electronics

MDPI

Article

Complementing JavaScript in High-Performance Node.js and Web Applications with Rust and WebAssembly

Kyriakos-Ioannis D. Kyriakou and Nikolaos D. Tselikas *

Communication Networks and Applications Laboratory, Department of Informatics and Telecommunications, University of Peloponnese, 221 00 Tripoli, Greece
* Correspondence: ntsel@uop.gr; Tel.: +30-2710-372216

Abstract: We examine whether the novel systems programming language named Rust can be utilized alongside JavaScript in Node.js and Web-based applications development. The paper describes how JavaScript can be used as a high-level scripting language in combination with Rust in place of C++ in order to realize efficiency and be free of race conditions as well as memory-related software issues. Furthermore, we conducted stress tests in order to evaluate the performance of the proposed architecture in various scenarios. Rust-based implementations were able to outperform JS by 1.15 by over 115 times across the range of measurements and overpower Node.js's concurrency model by 14.5 times or more without the need for fine-tuning. In Web browsers, the single-thread WebAssembly implementation outperformed the respective pure JS implementation by about two to four times. WebAssembly executed inside of Chromium compared to the equivalent Node.js implementations was able to deliver 93.5% the performance of the single-threaded implementation and 67.86% the performance of the multi-threaded implementation, which translates to 1.87 to over 24 times greater performance than the equivalent manually optimized pure JS implementation. Our findings provide substantial evidence that Rust is capable of providing the low-level features needed for non-blocking operations and hardware access while maintaining high-level similarities to JavaScript, aiding productivity.

Keywords: Node.js; JavaScript; Rust; WebAssembly; web software; scalability; software performance evaluation; software convergence; software interoperability

Citation: Kyriakou, K.-I.D.; Tselikas, N.D. Complementing JavaScript in High-Performance Node.js and Web Applications with Rust and WebAssembly. *Electronics* **2022**, *11*, 3217. https://doi.org/10.3390/electronics11193217

Academic Editors: Igor Mekterović, Juan Antonio Caballero-Hernandez, Marko Horvat and Manuel Palomo-Duarte

Received: 6 September 2022
Accepted: 1 October 2022
Published: 7 October 2022

1. Introduction

Node.js [1] is a JavaScript (JS) runtime built around V8 [2], the JS engine used in Chromium, which is the base for Google's Web browser. It has gained massive adoption by developers and organizations around the world because of its ease of development as well as the efficient event-driven and non-blocking input/output (I/O) model. Following JS's rise in popularity, the practicality of end-to-end JS cross-platform applications was studied in our previous work by creating a social application named Proximity [3]. Gains in productivity were observed when targeting multiple platforms from the same code-base, by minimizing the impact of context-switching, while moving from back-end to front-end logic and inversely. Another advantage that can be attributed to the success of JS-based systems is Node.js's package manager, npm [4], which houses the largest distribution of open-source libraries in the world. According to Modulecounts [5], a service monitoring language module repositories, npm is averaging over a thousand new modules/day, which is followed by Maven with about 200 modules/day. It is noteworthy that npm module submissions follow an exponential growth curve clearly outpacing all the other repositories. Publicly available modules provide building blocks for the rapid prototyping of systems with minimal effort through code re-use maximization. The possible benefits of implementing Web services utilizing Node.js have been investigated in our previous work, and evidence has been provided establishing JS as a viable option

for realizing efficient and scalable applications, even more so in conjunction with Web-based client implementations, where end-to-end JS may provide synergistical effects [6]. Such applications can be useful in practice when implementing, e.g., educational [7] and medical [8] interactive tools. Although Node.js applications are most commonly written in pure JS, the underlying interoperability with foreign compiled code is abstracted. According to Node.js' announcement for version 8.0.0 of the platform, 30% of all modules rely indirectly on native modules [9]. The reason is that JS is a highly dynamic, garbage-collected, single-threaded scripting language, lacking the capabilities required to directly access system resources and perform low-level operations, i.e., system-calls, threading, direct bit manipulations, etc. Such functionalities are a requirement for basically every I/O operation, be it logging to the screen, network access via sockets, filesystem access, etc. Hence, C and C++ modules have been directly bound to V8 and accessed through a JS Application Programming Interface (API).

In addition to the native core modules of Node.js, developers can provide their own bindings to C/C++ libraries in order to extend the platform's capabilities and optimize performance. For example, uWebSockets [10] is a popular WebSocket protocol implementation for Node.js that out-scales all known pure JS implementations. Moreover, libxmljs [11] provides bindings to the popular XML parsing C library, libxml, fulfilling the lack of XML support in Node.js. Another example is node-serialport [12] enabling access to serial ports for reading and writing via a JS API.

Although C and C++ have been used extensively as systems programming languages, they do not share much with JS's mental model. Furthermore, discipline and experience are integral requirements for avoiding memory-related faults, data races and other unwanted situations, setting the integrity of Node.js applications relying upon bindings to native libraries at risk. Mozilla, mostly known as the vendor of the open-source Web Browser Firefox, noticed challenges during the maintenance and development of non-trivial code bases in C/C++. They observed that whole classes of logical faults and bugs were caused by misuse of dynamic memory handling, which can be rather hard to detect [13]. For this reason, they were led to examine possible solutions to such cases where software could be driven into vulnerable states and be maliciously manipulated by adversaries. The goal was to maintain the high performance of systems programming languages while simultaneously increase memory-safety guarantees without incorporating a garbage-collector (GC). Furthermore, multiple cores of modern systems had to be taken advantage of safely through parallelism.

The Rust [14] programming language was developed at the Mozilla Research department in order to fulfill the aforementioned requirements. The compiler is based on the LLVM infrastructure [15], which is the toolchain responsible for the emission of efficient machine instructions theoretically for all platforms supported by it. Thereafter, a large group of independent programmers and researchers alike started contributing to Rust's development and studying its capabilities. The feasibility of a UNIX operating system has been studied by Alex Light [16]. The performance of the language has been evaluated in a cross-comparison with C and Go, in high-performance computing environments with positive results [17]. In a recent study, Rust appears to be suitable for performance-critical GC infrastructure when compared to the equivalent C implementation [18]. Furthermore, the integrity of the memory-corruption protection has been researched by Eric Reed [19]. A prototype Web browser, named Servo, has exhibited techniques of heterogenous parallelism in processors and graphics hardware alike, with innovative rendering of hundreds of frames per second [13,20]. Simultaneously, professional interest has driven technology companies such as Dropbox, which re-implemented central parts of their infrastructure based on Rust with multiple benefits [21].

Additionally, the WebAssembly format is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++ and Rust with a compilation target so that they can run on the Web. It is also designed to run alongside JavaScript, allowing both to work together. Currently,

it is implemented in all major Web browsers and is under the process of standardization via the W3C WebAssembly Working Group [22]. In our previous work, we supplied evidence that Web browser vendors appear to be investing heavily in optimizing the cost of WebAssembly function calls, prioritizing its performance even over their respective JS engine's implementations [23]. By acting as a compilation target for a conceptual machine, it features language and platform independence, safe execution and has shown promising performance gains of up to 86 times when replacing JS components with Rust code in real-use parsing scenarios [24,25]. In its present form, not all planned features have been finalized, and garbage collection, SIMD, etc. are in progress. Recent advancements have made the compilation of lower-level languages utilizing threads for parallelism to the WebAssembly binary format possible. Although numerous studies have been conducted to determine the performance benefits of WebAssembly [26–29], to the best of our knowledge, there have been none focusing on multi-threaded execution. Hence, we were motivated to explore how the performance of multi-threaded Rust code executed inside of Node.js would compare to cross-compiled WebAssembly executed inside of Web browsers.

On a different note, Bjarne Stroustrup, the father of C++, when interviewed about the unification of programming languages, has stated that the future of programming will not be a single language and instead, the unification of different paradigms will be achieved as a set of guidelines, leading to the interoperability among those paradigms [30]. Following this line of thought, in this paper, the examination of how Rust can be used to complement JS in Node.js application realization is presented, and the implications derived from such a coupling are discussed. Node.js provides access to the largest open-source module repository, and together with JS's dynamic, weakly typed and event-driven, object-oriented nature, the rapid-prototyping of systems can be achieved. Rust may provide efficient C bindings and low-level capabilities as well as enhanced performance, type safety, memory safety, and data-race freedom in a way that is more suitable for JS developers. We consider the exploration of the combined space that JavaScript, Node.js, Rust and WebAssembly occupy important, because it can potentially lead to innovation by setting the stepping stones for novel paradigms in computing.

In Section 2, the tendencies in Node.js application development are presented, coupled with arising challenges, as well as the ongoing related work. In Section 3, the design and implementation principles of various approaches leading to JS modules enhanced with Rust native code are presented, which set upon a pragmatic point of view. Section 4 describes the experimental environment and the methodology followed, while Section 5 presents the results as well as an elaboration on the findings. Finally, in Section 6, the findings of conducted research are summarized.

2. Current Tendencies and Challenges

2.1. Node.js Architecture

In order to examine how Node.js makes it possible to utilize JS as a scripting language and still perform system-related operations, its underlying architecture should first be analyzed. In Figure 1, an abstraction of the high-level architecture is presented. The higher up a component is depicted in the architecture, the higher the level of abstraction provided.

The `Applications/Modules` space is where all JS project files reside. They may make either direct or indirect use of precompiled foreign code. Additionally required external dependencies, which are re-used in the application's logic, belong in this space as well. The word `module` is used to describe building blocks, usually performing a single task, leading to composable instead of monolithic design patterns. Although JS has good parts, Node.js has to rely upon compiled code to perform I/O operations. The runtime's GC abstracts the potentially error-prone manual dynamic memory management, but there is no thread-safety mechanism present when performing I/O, making memory-related faults and race conditions possible [31]. Furthermore, failure points may be present in the underlying foreign compiled code, propagating to the higher levels and leading to unexpected behaviors and faults. Node.js utilizes C/C++ libraries internally in order to provide

access to the operating system resources. Such libraries provide efficient solutions to all I/O-related operations included as core functionalities. The `libuv` project [32] is such an example, providing the event-loop and asynchronous I/O access. Other depicted examples of libraries used internally by Node.js are `c-ares`, the C library for asynchronous DNS requests, as well as `zlib`, which provides compression functionality implemented using `Gzip` and `Deflate/Inflate`. Moreover, the `crypto` module includes a set of wrappers for `OpenSSL` methods, e.g., `hash`, `HMAC`, `cipher`, `decipher`, `sign`, `verify`, etc. The `C/C++ Bindings` space is where the functionality of such libraries is exposed via the core JS API. Some examples in this space are the `os` module for operating system-related utility methods, the `fs` module for file-related I/O, the net module for stream-based TCP servers and clients, the `http` module providing support for the `HTTP` protocol, etc. `Addons` [33] is a definition used by Node.js to refer to libraries and their corresponding bindings, which are not included in the core modules. They are usually written in C/C++ in order to extend Node.js' functionality or provide performance gains when a JS implementation is found to be lacking. The main focus in this paper will be on `Addons`; thus, the possible approaches to implement such modules are to be thoroughly examined in Section 3.2.

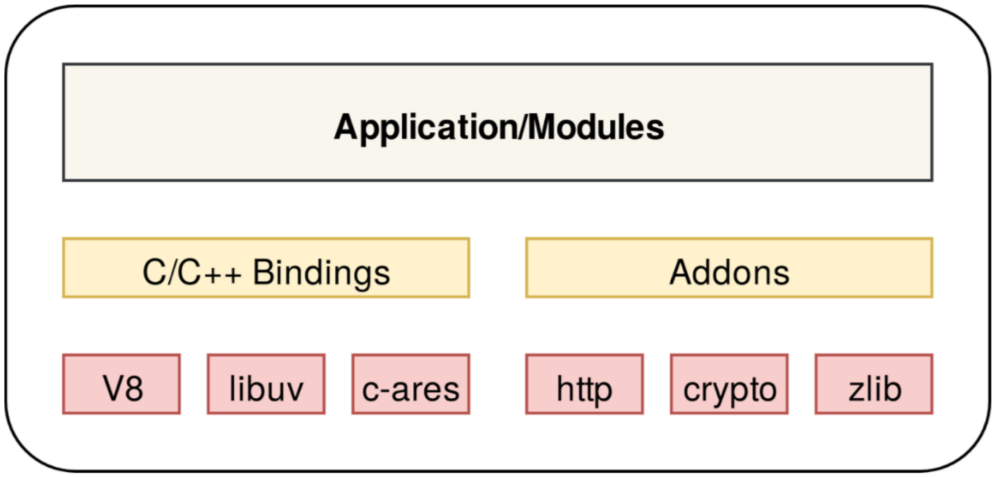

Figure 1. Abstracted Node.js application architecture.

2.2. C/C++ Inefficiencies in Node.js Addon Development and Rust Promises

Node.js relies heavily on C/C++ code, and additional native modules are commonly written in those languages. Hence, errors and vulnerabilities may bubble up to a presumed memory safe and correct JS application. The inefficiencies present in C/C++, capable of affecting `Addon` development, are described below. Numerical primitives are inconsistent, i.e., JS representations implement the IEEE 754 double precision point format [34]. Hence, porting arithmetic operations using C/C++'s `double` primitive may lead to different behavior than expected. In addition, JS has no embedded notion of any kind of integers, while C/C++ implement integer types of various bit-sizes (e.g., `char`, `short`, `int`, etc.). A lack of C/C++ programming experience may cause integer overflow/underflow, leading to undefined behavior. Type safety is not guaranteed by C/C++, and although programs may not exhibit type errors, undefined behaviors are incorporated in the standard specification leading compilers to not just produce unspecified results, but—instead—allow the program to do practically anything. A simple example of iterator invalidation leading to undefined behavior is demonstrated in the following listing.

```
std::vector v;
v.push_back(MyObject);
for (auto x : v) {
  v.clear();
  x->whatever(); // results in undefined behavior
}
```

If the contents of a container that is being iterated over are destroyed, the program is led into undefined behavior. This is an example of a perfectly valid code from a C++ compiler's perspective, which is capable of halting the Node.js process using it. Such cases of undefined behavior have already been investigated [35]. The errors in the manual management of the dynamically allocated memory can cause a Node.js process to crash as well. Memory safety is set at risk by `null pointer` dereferences (`NULL` in C and `nullptr` in C++) that cause programs to crash, dangling pointers allowing access to heap allocated resources that have not lived as long as they had to and buffer overruns allowing the program to access elements before the start or beyond the end of an array [36].

As Node.js is commonly used to implement services over the Internet, security can be of critical importance. In a case study of Heartbleed, the security bug in the popular OpenSSL cryptography library used internally by Node.js, Jens Getreu has determined that the Heartbleed bug would not have been possible if OpenSSL had been implemented in Rust [37].

Since the adoption of high-level languages for systems programming over the course of the past 50 years, two challenges remain largely unsolved: firstly, writing secure code. Since 1988, when the first analyzed Internet virus propagated by taking advantage of a buffer overflow bug [38] until the present day, malicious software has been taking advantage of the way C and C++ programs handle memory, and exploiting bugs in the code, to breach security layers. Secondly, writing multi-threaded code is another issue. The way to leverage the multi-core CPUs of modern systems can only be accomplished via threading. With even midrange mobile device having multiple cores, parallelizing operations is important for performance and other requirements, e.g., battery life economy. Concurrency introduces new classes of bugs, while at the same time, the reproduction of ordinary bugs is made even harder. Finally, C++ does not have a module system. Hence, Node.js developers must provide source files and make files manually, or incorporate complex build pipelines, which diverges from the common module-based, composable way of managing dependencies. Further details on the aforementioned concepts are also described by Jim Blandy in "Why Rust? Trustworthy, Concurrent Systems Programming" [39]. Rust was created in order to address these kinds of challenges. It follows the C++ philosophy of zero-cost abstractions [40] and takes a step further by incorporating memory safety and data-race free concurrency without the need for a GC. That is accomplished by statically tracking the ownership and lifetimes of all variables and their references. The ownership system enables Rust to automatically de-allocate and run destructors on all values immediately when they go out of scope and prevents values from being accessed after they are destroyed. Rust applies some established techniques from academia, e.g., `enums` as algebraic data types, common in the ML family of languages and `traits`, which enable polymorphism similar to Haskell's type classes. Just like JS, both procedural and functional paradigms are used, as examined by Poss [41], making Rust a good candidate for complementing Node.js' `Addon` development.

2.3. Related Work

An open source project named Neon provides a safe Rust abstraction layer for native Node.js modules [42]. It does so by protecting all handles to the JS heap, even when they are allocated on the Rust stack, ensuring that objects are always safely tracked by the GC. Furthermore, the project features a command-line tool that simplifies the creation and compilation of native modules. Although it is able to provide the memory safety mechanisms and the added performance derived from Rust, it is currently not stable enough as its API undergoes changes. Furthermore, Neon is not capable of fully replacing the current Node.js native module creation workflow, which will be examined in the next section.

3. Application Architecture and Implementation

3.1. Pure JavaScript Node.js Application Architecture

A complete analysis of the possibilities in architecting Node.js applications is well outside the scope of this study. For that reason, the creation of a relatively simple application will be demonstrated, following common practices in order to provide the basis for discussion in the next section. A CPU-bound and computation intensive task had to be chosen in order to provide customizable load on the CPUs. It is assumed that the assigned task is to prototype a pi number estimator using the Monte Carlo method. For this task, a pseudo-random number generator following a uniform distribution is needed, and since the goal is to determine its performance characteristics, seeding must be incorporated for reproducible deterministic outputs. The implementation steps are described in the following paragraphs. In pure JavaScript Node.js applications, the project workspace consists of a single directory in the file system. Figure 2 serves as visual reference to the examined workflow.

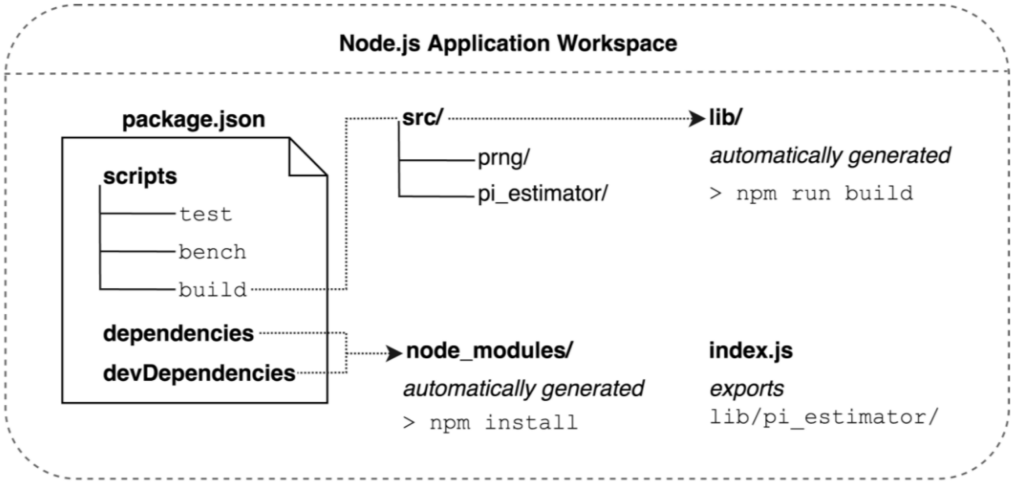

Figure 2. Pure JavaScript Node.js application architecture.

New projects are initialized by invoking `npm` with the `init` command, which requests for some input, i.e., a package name, the initial version number, a description, the entry point of the application, a test command, a git repository, keywords, authors and the license, interactively. This is how package.json, a standard JSON file, is created in the project's root directory [43]. This file contains metadata relevant to the project, handles dependencies and may hold scriptable commands to automate aspects of the development process, such as testing invocation, benchmarking, linting, documenting, publishing, etc.

Node.js utilizes a modules' specification known as CommonJS modules [44]. Application logic is broken down into discrete chunks of logic and is recombined to produce the desired behavior. Although JavaScript is not an ahead-of-time compiled language, compile-to-JavaScript languages, e.g., CoffeeScript [45] and TypeScript [46], are commonly employed, or different builds may be produced for debug, release, and other builds, hence the practice of keeping all code in a separate folder, usually named `src/`. Furthermore, since the release of the ES6 standard, newer JS features are added on an annual time-frame. Hence, developers may be agnostic toward the Node.js runtime on the targeted infrastructure, and JS-to-JS trans-compilers are employed to resolve this issue, e.g., Babel [47]. In our previous work, we examined such implications and proposed a solution based around the distribution of hygienic macros to address this challenge [48].

In `package.json`'s `script` field, a `build` command to produce the final module in the `lib/` directory is usually specified.

At this point, the first module can be written, the seedable Pseudo-Random Number Generator (PRNG). A modified version of the Park–Miller algorithm is chosen because of its simplicity [49]. By convention, modules can be referenced by path whenever an `index.js`

file is present inside the directory. A unit test should be written to determine whether the JavaScript implementations aligns with the assigned requirements. A commonly used practice to differentiate between different builds is to use the `NODE_ENV` environment variable. Node.js includes an assertion module which is chosen to be resolved only when we request a test invocation of the library by checking whether the `NODE_ENV` variable is set to `test`. Test cases are created for the seedable reproducibility and uniformity of results.

Additionally, a second implementation of the PRNG module was implemented, where the mathematical operations have been converted to more optimized bitwise operations.

In order to evaluate the performance, a benchmarking framework is included, `benchmark.js` [50], which will be resolved and run only when the `NODE_ENV` variable is set to `bench`. By issuing the command `npm install --save-dev benchmark`, the dependencies are installed and added as development dependencies in the `package.json` file. Then, benchmarks for integer and floating point numbers generation are created. Following the same workflow, the pi estimation module is written in the `pi_estimator/` directory, requiring the aforecreated `prng` module. Finally, a test is written in the same manner for checking the accuracy of the computed pi value and a benchmark.

3.2. Node.js Addon Architecture and Compilation

It is common to utilize the Node.js building tool, based on Google's Meta-Build system, `GYP` [51], for building `Addons`. The `node-gyp` [52] module, which bundles the `GYP` project used by the Chromium team and takes away the pain of dealing with the various differences in build platforms and Node.js versions, may be added as a dependency. Because developers are agnostic toward the targeted V8 engine, native bindings should use some form of abstraction mechanism in order to normalize the underlying API. The most popular choice is `NAN` [53], standing for Native Abstractions for Node.js. It consists of a C++ header file with helpful macros and utilities. Furthermore, a helpful tool to remove debug/release paths confusion is `bindings` [54]. Figure 3 illustrates the Addon architecture discussed in this section.

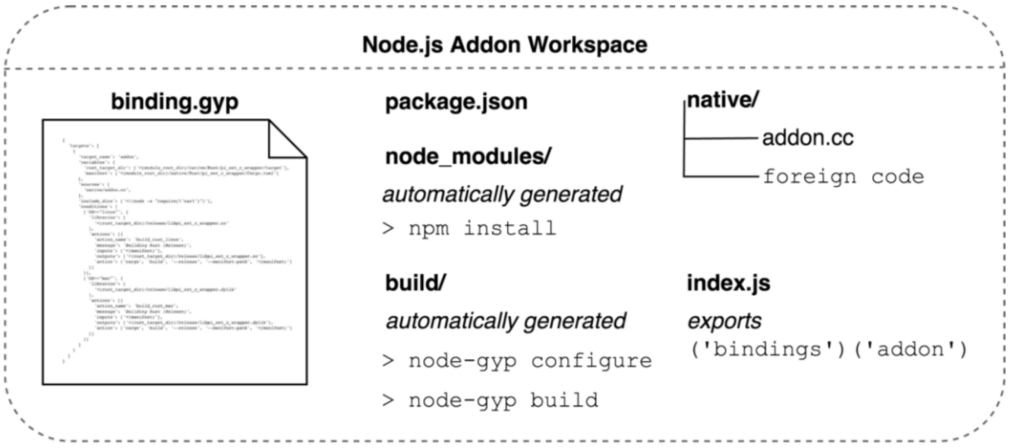

Figure 3. Node.js addon architecture.

The dependencies are included in the project with the command `npm install --save node-gyp nan bindings`. A `gypfile` is responsible for orchestrating the build process. It is by convention named `binding.gyp` and describes the configuration to build the native module in a JSON-like format. This file is placed in the root of the package alongside the `package.json` file. In order to keep native code separated, the native directory is created containing a C++ source file, `addonc.cc`, which will serve as the top-level module to export C++ constructs to JS. It is added to `binding.gyp`, the file instrumenting the build process, as a source file. By depending upon the `V8` and `NAN` header files, interoperability is achieved; hence, pre-existing or new C++ valid code may be included and called directly. Once the code is written, it is automatically built in the `build/` directory by issuing the

`node-gyp configure`, which is followed by `node-gyp build` commands. An `index.js` file is responsible for exporting the modules defined in `addon.cc` for re-use.

3.3. Porting a Sample Application from JavaScript to Rust

Rust, although a systems programming language, shares commonalities with the Node.js workflow and project structure. The equivalent workspace is a directory as well, and the term used to describe a module is `crate`. First, a `crate` is created in the `native` directory by invoking `cargo`, the equivalent of Node.js' `npm` tool, which is capable of declaring dependencies and ensuring the repeatability of builds. When the command `cargo new pi_estimator` is issued, a new directory named `pi_estimator` is produced, including a default structure similarly to how `npm init` works. Figure 4 holds the reference `crate` structure discussed in this section.

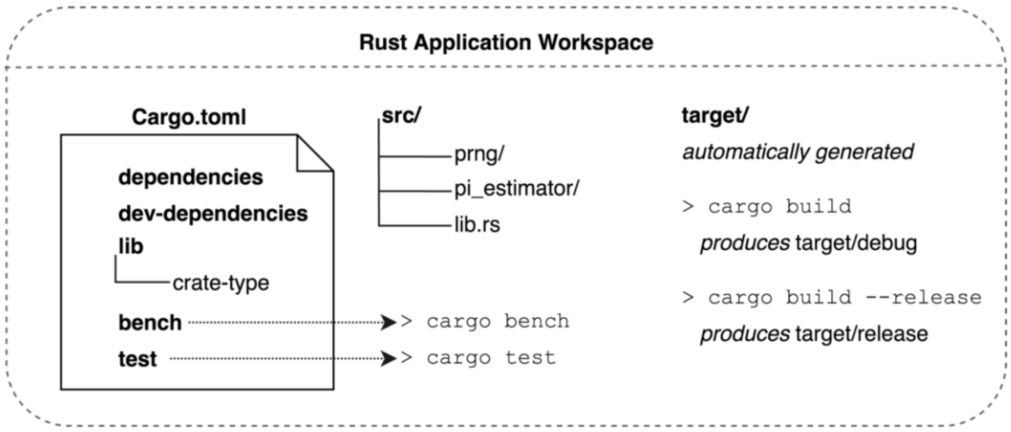

Figure 4. Rust crate architecture.

The equivalent of `package.json` is generated, named `Cargo.toml`, and all code resides in the `src/` directory by convention. Rust has the notion of modules, and the `PRNG` module can be re-created in the `src/prng/` directory, where the equivalent of `index.js` is `mod.rs`. It can be used from the entry point, `lib.rs`, similarly to the way Node.js modules are required in `index.js`.

The `PRNG` module can be ported from JS by translating each line of code to Rust syntax. In more complex applications, the logic may need to be revised, but thanks to Rust's zero-cost abstractions and syntactic similarities with JS, the procedure should not bear burdens. Rust's syntax may be familiar to JS developers because of the C syntax similarities and the first-class support for functional programming style, e.g., `closures`, series of element processing via `map`, `filter`, `fold`, etc. Although not as efficient as possible, numerical operations can be copied directly, because JS's number representation follows the same standard as Rust's double precision float, the IEEE 754 [34]. Optimizations can be performed at a later point by utilizing lower-level code, which JS may be incapable of producing. Furthermore, Rust does not utilize the classical inheritance of C++; instead, methods can be implemented for structures similarly to JS's prorotypal chain and factory functions. Tests may be ported as well, because assertions as seen in the JS module are present in Rust as well. The conditional execution of tests is performed by issuing the `cargo test` command, which acts in a similar fashion as the `npm test` command. A benchmarking library, i.e., `Bencher`, is included in Rust, and benchmarks may be ported over as well.

Having completed the porting of the PRNG module, the pi estimator function is ported as well, including tests and benchmarks. Direct performance comparisons can be made at this point to conclude whether converting from the JS implementation to the Rust version can offer performance benefits or not.

If the benchmarking results show promise, all that is left is to create bindings and connect them to Node.js. Because the `Addons` are written in C++ and have no notion of calling Rust code directly, a C compatible wrapper may be produced. This procedure is

possible, because Rust can pretend to be C by using C types and exposing C-compatible symbol definitions, just like a C++ library would have to expose a C-compatible API. This procedure is performed effortlessly, as the aforecreated `crate` can be re-used with the new function definition wrapping the pi estimator directly, and a header file defining the prototype of the function can be written. By adding the `cdylib` option to `Cargo.toml`, the library will be compiled as a C-compatible shared object. It must be noted that no extra configurations or Makefiles are required for a development build or an optimized production build. The commands `cargo build` and `cargo build - release` produce debug and production objects, respectively.

At this point, the Node.js Addon described in the previous section can be ordered to link against the C/Rust wrapper and expose the functionality as a JS function. The `binding.gyp` file may be updated to build the Rust object automatically whenever `node-gyp` is invoked.

4. Scenarios and Testbed

Following the aforementioned principles in Section 3, separate modules were created for various approaches to synchronous as well as asynchronous modules. Modules that utilize the event-loop are referenced to as **asynchronous** because they run on a separate thread without blocking the main thread, while modules that do not are referenced as **synchronous** or **blocking**. The basic implication of synchronous modules is the potential lock-up of the main thread, exhibiting diminished performance. Henceforth, the naming of the scenarios examined is made to contain the `Sync` or `Async` suffix for differentiating among synchronous and asynchronous operations, respectively. The `pureJSSync` module is the pure JS implementation as discussed in Section 3.1. All pure JS functions are synchronous; hence, no asynchronous implementation was derived. Additionally, the `pureJsBwSync` optimized version utilizing bitwise operations was created. An external library that automates the invocation of foreign functions is `ffi-napi`. It is the simplest method of using a C library for synchronous module invocation. Thus, the `ffiSync` module is created, requiring no additional work to be bridged to Node.js. By implementing separate bindings manually, the following modules are created.

The `addonSync` and `addonAsync` modules are the standard synchronous and asynchronous `Addons`, as discussed in Sections 3.2 and 3.3, respectively. The `addonAsync` module runs on a separate thread and thus does not block the single thread JS runs on. The `addonConAsync` module utilizes the `piEstRustAsync` module and splits the work into chunks to be independently handled by Node.js's thread-pool, potentially in parallel. The number of chunks assigned is 24 based on the available processing cores of the testbed. The corresponding optimized `addonBwSync`, `addonBwAsync` and `addonBwConAsync` versions utilizing bitwise operations were also created.

In order to leverage Rust's parallel data-race freedom, the `rayon` crate may be utilized. It implements a work-stealing algorithm [55] inspired by C's Cilk [56]. Unlike Node.js' thread pool, work stealing solves the problem of executing a dynamically multi-threaded computation, one that can "spawn" new threads of execution, on a statically multi-threaded computer with a fixed number of processors or cores. It does so efficiently both in terms of execution time, memory usage, and inter-processor communication. Hence, the concurrency strategy can easily be swifted from Node.js' thread-pool model on demand, without alteration to the JS source code, or Node.js' internals. Such modules produced are the blocking `addonWsSync` and the asynchronous `addonWsAsync`, potentially parallel modules by splitting the workload into chunks. Finally, the `addonBwWsSync` and `addonBwWsAsync` versions utilizing bitwise operations were also created. The source code is publicly accessible in a GitHub repository [57].

In order to evaluate the performance in the context of Web applications, the `addonBwSync` and `addonBwWsSync` implementations were cross-compiled to WebAssembly, implementing single and multi-threaded execution, respectively. In order to not block the main

thread, these versions rely on the Web Workers API as well as the SharedArrayBuffer API to achieve parallelism.

The first test regards the CPU utilization in order to prove that modules expected to utilize a single core do so, and modules with possible parallelism utilize additional cores. This approach helped detect misconfigurations early, leading to non-representative results. For this metric, the `perf` Linux tool was used. Each module was measured ten times for each power of ten from one to ten, according to the formula:

$$\{cpu\_utilization\} = \{\sum_{n=1}^{10} \frac{perf(pi\_est(10^n))}{10} \forall n \in [1, 10]\}$$

Similarly, the memory allocations were measured for each module. The `time` Linux tool was used to record the minimum, maximum and average kilobytes allocated per module, based on the following formula:

$$\{min\_mem, max\_mem, avg\_mem\} = \{\sum_{n=1}^{10} \frac{time(pi\_est(10^n))}{10} \forall n \in [1, 10]\}$$

The performance of the resulted modules was measured via the `benchmark.js` framework in the form of operations per second metric [50], in both Node.js and Web browsers. Each module was tested with inputs as powers of ten from one to ten, according to the following formula:

$$\{\frac{operations}{second}\} = \{bench(pi\_est(10^n)) \forall n \in [1, 10]\}$$

All the aforementioned scenarios were executed on the same testbed, which was based on the Asus Z10PA-D8 dual socket motherboard, dual Intel Xeon E5-2620 v3 CPUs clocked at 2.4 GHz with a max turbo frequency of 3.2 GHz, providing 24 total cores via Hyper-Threading and 4 Hynix 16GB 2133 MHz DDR4 ECC RAM modules for a total of 64 GB of available memory. The operating system was Arch GNU/Linux, running the 5.19.5 kernel, Node.js version 18.8.0, `gcc` version 12.2.0, `rustc` 1.65.0-nightly, `perf` version 5.19 and `GNU time` version 1.9. The open source Web browsers that were used to evaluate WebAssembly were Mozilla Firefox 104.0 and Chromium 104.0.5112.101.

5. Performance Evaluation: Results and Discussion

The results of the performance evaluation conducted are presented and discussed in this section. CPU utilization, maximum memory allocation and computation performance were the metrics evaluated, as presented in Section 4.

Figure 5 depicts the CPU utilization. The Y-axis represents the cores utilized, while the X-axis lists the names of individual implementations of the measured algorithm. The sampling rates provided to load the CPU are colored and grouped by separate cases. The modules incapable of any sort of parallelism have all achieved the utilization of about one core as expected, while the modules implementing concurrency strategies have started showing trivial gains at 1×10^5 samples, while at 1×10^6 or greater samples, the gains are more noticeable. In more detail, at the 1×10^6 samples mark, the `addonConAsync` implementation has utilized about three cores, while the work-stealing addons have all utilized about five cores. The gains are more pronounced at the 1×10^7 samples point, where all work-stealing approaches reach utilization of 12 cores, while the concurrent implementations reached utilization of only three cores. At the 1×10^8 samples point, about 22 cores are utilized, and the theoretical maximum limit of 24 cores is approached at 1×10^9 and 1×10^{10} samples showing more than 23.7 utilized cores. This metric has provided the insight required in order to fine tune the thread-pool of Node.js, which is utilizing the `libuv` project [32]. The `UV_thread-pool_SIZE` environment variable is responsible for statically defining the number of threads available to a Node.js process.

Since the default value is predefined to 4, our 24 core testbed showed diminished utilization in the async addon modules until it was fine-tuned. At the same time, the work-stealing addons did not suffer any utilization losses because of the dynamically multi-threaded computation capabilities provided by Rust's `rayon` library, i.e., the work-stealing strategy. This behavior was also apparent in the computational performance measured later in this section.

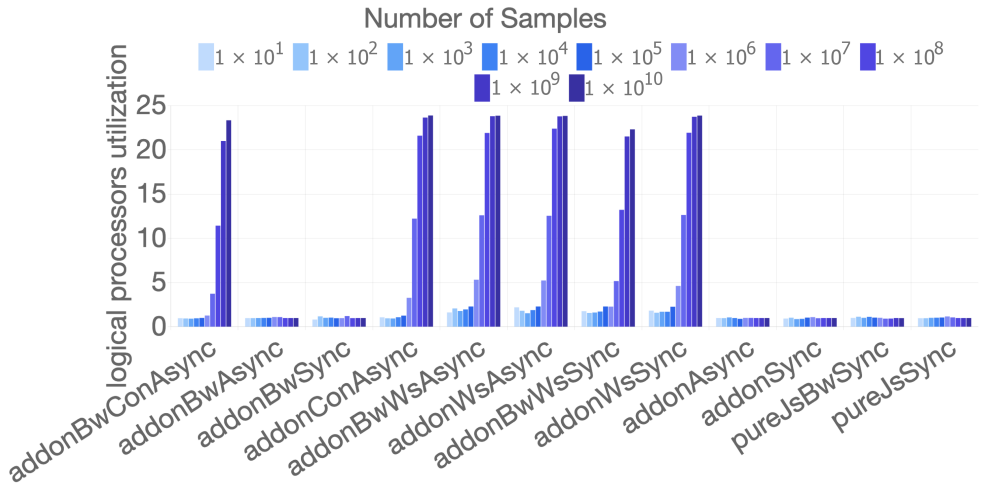

Figure 5. Average CPU utilization in Node.js.

Figure 6 displays the results of the memory allocation measurements per individual implementation. Following the aforementioned CPU utilization figure, the X-axis holds the names of individual implementations of the measured algorithm, and the Y-axis is formatted in MB of peak allocated memory. A minor fluctuation was observed, showing that initial memory allocation is not deterministic. Node.js modules included via the `require` method are read from the filesystem and stored in-memory. Hence, modules utilizing external libraries were expected to allocate more initial memory space. However, all implementations performed similarly, occupying 63 to 67 MB of system memory. This metric can be used as an indication that the algorithm tested is CPU bound, as there is no correlation evident between the increase in the number of samples and the memory allocations. Furthermore, complementing JS with Rust or any other foreign code can be expected to impose an optional overhead, which is attributed to helper modules binding the compiled `Addons`.

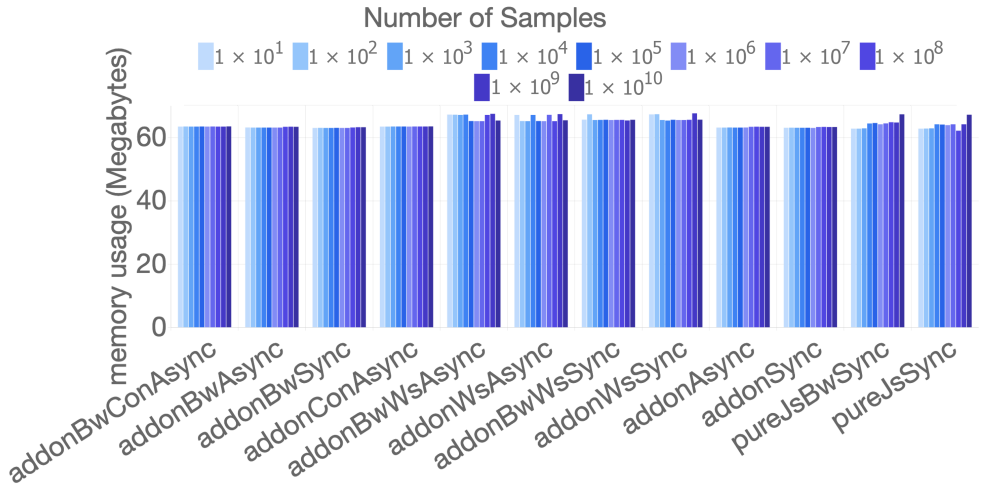

Figure 6. Average peak memory allocation in Node.js.

The computational performance of each approach is presented in Figures 7 and 8, for synchronous and asynchronous implementations, respectively. Both axes are log scaled in order to accommodate for the broad range of of measurements.

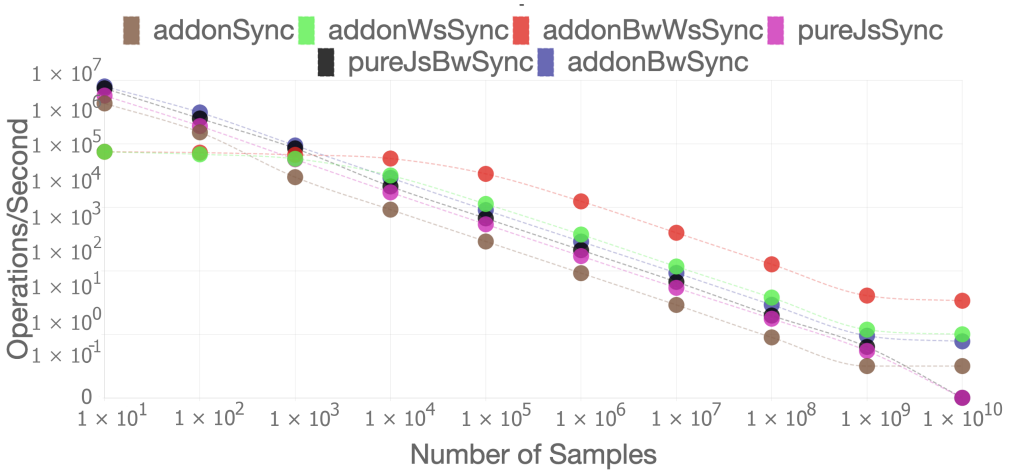

Figure 7. Average operations per second of synchronous implementations in Node.js.

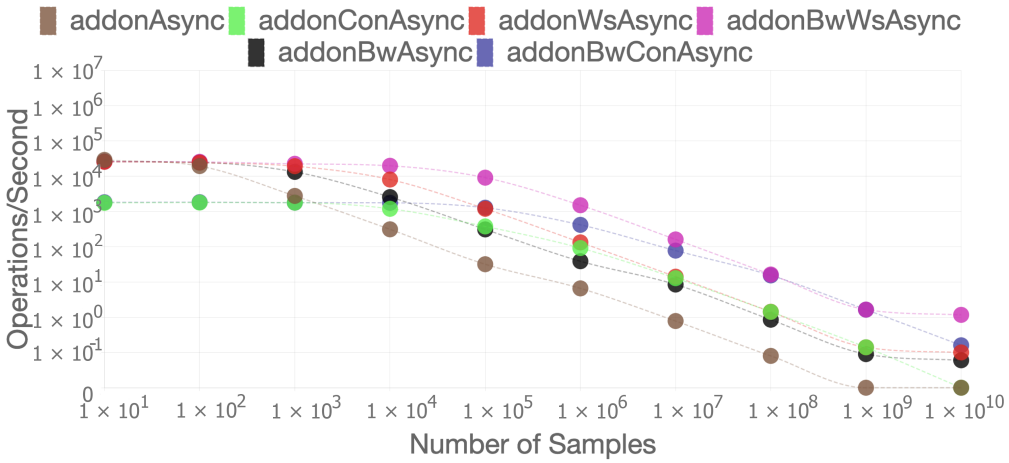

Figure 8. Average operations per second of asynchronous implementations in Node.js.

The `pureJSSync` implementation was found to be performing very well up to the 1×10^9 samples mark; however by the 1×10^{10} samples mark, the achieved operations per second reached zero. On the contrary, the `addonSync` implementation performed worse until the 1×10^{10} mark, where it outperformed the `pureJSSync` implementation. Because no evident overhead is induced when measuring the execution time of the `addonSync` version in relation to its raw performance outside of Node.js, JS's greater performance can be attributed to the Just-In-Time (JIT) compilation optimizations performed by V8 in relation to the unoptimized code translated directly to Rust. Disabling the JIT would de-optimize the JS code, leading to unrealistic comparisons. Hence, this observation can serve as a signal that the Rust ported code could benefit from manual optimizations.

For that reason, but also in order to compare how JS responds to optimizations in comparison to Rust, the bitwise optimized implementations were included. The `pureJSBwSync` implementation performed 1.3 to 2.2 times better than the `pureJSSync` implementation; however, by the 1×10^{10} samples mark, the achieved operations per second exhibited the same behavior by reaching zero. Even so, it was outperformed by the `addonBwSync` implementation by 1.15 to 2.18 times in the $[1 \times 10^1, 1 \times 10^9]$ samples range and over 6 times at the 1×10^{10} samples mark. Thus, Rust reacts much better to the same code optimizations, as it became over 10 times faster than the `addonSync` implementation.

The same pattern was observed in the `addonWsSync` and `addonBwWsSync` versions with bitwise implementation displaying over 10 times the performance of the unoptimized version for the majority of the samples. When compared to the pure JS implementations, both modules were performing worse until the 1×10^3 samples mark, which can be attributed to the overhead from `rayon` setting up its own thread-pool. However, they outperformed all other implementations starting at the 1×10^4 samples mark until the end of the measurements. Specifically, `addonWsSync` displayed 3.4 to over 10 times the performance of `pureJSSync`, and `addonBwWsSync` displayed 7.5 to over 115 times the performance of `pureJSBwSync`. Thus, by comparing the synchronous implementations, the observation is that porting JS code to a native Rust addon does not guarantee better performance; nonetheless, manual optimizations performed in Rust lead to better performance for quickly processed logic, and the ease of safely parallelizing work yields greater gains for more intensive tasks. By combining these observations, the synchronous Rust implementations were able to outperform JS by 1.15 to over 115 times across the range of measurements, producing greater gains as the workload was increased.

By examining Figure 8, the first observation is that utilizing Node.js's thread-pool leads to non-trivial performance degradation for quickly processed logic. All implementations, regardless of the level of parallelism and optimizations, were bottle-necked at 29,000 operations per second. Additionally, the implementations utilizing Node.js's thread-pool exclusively were limited to about 1800 operations per second. Compared to Figure 7, this translates as 0.4% to 15% the performance of the equivalent synchronous implementation in the range of (1×10^1, 1×10^3) samples, as seen in the case of `addonBwSync` and `addonBwAsync`.

In order to take advantage of the Node.js thread-pool, concurrency must be introduced as shown by the `addonConAsync` and `addonBwConAsync` implementations. These implementations display even worse performance until the 1×10^3 and 1×10^4 samples mark compared to `addonAsync` and `addonBwConAsync`, respectively. This is a sign of potential thread starvation, which is closely related to the `sUV_thread-pool_SIZE` environment variable as well as the number of chunks the work is split into. Thus, finding optimal values requires heuristics and is hardware dependent, since they are static values. This is where the work-stealing addon implementations with Rust's `rayon` crate shows promise.

The `addonWsAsync` and `addonBwWsAsync` implementations outperform their equivalent multi-threaded implementations, i.e., `addonConAsync` and `addonBwConAsync`, by 1.4 to 14.5 times and by 2 to over 14 times, respectively, for the majority of the data points. In addition, the performance of both `addonConAsync` and `addonBwConAsync` implementations appears to plummet when reaching the most intensive 1×10^{10} samples mark. By combining these observations, the asynchronous implementations utilizing the work-stealing technique were able to outperform Node.js's concurrency model by 14.5 times or more, producing greater gains for light as well as heavier workloads, without the need for fine-tuning.

In Figures 9 and 10, the computational performance of some implementations running inside the Chromium and Firefox Web browsers, respectively, is presented.

Both browsers have shown similar response, with the `wasm single thread bitwise` implementation outperforming the respective pure JS bitwise implementation by about four times in Firefox and by about two times in Chromium and without showing signs of abrupt slowdowns until the end of the measurements. The `wasm multi thread bitwise` implementation showed reduced performance until it reached the 1×10^4 samples mark when it outperformed all implementations. Relatively to the `wasm single thread bitwise` implementation, in Chromium, it reached 13 times the performance and in Firefox about nine times for the majority of data points. Firefox outperformed Chromium until the 1×10^4 samples mark by about three times, but for the rest of the datapoints starting at 1×10^6, Chromium showed slightly better performance. When Chromium is compared to the Node.js measurements, the potential of WebAssembly code running in the browser becomes apparent. In single-threaded performance, WebAssembly was able to deliver

93.5% the performance of `addonBwSync` at the 1×10^6 mark and 67.86% the performance of `addonBwWsAsync`, which translates to 1.87 to over 24 times greater performance than the equivalent manually optimized pure JS implementation.

Figure 9. Average operations per second in Chromium.

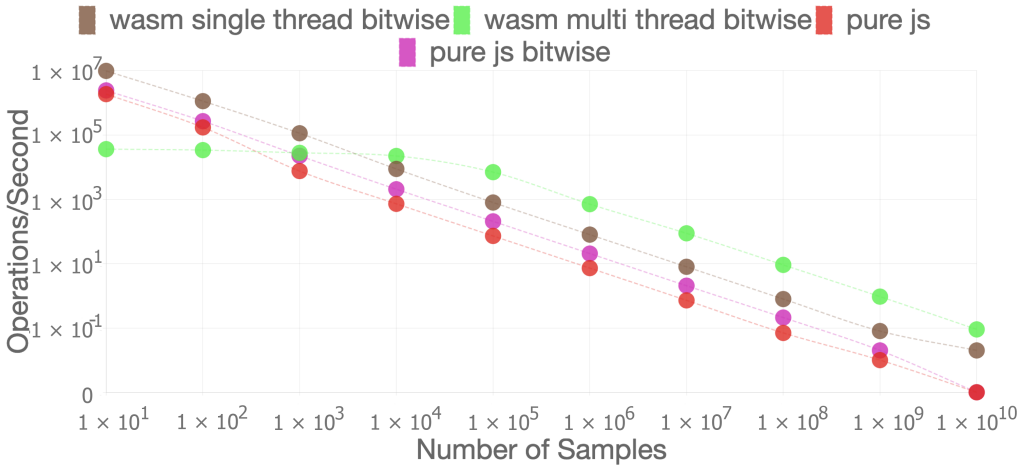

Figure 10. Average operations per second in Firefox.

6. Conclusions

The state of Node.js application development tendencies and practices was presented from the prototyping stage to the stage where optimizations and higher performance become a requirement. Additionally, the possible benefits that can be derived from utilizing the Rust programming language instead of the more canonical C/C++ approach were explored. The similarities in the mental model and workflow of JS and Rust were presented as well as the possibilities for attaining interoperability. This study was concluded with the results of the performance evaluation conducted for each possible approach over a large range of computational loads. In the case of computation-intensive algorithms, Rust was found to be an enhancement to Node.js' utilization of system resources with minimal effort. Furthermore, interoperability is easily achieved, while the produced modules are free of memory-related issues and data-races, which is something that was not possible to be guaranteed in the past by utilizing C/C++ code exclusively. Porting code to Rust was found to be a straightforward procedure, which was mostly because of the class-free structures and first-class functional programming practices available. For a JS developer, Rust can match their expectations more reasonably than C/C++ due to the availability of modules, a single build instruction, embedded tests and benchmarks, conformance to the same floating-point standard and the overall modern similarities, i.e., a similar package manager

and general workflow, etc. All of the benefits that Rust provides over C/C++, i.e., memory safety and data-race freedom, are applicable directly in Node.js Addon development and can eliminate whole classes of bugs, leading to more robust applications.

By performing the same optimizations to the equivalent Rust and JS implementations, Rust was shown to react much better to the optimizations, as it became over 10 times faster, while JS gained just 1.3 to 2.2 times its unoptimized performance. Rust's synchronous implementations were able to outperform JS by 1.15 to over 115 times across the range of measurements, producing greater gains as the workload was increased. Furthermore, the bottlenecks of utilizing Node.js's thread-pool exclusively were explored, potentially leading to thread starvation. Rust's asynchronous implementations utilizing the work-stealing technique were able to outperform Node.js's concurrency model by 14.5 times or more, producing greater gains for light as well as heavier workloads, without the need for fine-tuning. By performing the first-ever performance evaluation of multi-threaded execution of cross compiled Rust code to WebAssembly in browsers and the same implementation executed inside of Node.js, the benefits of utilizing Rust for high-performance Web Application's development were fortified. The single thread implementation outperformed the respective pure JS implementation by about 4 times in Firefox and by about 2 times in Chromium. WebAssembly executed inside of Chromium compared to the equivalent Node.js implementations was able to deliver 93.5% the performance of the single-threaded implementation and 67.86% the performance of the multi-threaded implementation, which translates to 1.87 to over 24 times greater performance than the equivalent manually optimized pure JS implementation.

This study suffers from a few limitations. First, it does not take into account cold starts vs. warm starts, and second, it is centered around a single algorithm. Additionally, in the Web browser benchmarks, the cost of WebAssembly compilation and memory allocations were not taken into account. In the future, we would like to experiment with other types of parameters such as comparing cold start versus warm start, calculating the actual cost of WebAssembly compilation, and to further extend our benchmarks to include more algorithms.

The concept of combining JS and Rust appears to be a reasonable approach to convergence, retaining the versatile prototyping characteristics of JS and complementing them with unmatched performance when required without the risks resulting from C/C++ misuse. This study has presented the evidence toward a path of convergence, as well as synergy between a safe system programming language and a high-level interpreted language, offering dynamic productivity via prototyping and performance when needed.

Author Contributions: Conceptualization, K.-I.D.K. and N.D.T.; methodology, K.-I.D.K. and N.D.T.; software, K.-I.D.K.; writing—original draft preparation, K.-I.D.K.; writing—review and editing, N.D.T.; visualization, K.-I.D.K.; supervision, N.D.T.; project administration, N.D.T.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Node.js. Available online: https://nodejs.org (accessed on 5 September 2022).
2. V8 JavaScript Engine. Available online: https://code.google.com/p/v8/ (accessed on 5 September 2022).
3. Chaniotis, I.K.; Kyriakou, K.I.D.; Tselikas, N.D. Proximity: A real-time, location aware social web application built with node. js and angularjs. In Proceedings of the International Conference on Mobile Web and Information Systems, Paphos, Cyprus, 26–29 August 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 292–295.
4. npm. Available online: https://www.npmjs.com/ (accessed on 5 September 2022).
5. Modulecounts. Available online: http://www.modulecounts.com/ (accessed on 5 September 2022).
6. Chaniotis, I.K.; Kyriakou, K.I.D.; Tselikas, N.D. Is Node. js a viable option for building modern web applications? A performance evaluation study. *Computing* **2015**, *97*, 1023–1044. [CrossRef]

7. Nasralla, M.M. An Innovative JavaScript-Based Framework for Teaching Backtracking Algorithms Interactively. *Electronics* **2022**, *11*, 2004. [CrossRef]
8. Rehman, I.U.; Sobnath, D.; Nasralla, M.M.; Winnett, M.; Anwar, A.; Asif, W.; Sherazi, H.H.R. Features of mobile apps for people with autism in a post covid-19 scenario: Current status and recommendations for apps using AI. *Diagnostics* **2021**, *11*, 1923. [CrossRef]
9. Node.js 8: Big Improvements for the Debugging and Native Module Ecosystem. Available online: https://medium.com/the-node-js-collection/node-js-8-big-improvements-for-the-debugging-and-native-module-ecosystem-58454861f2fc (accessed on 5 September 2022).
10. uNetworking/uWebSockets. Available online: https://github.com/uNetworking/uWebSockets (accessed on 5 September 2022).
11. libxmljs/libxmljs. Available online: https://github.com/libxmljs/libxmljs (accessed on 5 September 2022).
12. node-serialport. Available online: https://github.com/EmergingTechnologyAdvisors/node-serialport (accessed on 5 September 2022).
13. Anderson, B.; Bergstrom, L.; Goregaokar, M.; Matthews, J.; McAllister, K.; Moffitt, J.; Sapin, S. Engineering the servo web browser engine using Rust. In Proceedings of the 38th International Conference on Software Engineering Companion, Austin, TX, USA, 14–22 May 2016; pp. 81–89.
14. The Rust Project. Available online: https://www.rust-lang.org/ (accessed on 5 September 2022).
15. LLVM Compiler Infrastructure. Available online: https://llvm.org (accessed on 5 September 2022).
16. Light, A. Reenix: Implementing a Unix-Like Operating System in Rust. Undergraduate Honors Thesis, Brown University, Providence, RI, USA, 2015.
17. Wilkens, F. Evaluation of Performance and Productivity Metrics of Potential Programming Languages in the HPC Environment. Ph.D. Thesis, University of Hamburg, Hamburg, Germany, 2015.
18. Lin, Y.; Blackburn, S.M.; Hosking, A.L.; Norrish, M. Rust as a language for high performance GC implementation. *ACM SigPlan Not.* **2016**, *51*, 89–98. [CrossRef]
19. Reed, E. *Patina: A Formalization of the Rust Programming Language*; Tech. Rep. UW-CSE-15-03-02; Department of Computer Science and Engineering, University of Washington: Seattle, WA, USA, 2015; p. 264.
20. Zambre, R.; Bergstrom, L.; Beni, L.A.; Chandramowlishwaran, A. Parallel performance-energy predictive modeling of browsers: Case study of servo. In Proceedings of the 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), Hyderabad, India, 19–22 December 2016; IEEE: New York, NY, USA, 2016; pp. 22–31.
21. Friends of Rust. Available online: https://www.rust-lang.org/en-US/friends.html (accessed on 5 September 2022).
22. W3C WebAssembly Working Group. Available online: https://www.w3.org/wasm/ (accessed on 5 September 2022).
23. Kyriakou, K.I.D.; Tselikas, N.D.; Kapitsaki, G.M. Enhancing C/C++ based OSS development and discoverability with CBRJS: A Rust/Node. js/WebAssembly framework for repackaging legacy codebases. *J. Syst. Softw.* **2019**, *157*, 110395. [CrossRef]
24. Mozilla Hacks, Oxidizing Source Maps with Rust and WebAssembly. Available online: https://hacks.mozilla.org/2018/01/oxidizing-source-maps-with-rust-and-webassembly/ (accessed on 5 September 2022).
25. From Rust to beyond: The WebAssembly Galaxy. Available online: https://mnt.io/2018/08/22/from-rust-to-beyond-the-webassembly-galaxy/ (accessed on 5 September 2022).
26. Yan, Y.; Tu, T.; Zhao, L.; Zhou, Y.; Wang, W. Understanding the performance of webassembly applications. In Proceedings of the 21st ACM Internet Measurement Conference, Virtual Event, 2–4 November 2021; pp. 533–549.
27. Wang, W. Empowering Web Applications with WebAssembly: Are We There Yet? In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 15–19 November 2021; IEEE: New York, NY, USA, 2021; pp. 1301–1305.
28. De Macedo, J.; Abreu, R.; Pereira, R.; Saraiva, J. On the Runtime and Energy Performance of WebAssembly: Is WebAssembly superior to JavaScript yet? In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Melbourne, Australia, 15–19 November 2021; IEEE: New York, NY, USA, 2021; pp. 255–262.
29. De Macedo, J.; Abreu, R.; Pereira, R.; Saraiva, J. WebAssembly versus JavaScript: Energy and Runtime Performance. In Proceedings of the 2022 International Conference on ICT for Sustainability (ICT4S), Plovdiv, Bulgaria, 14–16 June 2022; IEEE: New York, NY, USA, 2022; pp. 24–34.
30. There Will Be a Unified Programming Language. Available online: http://bigthink.com/videos/there-will-be-a-unified-programming-language (accessed on 5 September 2022).
31. Daloze, B.; Marr, S.; Bonetta, D.; Mössenböck, H. Efficient and thread-safe objects for dynamically-typed languages. *ACM SigPlan Not.* **2016**, *51*, 642–659. [CrossRef]
32. Libuv, Asynchronous I/O Made Simple. Available online: http://libuv.org/ (accessed on 5 September 2022).
33. Node.js Documentation, C++ Addons. Available online: https://nodejs.org/api/addons.html (accessed on 5 September 2022).
34. Hough, D.G. The ieee standard 754: One for the history books. *Computer* **2019**, *52*, 109–112. [CrossRef]
35. Dietz, W.; Li, P.; Regehr, J.; Adve, V. Understanding integer overflow in C/C++. *Acm Trans. Softw. Eng. Methodol.* **2015**, *25*, 1–29. [CrossRef]
36. Tselikis, G.S.; Tselikas, N.D. *C: From Theory to Practic*; CRC Press: Boca Raton, FL, USA, 2017.
37. Getreu, J. *Embedded System Security with Rust*; Tallinn University of Technology: Tallinn, Estonia, 2016.
38. Denning, P.J. The science of computing: The Internet worm. *Am. Sci.* **1989**, *77*, 126–128.
39. Blandy, J. *Why Rust?*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2015.

40. Stroustrup, B. Abstraction and the C++ machine model. In Proceedings of the International Conference on Embedded Software and Systems, Hangzhou, China, 9–10 December 2004; Springer: Berlin/Heidelberg, Germany, 2004; pp. 1–13.
41. Poss, R. Rust for functional programmers. *arXiv* **2014**, arXiv:1407.5670.
42. Neon. Available online: https://github.com/neon-bindings/neon (accessed on 5 September 2022).
43. Bray, T. The Javascript Object Notation (json) Data Interchange Format. Technical Report. 2014. Available online: https://www.rfc-editor.org/rfc/pdfrfc/rfc7159.txt.pdf (accessed on 5 September 2022).
44. CommonJS Spec Wiki. Available online: http://wiki.commonjs.org (accessed on 5 September 2022).
45. CoffeeScript. Available online: http://coffeescript.org/ (accessed on 5 September 2022).
46. Welcome to TypeScript. Available online: http://www.typescriptlang.org/ (accessed on 5 September 2022).
47. Babel. Available online: https://babeljs.io/ (accessed on 5 September 2022).
48. Kyriakou, K.I.D.; Chaniotis, I.K.; Tselikas, N.D. The GPM meta-transcompiler: Harmonizing JavaScript-oriented Web development with the upcoming ECMAScript 6 "Harmony" specification. In Proceedings of the 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC), Vegas, NV, USA, 9–12 January 2015; IEEE: New York, NY, USA, 2015; pp. 176–181.
49. Park, S.K.; Miller, K.W. Random number generators: Good ones are hard to find. *Commun. ACM* **1988**, *31*, 1192–1201. [CrossRef]
50. Benchmark.js. Available online: http://benchmarkjs.com/ (accessed on 5 September 2022).
51. gyp. Available online: https://gyp.gsrc.io/ (accessed on 5 September 2022).
52. node-gyp. Available online: https://github.com/nodejs/node-gyp (accessed on 5 September 2022).
53. NAN. Available online: https://github.com/nodejs/nan (accessed on 5 September 2022).
54. Node-Bindings. Available online: https://github.com/TooTallNate/node-bindings (accessed on 5 September 2022).
55. Blumofe, R.D.; Leiserson, C.E. Scheduling multithreaded computations by work stealing. *J. ACM* **1999**, *46*, 720–748. [CrossRef]
56. Blumofe, R.D.; Joerg, C.F.; Kuszmaul, B.C.; Leiserson, C.E.; Randall, K.H.; Zhou, Y. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Not.* **1995**, *30*, 207–216. [CrossRef]
57. kenOfYugen/node_rust_interop. Available online: https://github.com/kenOfYugen/node_rust_interop (accessed on 5 September 2022).