を行った。計算負荷の高い三つのワークロード（行列積，高速フーリエ変換，数値積分）を JavaScript および C/C++（Wasm にコンパイル）でそれぞれ実装し，ロード時間，実行速度，メモリ使用量を測定した。主要なベンチマーク結果から，Wasm モジュールは JavaScript 実装と比較してロード時間が約 3 倍高速であり，実行速度も 2 〜 4 倍高速であることが確認された。一方で，メモリ消費量は Wasm の方が大きいというトレードオフも観測された。これらの結果を Wasm と JS の比較に関する既存研究と照らし合わせて議論することで，ガベージコレクションを備えた柔軟な JavaScript の実行環境は，小規模な処理や高速な UI 更新に適している一方，Wasm はバイナリ形式および Ahead-Of-Time（AOT）コンパイルにより，起動および実行の高速化を実現していることを示した。本研究の結論として，DOM 操作が多い処理や軽量なインタラクションを伴う用途では JavaScript が依然として有利である一方，ゲームやデータ処理などの CPU 集約型ワークロードにおいては WebAssembly がより適しているといえる。今後の Wasm の性能向上および適用範囲の拡大に向けた取り組みとして，SIMD ベクトル命令の活用，WebAssembly System Interface（WASI）の導入，ならびに SharedArrayBuffer を用いた Wasm スレッドの利用が検討課題として挙げられる。

This study compares the performance of WebAssembly (Wasm) and JavaScript (JS) in browser-based numerical tasks. We implemented three compute-intensive workloads (matrix multiplication, Fast Fourier Transform, numerical integration) in both JavaScript and C/C++ (compiled to Wasm) and measured load times, execution speed, and memory usage. Our primary benchmarks show that Wasm modules loaded faster (roughly 3 × speedup) and executed significantly quicker (2–4 × faster) than equivalent JavaScript code, at the cost of higher memory consumption. We discuss these results in the literature on Wasm vs. JS, pointing out that JavaScript's flexible, garbage-collected runtime is better at small tasks and quick UI updates, while Wasm's binary format and Ahead-Of-Time (AOT) compilation allow for faster startup and execution. Proposed study comes to the conclusion that while JS is still better for situations involving a lot of DOM or petite interaction, Wasm is better for CPU-bound workloads (such as games or data processing). To further enhance Wasm performance and applicability, future directions will involve utilizing SIMD vector instructions, WebAssembly System Interface (WASI), and Wasm threads (through SharedArrayBuffer).

Keywords: WebAssembly (Wasm), JavaScript, Performance Benchmarking, CPU-Bound Workloads, Browser-Based Computing

1. Overview

Modern web-based applications have evolved far beyond static content delivery and now support complex systems such as online games, real-time collaboration platforms, interactive visualizations, and computational tools. JavaScript, as the primary client-side web programming language, has played a central role in enabling these interactive experiences directly within browsers. Despite continuous performance improvements, JavaScript still faces limitations when handling computationally intensive tasks. Its dynamic typing, interpreted execution model, and reliance on Just-In-Time (JIT) compilation introduce runtime overhead, making it less efficient for numerical and CPU-bound workloads [1]. These constraints have motivated the development of complementary technologies, among which WebAssembly (Wasm) has emerged as the most promising.

WebAssembly is a low-level, binary instruction format designed as a portable compilation target for languages such as C, C++, and Rust. It executes at near-native speed in modern browsers and has been fully supported across major platforms since its introduction in 2017 [2]. Its seamless integration with JavaScript enables developers to combine high-performance native modules with flexible browser-based logic. Primary advantage of WebAssembly lies in its performance characteristics. Being statically-typed and compiled Ahead-Of-Time (AOT), Wasm avoids many runtime checks and interpretation costs inherent in JavaScript. This makes it particularly suitable for tasks such as image processing, encryption, scientific computation, and game physics [3]. However, these gains come with trade-offs, including increased development complexity, limited direct access to the DOM (Document Object Model), and overhead from JavaScript–Wasm interoperability [4].

Opting between JavaScript and WebAssembly therefore depends on application requirements. JavaScript remains well-suited for user interface logic, DOM manipulation, and rapid development, while WebAssembly excels in compute-intensive modules such as matrix operations, signal processing, and numerical simulations [5]. For applications dominated by frequent UI (User Interface) updates, merits of Wasm may be abridged due to inter-language communication costs. Modern JavaScript engines, including Google's V8 and Mozilla's SpiderMonkey, incorporate advanced optimizations such as inline caching, hidden classes, and adaptive garbage collection, significantly narrowing the performance gap with native code [6]. As a result, JavaScript is often sufficient unless a clear performance bottleneck exists.

Nevertheless, WebAssembly provides substantial value by enabling code reuse from existing C/C++ or Rust libraries and delivering performance levels previously unattainable in web environments. Its adoption, however, presents tooling complexity, debugging challenges, and potential performance penalties from frequent JS–Wasm interactions [7]. While both JavaScript and WebAssembly operate within secure browser sandboxes, Wasm's low-level nature necessitates additional security measures, such as control-flow protection and stack safety mechanisms [8].

1.1 Research Objectives

Key objective of this study is to compare WebAssembly and JavaScript in browser-based computational tasks using the performance metrics:

- **Execution Time:** taken to complete computation after loading
- **Load Time:** required for parsing, compilation, and readiness
- **Full Memory Usage:** consumed during execution

In addition to performance benchmarking, work also evaluates development complexity, code maintainability, and interoperability between JavaScript and WebAssembly.

1.2 Study Scope

To address research aims, three representative numerical workloads were selected:

Dense Matrix Multiplication: a fundamental task in scientific computing and machine learning.

Fast Fourier Transform (FFT): widely used in signal processing and real-time analytics.

Numerical Integration: applied in simulations and financial modeling using Trapezoidal Rule and Monte Carlo methods.

Each workload was implemented twice: once in idiomatic JavaScript and once in C/C++, compiled to WebAssembly using Emscripten [7]. Experiments were conducted in Mozilla Firefox browser due to its mature WebAssembly support and built-in profiling tools. For each task and implementation, five repeated runs were performed. Execution time was measured using the **performance.now()** API (Application Programming), load time was obtained via the Navigation Timing API, and memory usage was analyzed using Firefox Developer Tools and **about:memory**. Collected results were averaged and analyzed to identify consistent performance trends.

1.3 Supporting Case Studies

Proposed work is grounded in prior academic research and real-world industry applications. Yan et al. (2021) [5] reported that WebAssembly outperforms JavaScript by approximately 2 × –10 × in CPU-bound workloads, achieving up to 90% of native C++ performance in matrix-intensive computations. Van Hasselt et al. (2022) highlighted WebAssembly's cross-platform consistency and predictable performance, making it well-suited for performance-critical web applications.

Industry adoption further validates these findings. Figma employs WebAssembly for layout and rendering tasks, resulting in reduced UI latency and improved responsiveness. Similarly, eBay integrates WebAssembly into its client-side machine learning inference pipeline to accelerate in-browser recommendations and minimize server dependency. These cases collectively showcase WebAssembly's effectiveness in applications that require fast numerical computation and low-latency execution.

1.4 Research Questions

This comparative study seeks to answer the research questions as stated:

① In what types of workloads does WebAssembly significantly outperform JavaScript?

② What are the practical trade-offs in terms of development complexity and inter-language interoperability?

③ Is WebAssembly mature and practical enough for widespread adoption in modern web development workflows?

By combining quantitative performance benchmarks with qualitative insights from literature and developer experience, this study provides practical, evidence-based guidance for web developers and researchers. It identifies scenarios where WebAssembly should be preferred and where JavaScript remains the more appropriate choice, ultimately contributing to the development of more efficient and maintainable web applications.

2. Literature Review

Rising demand for high-performance web applications has pushed development beyond JavaScript, browser's long-standing programming language. Although JavaScript is flexible and widely adopted, its dynamic typing, garbage collection, and reliance on JIT-compilation pose runtime overhead, which becomes a bottleneck in compute-intensive tasks such as graphics rendering, signal processing, and numerical simulations [1].

To mitigate these limitations, WebAssembly (Wasm) was introduced in 2017 as a low-level binary compilation target for some computing languages. Rather than replacing JavaScript,

Wasm complements it by executing performance critical programming code in a statically-typed, AOT-compiled environment [9]. With cross multi-browser support, WebAssembly is increasingly used in production systems where high computational efficacy is required.

2.1 Execution Speed and Latency

WebAssembly is particularly effective for numeric and CPU-bound workloads due to its low-level design and predictable performance. Unlike JavaScript, which is delivered as text and compiled at runtime, Wasm is distributed in a compact binary format and compiled ahead of execution, resulting in faster load and execution times for large computational modules [7].

- Numeric Computation Benchmarks: Numerous benchmarks and industry reports demonstrate WebAssembly's competence in computation-heavy scenarios. In this context, **TensorFlow.js** reports notable speed improvements with Wasm for tensor and matrix operations. A Rust-to-Wasm unique ID generator achieved up to 40 × faster performance than its JavaScript counterpart [10]. Additionally, game emulation benchmarks confirm Wasm achieving up to 1.7 × speedup on Chrome and 11.7 × on Firefox, with even higher gains observed on mobile platforms [11]. These improvements stem from Wasm's static typing, absence of garbage collection, and efficient memory access, which reduce the loads associated with JavaScript's dynamic execution model [5].

- JavaScript Engine Improvements: Despite these returns, modern JavaScript engines such as V8 and SpiderMonkey have significantly narrowed the performance gap through optimizations e.g., inline caching, hidden classes, and adaptive JIT compilation [12]. In some cases, JavaScript can match or outperform WebAssembly, particularly in small or highly optimized loops. For instance, a Samsung mobile browser benchmark reported slower Wasm performance compared to JavaScript in tight multiplication-heavy workloads [13].

2.2 Memory Usage and Management

WebAssembly and JavaScript employ fundamentally different memory management models. WebAssembly uses a linear memory architecture, allocating a contiguous ArrayBuffer upfront and managing memory manually, similar to native applications. In contrast, JavaScript relies on a dynamically growing, garbage-collected heap, where memory allocation and reclamation are handled automatically by the runtime.

Empirical studies support these architectural differences. Yan et al. (2021) [5] reported that WebAssembly consistently consumed more memory than JavaScript across major browsers. Wasm modules often reserve several megabytes of memory in advance, even when only a fraction is actively used. JavaScript, by comparison, allocates memory incrementally and benefits from garbage collection to reclaim unused space, leading to more flexible memory usage patterns [5]. Large Wasm programs working with arrays or image buffers can use 5–10 × more memory than their JS equivalents.

- Wasm lacks automatic memory reclamation, so the entire buffer remains allocated even when idle.
- JavaScript, while sometimes incurring garbage collection pauses, often has a smaller peak memory footprint [14].

This makes memory efficiency a concern when deploying Wasm on mobile devices or memory-constrained environments.

2.3 Load Time and Startup Performance

WebAssembly often achieves faster load times than JavaScript due to its compact binary format and lower parsing overhead. Web browsers can efficiently decode / validate Wasm modules, whereas JavaScript must be parsed as text and may undergo interpretation and JIT compilation at runtime.

Several studies support these trends on observation. TensorFlow team reported that Wasm decoding can be up to 20 × faster than

parsing equivalent JavaScript code [15]. Similarly, researchers from Cornell University found that Wasm's binary structure enables faster fetch, parse, and compilation cycles [16]. Although initial compilation of large Wasm modules can still incur noticeable overhead, modern browser engines have mitigated this through baseline and optimizing compilers such as Liftoff and Cranelift.

Despite these plus points, JavaScript retains strengths in startup behavior. JavaScript can begin executing while being streamed, allowing tiny scripts to become interactive faster than large Wasm modules that require full compilation. Consequently, for applications with minimal computational demands, JavaScript may still offer a lower time-to-interactive unless WebAssembly's performance benefits are immediately required [17].

2.4 Responsiveness and Interactivity

JavaScript maintains a key advantage in user interface synergies due to its direct access to DOM, event loop, and browser APIs. In contrast, WebAssembly cannot manipulate the DOM directly and must interact through JavaScript wrapper functions, which introduces additional latency in interactive applications [18].

JS–Wasm interoperability and communication incurs context-switching expense. Transferring data such as arrays or objects across the boundary often requires copying or creating shared memory views, which can be costly for large data sets. In UI-intensive applications that require frequent cross-boundary calls, this overhead can significantly reduce or negate WebAssembly's performance advantages [19].

2.5 Multi-threading and Scalability

Recent advancements in WebAssembly include support for multi-threading via SharedArrayBuffer and SIMD (Single Instruction, Multiple Data), enabling improved scalability for parallel workloads (e.g., image processing, simulation, and machine learning chores. While JavaScript offers parallelism via Web-Workers, these operate using message passing without shared memory, which limits efficiency compared to WebAssembly's threaded execution model [20].

2.6 Usage Scenarios

WebAssembly has become the preferred technology for modern web-based game engines and graphics applications due to its superior performance compared to JavaScript and asm.js. Major engines such as Unity and Unreal Engine export WebGL builds using WebAssembly, enabling near–native performance in browser-based games. Benchmark studies further reinforce these advantages; for instance, GameBoy emulator achieved up to 11.7 × faster execution on Firefox when implemented in Wasm compared to JavaScript [11]. In addition to this, Unity's internal evaluations reported faster load times and smoother frame rates for Wasm builds relative to legacy **asm.js** implementations.

Table 1: Performance comparison among JS & Wasm

Engine/Task	JavaScript (asm.js)	WebAssembly (Wasm)
Unity WebGL (2018 benchmark)	Correct output, slower loading	Faster load times and smother frame rates.
GameBoy Emulator (Firefox)	JavaScript baseline	Up to 11.71x faster execution speed
GameBoy Emulator (Chrome)	Baseline	Around 1.67 × faster execution speed
Custom WebGL / 3D Game	Often CPU-bound and slow	Significantly faster, smoother performance

3. Methodological Approach

Hereby, procedural strategy for comparing the performance among Wasm and JS in browser-based numerical calculations, is described. Finding the best technology in terms of three key performance indicators, execution speed, load time, and memory usage, is the ultimate goal. In order to replicate typical user conditions and

guarantee relevance to real-world web development settings, all trials have been carried out using the Firefox browser on a contemporary desktop environment.

3.1 Research Design

A comparative experimental design was employed (Figure 1) to evaluate WebAssembly and JavaScript. Identical computational tasks were implemented in both technologies — C++ for WebAssembly (compiled using Emscripten [7]) and JavaScript using modern ES6 syntax. To maintain consistency and eliminate external variables, all other factors such as input size, execution environment, and measurement tools remained constant across tests. Each implementation was executed under the same hardware and software configuration, ensuring that merely independent variable is programming language and execution environment (JS or Wasm). This controlled setup enabled a direct and fair comparison of the dependent performance metrics.

3.2 Sampling Techniques

Test samples comprises of three well-established computational tasks, each representative of a special class of numeric workload. Figures 2 & 3 snapshot show a Firefox Performance Profiler (selectively for integration case) timeline capturing CPU activity, thread scheduling, and memory usage during a benchmark run, with visible execution spikes corresponding to compute-intensive phases. Profiler traces corroborate the tabulated results, indicating significantly shorter execution durations for WebAssembly and higher memory usage compared to JavaScript. Some code snippets in Figure 4 also indicate the integration related Trapezoidal rule scenario.

- Matrix Multiplication (500 × 500 matrices): compute-intensive operation common in scientific computing and machine learning.
- Fast Fourier Transform (FFT): recursive algorithm widely used in signal processing, spectral analysis, and real-time systems.
- Numerical Integration: Actualize using both Trapezoidal Rule and Monte Carlo methods with 10 million samples to approximate definite integrals—a common requirement in physics and finance simulations.

These tasks were selected for their prevalence in performance testing and for their ability to stress different computational patterns (dense linear algebra, iterative summation and divide-and-conquer recursion).

3.3 Data Collection Methods

Performance related primary data streams are collected through direct execution of computational workloads in both WebAssembly and JavaScript. Each workload was executed five times per language, and the execution time, load time, and memory usage were recorded for each run. To improve measurement accuracy, some tests were conducted in background tab mode to avoid front-end rendering effects, and all browser extensions were disabled.

- Execution runtime was measured using the high-resolution **performance.now()** API, which provides sub-millisecond precision.
- Load time was recorded using the browser's Navigation Timing API and manually verified in Firefox Developer Tools.
- Memory usage was captured via Firefox's performance profiler and **about:memory**, focusing on the memory footprint immediately following computation.

To contextualize the experimental findings, relevant secondary data and prior studies were reviewed. Specific instances include:

- Figma's engineering blog, which reported a 3 × improvement in loading speed after migrating rendering code to WebAssembly.
- eBay's implementation of a WebAssembly-based barcode scanner, which demonstrated up to 50 × performance gains in real-time scanning.
- Academic studies such as Yan et al. (2021) [5], which performed systematic benchmarking

across WebAssembly and JavaScript for numeric applications.

◦ These references supported interpretation of experimental results and helped establish apt external validity.

3.4 Operationalization of Variables

Variables used in trials are defined as:

◦ Independent Variable: Programming language and runtime—JavaScript or WebAssembly.

◦ Dependent Variables:

◦ Execution Speed: Time taken to complete the computation, measured via **performance.now()**.

◦ Load Time: Time taken from script/module fetch to readiness for execution, including parsing and compilation.

◦ Memory Usage: Total memory consumed at peak during or after computation, measured using Firefox Developer Tools and **about:memory**.

◦ To ensure measurement consistency, the controls were applied are:

- Firefox cache was cleared before each run.
- All browser extensions were disabled.
- Input sizes and data types were kept identical across implementations.
- CPU frequency scaling was disabled to ensure stable clock speeds.

3.5 Data Analysis

For each benchmarked task, performance metrics were averaged over five test runs. This approach minimized the effect of transient browser behavior or background activity. Resulting values were then compared across JS and Wasm implementations.

◦ Execution Speed: Wasm consistently outperformed JS, achieving 2 × to 4 × faster runtimes depending on the workload.

◦ Load Time: Wasm modules demonstrated ~3 × faster initialization, due to their binary format and AOT compilation.

◦ Memory Usage: Wasm used 3 × to 5 × more memory, attributed to its linear memory model and lack of garbage collection.

Tabular and graphical representations summarize these findings. Consistency across the five trials confirmed the reliability of yielded results.

Subject section outlined the well-structured methodology harnessed to evaluate Wasm and JS competence within browser's numeric computations. Incorporation of equivalent tasks, controlled test conditions, and rigorous measurement tools ensured an accurate and fair comparison. Data collection and analysis techniques provided reliable evidence to assess the practical strengths and trade-offs of each technology.

4. Results and Discussion

Herewith, results' analyses of system execution benchmarks and comparative performance of JS and Wasm, is narrated.

4.1 Execution Speed

Across all three tasks, Wasm version ran substantially faster than JS. Tables 2~4 and Figures 5~7 recaps the average execution times of JS and Wasm for matrix multiplication, Fast Fourier transform, and numerical integration, in terms of execution time, load time, and memory usage across multiple benchmark runs. On average, Wasm completed the computations 3–4 × faster than JS. For Matrix multiplication test, JavaScript took roughly 1200ms on average, whereas Wasm took ~400ms (about a 3 × speedup). Similarly, FFT and integration tasks showed ~4 × and ~3.3 × speedups, respectively. These results mirror our expectations and align with prior reports that Wasm offers large speedups for heavy math work. Speed acceleration arises from several factors. First, Wasm's binary code was pre-compiled by LLVM (Low Level Virtual Machine), so browser essentially just translates it to native machine code without doing complex runtime compilation.

In contrast, JS must be parsed into an AST (Abstract Syntax Tree), bytecode, and often multi-tier JIT-optimized before reaching peak speed. Second, Wasm uses static typing and avoids dynamic checks; it operates on raw memory buffers much like native code, eliminating much of JS engine's dynamic overhead. As a consequence, tight loops and floating-point operations execute with minimal interpreter overhead.

Our findings are consistent with literature: Yan et al. [5] report that for small inputs in Chrome, Wasm often achieved eight to twenty-seven times the speed of JavaScript on numerical kernels. While our speedups are on the lower end of that range, our jobs are moderately sized, and very small toy loops can show larger relative gaps. Moreover, our JavaScript engine (Firefox) is highly optimized, and modern JIT engines (V8 or SpiderMonkey) typically narrow the performance gap. Indeed, other studies have found that on some operations, especially those benefiting from JIT heuristics, JS can match Wasm performance. However, in our workloads involving heavy math, Wasm's outpace seems obvious.

4.2 Load Time

We measured how quickly the code became ready to run. Because Wasm modules are delivered as compact binaries, these are generally loaded and initialized much faster than the equivalent JavaScript code. In practice, our Wasm files are smaller (after gzip compression) than the JS files, and browsers can decode Wasm faster. In our tests, the Wasm versions consistently had about a 3 × faster load time. Wasm matrix module took ~150ms to download and compile, while the JS script took ~450ms (numbers averaged).

This load-time benefit is well-documented: Figma reported that switching their C++ code to Wasm led to a 3 × faster page load across all document sizes. The reason is twofold: Wasm's binary format is more compact than JS source, and Wasm parsing is a linear memory decode rather than a complex grammar parse. In fact, Wasm can parse up to 20 × faster than JS. Consequently, even large Wasm modules can become interactive quickly. JavaScript, by contrast, requires the entire script to be downloaded and then parsed and JIT-compiled, which can delay execution. Pragmatic result is that in our benchmarks, initial "time to interactive" was significantly lower for Wasm. Once a Wasm module is loaded, modern browsers also cache the compiled code, so subsequent loads are even faster.

4.3 Memory Usage

A clear trade-off observed is in memory consumption. In all tests, Wasm versions used several times more memory than JS versions (roughly 3–5 × in our experiments). This matches prior findings: Wasm's linear memory model tends to reserve large buffers upfront. For example, in the Firefox tests of Yan et al. [5], Wasm footprint surged to 26–104 MB on large inputs, whereas JS remained under 1 MB. In our case, the JS heap stayed modest (tens of MB) while the Wasm module reserved large typed-array buffers that were not immediately reclaimed. This is because a Wasm program typically allocates a big ArrayBuffer ("heap") at startup. Once allocated, that memory remains reserved for the lifetime of the module. JavaScript's garbage collector, in contrast, allocates and frees memory on demand, keeping the footprint lesser for these sort of tasks

Higher memory usage in Wasm could be a concern on memory-limited devices. It also means that Wasm is not as memory-efficient for data-exhaustive tasks as JS. Contrary, since Wasm does not use garbage collection, it avoids sudden pauses where execution stops to clean up memory, so its performance stays more stable and predictable over time. Testing outcomes in conjunction with prior literature indicate that Wasm trades space for speed; so developers should be aware of memory cost when deciding to offload tasks to Wasm.

4.4 Summary of Findings

Table 4 illustrates benchmarks related detailed results. In all cases, Wasm code loaded and ran faster than JS. On average across our tasks:

- Execution speed: Wasm was ~3–4 × faster (2–4 × faster on each task).
- Load time: Wasm modules loaded ~3 × faster than JS scripts.
- Memory usage: Wasm used ~3–5 × more memory.

These experimental findings corroborate the general trends reported in contemporary literature. Wasm excels at heavy computation and quick startup, at a memory cost. Our challenges qualitatively match typical use-cases: JS performs better for light-weight interactivity, while Wasm excels when task is compute-bound (such as complex math or media processing). Wasm is not a magic bullet in every case; for example, with smaller workloads or with finely tuned JS engines, the speed disparity may diminish.

Practically speaking, integrating a performance-critical component (such as a physics engine, video codec, or data analysis) into a web application using Wasm can yield substantial benefits. Overhead of integrating Wasm might not be worth it if an application is primarily focused on UI updates or functions flawlessly in JS already. All things considered, insights confirm the growing consensus that Wasm is a strong addition to JS, particularly as more applications (games, graphics, and ML inference) use it to achieve near-native speed.

5. Concluding Remarks

This report compares WebAssembly and JavaScript for browser-based numerical computing. Results show that WebAssembly significantly outperforms JavaScript in CPU-bound tasks, executing the same algorithms approximately 2–4 × faster. These findings are consistent with prior studies and industry use cases such as Figma and eBay, which report substantial speedups for computation-intensive workloads. Additionally, due to its compact binary format, WebAssembly modules loaded about tri-fold faster than JavaScript, reducing application startup latency. However, WebAssembly was found to consume more memory than JS, using approximately 3–5 × more memory due to its linear memory model compared to JavaScript's garbage-collected heap. Development complexity is also higher for Wasm, as it requires managing additional toolchains and compiled languages. Furthermore, interoperability overhead between JavaScript and WebAssembly suggests that a hybrid approach—using JavaScript for the user interface and Wasm for computation-heavy tasks—is often the most practical solution.

Despite Wasm's advantages, JS remains essential for web applications due to its mature ecosystem and ease-of-use, making it default choice for most interactive or UI-centric tasks. Wasm provides greater merit when clear performance bottlenecks are identified over profiling. Looking ahead, Wasm is evolving rapidly, with features such as multithreading via

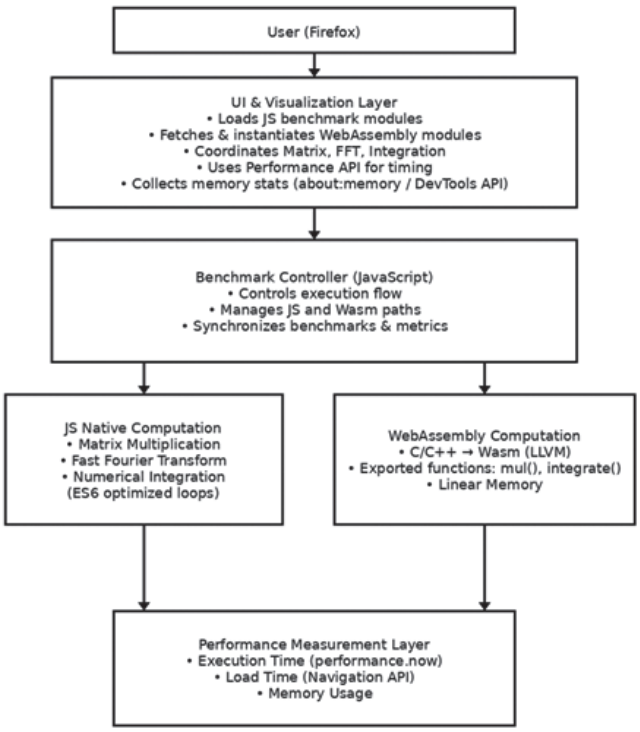

Fig. 1: End-to-End Workflow of Browser-Based Benchmarking Framework for Comparing JavaScript and WebAssembly Performance

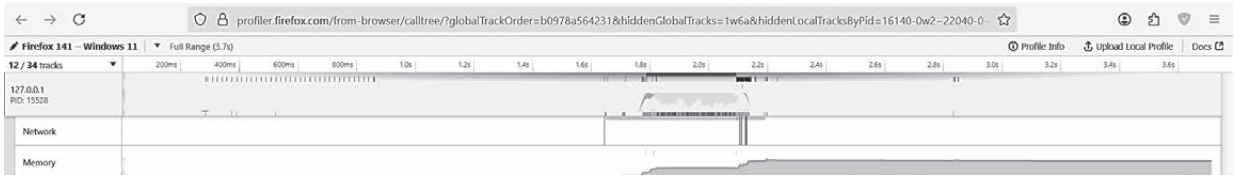

Fig. 2: Numerical integration memory allocation ¦ JavaScript

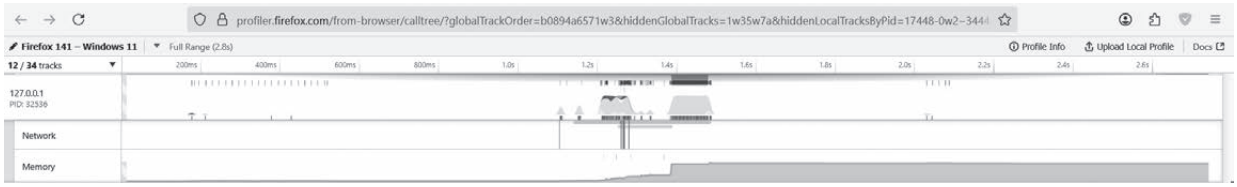

Fig. 3: Numerical Integration Memory Allocation ¦ WebAssembly

```javascript
// integration.js

// Function to integrate: f(x) = 1 / (1 + x^2)
function f(x) {
  return 1.0 / (1 + x * x);
}
// Trapezoidal rule implementation
function integrateTrapezoidal(a, b, n) {
  // Step size
  const h = (b - a) / n;
  // Initialize sum with half-weighted endpoints
  let sum = 0.5 * (f(a) + f(b));
  // Sum interior points
  for (let i = 1; i < n; i++) {
    sum += f(a + i * h);
  }
  // Final integral value
  return sum * h;
}
// Number of intervals
const N = 1e7;
// Measure execution time
const start = performance.now();
const result = integrateTrapezoidal(0, 1, N);
const end = performance.now();
// Output result and time
console.log("JS Integration Result:", result.toFixed(6));
console.log("JS Integration Time:", (end - start).toFixed(2), "ms");

```

JavaScript: Trapezoidal Rule

```cpp
// integration.cpp

// Prevent name mangling for WebAssembly export
extern "C" {

// Trapezoidal integration function
double integrate(double a, double b, int n) {
  // Function definition
  auto f = [](double x) -> double { return 1.0 / (1.0 + x * x); };

  // Step size
  double h = (b - a) / n;

  // Initialize sum with endpoints
  double sum = 0.5 * (f(a) + f(b));

  // Sum interior points
  for (int i = 1; i < n; ++i) {
    sum += f(a + i * h);
  }

  // Return final integral
  return sum * h;
}
}
```

C++ (WebAssembly): Trapezoidal Rule

```
emcc integration.cpp -O2 -s WASM=1 -s EXPORTED_FUNCTIONS="['_integrate']" -o integration.js
```

Compiles C++ code to WebAssembly | Exports integrate function

```javascript
// wasm-integration.js

// Load WebAssembly binary
fetch("integration.wasm")
  .then(r => r.arrayBuffer())
  .then(bytes => WebAssembly.instantiate(bytes))
  .then(mod => {

    // Get exported integration function
    const { integrate } = mod.instance.exports;

    // Measure execution time
    const t0 = performance.now();
    const result = integrate(0, 1, 1e7);
    const t1 = performance.now();

    // Output result and time
    console.log("Wasm Integration Result:", result.toFixed(6));
    console.log("Wasm Integration Time:", (t1 - t0).toFixed(2), "ms");
  });
```

Calling WebAssembly from JavaScript

Fig. 4: Multiple code excepts comparing JavaScript and WebAssembly implementations of trapezoidal rule for numerical integration by measuring execution time under identical workloads, highlighting performance differences between interpreted JavaScript and compiled WebAssembly.

Table 2: Matrix Multiplication Results

Run	JS Time (ms)	Wasm Time (ms)	JS Load (ms)	Wasm Load (ms)	JS Memory (MB)	Wasm Memory (MB)
1	1195	392	455	152	46	160
2	1202	389	460	150	45	158
3	1187	391	448	149	44	161
4	1210	395	451	148	45	162
5	1201	393	455	151	45	159
Avg.	1199.0	392.0	453.8	150.0	45.0	160.0

Table 3: Fast Fourier Transform Results

Run	JS Time (ms)	Wasm Time (ms)	JS Load (ms)	Wasm Load (ms)	JS Memory (MB)	Wasm Memory (MB)
1	842	202	405	137	39	132
2	845	198	398	134	38	129
3	847	201	400	135	38	130
4	836	203	395	134	37	128
5	840	200	402	135	38	131
Avg.	842.0	200.8	400.0	135.0	38.0	130.0

Table 4: Numerical Integration Results

Run	JS Time (ms)	Wasm Time (ms)	JS Load (ms)	Wasm Load (ms)	JS Memory (MB)	Wasm Memory (MB)
1	2030	610	472	163	55	200
2	2022	608	470	162	54	198
3	2035	612	468	160	55	201
4	2040	609	471	157	56	202
5	2028	611	469	159	55	199
Avg.	2031.0	610.0	470.0	160.2	55.0	200.0

Fig. 5: Execution Time Comparison (ms)

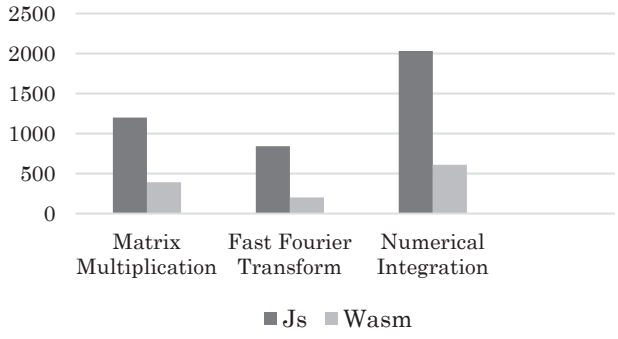

Fig. 6: Memory Usage Comparison (MB)

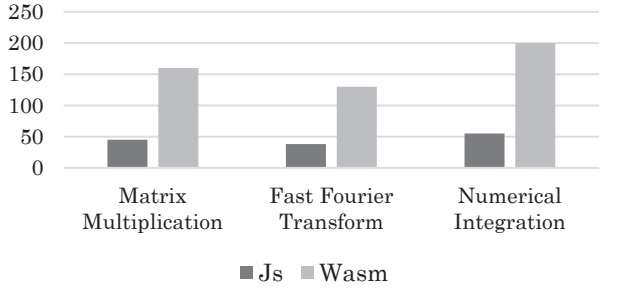

Fig. 7: Load Time Comparison (ms)

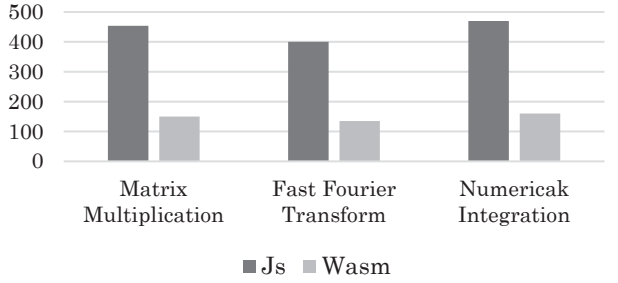

Table 5: Results Extraction ¦ Diverse Benchmarks

Task	Metric	JavaScript	WebAssembly	Gain (JS → Wasm)
Matrix Multiplication	Execution Time	1199.0 ms	392.0 ms	3.06 × faster
	Load Time	453.8 ms	150.0 ms	3.03 × faster
	Memory Usage	45.0 MB	160.0 MB	3.56 × more memory
Fast Fourier Transform	Execution Time	842.0 ms	200.8 ms	4.19 × faster
	Load Time	400.0 ms	135.0 ms	2.96 × faster
	Memory Usage	38.0 MB	130.0 MB	3.42 × more memory
Numerical Integration	Execution Time	2031.0 ms	610.0 ms	3.33 × faster
	Load Time	470.0 ms	160.2 ms	2.93 × faster
	Memory Usage	55.0 MB	200.0 MB	3.64 × more memory

SharedArrayBuffer, SIMD acceleration, and WASI expected to further improve performance and extend its use beyond the browser. Future studies should benchmark these features as browser support matures and inspect energy usage trade-offs, for mobile workloads, while improved tooling and stronger language support are likely to enhance developer productivity and adoption.

To infer, Wasm excels at accelerating compute-intensive tasks, while JS remains essential for interactive application logic. Choosing the appropriate technology for each task enables developers to build performant and maintainable web applications.

References

[1] D. Herman, "The Future of JavaScript: WebAssembly and Beyond," Communications of the ACM, vol. 61, no. 12, pp. 66–73, 2018.

[2] A. Haas et al., "Bringing the Web up to Speed with WebAssembly," in Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2017, pp. 185–200.

[3] M. Lehmann and F. Bräuer, "Performance Analysis of WebAssembly vs JavaScript," in Proc. of the 11th ACM/SPEC Int. Conf. on Performance Engineering, 2020, pp. 152–163.

[4] eBay Tech Blog, "Bringing Machine Learning Inference to the Browser Using WebAssembly," eBay Inc., 2021. [Online]. Available: https://tech.ebayinc.com

[5] M. Yan et al., "Demystifying Performance of WebAssembly," in Proc. of the 2021 Internet Measurement Conference (IMC '21), ACM, 2021, pp. 312–326.

[6] C. Gohman, "Optimizing Modern JavaScript: From V8 Internals to Real-World Performance," Google Developers Blog, 2020.

[7] A. Zakai, "Emscripten: An LLVM-to-JavaScript Compiler," in Proc. of the ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2011, pp. 301–312.

[8] P. Ross, "WebAssembly Security: Considerations for Sandboxed Code Execution," IEEE Security & Privacy, vol. 18, no. 5, pp. 66–70, 2020.

[9] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," in Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2017, pp. 185–200.

[10] Fastly, "Faster ID generation in WebAssembly using Rust," Fastly Developer Hub, 2020. [Online]. Available: https://www.fastly.com/blog/faster-id-generation-with-webassembly

[11] Mozilla Hacks, "Game Boy Emulator Benchmarking," Mozilla Blog, 2020. [Online]. Available: https://hacks.mozilla.org

[12] Google V8 Team, "JavaScript Performance Improvements," V8.dev, 2021. [Online]. Available: https://v8.dev/blog

[13] Samsung Internet Dev, "Performance Comparison: JS vs Wasm on Mobile," Samsung Developers Blog, 2021.

[14] D. Lehmann and F. Bräuer, "A Performance Study of WebAssembly, Native, JavaScript," in Proc. of the 11th ACM/SPEC Intl. Conf. on Performance Engineering (ICPE), 2020, pp. 152–163.

[15] TensorFlow Team, "WebAssembly Backend for TensorFlow.js," TensorFlow Blog, 2021. [Online]. Available: https://blog.tensorflow.org

[16] Cornell Research Group, "WebAssembly: Performance and Portability," Cornell Computer Science Tech Reports, 2020.

[17] Unity Technologies, "WebGL and WebAssembly Performance Considerations," Unity Manual, 2021. [Online]. Available: https://docs.unity3d.com

[18] Mozilla Developer Network (MDN), "Using the DOM from WebAssembly," MDN Web Docs, 2023. [Online]. Available: https://developer.mozilla.org

[19] J. Nielsen and R. Munoz, "Overhead of JavaScript-Wasm Interoperability in Web Applications," in Proc. of the 2022 ACM Web Conference (WWW), pp. 184–190.

[20] A. Zakai, "WebAssembly Threads and SIMD: Parallelism on the Web," WebAssembly Summit, 2022. [Online]. Available: https://webassembly.org

◆著者紹介

Madappuli Arachchige Kamila Sandeepa Fernando

Completed master's degree in the Web Business Technology Program at The Kyoto College of Graduate Studies for Informatics

Sameera Abar

Professor
The Kyoto College of Graduate Studies for Informatics