

ORIGINAL

## Code optimization opportunities in the JavaScript ecosystem with Rust

## Oportunidades de optimización del código en el ecosistema JavaScript con Rust

Volodymyr Kozub<sup>1</sup>  

<sup>1</sup>National Aviation University, Kyiv, Ukraine.

Cite as: Kozub V. Code optimization opportunities in the JavaScript ecosystem with Rust. LatIA. 2024; 2:68. <https://doi.org/10.62486/latia202468>


Submitted: 20-01-2024

Revised: 15-05-2024

Accepted: 14-11-2024

Published: 15-11-2024

Editor: Dr. Rubén González Vallejo 

Corresponding author: Volodymyr Kozub 

### ABSTRACT

This paper explores the potential of optimizing node.js applications by integrating rust. In particular, in processing cpu-intensive tasks where javascript faces performance limitations due to its single-threaded architecture. Rust's memory safety and parallelism model, which eliminates the need for a garbage collector, makes it an attractive alternative to traditional c/c++ modules for extending the capabilities of node.js. This study explores the performance gains achieved by integrating rust, both through native bindings and WebAssembly, demonstrating significant improvements in computational efficiency, especially in parallel processing scenarios. Rust's ability to efficiently handle computation-intensive workloads with work interception algorithms is emphasized as a key factor in overcoming javascript bottlenecks. The study includes a detailed performance evaluation that compares synchronous and asynchronous modules in node.js with rust implementations. Tests demonstrate how rust optimizations outperform javascript by up to ten times in certain computational tasks. The study also evaluates cross-compiled rust modules using WebAssembly in the browser environment, which once again illustrates the advantages of rust in providing near-native performance. The results emphasize the potential of rust to enhance node.js applications by making them more scalable, reliable, and efficient for high-performance web applications.

**Keywords:** Node.js Optimization; Rust Integration; Integration of Programming Languages; Node.js Modules.

### RESUMEN

Este artículo explora el potencial de optimización de las aplicaciones node.js mediante la integración de rust. En particular, en el procesamiento de tareas intensivas en cpu, donde javascript se enfrenta a limitaciones de rendimiento debido a su arquitectura monohilo. La seguridad de memoria y el modelo de paralelismo de Rust, que elimina la necesidad de un recolector de basura, lo convierten en una alternativa atractiva a los módulos c/c++ tradicionales para ampliar las capacidades de node.js. Este estudio explora las ganancias de rendimiento conseguidas mediante la integración de rust, tanto a través de bindings nativos como de WebAssembly, demostrando mejoras significativas en la eficiencia computacional, especialmente en escenarios de procesamiento paralelo. La capacidad de Rust para manejar eficientemente cargas de trabajo intensivas en computación con algoritmos de interceptación de trabajo se enfatiza como un factor clave para superar los cuellos de botella de javascript. El estudio incluye una evaluación detallada del rendimiento que compara módulos síncronos y asíncronos en node.js con implementaciones de rust. Las pruebas demuestran cómo las optimizaciones de rust superan a javascript hasta diez veces en determinadas tareas computacionales. El estudio también evalúa módulos de rust compilados de forma cruzada utilizando WebAssembly en el entorno del navegador, lo que

ilustra una vez más las ventajas de rust a la hora de proporcionar un rendimiento casi nativo. Los resultados enfatizan el potencial de rust para mejorar las aplicaciones node.js haciéndolas más escalables, fiables y eficientes para aplicaciones web de alto rendimiento.

**Palabras clave:** Optimización de Node.js; Integración de Rust; Integración de Lenguajes de Programación; Módulos de Node.js.

## INTRODUCTION

Optimization of Node.js code in web applications has become an urgent task for developers who face performance and efficiency issues when working with a single-threaded environment. Despite the flexibility and convenience of JavaScript, which allows you to quickly create scalable systems, the limitations caused by single-threading and dynamic memory management sometimes make it difficult to perform resource-intensive tasks (Pratama & Raharja, 2023).

With the growing demands for parallel processing and complex computing, approaches such as integrating C/C++ modules or using the modern Rust language are becoming critical to solving these problems. Rust's capabilities make it a good match for Node.js, especially for tasks that require high parallelism and efficient resource utilization. In addition, the emergence of WebAssembly has further expanded the possibilities for integrating Rust into web applications, offering near-native performance and the ability to work seamlessly with JavaScript. Using Rust can complement JavaScript in Node.js applications and improve not only performance but also security. By integrating Rust, developers can retain the flexibility of JavaScript while gaining the performance benefits of a system programming language. This is necessary in CPU-intensive tasks and applications that require high parallelism (Serefaniuk, 2024).

The study examines the methodology for optimizing web applications to improve their reliability and scalability. Particular attention is paid to the use of distributed caching, architectural approaches, and programming languages that allow for better performance. The methodology is focused on building and testing Node.js application architecture, as well as exploring opportunities to improve efficiency by porting resource-intensive components to Rust. This allows you to analyze the impact of various architectural solutions and tools on the performance and scalability of web applications.

The basic structure of pure JavaScript web applications in Node.js involves a server side that processes HTTP requests, performs routing, data processing, and is responsible for interacting with databases. The event management model is used, in which each request is queued and processed sequentially. This allows you to process many requests simultaneously, but limits the efficiency at high computing loads. As an example, a simple project to calculate a number using the Monte Carlo method is used to learn the architecture of Node.js applications. This task requires a pseudo-random number generator (PRNG) that must use an initial value to ensure reproducible results. In a typical Node.js application, the workspace is structured in a single directory, which provides a visual representation of the project workflow.

The project is initialized using npm init, creating a package.json file. This file contains metadata, dependencies, and scripts for tasks such as testing, benchmarking, and building the application. Node.js uses the CommonJS module system, which allows developers to break logic into separate reusable modules. In this project, the first module implemented is the RNG, which uses a modified Park-Miller algorithm, chosen for its simplicity. Tests are configured using the NODE\_ENV environment variable, which allows you to distinguish between test and production environments.

The second version of the RNG is built using bitwise operations for better performance. To measure performance, the benchmark.js framework is included in the project. The number evaluation module is written using the RNG, and tests are added to check the accuracy of the number value. Next, benchmarks are run to evaluate the overall performance of the system. Node.js offers modules that allow you to create multi-threaded request processing. Thus, the node-gyp module is a critical tool for compiling native modules into Node.js, especially when you need to integrate native code written in C or C++ into the Node.js environment. The tool is based on Google GYP (Generate Your Projects), a Meta-Build system that allows you to create configuration files for different platforms, making it easier to support cross-platform development. Node-gyp automatically generates and customizes build files (e.g., Makefile or Visual Studio Solution), taking into account different versions of Node.js and the features of the platform on which it is executed.

Native modules allow you to perform computationally intensive tasks outside of the JavaScript interpreter, which significantly improves application performance. Node-gyp allows you to integrate C/C++ code into Node.js, turning it into a native module. This tool automatically compiles and links native code for the target platform, which ensures its compatibility with the Node.js API and makes direct integration with JavaScript

possible.

One of the main challenges in developing native modules is that different platforms (Linux, macOS, Windows) have their own build and configuration systems. Node-gyp automatically generates configuration files specific to each of them, making it easy to work with C/C++ code regardless of the platform. Thanks to GYP support, node-gyp can be customized for different versions of Node.js, automatically taking into account changes in the ABI (Application Binary Interface) between versions. This allows native modules to maintain backward compatibility with different versions of Node.js without the need for significant code changes.

## METHOD

When building native Node.js applications, a common approach is to use the node-gyp tool, which is based on Google's Meta-Build system, GYP. This tool allows you to cope with the complexities of working with different build platforms and versions of Node.js. The key advantage is that developers don't have to worry about the underlying V8 engine. Instead, they can use abstraction layers such as NAN (Native Abstractions for Node.js), which simplifies the process of creating native bindings. NAN provides a C++ header file with macros and utilities that help manage the interaction of native code with Node.js. To integrate native code into the project, dependencies are included by running `npm install --save node-gyp nan`. The configuration for building the application is saved in a file called `binding.gyp`, which defines how to compile C++ code into a native module. This file is placed together with `package.json` in the project root.

Native code written in C++ is stored in a separate directory (usually called `native`), and the main C++ source file, `addon.cc`, is used to export C++ functionality to JavaScript. This file is referenced in `binding.gyp` as the source file. The compilation process includes binding the code to the V8 and NAN header files, which ensures seamless compatibility between C++ and JavaScript. Once the native code is written, it can be built by running `node-gyp configure` and then `node-gyp build`, which compiles the native module. The resulting files are placed in the `build/` directory. The `index.js` file in the project is responsible for exporting the compiled C++ modules, making them suitable for reuse in the JavaScript environment.

For tasks that require significant computing power, such as math, image processing, or encryption, parts of the code are rewritten in Rust. Rust provides high performance and memory control without the need for a garbage collector like JavaScript. Although Rust is a system programming language, it can be easily integrated into the workflow and structure of a Node.js project. Similar to Node.js, Rust projects are organized in a directory called a `crate`, which is the equivalent of a Node.js module. To start the migration, a container is created in the directory using the `cargo` command, a Rust build tool similar to `npm`. Running `cargo new pi_estimator` creates a directory named `pi_estimator` with a structure similar to how `npm init` works in Node.js. Using WebAssembly to compile Rust code and integrating it into Node.js allows you to execute Rust functions at high speed while interacting with JavaScript. WebAssembly provides low access to system resources and high performance.

The equivalent of `package.json` in Rust is the `Cargo.toml` file, which manages project dependencies and configurations. All source code is placed in the `src/` directory. An RNG module originally written in JavaScript can be ported to Rust by translating the JavaScript code line by line to Rust syntax. Rust syntax is similar to JavaScript syntax. Especially with support for programming functionality such as loops, maps, and filters, which makes it easier for developers familiar with JS to transition.

While direct numeric operations may not be fully optimized when first translated, Rust's performance can be improved over time by refining low-level code, something that JavaScript cannot offer. Rust also differs from C++ in that it doesn't rely on classical inheritance, instead allowing you to implement methods for structures in a way that resembles the JavaScript prototype chain. JavaScript implementation tests can also be ported to Rust. The `cargo test` command in Rust works similarly to the Node.js `npm test`. Additionally, Rust has benchmarking tools, such as `Bencher`, that can be used to evaluate the performance of both the original JavaScript implementation and the new version of Rust.

After the RNG module is ported and tested, the number estimation function will also be ported to Rust, including tests and benchmarks. You can compare the performance between the JavaScript and Rust implementations to see if Rust offers any noticeable improvements. If the Rust implementation proves to be better, the last step is to create bindings to link it to Node.js. Since Node.js applications are written in C++, Rust code must be wrapped in a C-compatible interface to interact with Node.js. Rust can mimic C by using C-compatible types and C-style function definitions, allowing a Node.js application to call Rust code directly. Rust compiles as a shared object using the `cdylib` option in the `Cargo.toml` file. No additional configuration files or Makefiles are required, as Rust can build both the test and production versions using the `cargo build` and `cargo build - release` commands. Once compiled, the Node.js application described in the previous section can bind to the Rust object, exposing its functionality to JavaScript. The `binding.gyp` file can be updated to automatically build the Rust object when node-gyp is called.

## RESULTS

### Performance evaluation overview

This section presents the results of the performance evaluation, focusing on CPU utilization, memory allocation, and computational performance. The results are visualized through a series of charts that provide a detailed comparison of each approach. The analysis shows that modules that lack parallel execution capabilities, such as purely synchronous implementations, consistently use a single processor core. As shown in Figure 4, the CPU load of synchronous modules remained unchanged regardless of the increase in sample size.

In contrast, modules with parallel mechanisms, such as `addonConAsync`, showed performance improvements with increasing input size, with more noticeable improvements starting with  $1 \times 10^6$  samples.

For example, at this level, `addonConAsync` ran on about three cores, while Rust-based work interception applications such as `addonWsAsync` used about five cores. As the workload increased, all work interception approaches peaked at 13 cores at the sample  $1 \times 10^7$  mark, while more traditional parallel implementations remained limited to about three cores. The highest utilization was observed at  $1 \times 10^8$  samples and above, with modules such as `addonWsAsync` approaching the theoretical benchmark maximum of 24 cores (figure 1).

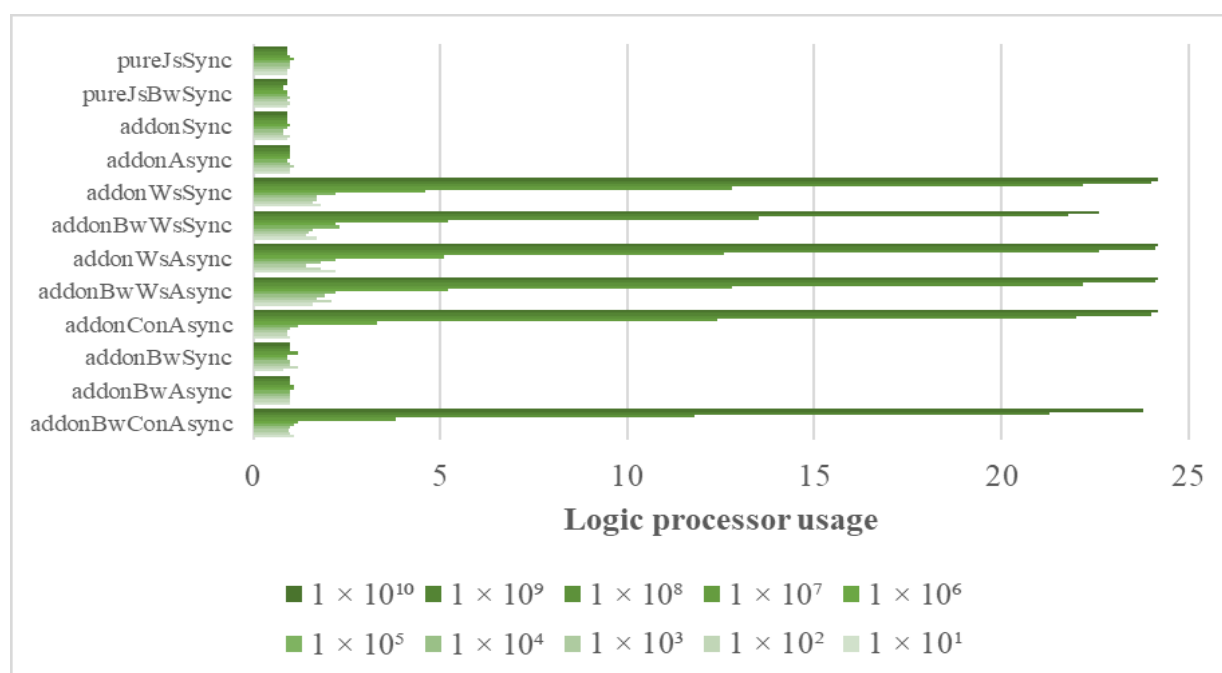


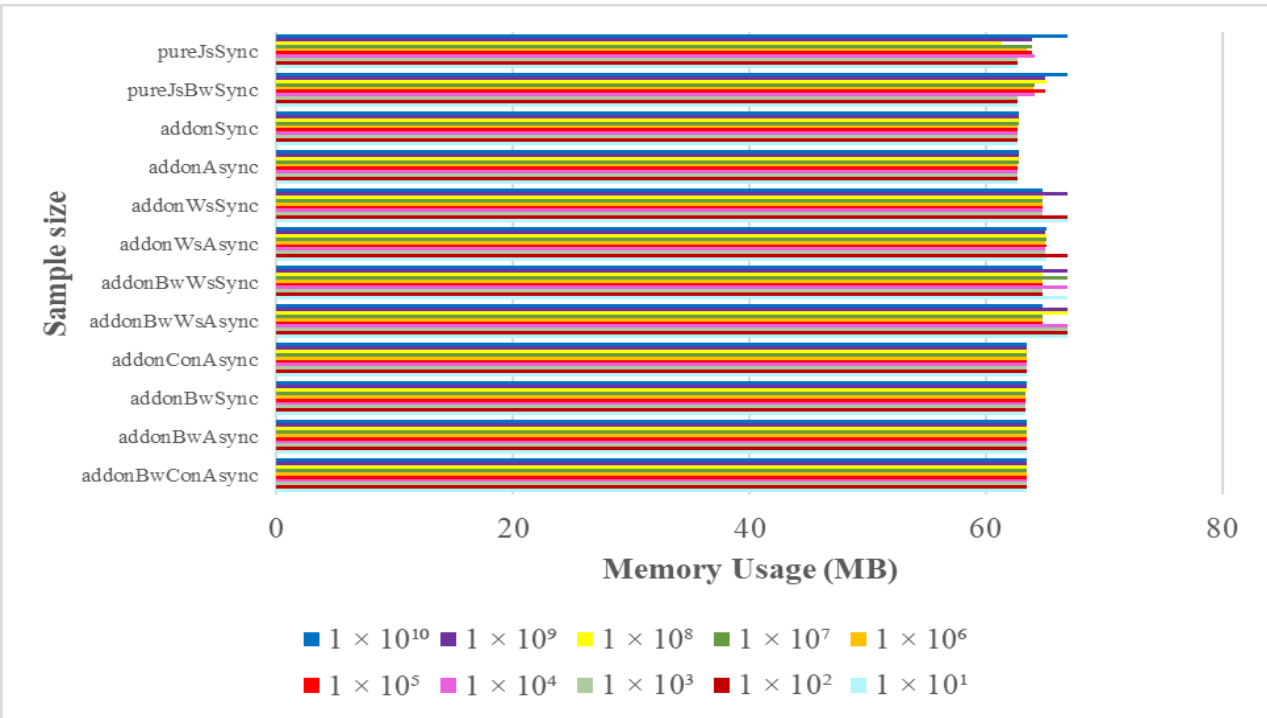
Figure 1. Average CPU load in Node.js

Source: developed by the author based on Kyriakos-Ioannis & Nikolaos (2022).

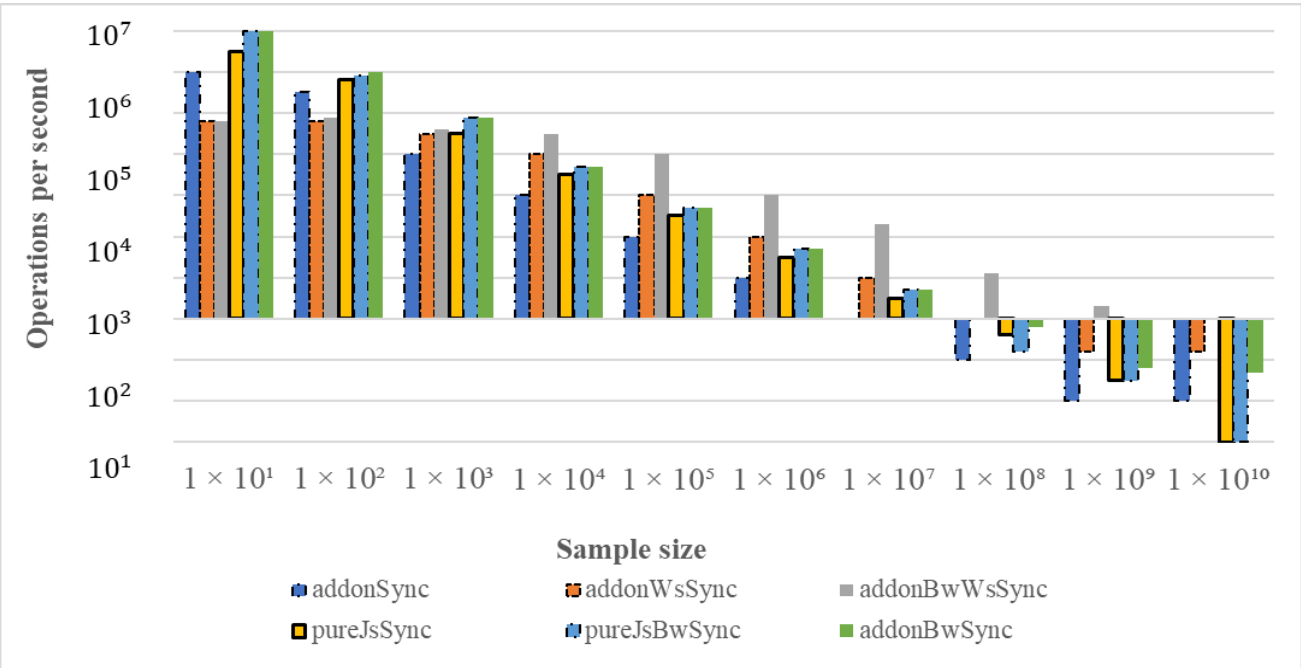
These findings were used to adjust the Node.js thread pool, in particular, to adjust the `UV_THREADPOOL_SIZE` environment variable, which determines the number of threads available to a Node.js process. By default, this value is set to four threads, which led to underutilization on a 24-core system until it was adjusted manually. However, Rust's work hijacking model implemented through `rayon` crate was independent of this limitation, allowing it to dynamically scale across multiple threads without additional configuration (figure 2).

The memory usage measurements summarized in figure 2 show minimal variation between the different modules, with usage ranging from 62 to 68 MB. This consistency indicates that the tested algorithm is more dependent on CPU resources than memory. Although it was expected that modules that use external libraries would initially allocate slightly more memory due to their load dependencies, the overall impact on peak memory usage remained negligible. This indicates that the introduction of Rust or other third-party code into the Node.js environment does not lead to a significant increase in memory, except for the minimal increase associated with the connection of auxiliary modules.

The comparative analysis of synchronous and asynchronous implementations visualized in figure 3 shows that Rust-based modules significantly outperform their JavaScript counterparts, especially under heavy load. Synchronous implementations such as `pureJSSync` worked well before  $1 \times 10^9$  samples, but showed a sharp decrease in the number of operations per second during samples. Conversely, `addonSync` initially lagged behind but outperformed `pureJSSync` with more samples, indicating that the initial advantage of JavaScript JIT optimizations diminishes as the workload grows.



**Figure 2.** Average maximum memory allocation peak in Node.js  
**Source:** developed by the author based on Kyriakos-loannis & Nikolaos (2022)



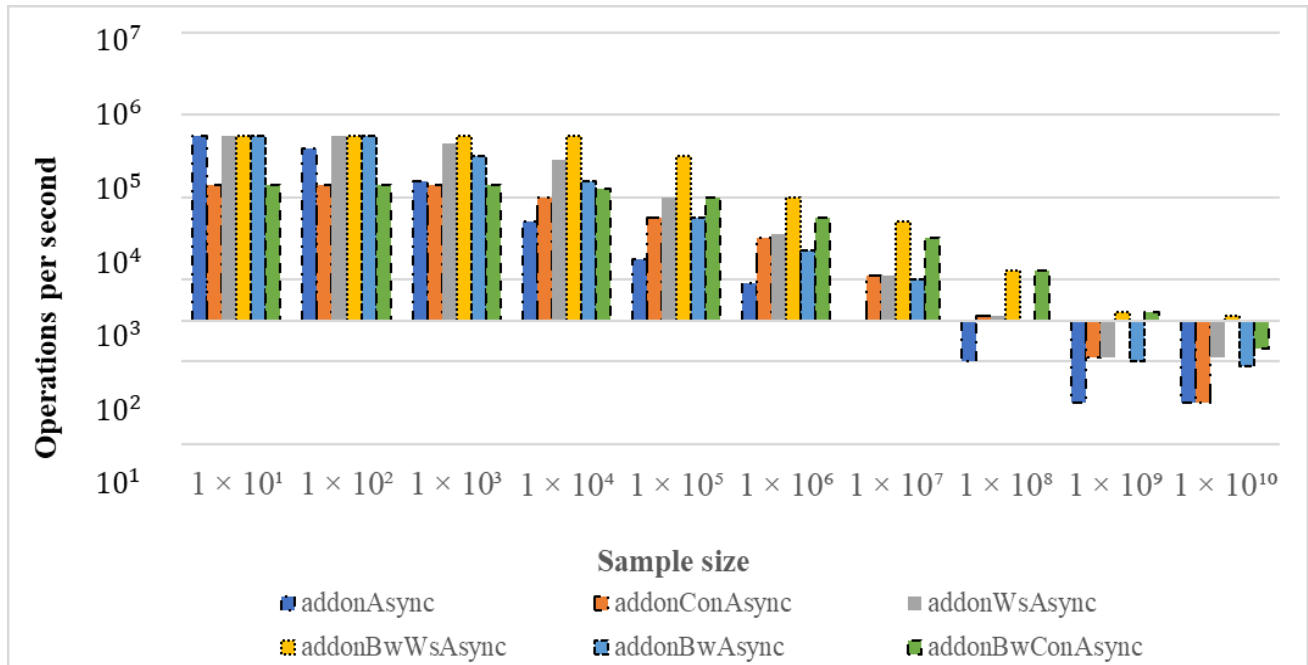
**Figure 3.** Average number of operations per second of synchronous implementations in Node.js  
**Source:** developed by the author based on Kyriakos-loannis & Nikolaos (2022)

Synchronous implementations such as pureJSSync performed well before samples, but showed a sharp decrease in the number of operations per second with  $1 \times 10^{10}$  samples. Conversely, addonSync initially lagged behind but outperformed pureJSSync with more samples, indicating that the initial advantage of JavaScript JIT optimizations diminishes as the workload grows. Including the bitwise optimized versions showed that JavaScript and Rust react differently to such code optimization. While the pureJSBwSync implementation showed a 1,45-2,4x performance increase over pureJSSync, the BwSync addon for Rust was able to outperform the optimized JavaScript version by up to 6x per sample. This suggests that Rust's low-level optimization capabilities can lead to significant performance gains over JavaScript, especially for computing heavyweight tasks.

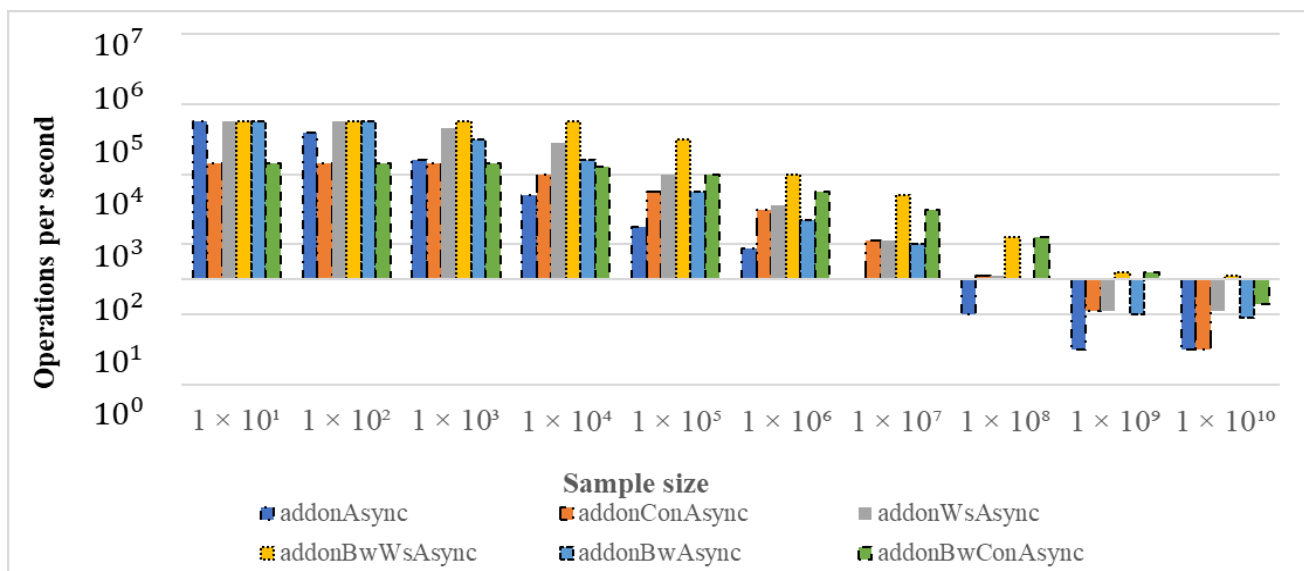
The Rust implementations of asynchronous interception of work (addonWsAsync and addonBwWsAsync)



showed even more significant results. These modules outperformed Node.js thread pool-based approaches by 1,6-15,1 times, which indicates their effectiveness in handling parallel loads. Importantly, these implementations did not require detailed manual tuning to achieve high levels of parallel performance, unlike the default thread pool mechanisms in Node.js. The evaluation also extended to WebAssembly implementations, comparing single-threaded and multi-threaded performance in browsers such as Chromium and Firefox. The results showed that Rust-based WebAssembly implementations achieved almost four times the performance of similar JavaScript modules (figures 4 and 5).



**Figure 4.** Average number of operations per second in Chromium  
Source: developed by the author based on Kyriakos-Ioannis & Nikolaos (2022).



**Figure 5.** Average number of operations per second in Firefox  
Source: developed by the author based on Kyriakos-Ioannis & Nikolaos (2022).

The single-threaded version of WebAssembly achieved 94,8 % of the performance of its Node.js counterpart in Chromium and 69,4 % in multi-threaded scenarios, demonstrating the potential of WebAssembly for high-performance web applications.

#### Testing: CPU usage, memory allocation and productivity.

For testing purposes, several modules have been created to explore synchronous and asynchronous

approaches. Asynchronous modules run in separate threads using an event loop that prevents the main thread from blocking. In contrast, synchronous modules can cause the main thread to block, resulting in a decrease in performance. Module names include the suffix “Sync” or “Async” to distinguish between synchronous and asynchronous operations. The pureJSSync module is a fully synchronous JavaScript implementation. The optimized version of pureJsBwSync uses bitwise operations for better performance. The ffiSync module is built using the ffi-napi library, which allows you to synchronously call C libraries in Node.js without an additional bridge.

Two main modules were created: `addonSync` (synchronous) and `addonAsync` (asynchronous). The `addonAsync` module runs in a separate thread to avoid blocking single-threaded JavaScript execution. The more advanced version of `addonConAsync` splits tasks into blocks that can be processed in parallel using a Node.js thread pool with 24 blocks assigned based on the available cores on the test system. There were also created bitwise optimized versions (`addonBwSync`, `addonBwAsync`, and `addonBwConAsync`).

To make full use of Rust’s parallelism, we used `rayon` crate, which implements a work interception algorithm. Unlike the Node.js thread pool, `rayon` efficiently manages multi-threaded computing and allows threads to “intercept” the task dynamically. This makes it easy to switch from the Node.js thread pool model to Rust parallel execution without modifying the JavaScript code. This approach made it possible to create both synchronous (`addonWsSync`) and asynchronous (`addonWsAsync`) modules, the optimized versions of which (`addonBwWsSync` and `addonBwWsAsync`) use bitwise operations.

For web applications, the `addonBwSync` and `addonBwWsSync` modules were cross-compiled with WebAssembly, implementing both single-threaded and multi-threaded execution. These modules used the Web Workers API and the `SharedArrayBuffer` API to provide parallelism without blocking the main thread.

The Linux `perf` tool was used to test the single-core modules’ use of a single core, while the parallel modules efficiently used additional cores. The CPU utilization of each module was measured 10 times for each degree from 1 to 10 and calculated as:

$$\{cpu\_usage\} = \left\{ \sum_{n=1}^{10} \frac{perf(pi\_est(10^n))}{10} \right\}, \quad (1)$$

where *perf* is a tool used to measure the performance of functions;

*pi\_est*(10<sup>n</sup>) is the function for estimating a number using the Monte Carlo method.

The time tool recorded the minimum, maximum, and average memory usage in kilobytes for each module.

The benchmark.js framework measured performance in operations per second for each module in both Node.js and web browsers. It is calculated as follows:

$$\left\{ \frac{operations}{second} \right\} = \{bench(pi\_est(10^n))\}, \quad (2)$$

where *bench* is a tool for measuring the number of operations that can be performed per unit of time, which gives an idea of the speed of execution of a particular code.

The tests were performed on an Asus Z10PA-D8 motherboard with two Intel Xeon E5-2620 v3 processors (2.4 GHz, 24 cores with Hyper-Threading) and 64 GB of RAM. The system was running Arch GNU/Linux (kernel 6.9.3) with Node.js 20.3.0, GCC 14.1.0, and Rust 1.71.0 on a nightly basis. WebAssembly was tested in Mozilla Firefox 115.0 and Chromium 115.0.5790.98

## DISCUSSION

Integration of Rust with Node.js has become an effective solution to the problems associated with JavaScript single-threading. Rust’s concept focuses on providing memory safety and concurrency without using a garbage collector, making it an ideal choice for extending the functionality of Node.js (Popescu et al., 2021). This is useful for managing CPU-intensive tasks where JavaScript often encounters performance bottlenecks.

Rust’s compatibility with WebAssembly (Wasm) has further expanded its use by allowing Rust code to run in browsers simultaneously with JavaScript. This combination provides performance close to native. This significantly improves the performance of compute-intensive applications such as real-time data processing and Internet of Things (IoT) solutions (Dahiya & Dharani, 2023; Pratama & Raharja, 2023). Using Wasm, Rust modules can be easily integrated into Node.js environments, offering significantly faster execution than pure JavaScript implementations (Ray, 2023).

The complementary strengths of Rust and JavaScript have been revealed in studies comparing Rust’s parallelism model with Node.js’ event loop. Rust’s ability to implement work hijacking algorithms allows for more efficient parallel processing, overcoming typical bottlenecks in JavaScript’s event-driven architecture

(Tushar & Mohan, 2022). This makes Rust particularly suited to offloading complex computational tasks from Node.js, allowing Node.js to retain its non-blocking I/O benefits while benefiting from Rust's computational efficiency.

Although Rust's memory management system based on the concepts of ownership and borrowing requires some learning curve, it offers significant benefits. It significantly minimizes the risks of memory leaks and data races, making it ideal for applications that require high reliability (Hoffman, 2019; Popescu et al., 2021). Unlike C/C++ bindings, which require careful management of pointers and memory, Rust provides compile-time guarantees that simplify integration while maintaining high performance (Goyal, 2023).

Performance benchmarks consistently demonstrate the benefits of using Rust for compute-intensive Node.js applications. Studies show that Rust can achieve speeds up to 10 times faster than equivalent JavaScript implementations in CPU-intensive scenarios, especially when both languages are optimized for high performance (Ardito et al., 2021). These results support the use of Rust for backend services where minimizing latency is critical, such as in database systems and cryptographic processing (Dahiya & Dharani, 2023).

Despite its benefits, integrating Rust into Node.js workflows comes with some challenges. Developers need to familiarize themselves with Rust's unique programming paradigms, and setting up native bindings or compiling Rust to Wasm can complicate the development process (Ray, 2023). However, the long-term benefits of stability and scalability often outweigh these initial difficulties, especially in projects where performance is a critical factor (Hoffman, 2019; Kyriakos-Ioannis & Nikolaos, 2022).

Combining the computational power of Rust with the flexibility of JavaScript allows developers to create versatile applications that can handle a variety of workloads. This hybrid approach allows for rapid prototyping with JavaScript while using Rust for module performance, striking a balance between development speed and computational efficiency (Tushar & Mohan, 2022).

## CONCLUSIONS

This study examined the process of developing Node.js applications, from initial prototyping to the stages where optimization and performance enhancement become crucial. The potential advantages of using Rust as an alternative to traditional C/C++ solutions to extend the capabilities of Node.js are emphasized. Rust offers a familiar workflow for JavaScript developers with similar module models and structures. This facilitates compatibility between the two languages. In addition, Rust provides increased memory safety and no data races, solving the problems that C/C++ faced in previous integrations with Node.js.

Performance evaluations showed that Rust significantly improves the resource utilization of Node.js, especially in computationally intensive tasks. Rust implementations, both synchronous and asynchronous, performed better than their JavaScript counterparts, showing more than a tenfold improvement in certain scenarios. Rust's intercepting parallelism models also delivered superior performance, handling various computational workloads more efficiently than Node.js' default thread pool.

Cross-compiling Rust with Wasm further proved Rust's advantages in browser-based environments, achieving near-native performance. Wasm modules based on Rust outperformed JavaScript implementations in real-time data processing tasks, demonstrating up to four times the performance gain in Firefox and twice the performance in Chromium.

Despite its strengths, this study revealed some limitations. It did not consider the differences between cold and warm start scenarios, focused on a single algorithm, and did not take into account the cost of compiling Wasm in browsers. Future research could explore these areas, as well as expand the benchmarks to include a wider range of algorithms.

Rust's integration with Node.js provides a balanced approach, combining the rapid prototyping capabilities of JavaScript with the robust performance and security of Rust. This synergy paves the way for building modern web applications that can adapt to different performance requirements without the risks associated with C/C++ code. Rust's potential to improve the efficiency and stability of Node.js applications makes it a valuable asset in the high-performance development landscape.

## REFERENCES

1. Ardito, L., Barbato, L., Coppola, R., & Valsesia, M. (2021). Evaluation of Rust code verbosity, understandability and complexity. *PeerJ Computer Science*, 7, 1-33. <https://doi.org/10.7717/peerj-cs.406>.
2. Dahiya, K. & Dharani A. (2023). Building high-performance Rust applications: A focus on memory efficiency. *Social Science Research Network*, 1-5. <https://doi.org/10.2139/ssrn.4518760>.
3. Goyal, A. (2023). Improving Node.js performance using Rust. Retrieved from <https://blog.logrocket.com/improving-node-js-performing-rust>
4. Hoffman, K. (2019). Programming WebAssembly with Rust: Unified development for web, mobile, and



embedded applications. Stanford: Pragmatic Bookshelf.

5. Kyriakos-Ioannis, K. & Nikolaos, T. (2022). Complementing JavaScript in high-performance Node.js and web applications with Rust and WebAssembly, *Electronics*, 11(19), 3217. <https://doi.org/10.3390/electronics11193217>

6. Popescu, N., Xu, Z., Apostolakis, S., August, D.I., & Levy, A. (2021). Safer at any speed: automatic context-aware safety enhancement for Rust. *Proceedings of the ACM on Programming Languages*, 5, 1-23. <https://doi.org/10.1145/3485480>.

7. Pratama, I.P. A. E. & Raharja, I. M. S. (2023). Node.js performance benchmarking and analysis at Virtualbox, Docker, and Podman environment using node-bench method. *International Journal on Informatics Visualization*, 7(4), 2240-2246. <https://doi.org/10.30630/ijov.7.4.1762>.

8. Ray, P. P. (2023). An overview of WebAssembly for IoT: background, tools, state-of-the-art, challenges, and future directions. *Future Internet*, 15(8), 275. <https://doi.org/10.3390/fi15080275>.

9. Serefaniuk, B. (2024). Understanding rust and its integration with Node.js & front-end applications. Retrieved from <https://medium.com/@bserefaniuk/understanding-rust-and-its-integration-with-node-js-front-end-applications-2da705a0bf1b>

10. Tushar, B. R. & Mohan, M. (2022). Comparative analysis of JavaScript and WebAssembly in the browser environment. *IEEE 10th Region 10 Humanitarian Technology Conference* (pp. 232-237). Hyderabad: IEEE. <https://doi.org/10.1109/r10-htc54060.2022.9929829>.

## FINANCING

The authors did not receive financing for the development of this research”.

## CONFLICT OF INTEREST

The authors declare that there is no conflict of interest.

## AUTHORSHIP CONTRIBUTION

*Conceptualization:* Volodymyr Kozub.

*Data curation:* Volodymyr Kozub.

*Formal analysis:* Volodymyr Kozub.

*Research:* Volodymyr Kozub.

*Methodology:* Volodymyr Kozub.

*Project management:* Volodymyr Kozub.

*Resources:* Volodymyr Kozub.

*Supervision:* Volodymyr Kozub.

*Display:* Volodymyr Kozub.

*Writing - proofreading and editing:* Volodymyr Kozub.