Rust – The Programming Language for Every Industry

Cosmin CARTAS
Department of Economic Informatics and Cybernetics
The Bucharest University of Economic Studies
cosmin.cartas@csie.ase.ro

Programming languages are continuously evolving by focusing on the following criteria: speed and ease of use. Rust is a system programming language developed by Mozilla Foundation, which offers features of a high-level language implemented by the principle of zero cost abstraction and is very efficient in terms of performance.
It provides a memory safety mechanism without using a garbage collector that could affect the performance, called borrow checker. Rust is providing out of the box a package manager used for importing dependencies, building and distribution of a project.
Keywords: *Rust, Web, WebAssembly, Electron, Performance*
DOI: 10.12948/ei2019.01.05

1 Rust Overview

Rust is a system programming language created by Mozilla Foundation that offers the feeling of a highly abstracted language and a safe memory allocation mechanism. It encapsulates the most important aspects that needs to be present in a complete development environment. New functionalities are implemented by using the principle of zero cost abstraction which implies that if a new feature is added that solves a complicated use case, it shouldn't affect the performance in any way [1].
Microsoft states that 70% of security flaws discovered in their systems from every year are related to memory safety. Rust solves this issue out of the box due to its borrow checker mechanism [9].
The borrow checker works by a very simple principle: if a variable is declared in a specific context, at the end of the execution of that context the memory allocated for the variable will be freed. If a variable is passed to another function by value, the function will free the memory allocated at the end of its execution. This is made possible due to a property of variable bindings, called ownership [7]. A variable in Rust can be bound to a single context and the ownership could be passed from an execution context to another (Fig.1.1). However, there is a possibility to pass variables to other functions without reference if the type of the variable implements the Copy trait without switching the ownership as well [7] [8].
In the example below we have a simple string variable that is initialized in the main function of the program. The second function takes a string as an argument, concatenates it and returns the new value. When the variable is passed to *borrow_function* by value, it will gain ownership of value. According to Rust's principle of memory deallocation, at the end of the execution of *borrow_function* value will be freed. Returning to main function there is one more line that needs to be executed but the variable that is trying to access no longer exists in memory, resulting in a compiling error.

```rust
fn main() {
    let value: String = String::from("Rust Ownership");
    println!("{}", borrow_function(value));
    println!("Value is: {}", value);
}

fn borrow_function(value: String) -> String {
    let mut result = String::from("Hello ");
    result.push_str(value.as_str());
    result
}
```

Fig. 1. Borrow checker example

Rust's main focus is safety and all the variables are immutable by default. By using this principle, the compiler can tell you if a mutation occurred when it wasn't intended. This mechanism is very helpful when a reference is passed in multiple contexts or accessed across multiple execution threads. Of course, there is the option to create a mutable variable by adding the *mut* keyword in front of it [3].

When choosing a programming language one of the aspects that should be taken care of would be interoperability. There might exist situations where a needed third-party library is only available for C/C++. Rust makes possible the integration with any C/C++ library due to its function foreign interface [7].

Probably one of the greatest features of the language is the package manager, called cargo. Having the experience of development in C++ on a production level project, would definitely require the use of make and Makefiles. That adds a new layer of complexity to the process and implies learning a new framework. Rust learnt from this approach and the language wasn't released until it had the mandatory tooling available. The mechanism used for dependency management is straight forward and only a manifest file is needed to configure the project for any scenario possible [5] [6].

2 Rust Applied in Different Industries

Being developed by Mozilla Foundation, Rust has a very strong integration with web technologies, but it's not limited to that. This was made possible due to release of a new standard called WebAssembly that compiles Rust code to proprietary bytecode and is ran in a safe and encapsulated sandbox within the browser [3]. From a high-level architecture most of the web applications follow the client server bidirectional communication over HTTP protocol. Rust could be used on both sides in order to increase the performance of the application, by injecting WASM modules directly into the frontend or by integrating the server with native modules. There is no doubt that there might be scenarios where the performance gain is not significant or not even needed, so the decision to make this integration should be taken after a deep analysis was made [4].

Cross platform applications are highly demanded lately from a lot of publishers and developers are continuously looking for alternatives to develop them. The true cross platform framework that is globally used is Electron and the idea behind it encapsulates the V8 JavaScript Engine combined with a WebView and the web technologies. So, this solution gives the possibility for the majority of developers to create an application with their current set of skills (for web developers). There are situations where JavaScript is not fast enough for the requirements of the application and the solution would be to integrate it with a more performant language that could bind to it. NodeJS API for integration with native modules is written in C++ but Rust can be used as well due to its foreign function interface. Some of the scenarios where Rust fits very well are the following: binary files

processing, number operations, collection manipulation [3].

Discussing about the development process in the world of Internet of Things and embedded devices requires a set of features that needs to be checked, depending on the business case: interoperability, strong JSON parsing, serial communication, low runtime requirements and security [6]. Development in IoT is different than the one from enterprise applications because there are a lot of restrictions related to power consumption, limited hardware resources and latency and environment conditions. Rust is one of the best candidates for the scenario described above because of the principles based on what it was developed. It is secure and can run even on bare metal without the standard library for a specific operating system with minimal power consumption (Fig 2.1).

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

Fig 2. Energy Consumption Benchmark on different algorithms [2]

3 References, citations and authors descriptions

In this chapter a more in-depth analysis will be performed in order to showcase the ability of Rust to improve the performance of an application. For the test we will mainly focus on the first two categories of applications: web and cross-platform, written in JavaScript. The comparison with JavaScript will be done on multiple aspects: DOM Rendering, working with numbers and strings on the server and client side and serialization. The client-side rust code will run as a WebAssembly module and for the server side we will perform profiling on a native module written in Rust but also native code.

This benchmark will showcase a scenario of creating multiple DOM Elements, add inner HTML to them and append them to the page. The tool used for this test is wasm-bindgen that compiles the Rust library to WebAssembly module.

Table 1. DOM Rendering Benchmark

Scenario	Rust WASM Module	Javascript
Rendering 10 elements	0.454833984375ms	0.391845703125ms
Rendering 100 elements	1.022216796875ms	0.976806640625ms
Rendering 1000 elements	11.962890625ms	6.586181640625ms
Rendering 10000 elements	73.955078125ms	54.496826171875ms
Rendering 100000 elements	798.493896484375ms	586.662109375ms

After analyzing the results, we can notice that Rust can be even twice slower than JavaScript for DOM manipulation, but its main purpose is not for building frontend applications (Table 1). One thing that should be taken care of when choosing Rust as programming language for developing on the client side is the complexity that could be achieved in enterprise applications. At this aspect JavaScript frameworks are more mature and easier to use.

The next scenario that will be tested will be to compute the result of a simple function that will return the n-th number from a Fibonacci sequence. Same implementation was made for Rust and Javascript with the same tooling (wasm-bindgen) for converting the library into a WebAssembly module.

Table 2. Fibonacci Benchmark

Scenario	Rust WASM Module	Javascript
Fibonacci of 10	0.1023853265ms	0.1534635483ms
Fibonacci of 100	0.2509765625ms	0.427978515625ms
Fibonacci of 1000	0.155029296875ms	0.429931640625ms
Fibonacci of 10000	0.1611328125ms	0.679931640625ms

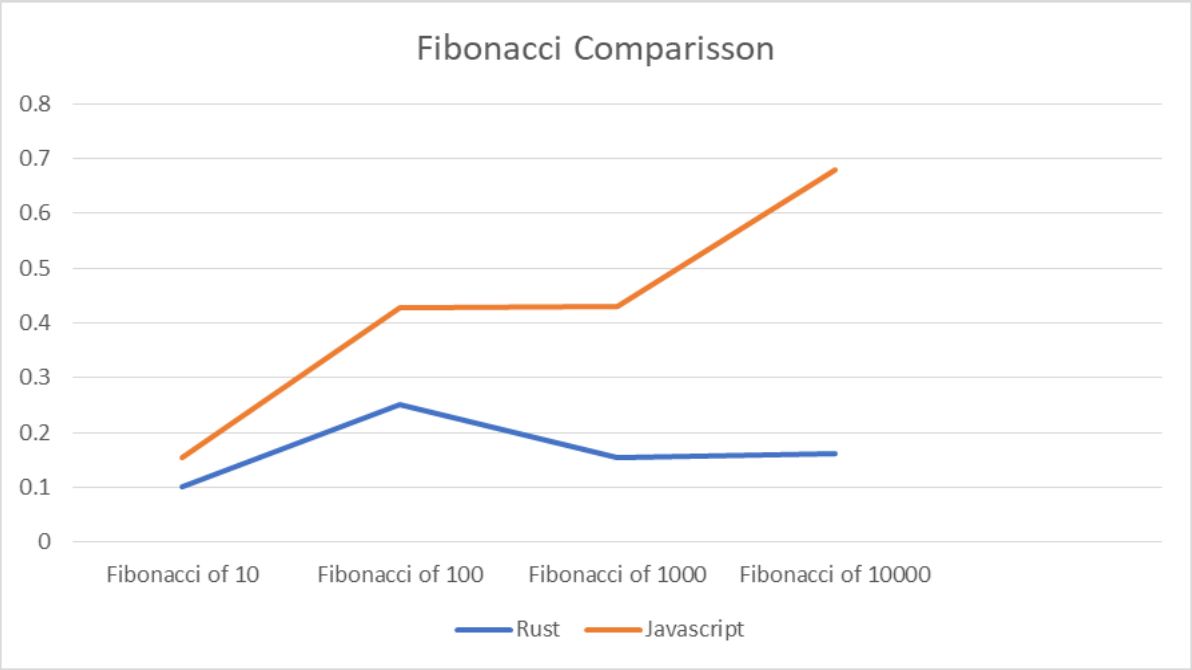

Fig. 3. Fibonacci Benchmark

The results from Table 2. will show that Rust is significantly faster especially when passing a larger number as a parameter for the function. One downside of this approach will be the fact that WebAssembly does not support BigInt as a type yet, even though BigInt is present in Rust, so a larger number would print an irrelevant result. This aspect will be solved

in the next test from the server side. This comparison will analyze the capability of Rust and JavaScript to manipulate strings and the function implemented will take an empty string and append multiple values to it.

Table 4. String concatenation Benchmark

Scenario	Rust WASM Module	Javascript
Concatenate 10 values	1,206787109375ms	0,604736328125ms
Concatenate 100 values	1,418212890625ms	0,594970703125ms
Concatenate 1000 values	1,5732421875ms	0,3232421875ms
Concatenate 10000 values	1,865234375ms	0,5361328125ms
Concatenate 100000 values	2,173285642ms	0,7732452923ms

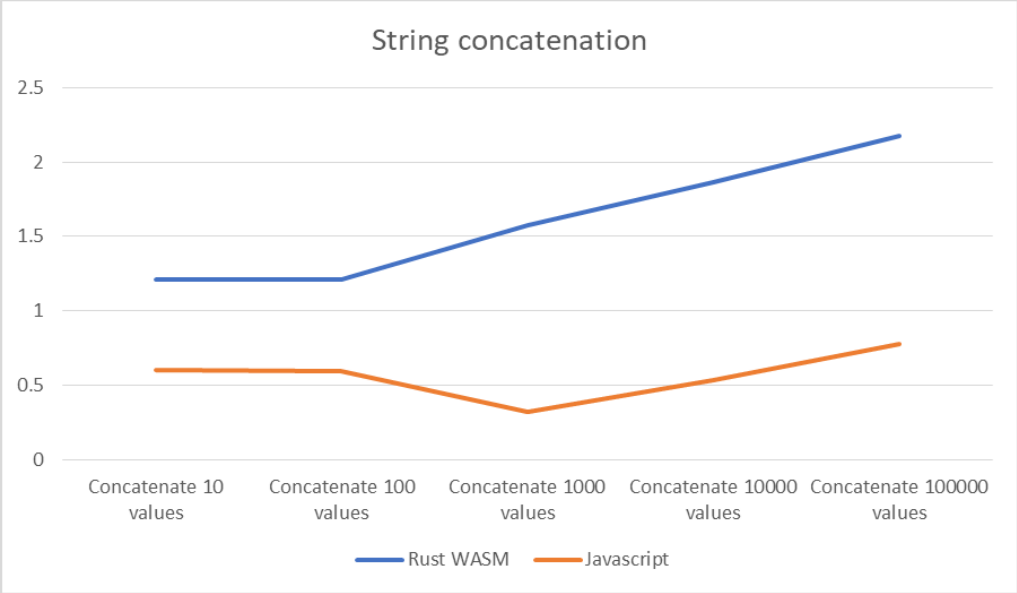

Fig. 4. String concatenation benchmark

The result from this test would point out that effort put in the JavaScript engine for parsing the main format used for web applications (JSON) pays off. From the benchmark could be noticed that JavaScript could be even 3 times faster than Rust on string manipulation. For the serialization test was applied a very common scenario where a specific payload in JSON format needs to be serialized to a file. NodeJS comes in advantage here because in JavaScript objects have the structure of JSON natively. For serialization in Rust the library serde was used in order to transform an array of structs to a JSON string. We will benchmark not only Rust and JavaScript but also a NodeJS module that is written in Rust.

Table 5. JSON Object Serialization

Scenario	Rust	Node Native Module	Javascript
Serialize 10 objects	0,154617ms	0,15623499ms	0,877ms
Serialize 100 objects	0,18937099ms	0,188241ms	1,011ms
Serialize 1000 objects	0,391794ms	0,481081ms	1,478ms
Serialize 10000 objects	2,24886099ms	3,63775399997ms	9,015ms
Serialize 100000 objects	24,165125ms	43,7603159996ms	81,555ms

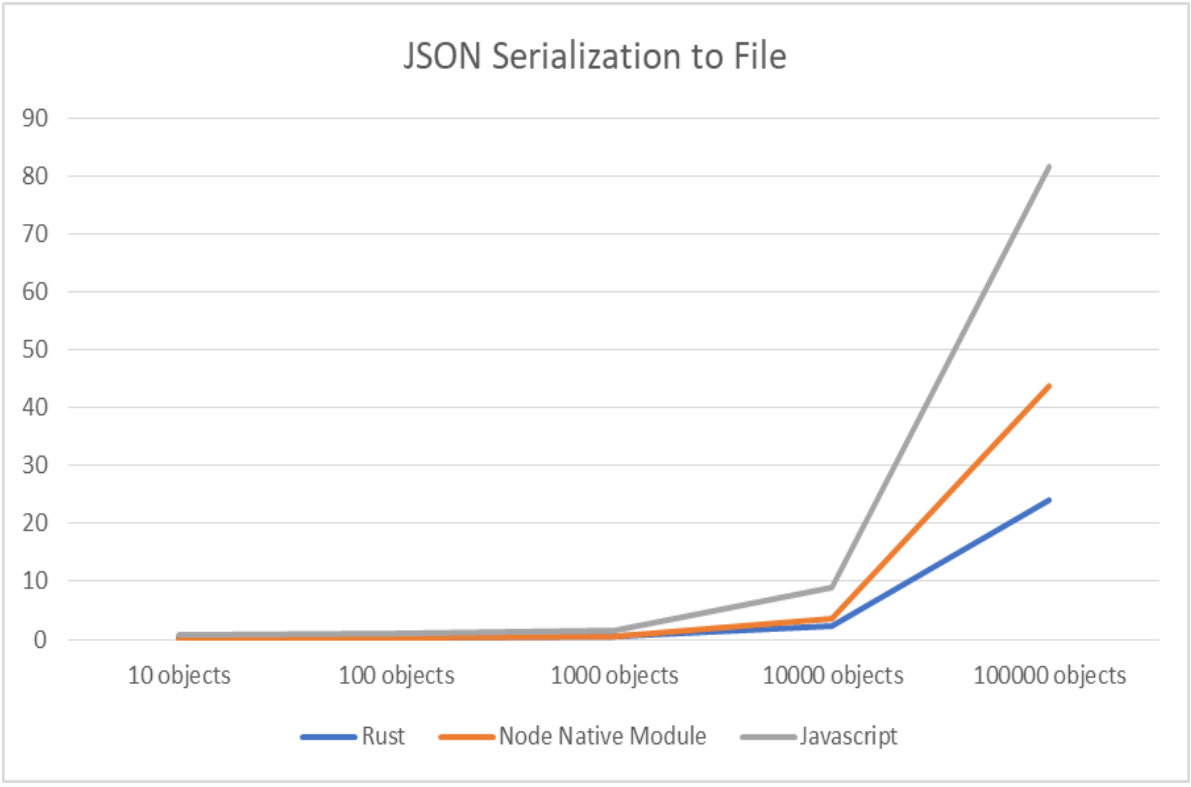

Fig. 5. JSON Serialization to File

The results from the table above are self-explanatory it's obvious that Rust is from two to four times faster than Javascript.

4 Conclusions

Rust could bring huge improvements in terms of performance to an application but there are cases where the complexity could grow significantly, so the effort to make the implementation is not worth it. Before making a decision to add or rewrite a module of a project using Rust, an in-depth analysis should be performed to be sure that it will bring any benefits to the solution. One thing that should be considered is the maturity of the language in a short period, first release of Rust was done in 2015. It has grown a huge community with a great evolution, with more and more features added over the years. The language provides a lot of tools and libraries for different use-cases, from native UI development to integration with any C/C++ library out of the box and building WASM or WASI modules. Rust will definitely not replace Javascript in any future, but it will combine together to improve the aspects where Javascript is not giving the best results. If WASM would be mature enough to support all data types and create close to zero time to initialize it will definitely worth the switch from non-sensitive operations from server-side to client side in order to reduce costs and without losing any performance. The main advantage of using Rust as your development programming language is the memory safety mechanism that it offers. The compiler will not allow you to do mistakes in terms of ownership and mutability that leads to code of a decent quality without the need of having huge experience with the language.

References

[1] S. Klabnik and C. Nichols, "*The Rust Programming Language*", No Starch Press, 2019, ISBN 978-1593278281
[2] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes and J. Saraiva, "*Energy Efficiency across Programming Languages*", International Conference of Software Engineering 2017
[3] K. Hoffman, "*Programming WebAssembly with Rust*", The Pragmatic Programmers, March 2019
[4] M. Rourke, "*Learn WebAssembly*", Pakt Publishing, 2018

[5] J. Blandy and J. Orendoff, "*Programming Rust: Fast, Safe Systems Development*", O'Reilly Media, 2019
[6] J. Hiner, "*We Rewrote Our IoT Platform in Rust and Got Away with It*", Medium.com, July 2019, [Online], https://medium.com/dwelo-r-d/we-rewrote-our-iot-platform-in-rust-and-got-away-with-it-2c8867c61b67
[7] C. Matzinger, "*Data Structures and Algorithms with Rust*", Pakt Publishing, 2019
[8] B. L. Troutwine, "*Concurrency with Rust*", Pakt Publishing, 2018
[9] R. Levick, "*Why to choose Rust as your next programming language*", Opensource.com, 2019, [Online], https://opensource.com/article/19/10/choose-rust-programming-language

Cosmin Cartas has always been passionate about technology and interested in learning from others. He is a Full Stack Developer with work experience demonstrated in different industries as well as a programmer with experience in developing modern web applications and complex distributed systems.
Throughout his career he has had several roles such as Java Developer, Web Developer or Security Engineer. As a Security Engineer, he was responsible, among other things, for identifying security deficiencies in software solutions developed in different programming languages such as Java, JavaScript, Android, C ++ or iOS. More than that, he also worked on developing solutions to address identified vulnerabilities, DevOps security and threat modeling. He has bachelor degree in Computer Science and a master's degree in IT&C Security from The Bucharest University of Economic Studies and is doing now a PhD in Optimization of Distributed Systems.