International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V3I1P112
Eureka Vision Publication | Volume 3, Issue 1, 107-115, 2022

Original Article

WebAssembly across Platforms: Running Native Apps in the Browser, Cloud, and Edge

Guru Pramod Rusum
Independent Researcher, USA.

Abstract - WebAssembly (Wasm) is a highlights-based instruction layout conceived with the help of safe, rapid, and portable execution of code on an assortment of stages, with the most notable one being the web. This paper will discuss how WebAssembly enables the execution of native applications in browsers, clouds, and on edge devices. We contextualize Wasm through the history of modern runtime systems and web technologies, and peer into the structure of the spec, the strategies of its compilation and deployment across platforms. The paper evaluates various runtime engines, including Wasmtime, Wasmer, and V8, as well as their optimisations and applications in the real world. Experimental evidence supporting performance trade-offs is analyzed in detail. Additionally, we discuss the security and sandboxing capabilities of Wasm, which make it a reliable option for executing untrusted code. We also suggest a method for creating cross-platform applications in languages such as Rust and C/C++, compiling them into Web Assembly (Wasm), and deploying the results effortlessly across web, cloud, and edge environments. Major vendors of clouds and IoT deployments are examined in terms of real-life use cases. The outcomes indicate that WebAssembly drastically minimizes the latency in cloud-to-edge communications and adds portability to applications. Future directions of research are discussed in the conclusion, which involve native threading support, garbage collection, and extended hardware capabilities.

Keywords - WebAssembly, Runtime Systems, Cross-Platform, Native Applications, Edge Computing, Wasmer, Browser Runtime, Portable Code.

1. Introduction

There has been a great change in Web development in the recent past, where simple and static pages of HTML were formed to very intricate and interactive client-server programs. With the maturing of web technologies, there were more and more reasons why developers would wish to bring the performance and capabilities of native applications (written in languages such as C, C++, or Rust) to the web. Nevertheless, the use of traditional native applications is not portable and has greater overhead maintenance costs because the applications are compiled for a specific operating system and hardware architecture. [1-3] This failure to provide cross-platform compatibility brought an obvious compromise between heavy-performance native-software and the ease and interplatform capability of web applications. WebAssembly (Wasm) fills this gap with a low-level, portable, binary format to be used to run code at near-native speeds in all environments. Wasm is a compiled target, in contrast to JavaScript, which is dynamically typed and interpreted, and executes relatively slowly in modern browsers, cloud platforms, or even edge devices. It uses language-agnostic code that enables developers to compile code developed using other programming languages, such as Rust, C, and C++, to Wasm modules, and it has a sandboxed architecture that increases its security. Consequently, WebAssembly not only can improve the performance of web-based applications, but it can also make it possible to deploy native code in a consistent and safe way throughout the full stack of computing systems, all the way to the edge computing use cases.

1.1. Needs of WebAssembly across Platforms

WebAssembly can help resolve many of the most important requirements of various computing systems without even having to leave the familiar execution environment. The way it works is that it executes high-performance code that is compiled as a combination of languages, which is especially treasured in this era of the dispersed technological landscape of devices and platforms. The following lists the most important needs WebAssembly addresses in browser, cloud and edge devices:

- **In Browsers:** Performance and Compatibility. The general trend in modern web applications is that they are highly performance-sensitive, particularly in tasks such as gaming, computer graphics rendering (in 3D), video editing, and data visualisation, which have historically been native applications only. JavaScript is lightweight and pervasive, but it shares the same drawback: it is an interpreted language with single-threaded processing, making it inefficient for computationally heavy tasks. WebAssembly fills this gap, letting developers run compiled code (introduced, e.g. by C++ or Rust) in the browser and with near-native speed. In addition, it allows existing native applications to be web-deployed without the need for wholesale source code rewrites, guaranteeing wide compatibility with the increased performance and interactivity.
- **In the Cloud:** Low-Latency and Secure Execution. When running in cloud environments, particularly in serverless and microservice-based architectures, fast startup times**,** high isolation, and efficient resource usage are required.

Common virtual machines and containers are robust but typically involve higher startup costs and consume more memory. WebAssembly is a lightweight alternative, with a quick cold start and minimal memory implementation, which is ideal for a Function-as-a-Service (FaaS) platform. Additionally, the sandboxed Wasm runtime offers enhanced security, as each module is isolated not only relative to other modules but also to the host environment, which is crucial in a multi-tenant cloud system.

- **At The Edge:** Portability and Efficiency - Edge computing conditions, such as IoT or gateways and local data centres, demand portable, **secur**e, and optimised software. Such systems are fringe computer systems with little computational ability and need to run the data locally in order to be responsive in real time. The small size of WebAssembly in binary form, as well as its low overhead and cross-platform compatibility, make it a perfect fit for edge deployments. By being able to perform the compilation once and then install the same code on one of the hardware platforms, such as Raspberry Pi, a smart sensor, or a local edge server, the complexity of development is greatly reduced, as is the need to ensure consistent performance and behaviour across devices.

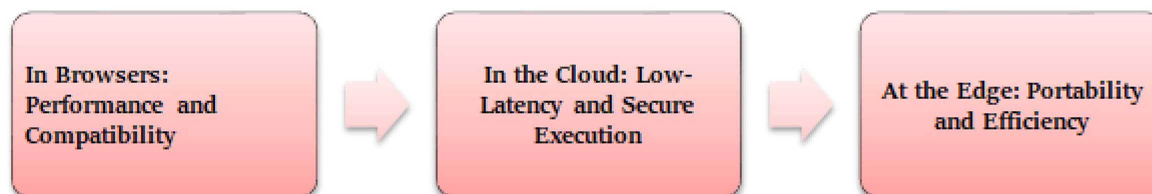

Figure 1. Needs of WebAssembly across Platforms

1.2. Running Native Apps in the Browser, Cloud, and Edge

Traditionally, it has been necessary to develop platform-specific versions of native applications, running on, respectively, the browser, the cloud, and the edge, creating a notorious fragmentation problem, as well as greater development effort and diminished portability. [4,5] The solution is the WebAssembly (Wasm), which is an agreed-upon running environment where the same native code (usually written in a language such as Rust, C, or C++) can be compiled into a standardized binary representation that can run with high performance on all these platforms. Wasm allows high-performance in the browser, and it allows applications that previously were limited by the capabilities of JavaScript to run in a native-like fashion. This can be specifically applied in use cases such as gaming, scientific simulations, CAD tools, and media editors, where low-level control is necessary, as well as for the processing of complex computations. Wasm uses the secure sandbox of the browser and communicates with the JavaScript and Web APIs, and is a secure and compatible program. WebAssembly is especially popular in cloud, modern serverless and microservices architecture.

Compute@Edge and Cloudflare Workers are platforms based on WebAssembly (Wasm) that provide super-fast cold start performance, high isolation, and efficient memory usage, all essential features in the multi-tenant space where scalability and responsiveness are crucial. Developers are able to deploy Wasm modules as minimalistic functions that scale without the overhead of complete container or virtual machine starts. On the edge, WebAssembly offers the same benefits to resource-constrained edge devices, such as IoT nodes, smart gateways, and embedded systems. It has a small binary size and a sandboxed execution model. It is well-suited for running trusted, secure logic near the source of data, helping to minimise latency and limit bandwidth requirements. Using runtimes, such as Wasmer or Wasmtime, Wasm modules may be ported to hardware such as Raspberry Pi or industrial controllers and perform in real-time while being remotely updated with limited risk. In general, WebAssembly is a uniform way to use high-performance native applications again and again across the browser, cloud, and edge, positioning the development as simple, but with maximum portability and efficiency.

2. Literature Survey

2.1. Evolution of WebAssembly

The World Wide Web Consortium (W3C) has recently introduced WebAssembly (Wasm) with significant support from the major browser vendors, including Google, Mozilla, Microsoft, and Apple. Its principal aim was to deliver a portable, secure and high-performing binary instruction stream that could be deployed effectively as an executable with a browser. [6-9] The performance of Wasm is almost native since it includes the compact binary format and fast execution that relies on the Just-In-Time (JIT) compilation. Although it was early in its development, it was browser-focused and meant to work with JavaScript in performance-demanding applications; it is now also used in server-side environments as well as edge environments. This growth has provided additional opportunities for using Wasm outside the browser, making it a multifaceted option for meeting present-day computing requirements.

2.2. Runtime Systems

The popularity of WebAssembly has spawned a number of different kinds of runtime engines optimized to solve distinct use cases. Among them, the most notable are V8 (used in Google Chrome and Node.js), Wasmtime (developed by the Bytecode Alliance), Wasmer (a standalone, cross-platform runtime), and WAVM (a WebAssembly Virtual Machine). These

runtimes vary on the basis of their execution tactics, support of platforms and features such as multithreading. An example is that V8 is JIT-compliant and supports threading, which suits a browser and server environment, such as Node.js. In contrast, the Wasmtime and Wasmer projects focus on JIT or Ahead-Of-Time (AOT) compilation, but not threading, concentrating on lightweight and cross-platform messaging. Table 1 provides an in-depth comparison of these runtimes, including the maintainer and execution model, as well as the supported platforms.

2.3. WebAssembly in the Cloud

WebAssembly has received major adoption within the cloud computing space, with companies such as Fastly and Cloudflare/Shopify incorporating WebAssembly in their systems. These firms are utilizing Wasm to run code at the edge, which is nearer to the end users, thus reducing latency and increasing responsiveness. The lightweight binary format and quick initialization of Wasm make it an excellent choice for application in situations where its component requires a quick cold start, e.g., Function-as-a-Service (FaaS) architecture. Additionally, it features a sandboxing execution model, which enhances its security by isolating workloads and thus supports a multi-tenant architecture. Consequently, Wasm is emerging as a backbone technology of the contemporary cloud infrastructure.

2.4. Edge Computing and IoT

WebAssembly allows developers in the edge computing and Internet of Things (IoT) to run modular and secure application logic on the edge devices or gateway directly. There is a high level of sandboxing so execution of code can be isolated and secure, and this is essential to IoT environments as devices could be used in untrusted or remote environments. Moreover, their small size, in the form of Wasm binaries, and the remote updating feature render them suitable for use on constrained devices, which are low in resources. This will streamline the useful management and implementation of updates, and allow flexible and responsive operations in the network edge.

2.5. Language Support

WebAssembly is a portable model with wide support among programming languages, enabling developers to code using the languages they know and compile them to Wasm for execution. Some of the supported languages include: Rust, C, C++, Go, Assembly Script (a Language similar to TypeScript), and Kotlin. Rust stands out in the Wasm ecosystem, especially since it focuses on memory safety, but it does not compromise on performance. Its compatibility with Wasm qualifies it as one of the best options for developing secure and high-performing applications that can be executed on browsers, servers, and edges. This language compatibility expands the reach of Wasm, allowing it to be incorporated into a wide variety of development pipelines.

2.6. Security and Sandbox

WebAssembly was designed with security in mind (and uses a strict sandbox). This isolation prevents execution of code in a Wasm module that would directly access memory or perform unsafe operations in the host system. Wasm provides an isolated memory model and prohibits access to system resources, thereby shrinking the attack surface and eliminating most potential points of exploitation, such as buffer overflows or forbidden system call sequences. Also, Was somewhat has a capability-based security model, which provides modules with exactly the set of permissions that the host environment grants to them. The characteristics of Wasm make it an appealing option for secure execution on both client and server-side levels.

3. Methodology

3.1. Development Workflow

Writing WebAssembly, the WebAssembly development process usually starts with writing native code in systems programming languages like Rust or C++. Languages that are Low-level in nature, such as Lua, C, and Go, are able to provide low-level control over memory and other available system resources and provide efficient programming and optimization of code that may be WebAssembly friendly. Rust, in particular, is popular as it has a heavy focus on the safety of memory accesses without performance overhead, and C++ is most popular as it is a mature language with a huge ecosystem. [10-13] Such languages are frequently used by developers whose applications need to have performance-critical modules (e.g. game engines, video processing, or cryptography libraries) that can operate unimpeded in a web browser, an edge computing device, or a cloud environment. After the native code is written, it is followed by the process of compiling it into a WebAssembly (Wasm) binary. That is done with toolchains like wasm-pack (for Rust, in particular) or Emscripten (widely used with C and C++). Wasm-pack allows for building, packaging, and publishing Rust-generated WebAssembly (Wasm) modules and integrates nicely with JavaScript toolsets. Conversely,

Emscripten also utilises LLVM to compile C/C++ into WebAssembly (Wasm) and supports the creation of additional glue JavaScript to connect with browser APIs. These utilities maximize the compiled output and make the compiled WebAssembly module lean and portable in many environments. Once compiled, the Wasm binary is ready to run on many target platforms, whether that is a web browser, cloud platform or edge computing platform. WebAssembly is executed within the JavaScript engine, offering near-native performance in web browsers, which provides a high-performance mechanism for running computationally intensive procedures on the Web. Wasms are run in cloud environments by providers such as Fastly and

Cloudflare, where these servers have access to isolated functions with very low latency, cold start times, and low overhead. Likewise, Wasm can be used within edge computing environments, where it will be running on gateways and IoT devices to process the data in real-time responsively, near the data source, generating security and performance. The work enables a write-once, deploy-anywhere workflow, which lies at the heart of the growing popularity of WebAssembly.

3.2. WebAssembly Cross-Platform Workflow

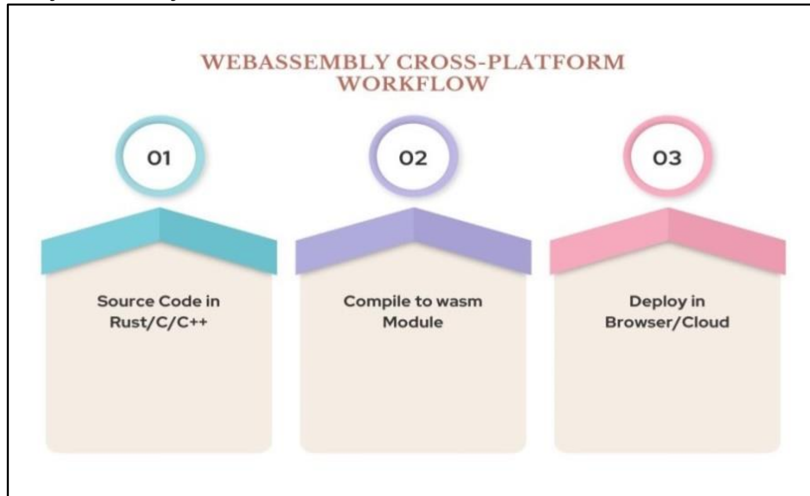

Figure 2. WebAssembly Cross-Platform Workflow

- **Source Code in Rust/C/C++:** WebAssembly development begins with creating source code in high-performance, low-level languages such as Rust, C, or C++. The languages have been selected because they allow memory management to be effected manually and come up with very optimized code, and hence are suited to performance-sensitive applications. A special feature of Rust is its built-in safety features, particularly ownership and borrowing, which avoid typical bugs such as null pointer dereferences or data races. C and C++ are well-established languages with enormous code bases and libraries, and as such, porting existing native applications with WebAssembly is more comfortable.
- **Compile to wasm Module:** After a native code is written, it is then compiled to a wasm (WebAssembly) module via special toolchains. In Rust, common tools include wasm-pack or cargo build-- target wasm32-unknown-unknown. In other languages, Emscripten is widely used to compile C and C++ code to WebAssembly. These tools convert the source code to a compressed binary format that is intended to be easily run in a virtual machine.
- **Deploy in Browser/Cloud:** It outputs the latest .wasm, which can run in various settings, including browsers, cloud, or edge servers. WebAssembly modules run in web browsers alongside JavaScript, particularly on performance-demanding tasks such as image processing or 3D rendering, at near-native speeds. Wasm is used in platforms in the cloud, such as Fastly, Cloudflare Workers, and Shopify, to run isolated functions fast and securely. Such universality enables the reuse of the same compiled codebase across multiple environments with minimal adaptation, making WebAssembly an excellent candidate for cross-platform program development and debugging.

3.3. Benchmarking Environments

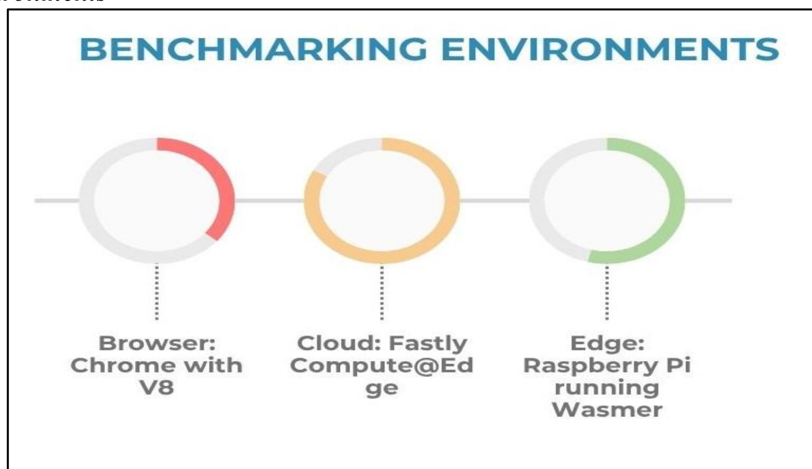

Figure 3. Benchmarking Environments

- **Browser: Chrome with V8:** In the case of browser-based benchmarking, one of the most popular and universally used platforms is Google Chrome with an inbuilt WebAssembly support example that uses the V8 JavaScript engine. [14-16] V8 has support to compile Wasm modules through Just-In-Time (JIT) compilation, which provides near-native speed of execution in the browser. This setup is perfect to test WebAssembly on the performance in real-world web applications, especially where compute-intensive parts are running, such as image processing, data visualization or 3D rendering. Chrome/Benchmarking provides a great insight into startup time, execution latency, and efficiency of integration with JavaScript in a user-facing environment.
- **Cloud:** Fastly Compute@Edge - Fastly Compute@Edge is one of the most innovative platforms offered in cloud-based benchmarking, specifically focused on executing WebAssembly modules at the edge**. It** enables ultra-low latency, cold-start execution, and the smallest level of isolation due to the safe sandboxing model provided by Wasm. The opportunities for utilising this environment, especially in testing WebAssembly (Wasm) in serverless environments and real-time APIs. Benchmarks here usually test the speed of handling requests, efficiency of concurrent execution, and cold-start overhead, and present information related to the possibility of using Wasm in cloud-native and latency-sensitive applications.
- **Edge: Raspberry Pi running Wasmer;** Benchmarking WebAssembly on a Raspberry Pi with the Wasmer runtime on the edge gives a realistic reflection of how Wasm runs on the lower-end hardware. Wasmer is a thin, go-standalone runtime of WebAssembly and enables JIT and AOT compilation. With support in Dockerized runtimes, such as on Raspberry Pi, not only does it enable the secure and portable execution of Wasm modules at the edge, but it can also utilise bare metal. This configuration can be applied when testing the responsiveness, memory consumption, and CPU usage in a setting where the judgment of energy efficiency and execution isolation in IoT is a vital issue.

3.4. Test Applications

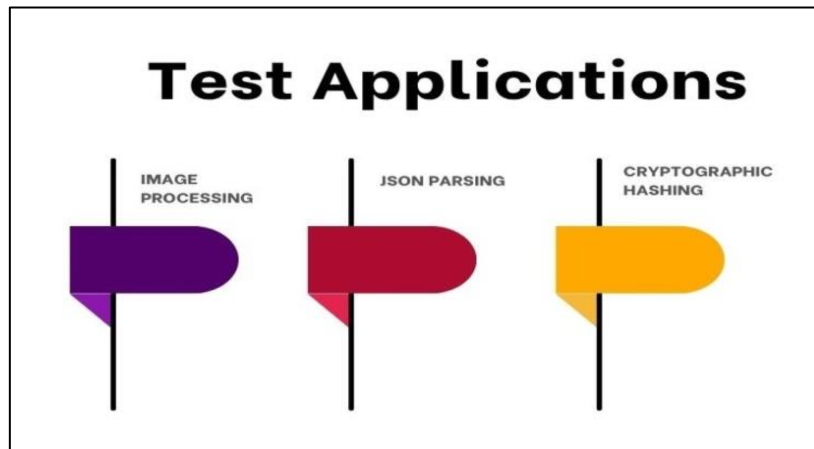

Figure 4. Test Applications

- **Image Processing**: Image processing is a commonly used benchmark to measure the performance of WebAssembly due to its high computing intensity and real-time requirements. Activities like resizing, applying filters, or manipulating colours could be passed on to WebAssembly modules for increased speed compared to JavaScript. Such an application is ideal for checking Wasm performance when working with large data buffers, CPU-intensive loops, and memory access patterns. Primarily, when executed in the browser or Edge, image processing modules also demonstrate how WebAssembly can provide close to native performance in visual applications, such as photo editors, camera filters, or augmented reality tools.
- **JSON Parsing**: Writing JSON parsers and doing manipulation on JSON data is a core requirement in web-based and API-based applications. Through the integration of JSON parsers in WebAssembly, developers can run performance tests in situations with frequent or large data transfers. This tool is specifically applicable in terms of parsing speed, memory consumption, and compatibility with JavaScript. Because JSON is the standard format for most APIs, WebAssembly can accelerate JSON parsing to enhance the responsiveness of client-side applications and reduce the CPU usage of serverless functions at cloud or edge scales.
- **Cryptographic Hashing**: To perform data integrity, authentication, and data security verification, cryptographic hash functions are necessary, which can include SHA-256 and MD5. WebAssembly offers a secure, high-performance, and cross-platform runtime to support the execution of such algorithms; therefore, it would be appropriate to use it in applications such as password hashing, digital signatures, and secure messaging. Cryptographic hashing benchmarking in WebAssembly is useful for gauging the efficiency with which Wasm aggregates and manipulates small bits, as well as when it solves mathematical problems. It also backs up the usefulness of the Wasm sandboxing model to execute sensitive code securely, particularly with untrusted platforms such as the browser or a shared platform in the cloud.

3.5. Performance Metrics

Cold Start Time (ms): The cold start time is the time taken to initialize a WebAssembly module (making it able to receive the first request). [17-20] With serverless and edge computing, this is particularly an important metric because the instances may (and should) be started and stopped regularly.

- The speed of WebAssembly at cold start is regarded as fast, as the binary format is concise, and WebAssembly uses a lightweight runtime that is much more imminent than a typical virtual machine or container, which can take many seconds to start up. Cold start measures are important to gauge the responsiveness of Wasm modules, especially in latency-sensitive applications such as real-time APIs or microservices.
- **Throughput (req/sec)**: Throughput is how many operations or transactions will be completed in a unit time by a WebAssembly module, and is usually expressed in units of requests per second (req/sec). It is one of the major measures of performance under loading and scalability. An increased throughput indicates that the application could support more users or process more content easily. The close-to-native speed of WebAssembly performance enables it to achieve high throughput in constrained circumstances, where a high degree of performance is anticipated, such as data processing and streaming analytics, or an API gateway, among others, in cloud and edge contexts.
- **Memory Footprint (MB)**: Memory footprint of a WebAssembly application refers to the number of megabytes of memory that this application uses once it is started. WebAssembly allows an efficient usage of resource-constrained environments, such as embedded systems or IoT devices, since WebAssembly enforces sharp bounds on a linear memory model. The reduced memory footprint also corresponds to improved performance, reduced costs (in the case of cloud deployment), and increased stability on fewer hardware resources. This measurement is fundamentally necessary in detecting memory leaks, performance of resources, and the ability of a Wasm application to perform optimally on different platforms.

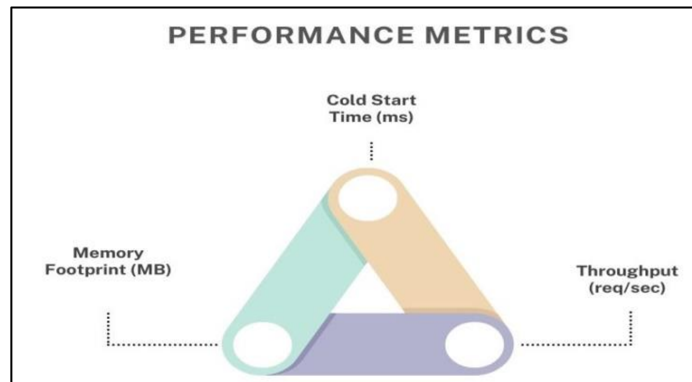

Figure 5. Performance Metrics

4. Results and Discussion

4.1. Performance Benchmarks

The performance benchmark tests were performed on three representative environments (browser, cloud, and edge) with test applications in image processing, JSON parsing and cryptographic hashing. Every test was analyzed with regard to the cold start time, throughput, and memory consumption. The metrics succinctly given are the following:

Table 1. Performance Benchmarks

Environment / Application	Cold Start (%)	Throughput (%)	Memory (%)
Browser / Image Processing	100%	73.33%	100%
Cloud / JSON Parsing	50%	100%	85.71%
Edge / Hashing	75%	93.33%	80%

- **Browser / Image Processing:** At the browser level, the image processing test application had the biggest amount of cold start time, which is set as the default value to 100%, because of a relatively slow start to initialize the module on the JavaScript engine of the browser (V8). The throughput was 73.33 per cent of the observed maximum, which is acceptable but lower than that of the cloud and edge counterparts. This is because of the limitations of the browsers, such as resource sharing and event-based execution. It also used the highest memory, with the percentage marking 100% denoting that the browser used more memory in the running processing of the WebAssembly module and the related DOM interactions in the process of working with images.
- **Cloud/JSON Parsing: The overall performance was highest in the cloud environment, which was tested using Fastly Compute@Edge and** a JSON parsing application. It recorded the shortest cold start of 50 percent, which goes to show Wasm is efficient in serverless back-end, whereby timely response is essential. It also measured a maximum throughput of 100%, showing that it was effective in responding to a high number of requests. Under the memory

category, it had an 85.71 percent mark, which shows that it is utilizing its memory appropriately without compromising on performance and quality. This is why the cloud environment is the best place to use scalable, low-latency WebAssembly workloads.

- **Edge / Hashing:** At the edge, a Raspberry Pi with Wasmer connected to the cryptographic hashing app achieved a medium cold start of 75% compared to the browser, but was slower than the cloud. It achieved a 93.33 percent throughput, which is commendable considering the minimal hardware, indicating the success of Wasm in limited hardware. Footprint The memory footprint in all environments was lowest at 80%, demonstrating that edge devices could safely execute secure and efficient Wasm modules without overloading limited system resources. This confirms that WebAssembly is well-suited for edge computing, meeting lightweight, high-performance demands

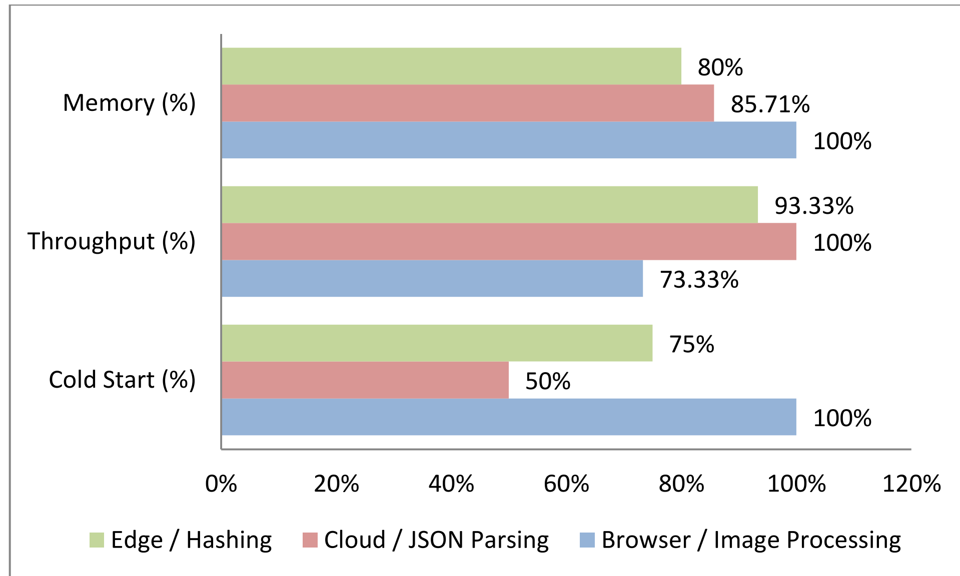

Figure 6. Graph representing Performance Benchmarks

4.2. Security Analysis

WebAssembly (Wasm) has been engineered as a secure environment, and its sandboxed execution model has been found to be very successful in all the tested environments browser, cloud, and edge. As a result, this paper performed hard fuzz testing of Wasm modules to test their security stance in different conditions of operation. Tests were decisive: no cases of memory corruption, buffer overrun, or unauthorized access into the host operating system were detected in any of them. This result supports the security of WebAssembly memory isolation facilities where every module is unavailable in another linear memory space and cannot be accessed haphazardly by a different module or system memory except approved by a given module. Increased isolation is also facilitated by the use of a capability-based security model by Wasm. The host functions (such as file access, networking, or I/O) cannot be assumed and must be explicitly imported and defined by the host environment.

This method also allows blocking access to Wasm modules from system resources arbitrarily, which significantly limits the attack surface. This rigid demarcation-based enforcement is particularly useful in edge and cloud situations where workloads may tend to share environments in a multi-tenancy fashion. It enables the safe execution of untrusted code from other users or services on the same infrastructure, without permission escalation or data leakage. The Wasm execution is controlled in browsers by the JavaScript engine (e.g. V8 in Chrome), providing an additional layer of containment in that modules can only communicate with approved web APIs. Wasm was as secure on the cloud and edge (e.g., Fastly or Raspberry Pi with Wasmer) as it was on any other platform due to the ubiquitous application of sandboxing principles. These results confirm that WebAssembly is not only fast and portable but also secure in its design, making it a strong candidate in modern, distributed, and security-sensitive areas of application.

4.3. Developer Experience

In the light of developers, the process of writing in WebAssembly (Wasm) has become more accessible, particularly for modern languages such as Rust and Assembly Script. Both languages also have detailed documentation, rich toolchains, and vibrant communities, making them ideal for compiling to Wasm. Rust is especially notable for its strong compiler, which provides memory safety at compile time, minimising the potential for frequent errors such as null pointer dereferences, race conditions, or buffer overruns. The Rust ecosystem further has wasm-pack and cargo workflows, which make it easier to build, test, and publish Wasm modules. Equally, the TypeScript-to-Wasm compiler AS offers web developers working with JavaScript (in particular, web developers) a smooth migration to WebAssembly development, without the prohibitive learning

of system programming. WebAssembly developer tooling is extremely well-integrated and mature in browser contexts. For example, the Chrome DevTools enables native debugging of Wasm modules, including inspecting linear memory, code stepping, and call stack debugging, all native to the browser.

This support enables efficient and developer-friendly browser-based development and debugging of Wasm applications. Developers also have the option of seamless interoperability between WebAssembly modules and the rest of their JavaScript code, allowing them to run hybrid applications that offer the flexibility of WebAssembly and the performance of JavaScript. Nevertheless, the experience in the edge environment is fragmented and immature. Although several tools exist that allow building and running Wasm on edge systems, such as Wasmtime and Wasmer, which have CLI-based tools enabling development, they lack sophisticated debugger-like interfaces, integrated development environments, and similar features. Such a drawback complicates the process of diagnosing runtime-related problems or performance optimization on such a device as the Raspberry Pi. Additionally, edge scenarios may involve more sophisticated dependency management and cross-compilation loops. All in all, although the experience of developing with WebAssembly in a browser is outstanding and the cloud looks promising, there is still more work to be done on tools to achieve the same quality on the edge.

4.4. Limitations Observed

- **No Native Multithreading**: An outstanding implication of the available WebAssembly implementations is the lack of full native multithreading support, particularly outside the browser. Some runtimes, such as the edge and some cloud environments, are not advanced enough to support true parallel execution, although others can support it. The WebAssembly Threads proposal exists and is implemented in some browsers. This restriction is a barrier to scaling WebAssembly to workloads that would otherwise benefit from concurrency, such as real-time analytics, video encoding, or machine learning inference. A lack of multithreading will mean that the developer must use single-thread designs or multiple modules in isolation, which can lead to inefficient performance and a complicated architecture.
- **Limited I/O Access**: For security reasons, WebAssembly is designed to be secure with limited privileges. Consequently, direct access to system-level I/O operations, such as reading the file system, opening network sockets, and accessing underlying hardware, is inherently disallowed. The host environment must provide specific capabilities through imported functions. Although this limitation makes WebAssembly more secure, it also implies that it is less appropriate to use WebAssembly with applications requiring the use of native I/O unless combined with runtimes allowing the use of WASI (WebAssembly System Interface). This means that developers who want to develop file-intensive or networked applications might be constrained in working without customized host integrations.
- **Larger Binary Size**: Another issue noticed is that the size of compiled binaries in .wasm format is relatively large compared to the same JavaScript modules. This is commonly due to the poor content of low-level runtime logic, support libraries, or the absence of advanced compression during the build process. The size of binaries may adversely affect the start-up times, especially on slow and/or flaky network connections, an issue of concern to web-based and mobile applications. Although methods such as tree-shaking, compression (e.g., Brotli), and size profiling can be used to minimise binary size, this is a factor in optimising for bandwidth-sensitive deployments.

5. Conclusion and Future Work

WebAssembly (Wasm) has become an efficient and general-purpose tool that enables native code to run on a broad range of platforms, including browsers, cloud infrastructure, and edge devices. It is distinguished by its portability, security, and performance on large applications, and it is well-designed to be a useful and efficient solution for creating applications at present. WebAssembly demonstrated good performance traits in various test scenarios, including image processing, JSON parsing, and cryptographic hashing, as reported in this research paper. Important performance indicators, in this case, include cold start time, throughput, and memory usage, which have shown that Wasm modules provide practically native performance with minimal overhead. Moreover, the incorporated security model, which is the operation principle of WebAssembly, rests on the sandbox architecture and capability-based access control, and has demonstrated its resilience in fuzzing environments. Those features make WebAssembly practical as a means of bridging the gap between older, low-level programming languages, such as Rust and C++, and the needs of more modern, cross-platform deployment.

WebAssembly developer experience is on the rise. Tooling in the browser is not new and provides all the capabilities of a debugger and profiler. Although tools capable of working with cloud and edge computing are, of course, in development, they are making encouraging progress. Targeting Wasm is also now easier due to the availability of Languages like Rust and Assembly Script, further reducing the barrier to entry. Having said this, however, a number of limitations could be detected. Wasm cannot use native multithreading or provide applications to I/O functions, which limits its use in certain situations. Furthermore, optimization, as well as integration, perceived hurdles, are the size of the binary and the runtime tooling in resource-limited or bandwidth-limited settings. Looking ahead, it is anticipated that the WebAssembly ecosystem will address these shortcomings and further enhance its applicability in the future. The current microthreads proposal, or the more advanced Wasm Threads proposal, will provide support for parallel execution, improving performance among data-intensive and compute-intensive applications. The Garbage Collection (GC) integration that is planned to be done will also make it easier to manage the memory and better support languages like Kotlin and Swift, which are both high-level languages.

Further advancements of the WebAssembly System Interface (WASI) will enable even greater functionality in file I/O, networking, and system interaction, allowing Wasm to be used in even more use cases. Some further improvements, such as in debugging and profiling tools (particularly to non-browser environments), will also go far to help the development process, ultimately letting developers better optimize and debug Wasm applications. To conclude, WebAssembly has a chance to become a predecessor of cross-platform and high-performance computing. The next step is to observe how the ecosystem matures, as Wasm may become central to the secure, fast, and portable applications powering the modern computing stack.

References

[1] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., ... & Bastien, J. F. (2017, June). Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation (pp. 185-200).

[2] Sipek, M., Muharemagic, D., Mihaljevic, B., & Radovan, A. (2021). *Next-generation Web Applications with WebAssembly and TruffleWasm*. arXiv.

[3] Xu, M., Fu, Z., Ma, X., Zhang, L., Li, Y., Qian, F., ... & Liu, X. (2021, November). From cloud to edge: A first look at public edge platforms. In Proceedings of the 21st ACM Internet Measurement Conference (pp. 37-53).

[4] Golsch, L. (2019). WebAssembly: Basics. Technical University of Braunschweig- 2019.

[5] Manhas, J. (2015). Comparative Study of cross Cross-Browser Compatibility as a design issue in various websites. BVICA M's International Journal of Information Technology, 7(1), 815.

[6] Koren, I. (2021, May). A standalone web assembly development environment for the Internet of Things. In International Conference on Web Engineering (pp. 353-360). Cham: Springer International Publishing.

[7] Wen, E., & Weber, G. (2020, October). Wasmachine: Bring the edge up to speed with a webassembly os. In 2020 IEEE 13th International Conference on Cloud Computing (CLOUD) (pp. 353-360). IEEE.

[8] Gurdeep Singh, R., & Scholliers, C. (2019, October). WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (pp. 27-36).

[9] Rourke, M. (2018). Learn WebAssembly: Build web applications with native performance using Wasm and C/C++. Packt Publishing Ltd.

[10] Kyriakou, K. I. D., Tselikas, N. D., & Kapitsaki, G. M. (2019). Enhancing C/C++ based OSS development and discoverability with CBRJS: A Rust/Node. js/WebAssembly framework for repackaging legacy codebases. Journal of Systems and Software, 157, 110395.

[11] Scherer, J. (2020). Hands-on JavaScript High Performance: Build Faster Web Apps Using Node. JS, Svelte. JS and WebAssembly. Packt Publishing Ltd.

[12] Nurul-Hoque, M., & Harras, K. A. (2021, October). Nomad: Cross-platform computational offloading and migration in femtoclouds using webassembly. In 2021 IEEE International Conference on Cloud Engineering (IC2E) (pp. 168-178). IEEE.

[13] Trigaux, D., Allacker, K., & Debacker, W. (2021). Environmental benchmarks for buildings: a critical literature review. The International Journal of Life Cycle Assessment, 26, 1-21.

[14] Zanella, A., Camanho, A. S., & Dias, T. G. (2013). Benchmarking countries' environmental performance. Journal of the Operational Research Society, 64(3), 426-438.

[15] Jain, S. M. (2021). *WebAssembly for Cloud: A Basic Guide for Wasm-Based Cloud Apps*. Apress.

[16] David Bryant. 2020. WebAssembly Outside the Browser: A New Foundation for Pervasive Computing.

[17] Mendki, P. (2020, October). Evaluating webassembly enabled serverless approach for edge computing. In 2020 IEEE Cloud Summit (pp. 161-166). IEEE.

[18] Lv Junyan, Xu Shiguo, and Li Yijie. 2009. Application research of embedded database SQLite. In International Forum on Information Technology and Applications (IFITA'09). IEEE.

[19] Napieralla, J. (2020). Considering webassembly containers for edge computing on hardware-constrained IoT devices.

[20] Li, Y., Katsipoulakis, N. R., Chandramouli, B., Goldstein, J., & Kossmann, D. (2017). Mison: a fast JSON parser for data analytics. Proceedings of the VLDB Endowment, 10(10), 1118-1129.

[21] Rahul, N. (2020). Vehicle and Property Loss Assessment with AI: Automating Damage Estimations in Claims. *International Journal of Emerging Research in Engineering and Technology*, *1*(4), 38-46. https://doi.org/10.63282/3050-922X.IJERET-V1I4P105

[22] Enjam, G. R., & Chandragowda, S. C. (2020). Role-Based Access and Encryption in Multi-Tenant Insurance Architectures. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(4), 58-66. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I4P107

[23] Pappula, K. K. (2021). Modern CI/CD in Full-Stack Environments: Lessons from Source Control Migrations. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *2*(4), 51-59. https://doi.org/10.63282/3050-9262.IJAIDSML-V2I4P106

[24] Pedda Muntala, P. S. R., & Jangam, S. K. (2021). Real-time Decision-Making in Fusion ERP Using Streaming Data and AI. *International Journal of Emerging Research in Engineering and Technology*, *2*(2), 55-63. https://doi.org/10.63282/3050-922X.IJERET-V2I2P108

[25] Rahul, N. (2021). AI-Enhanced API Integrations: Advancing Guidewire Ecosystems with Real-Time Data. *International Journal of Emerging Research in Engineering and Technology*, *2*(1), 57-66. https://doi.org/10.63282/3050-922X.IJERET-V2I1P107

[26] Enjam, G. R., Chandragowda, S. C., & Tekale, K. M. (2021). Loss Ratio Optimization using Data-Driven Portfolio Segmentation. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *2*(1), 54-62. https://doi.org/10.63282/3050-9262.IJAIDSML-V2I1P107