DOI: 10.2507/33rd.daaam.proceedings.xxx

RUST AND WEBASSEMBLY FOR FAST, SECURE, AND RELIABLE SOFTWARE

Zlatan Moric, Loïc Branstett & Robert Petrunic

This Publication has to be referred as: Moric, Z[latan]; Branstett, L[oïc]; Petrunic, R[obert] (2022). Rust and WebAssembly for fast, secure and reliable software, Proceedings of the 33rd DAAAM International Symposium, pp.xxxx-xxxx, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-xx-x, ISSN 1726-9679, Vienna, Austria
DOI: 10.2507/33rd.daaam.proceedings.xxx

Abstract

Rust is a new systems programming language that promises to overcome the seemingly fundamental tradeoff between high-level safety guarantees and low-level control over resource management. WebAssembly (Wasm) is a new binary instruction format for a stack-based virtual machine, it is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications. This paper explores these two new technologies and how they interact with each other. Our findings showed that the combination of those two technologies could revolutionize the way the entire industry design fast and secure software.

Keywords: Rust; WebAssembly; C++; memory security; reliable software.

1. Introduction

Systems programming languages like C and C++ give programmers low-level control over resource management at the expense of safety, whereas most other modern languages give programmers safe, high-level abstractions at the expense of control. It has long been a holy grail of programming languages research to overcome this seemingly fundamental trade-off and design a language that offers programmers both high-level safety and low-level control. Rust comes closer to achieving this holy grail than any other industrially supported programming language to date, compared to mainstream safe languages Rust offers both lower-level control and stronger safety guarantees [1]. Some safe languages like Java or C# use a virtual machine but unlike WebAssembly, which is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web. WebAssembly has been designed with formal semantics from the start making it a strong contender for secure and reliable software [2].

From the perspective of a web browser, WebAssembly is a type of code that web browsers can execute directly – with its compact binary format we can expect to have near-native performance while using different programming languages as compilation targets. While it's debatable how useful that is, WebAssembly is also capable of running within a web page that already contains JavaScript code, so mixed use of these technologies is possible.

In this paper, we are going to focus on Rust as a source for WebAssembly, as this seems to be a very popular choice for developers. We also consider it to be a sensible approach, as Rust offers advantages over many existing programming languages in terms of security and semantics. So, let's explore Rust first with a brief overview, then move on to WebAssembly and finally check how they would partner for a high-performance web application.

2. Rust

In this section, we give a brief overview of some of the central features of the Rust programming language. As a programming language, Rust has a different focus than many other programming languages – it's very efficient with memory, very fast, and prioritizes memory security which seems to be a big problem for applications and computer architecture in the past ten or so years with a variety of different memory and CPU-related issues hampering application development and speed. It's well-documented and recently made it to the Linux kernel as a development platform (version 6.1).

A hugely popular feature of Rust is its capability to work with JavaScript and publish to the npm registry, which makes it a very attractive platform for web developers – so it's not meant to be used for non-web applications only. Also, with the proliferation of IoT devices and the general growth of the IoT concept, its capability to run decently on devices that are low on resources seems like a reasonable value proposition to use it.

Let's now look at some basic capabilities and advantages of Rust.

2.1. Ownership

Ownership is Rust's most unique feature and has deep implications for the rest of the language. It enables Rust to make memory safety guarantees without needing a garbage collector.

The ownership rules are quite simple:

- Data is assigned to a variable.
- The variable becomes the 'owner' of the data.
- There can only be one owner at a time.
- When the owner goes out of scope, the data will be dropped.

Primitive types are popped from stack memory automatically when they go out of scope (e.g., when a function block ends), while complex types must implement a drop function that Rust will call when out of scope (to explicitly deallocate the heap memory).

2.2. Borrowing

The concept of borrowing is designed to make dealing with ownership changes easier. It does this by avoiding the moving of owners, and by letting your program provide a 'reference' to the data. This means the receiver of the reference (e.g., a function, struct field, or a variable) can use the value temporarily without taking ownership of it.

There are two main rules to the concept of borrowing:

- At any given time, you can have either (but not both of) one mutable reference or any number of immutable references.
- References must always be valid.

Those rules prevent concurrent access to the same underlying memory location and ensure that the underline memory location is safe to access at any time.

These effectively make data races impossible [3].

2.3. Lifetimes

Lifetimes are tightly coupled to references. They are a way for the compiler to know how long a reference lives, and to be sure that any reference that is currently active doesn't refer to data that no longer exists (i.e., a 'dangling pointer').

At the language level lifetimes are just an annotation that has a specific naming convention: '<T> where <T> is a letter like 'a or 'b. The letters don't mean anything special, they're just a way for the user to differentiate them. [4]

The below code example highlights how defining a single lifetime called 'a and assigning it to both arguments (and to the return value) allows the compiler to track these references and ensure they both live long enough to prevent any errors at runtime.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
   if x.len() > y.len() {
      x
   } else {
      y
   }
}
```

Code 1. Example of usage of lifetimes in a Rust code

2.4. Interior Mutability

Interior mutability is a design pattern in Rust that allows you to mutate data even when there are immutable references to that data; normally, this action is disallowed by the borrowing rules. To mutate data, the pattern uses unsafe code inside a data structure to bend Rust's usual rules that govern mutation and borrowing.

The basic idea behind Interior Mutability is UnsafeCell [5] a basic primitive in Rust that allows checks normally done by the borrow checker to be deferred (not bypassed) to the library code.

RefCell [6] is a type that is based on UnsafeCell and does the checks at runtime. This still provides the same safety guarantees as the borrow checker would give at compile-time. This allows developers to still be safe even when outside of the standard borrowing rules.

Rust has many features to make the code safe but how do these features ensure memory safety? Let's look at a few examples of programs that should be memory unsafe and see how Rust identifies their errors.

2.5. Memory safety – dangling pointers

Let's take a look at the following code example:

```
fn foo() -> &i32 {
   let n = 0;
   &n
}
```

, and result from its compilation:

```
error[E0106]: missing lifetime specifier
--> test.rs:1:13
|
1 | fn foo() -> &i32
| ^ expected lifetime parameter
|
= help: this function's return type contains a borrowed value, but there is no value
  for it to be borrowed from
= help: consider giving it a 'static lifetime
```

If we attempt to return a pointer to a value owned by n, where the pointer would nominally live longer than the owner. Rust identifies that it's impossible to return a pointer to something without a "value for it to be borrowed from."

2.6. Memory safety – use-after-free

Let's take a look at the following code example to create a variable and pass it to the drop function:

```
use std::mem::drop; // equivalent to free()
fn main() {
let x = "Hello".to_string();
drop(x);
```

```
println!("{}", x);
}
```

, and result from its compilation:

```
error[E0382]: use of moved value: `x`
--> test.rs:6:18
|
5 | drop(x);
| - value moved here
6 | println!("{}", x);
| ^ value used here after move
|
= note: move occurs because `x` has type `std::string::String`, which does not implement
  the `Copy` trait
```

Due to Rust's ownership semantics, when we free a value, we relinquish ownership of it, which means subsequent attempts to use the value are no longer valid. This also protects against double frees, since two calls to drop would encounter a similar ownership type error.

2.7. Development safety

Rustc (the Rust official compiler) won't compile programs that attempt unsafe memory usage. Most memory errors are discovered when a program is running. Rust's syntax and language metaphors ensure that those common memory-related problems in other languages null or dangling pointers, data races, and so on—never make it into production. This makes the developers more confident about the safety and reliability of their code. [7]

Let's now move on to WebAssembly - as a near-native-performance platform that's able to use various programming languages as sources, WebAssembly might play a significant role in the future of application development, especially when talking about web applications.

3. WebAssembly

WebAssembly is based on the principle of a stack-based virtual machine. It's an abstraction of a computer, that emulates a real machine. In this case, the emulation is based on a stack pointer that follows the instructions of the code. Its capability to execute binary code on the web while being independent of the programming language makes it a very attractive proposition for running fast, near-native performance web applications.

WebAssembly has modules that are binary compiled by the browser (or a virtual machine). It has its own ISA (Instruction Set Architecture), and – at its core – it's not a project that wants to replace the ever-present JavaScript, as it tries to work alongside it to make the best of both worlds.

Let's now discuss some WebAssembly specifics that make it such a good platform for web application developers.

3.1. Modules

A Wasm program is designed to be a separate module containing collections of various Wasm-defined values and program-type definitions. Every module is separated from the other modules. This provides native isolation between the code, meaning that the code from module A cannot access anything from module B that isn't explicitly exported/allowed by module B. This is particularly useful in our modern world where malware tries to access the memory of a program to give them privileges.

3.2. Linear memory

The main storage of a WebAssembly program is a large array of bytes referred to as linear memory or simply memory. This technique is fairly standard and has some advantages as the CPU can directly (and linearly) address all of the available memory locations without having to resort to any sort of memory segmentation or paging schemes [8].

3.3. Security

The linear memory is disjoint from code space, the execution stack, and the engine's data structures; therefore, compiled programs cannot corrupt their execution environment, jump to arbitrary locations, or perform other undefined

behavior. At worst, a buggy or exploited WebAssembly program can make a mess of the data in its memory [9]. This means that even untrusted modules can be safely executed in the same address space as other code. It also allows a WebAssembly engine to be embedded into any other managed language runtime without violating memory safety, as well as enabling programs with many independent instances with their memory to exist in the same process.

3.4. Determinism

The design of WebAssembly has sought to provide a portable target for low-level code without sacrificing performance. Where hardware behavior differs it usually is corner cases such as out-of-range shifts, integer divide by zero, overflow or underflow in floating point conversion, and alignment.

WebAssembly gives deterministic semantics to all of these across all hardware with only minimal execution overhead. However, there remain three sources of implementation-dependent behavior that can be viewed as non-determinism [10]:

- NaN Payloads: WebAssembly follows the IEEE-754 standard for floating-point arithmetic [11]. However, this standard doesn't define the representation of the NaN payload and instead only puts a rule for what it cannot be. This leads to CPUs handling and representing NaN payload differently.
- Resource Exhaustion: Available resources are always finite and differ wildly across devices. An "out of memory" error is unpredictable and therefore nondeterministic.
- Threads: Executing concurrently some code is highly dependent on the CPU Scheduler and subject to the difference between runs as the CPU has a different schedule profile [12].

Nevertheless, these nondeterminisms are limited and local, meaning that WebAssembly is as deterministic or even better than every other program.

3.5. Control-flow integrity

Within WebAssembly, Control-flow integrity (CFI) is enforced by the Memory Safety API. CFI protects against code-reuse attacks, such as Return Oriented Programming (ROP) [13]. The efficacy of control-flow integrity can be measured based on its completion. Generally, three external control-flow transitions need to be safeguarded because the caller may not be trusted. Those are direct function calls, indirect function calls, and returns.

3.6. Memory safety

To ensure memory safety, WebAssembly will use an expected control flow graph when it is compiled. Vectors of computer-readable instructions can be made safe by inserting runtime instrumentation at every call site to verify that the transition is safe.

3.7. Memory isolation

The Memory Isolation API provides capabilities to protect against shared memory writes between two modules with different permissions (e.g., one module can read and change data while another cannot) [14]. This protects against vulnerabilities such as dangling pointers where an attacker might use knowledge of a pointer location from one context to attack it in another. The Memory Isolation API takes advantage of type system features available only when compiling to WebAssembly, such as linear types, which provide transparent compile-time enforcement at runtime.

3.8. Development safety

As for WebAssembly, its determinism of it makes developers also more confident and more productive as they don't have to worry about non-determinism and platform-specific methods/conversions or APIs. And, because WebAssembly methods are typed misuse of them (e.g., passing a short instead of an int) is impossible reaffirming the safety to the developers.

4. Speed of execution

As stand, previous WebAssembly is based on the principle of a stack-based virtual machine. This technique isn't new and is used in many implementations like the JVM (Java Virtual Machine) or the CLR (.NET Common Language). These implementations of a stack-based virtual machine prove to be extremely high-performant with only a small performance penalty [15]. However, unlike most JVM or CLR implementations many WebAssembly interpreter has also a compiler mode to compile WebAssembly to the native binary format saving the cost of the interpretation and allowing many optimizations making these binary close to native speed.

Rust is based on the now robust LLVM project a backend for compiler optimization, its most famous frontend is Clang, a C/C++ compiler. LLVM caries many optimizations accumulated over more than 2 decades. Studies as found that in general the same C/C++ program compiled with GCC, and Clang only has a +5% percent difference in performance [16].

This greatly helps Rust achieve the same or better performance than an equivalent C++ program compiled with Clang. Note that there is ongoing work to provide GCC as an alternative backend for the Rust compiler. This could potentially make Rust programs even faster than with LLVM [17].

For comparison, we executed a series of number operation counter code written in JavaScript and compare it with Rust code.

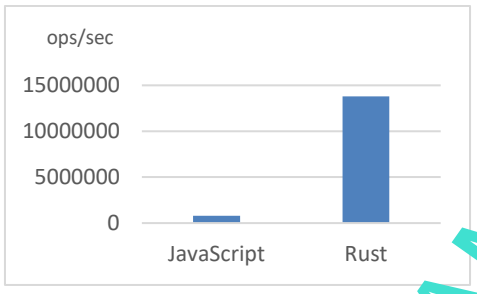

Fig. 1. JavaScript vs Rust speed comparison on simple counting loop

In this case (Fig. 1.), Rust outruns JavaScript by far. Next, we tried a similar test but now with strings to see how the technologies behave with a more complex set of data. Again, in both technologies, we create functions that accept two strings and check if the second string is part of the first.

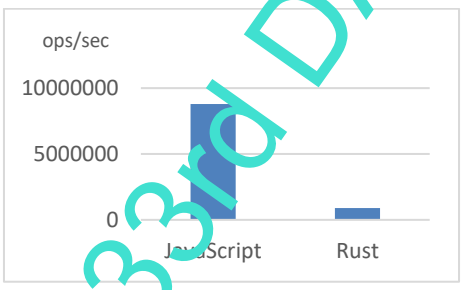

Fig. 2. JavaScript vs Rust speed comparison with strings

It was strange that JavaScript is so faster than Rust (Fig. 2). The raw execution of an algorithm in WASM is almost always faster than in JavaScript. However, when passing data into the WASM module's memory, the cost can be so high that it negates the benefit of using WASM. To verify this assumption, we conduct another set of experiments. Instead of just calling a method with parameters, we will call a method with static parameters, that are already in memory, and loop 1000 times.

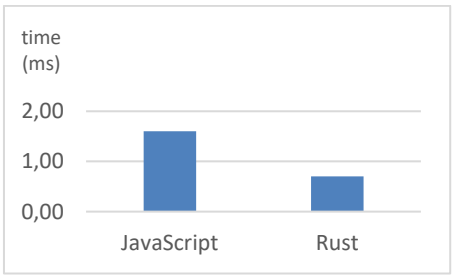

Fig. 3. JavaScript vs Rust speed comparison with static parameters on 1000 iterations

Rust-based WASM alternative outruns JavaScript by far (Fig. 3.). Writing complicated data structures into WASM memory is expensive. For example, to give a basic string to a WASM module, you must first encode the string, then find some free memory, and finally, write the encoded string into that memory. This can become slow depending on the memory management strategy being used. Additionally, the operation runs slower the longer the string.

5. Conclusion

Rust and WebAssembly, both have been designed with security and reliability in mind. This makes the combination of them through features like borrow checking and ownership for Rust and modularization and determinism in WebAssembly even more (data races, invalidation, …) secure than any other alternative like C/C++ and JVM or CLR.NET even though most of them can be compiled to WebAssembly. The speed of WebAssembly and Rust also greatly improve their possible massive adoption as a secure combination for fast, secure, and reliable software. The capability of Rust/WebAssembly combination to work alongside JavaScript seems like a very good adoption point as it creates a non-destructive environment where WebAssembly applications can be seamlessly integrated into a rich ecosystem of JavaScript applications without development wars raging all over the world. Also, if the non-existence was to be a standardized way of deploying future web applications, it would mean that we would also get the best of both worlds in terms of functionality, security, and speed, which should be the point of developing web applications. At the end of the day, we do live in an age where almost anything needs to be a web application or have a web-application frontend. Related to performance, WebAssembly is a good choice either if you are only handling numeric data or if your algorithm is complex enough for the cost of passing arguments to the WebAssembly module to become negligible. Another advantage is related to the obfuscation of code in browsers which increases the complexity of reverse engineering.

In the future, we expect to see a massive increase in the usage of Rust and WebAssembly. In our future research, we will focus on trying to determine if the security of these technologies is a decisive factor in favor of adoption or if there are other key points besides the non-destructive nature of WebAssembly that could lead to an even better adoption rate.

6. References

[1] Jung, R.; Jourdan J.; Krebbers, R. & Dreyer D. (2018). RustBelt: Securing the Foundations of the Rust, Proc. ACM Program. Lang. 2, POPL, Article 66 (January 2018), 34 pages. DOI:10.1145/3158154 https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf

[2] https://webassembly.org/docs/non-web (2022). The WebAssembly Contributors, Accessed on: 2022-06-30

[3] https://doc.Rust-lang.org/book/ch04-02-references-and-borrowing.html#mutable-references (2022). References and Borrowing, The Rust Contributors, Accessed on: 2022-06-30

[4] https://doc.Rust-lang.org/book/ch10-03-lifetime-syntax.html (2022). Lifetime Syntax, The Rust Contributors, Accessed on: 2022-06-30

[5] https://doc.Rust-lang.org/core/cell/struct.UnsafeCell.html (2022). UnsafeCell, The Rust Contributors, Accessed on: 2022-06-30

[6] https://doc.Rust-lang.org/std/cell/struct.RefCell.html (2022). RefCell, The Rust Contributors, Accessed on: 2022-06-30

[7] Jedlicka, M., et al. (2009). Reliability assessment using UML models, DAAAM International Scientific Book, pp. 53+

[8] Matei, R. (2021). A practical guide to WebAssembly memory, https://radu-matei.com/blog/practical-guide-to-wasm-memory, Accessed on: 2022-06-30

[9] Lewycky, N. (2021). Why is WebAssembly safe and what is linear memory model, https://stackoverflow.com/a/65933986, Accessed on: 2022-06-30

[10] Richey J. (2017). Nondeterminism in WebAssembly, https://github.com/WebAssembly/design/blob/main/Nondeterminism.md, Accessed on: 2022-06-30

[11] Hough D., et al. (2019). IEEE Standard for Floating-Point Arithmetic, IEEE, New York

[12] https://www.iitk.ac.in/esc101/05Aug/tutorial/essential/threads/priority.html (2011). Thread Scheduling, Sun Microsystems, Accessed on: 2022-06-30

[13] https://webassembly.org/docs/security/#control-flow-integrity (2021). Control Flow Integrity, Accessed on: 2022-06-30

[14] https://webassembly.org/docs/security/ (2020). WebAssembly - Security, Accessed on: 2022-06-30

[15] Lippert, E. (2011). Why have a stack?, https://web.archive.org/web/20120328082950/http://blogs.msdn.com/b/ericlippert/archive/2011/11/28/why-have-a-stack.aspx, Accessed on: 2022-06-30

[16] Kornelski, D. (2021). Speed of Rust vs C, https://kornel.ski/Rust-c-speed, Accessed on: 2022-06-30

[17] Antoyo (2020). Rustc_codegen_gcc, https://github.com/Rust-lang/Rustc_codegen_gcc, Accessed on: 2022-06-3