ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
DIPARTIMENTO DI INFORMATICA-SCIENZA E INGEGNERIA (DISI)
Corso di Laurea in Ingegneria Informatica

Developing and Managing Applications on Top of the WebAssembly System Interface (WASI)

Candidato:
Luca Giovannini

Relatore:
Prof. Paolo Bellavista

Anno Accademico 2021/2022
Sessione IV

Summary

1. Web and WebAssembly landscape

Introduction

In this thesis we are going to discuss one of the most significant technological advancements in recent years in the field of computer science: the development of WebAssembly System Interface (WASI), a new standard that allows developers to run code in a safe and sandboxed environment.

To understand the relevance of WASI, first we need to get familiar with WebAssembly. WebAssembly (Wasm) is a relatively modern technology that was introduced in 2015 as a joint effort by major browser vendors, including Mozilla, Google, Apple, and Microsoft. The goal was to create a low-level bytecode that can be executed in a browser environment, which can be used as a target for various programming languages. The main idea was to bridge the gap between high-level programming languages and the low-level hardware, allowing for efficient execution of code in the browser without the need for plugins or native extensions. Since its introduction, WebAssembly has gained a lot of traction and is now supported in all major browsers, making it a viable option for web development.

Due to the popularity of WebAssembly, developers sought a way to utilize Wasm code beyond the browser environment. Several attempts were made, some of which resulted in inefficient solutions that attempted to simulate the browser in a native environment.

This is what WASI aims to do: provide an interface to the system for WebAssembly, offering efficiency and effectiveness, with the potential for lasting several decades.

This writing comprises three chapters. In Chapter 1, the current landscape of the web and WebAssembly will be examined. This includes an analysis of existing and developing technologies, leading up to the introduction of our main topic: WebAssembly System Interface (WASI). Chapter 2 delves deeply into WASI, clarifying its design principles and implementation. It also introduces the reader to the usage of this technology by means of a practical example. Chapter 3 will assist the reader in framing WASI within a stack of contemporary technologies by providing a real-world application and guiding them through each step of the process.

This document plans to analyse WASI comprehensively and provide the reader with a complete overview of the technology.

As WASI is still in its initial stages, it may be challenging to make precise predictions about its potential and future developments. However, this review will strive to be as precise and scientific as possible.

Specifically, this document will explore the benefits and trade-offs of using WASI for developing future cross-platform native applications.

Background

Software technologies are constantly evolving, and new programming languages, libraries, and frameworks are being developed. Specifically, the Web has a unique history, with various aspects that have been emphasized over time.

The approaching end of Moore's Law[1] due to the physical limitations of technology puts particular emphasis on the optimization of resources over an increase in raw performance, which is difficult to achieve with current technology.

The Web

In traditional Web development, client logic is managed using Javascript[2] code.

JavaScript was developed in 1995 as a dynamically typed language with the goal of providing a simple scripting language for animating web pages and turning the browser into an application platform. This focus on simplicity has resulted in a language with a simple syntax and structure, making it one of the easiest programming languages to learn. Over time, JavaScript has evolved into one of the most popular programming languages in the world, with its usage expanding beyond the browser to include servers and native applications. However, to execute web applications, JavaScript code needs to be transferred from the server to the client machine,

[1] Moore's law is the observation that the number of transistors in a dense integrated circuit doubles about every two years. Moore's law is an observation and projection of a historical trend. Rather than a law of physics, it is an empirical relationship linked to gains from experience in production.
[2] Often abbreviated as JS

where it is just-in-time compiled, optimized, and executed. This sequence of steps has been necessitated by the variety of JavaScript engines distributed with different web browsers and is unlikely to change.

WebAssembly

In this context, in 2015, WebAssembly was announced as a solution to these performance limitations of Javascript.

WebAssembly, often abbreviated as WASM, is a binary instruction format for a stack-based virtual machine. Wasm is designed to be a portable compilation target for programming languages, enabling deployment on the web for client and server applications.

Wasm is by nature faster to process and to analyse than Javascript, however it is not designed to be a substitution, but an integration to Javascript: they are meant to work together inside the Web ecosystem. WebAssembly is particularly suitable to improve the performance of critical and computationally heavy parts of a Web application.

Since 2019, WebAssembly has been a W3C standard, maintained in collaboration with Mozilla, Microsoft, Google, and Apple, and is currently supported by the main modern browsers such as Chrome, Edge, Firefox, and Safari.

It is possible to write WebAssembly applications directly in a text format called WAT (WebAssembly Text), which can be compiled and translated to binary code.

Although it is not a usual practice, as this format is very close to Assembly language and lacks any sort of abstraction, resulting in an impractical way of writing code.

Being no more than a set of binary instructions, it is also possible to compile applications written in several programming languages into Wasm. Among these, C/C++, C#, and Rust stand out; expressly for them, some tools have been developed to compile code to Wasm. For instance, Emscripten was one of the first and most popular ones: it allows to compile even complex C/C++ applications to Wasm.

As an alternative, a new programming language called AssemblyScript was developed. The name recalls TypeScript, and the syntax does as well, but with some key differences; for

example, AssemblyScript has strict static typing and allows no dynamic objects, which must be replaced by maps.

WebAssembly is executed directly inside Javascript Runtime thanks to some API's which are developed ad hoc and incorporated into it.

Wasm was designed with a specific goal in mind: the efficient and high-performance execution of browser-based applications, without compromising compatibility and security.

The web platform can be thought of as having two parts:

- A virtual machine (VM) that runs the Web app's code.
- A set of Web APIs that the Web app can call to control web browser/device functionality and make things happen (DOM, CSSOM, WebGL, IndexedDB, Web Audio API, etc.).

WebAssembly is a different language from JavaScript, designed to complement and work alongside it, allowing web developers to take advantage of both languages' strong points:

- JavaScript is a high-level language, flexible and expressive enough to write web applications. It has many advantages — it is dynamically typed, requires no compile step, and has a huge ecosystem that provides powerful frameworks, libraries, and other tools.
- WebAssembly is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages with low-level memory models such as C++ and Rust with a compilation target so that they can run on the web. (Note that WebAssembly has the high-level goal of supporting languages with garbage-collected memory models in the future.)

The different code types can call each other as required — the WebAssembly JavaScript API wraps exported WebAssembly code with JavaScript functions that can be called normally, and WebAssembly code can import and synchronously call normal JavaScript functions. In fact, the basic unit of WebAssembly code is called a module and WebAssembly modules are symmetric in many ways to ES modules[3].

[3] Short for ECMAScript module, it is a standard for organizing and sharing JavaScript code. It is a way to include and reuse code in JavaScript applications, providing better structure and maintainability.
An ES module is defined using the "**export**" keyword to export functions, objects, or values, and the "**import**" keyword to import them into another module. The modules are stored in separate files and have their own scope,

There are several key concepts needed to understand how WebAssembly runs in the browser. All these concepts are reflected 1:1 in the WebAssembly JavaScript API.

- Module: Represents a WebAssembly binary that has been compiled by the browser into executable machine code. A Module is stateless and thus, like a Blob, can be explicitly shared between windows and workers (via postMessage()). A Module declares imports and exports just like an ES module.
- Memory: A resizable ArrayBuffer that contains the linear array of bytes read and written by WebAssembly's low-level memory access instructions.
- Table: A resizable typed array of references (e.g., to functions) that could not otherwise be stored as raw bytes in Memory (for safety and portability reasons).
- Instance: A Module paired with all the state it uses at runtime including a Memory, Table, and set of imported values. An Instance is like an ES module that has been loaded into a particular global with a particular set of imports.

The JavaScript API provides developers with the ability to create modules, memories, tables, and instances. Given a WebAssembly instance, JavaScript code can synchronously call its exports, which are exposed as normal JavaScript functions. Arbitrary JavaScript functions can also be synchronously called by WebAssembly code by passing in those JavaScript functions as the imports to a WebAssembly instance.

Since JavaScript has complete control over how WebAssembly code is downloaded, compiled, and run, JavaScript developers could even think of WebAssembly as just a JavaScript feature for efficiently generating high-performance functions.

In the future, WebAssembly modules will be loadable just like ES modules (using <script type='module'>), meaning that JavaScript will be able to fetch, compile, and import a WebAssembly module as easily as an ES module.

---

which means the variables defined inside an ES module are not visible in other modules unless they are explicitly exported.
This provides a clear boundary between the various parts of a codebase, making it easier to manage and reuse code in a scalable way. ES modules are supported natively in modern web browsers and in Node.js, the popular JavaScript runtime environment.

A comparison between WebAssembly and JavaScript

Among the pros of the usage of Wasm over Javascript we have:

- Performance: thanks to static typing and ahead of time compilation. Code is highly optimized before reaching the browser, where it executes at near-native speed. Its binary files are considerably smaller than JavaScript's, resulting in significantly faster loading times.
- Cross-Platform Support: As stated earlier, one of the biggest drivers for WASM adoption is that developers can write code for the web in languages other than JavaScript and port existing applications over the web. Portability is a prominent feature of WebAssembly from the beginning, and this makes it worthwhile outside the browser powering efficient and performant applications on various operating systems (Windows, Linux, OSX, …) and architectures (Arm32/64, x64, RISC-V, …).
- Top-Notch Security**:** WebAssembly was built with security in mind. Its goal is to protect users from potential web insecurities while empowering developers to produce secure applications. WebAssembly provides a secure application experience by isolating module execution in a sandboxed environment while enforcing known browser security policies.

There are also some drawbacks to consider when adopting WebAssembly for browser applications:

- WASM is still in its early stages of development, and it will take some time before it builds the rich environment that JS had over 20 years to create. WebAssembly, for example, currently lacks document object model (DOM) and garbage collection features, and it still relies on JavaScript for full platform access.
- Imperfect Security**:** Although WASM was built with security in mind, some features make it useful for malicious attackers. Furthermore, while the sandbox feature was designed to contain exploits, recent findings have proven this to be not entirely accurate. These concerns might be linked to the teething problems of a new language. WebAssembly may prove to be the solution that minimizes browser-based attacks as more features are developed.

Why WASI?

Since WebAssembly provides a new fast, scalable, secure way to run the same code across all machines, Mozilla developers decided to push Wasm beyond the browser.

In 2019, therefore, they announced WASI, the WebAssembly System Interface, the begin of a new standardization effort.

WebAssembly was powerful but, at the time, it did not have a way to communicate with the underlying system, hence it needed a solid system interface to run outside the browser.

Just as WebAssembly is an assembly language for a conceptual machine, WebAssembly needs a system interface for a conceptual operating system, not any single operating system. This way, it can be run across all different OSs.

This is what WASI is: a system interface for the WebAssembly platform.

WASI: history, state of art and future

WASI started as a different name in 2016, CloudABI, just one year after the announcement of Wasm, as an attempt to create a portable system API.

After the release of WebAssembly MVP[4], developers started thinking more systematically about this concept and CloudABI was deprecated.

At that juncture, considerable effort was dedicated to modularization, which proved significant as diverse environments sought to utilize WebAssembly, with a limited capacity to expose all available interfaces under WASI. Consequently, the WASI specification was subdivided into multiple modules, such that platforms could only support the necessary APIs. When the initial snapshot of WASI was released in 2020, it included several crucial interfaces such as clocks, filesystems, networking, and arguments. However, it was inadequate because it lacked modularity. The current state of WASI development, as of February 2023, indicates that Networking, a crucial component, has some limitations in terms of functionality. The forthcoming versions of WASI are expected to evolve based on practical experience and

[4] Minimum Viable Product

feedback gathered from the initial release, thereby incorporating new features to address emerging use cases. It is also possible that these future releases may undergo substantial architectural modifications. The following image represents the hierarchy and the levels of abstraction in the WASI architecture at the state of art.

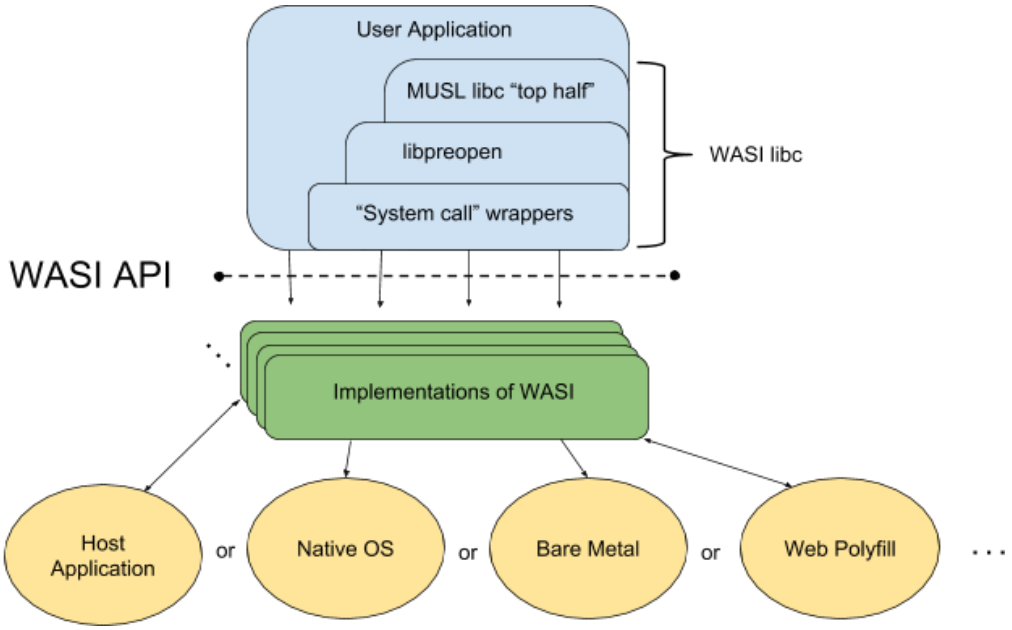


Figure 1: WASI software architecture

What is exactly a system interface?

Before jumping right into the WebAssembly System Interface, it is appropriate to explain in a detailed manner what a system interface is.

Every programming language, even low-level ones like the C language, are unable to have direct access to system resources such as opening and creating files or accessing memory. These resources are crucial for the stability and security of the system. If one program accidentally corrupts the resources of another, it could cause the program to crash. If a program or user intentionally tampers with the resources of another, it could result in the theft of sensitive data. As a result, a method is required to regulate which programs and users have access to which resources. Early on, a solution was developed to provide this control known as protection ring security[5].

[5] Protection ring security is a computer security model that uses multiple levels of privilege to protect resources and prevent unauthorized access. In this model, the operating system and software are divided into different rings or levels of access, with higher levels having more access and control than lower levels.

Protection ring security is a mechanism implemented by the operating system to safeguard system resources from unauthorized access. The kernel, which is the heart of the operating system, acts as a protective barrier around these resources, and is responsible for operations such as creating and opening files, and establishing network connections. User programs run outside the kernel, in a mode known as user mode, and must request access to resources via system calls. When a program needs the kernel to perform an operation on its behalf, it sends a system call, which allows the kernel to verify the user's identity and access rights before carrying out the requested operation. This ensures that programs cannot inadvertently or maliciously interfere with the resources of other programs or the system as a whole, thus enhancing system stability and security.

The good news is that you don't need to have different versions of your code for each operating system, even though each operating system may have its own set of system calls. This is because programming languages and compilers have created an abstraction layer between the code and the operating system. This layer provides a standard interface for system calls, so the same code can be compiled to work on different operating systems without having to rewrite the system call code. This approach allows for cross-platform compatibility and makes it easier for developers to create software that can run on multiple systems.

How is this problem solved? Abstraction.

The majority of programming languages offer a standard library that allows programmers to use a uniform interface without being aware of the targeted system. During the compilation process, the toolchain selects the appropriate implementation of the interface based on the system being targeted. This implementation utilizes functions from the operating system's API, making it specific to that particular system. This is where the system interface plays a role. As an example, when printf is compiled for a Windows machine, it utilizes the Windows API to interact with the machine. If a code is being compiled for Mac or Linux, it will use POSIX[6] instead. This poses a problem for WebAssembly, though.

When compiling code to WebAssembly, it's not possible to know which specific operating system the code will run on. Therefore, a WebAssembly implementation of a standard library cannot rely on any particular operating system's system interface. Instead, WebAssembly

[6] **P**ortable **O**perating **S**ystem **I**nterface for Uni**x** is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

requires a system interface for a conceptual operating system that is independent of any real operating system.

Fortunately, there exist runtimes that can execute WebAssembly code outside the web browser, even in the absence of a complete system interface. These runtimes offer a secure environment in which WebAssembly modules can operate with restricted access to system resources. As the WebAssembly technology continues to advance, there will likely be an increased demand for a more comprehensive system interface, and various initiatives are already underway to create and standardize such an interface.

Emscripten was the first tool for producing WebAssembly and it created its own implementation of libc[7] to emulate a particular OS system interface, POSIX, on the web. This implementation was split into two parts: one part was compiled into the WebAssembly module, and the other part was implemented in JS glue code. The JS glue code would call into the browser, which would then talk to the OS. However, before WASI, there was not a standard way to run Emscripten-compiled code outside the browser. People started creating their own runtime based on the functions in the glue code, but this was problematic because the interface provided by the JS glue code was not designed to be a standard or a public-facing interface. It was not intended to solve this problem. The image below (by Lin Clark) can give a playful idea of how these runtimes are designed. We need a WebAssembly ecosystem that lasts for decades; therefore, it must be built upon solid foundations. This means our de facto standard cannot be an emulation of an emulation.

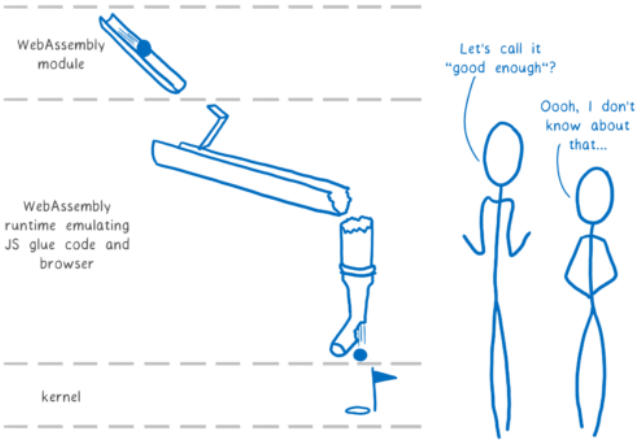


Figure 2: WASI predecessors architecture design (from https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/)

[7] The C Standard Library, often called libc, is a set of functions and macros that provide commonly used functionality to C programs. It provides a standardized and portable interface for interacting with the operating system and other system resources.

2. WASI

Design principles

WebAssembly has portability and security as two of its fundamental principles. These principles are crucial for developing and deploying software outside the browser, as well. With portability, it becomes simpler to create and distribute software, while providing tools for hosts to secure themselves or their users becomes an essential requirement.

POSIX offers source code portability by allowing developers to compile the same source code using different versions of libc to target various machines. However, WebAssembly needs to go one step further by allowing for compilation once and running across a wide range of machines. Portable binaries are necessary for this level of portability, making it easier to distribute code to users.

WebAssembly also takes security a step further than POSIX by providing a sandboxed environment for code execution. Sandboxing is a technique used to isolate code from the rest of the system to prevent it from accessing resources it should not. In the case of WebAssembly, the sandboxing is done at the binary level, which means that a WebAssembly module can be executed securely even if it comes from an untrusted source.

The WebAssembly runtime achieves this by enforcing a set of constraints on the module, including limits on memory access, I/O operations, and CPU usage. These constraints are enforced by the runtime and cannot be circumvented by the module. Additionally, WebAssembly uses a stack-based execution model, which makes it more difficult for malicious code to exploit vulnerabilities in the runtime or the host system.

However, sandboxing alone is not enough to ensure complete security. As mentioned earlier, WebAssembly code often relies on interacting with the host system, which can be a potential avenue for attacks. This is where the concept of "capabilities" comes in. Capabilities are a way to provide fine-grained control over what resources a WebAssembly module can access. By limiting the module's capabilities, the host can prevent it from accessing sensitive resources or performing malicious actions. If a library is granted access to perform any action on a system, such as opening files and sending them over a network, then a code utilizing this library can potentially cause severe damage.

WebAssembly approaches security differently through sandboxing. The sandboxing feature of WebAssembly prevents code from directly communicating with the OS. However, this raises the question of how code can interact with system resources. To address this, the host, whether it be a browser or a wasm runtime, provides functions within the sandbox that can be utilized by the code.

WASI is built using capability-based security principles. Access to external resources is always represented by handles, which are special values that are unforgeable[8].

WASI aims to eliminate ambient authorities by preventing the provision of a handle solely by supplying a user-controlled identifier such as a string that provides the name of a resource. Instead, the only ways to access resources are to be explicitly granted handles or to perform operations on handles that return new handles. This notion of "capability" differs from Linux capabilities or the withdrawn POSIX capabilities, which are per-process rather than per-resource.

Handles are represented as values of reference type, which are inherently unforgeable in WebAssembly. This means that they can be used to represent handles directly. However, some programming languages such as C, C++, and Rust primarily operate within linear memory and do not have a straightforward way to use references in normal code. Even if references become usable, it may not always be practical to use them without annotations. In such cases, references are stored in a table called a c-list, with integer indices into the table serving to identify resources that can be easily passed around or stored in memory. These indices are sometimes referred to as file descriptors, as they are similar to what POSIX uses that term for. Fortunately, there are tools available to simplify this process.

Integer indices themselves may be forgeable, but a program can only access handles within the c-list it has been granted access to. Thus, isolation can still be achieved between libraries that use integer indices internally by restricting access to each library's c-list. Instances can be given

[8] One of the key words that describes capabilities is *unforgeable*.
A pointer in C is forgeable, because untrusted code could cast an integer to a pointer, thus *forging* access to whatever that pointer value points to.
MVP WebAssembly does not have unforgeable references, but what we can do instead is just use integer values which are indices into a table that is held outside the reach of untrusted code. The indices themselves are forgeable, but the table is the thing which holds the actual capabilities, and its elements are unforgeable. There is no way to gain access to a new resource by making up a new index.
When the reference-types proposal lands, references will be unforgeable, and will subsume the current integer-based APIs, at the WASI API layer.

access to some c-lists and not others, or even no c-lists at all, so it is still possible to establish isolation between instances.

There are two levels of capabilities: static and dynamic. The static capabilities of a wasm module are its imports, which essentially declare the set of "rights" the module will be able to request. However, it is important to note that this does not consider capabilities that may be passed into an instance at runtime.

The dynamic capabilities of a wasm module are a set of Boolean values associated with a file descriptor, indicating individual "rights" such as the right to read or write using a given file descriptor.

filesystem hierarchy, and this is granted based on the system's security policies. While this doesn't violate the capability model, it doesn't fully leverage its benefits. In contrast, some capability-oriented systems make use of the hierarchical nature of the filesystem by requiring untrusted code to possess a capability for a directory to access the files within that directory.

However, sockets are not naturally hierarchical, so developers will need to figure out how to incorporate capabilities for sockets. While this is currently an unimplemented area, one possible approach is for users to launch programs with the sockets they need already created, which could work for simple use cases. Overall, the goal is to use capability-based security to limit a program's actions on a program-by-program basis, rather than allowing it to act on behalf of the user with full permissions for all system calls.

In CloudABI, programs are launched with the sockets they need already created, which is a potential starting point for capability-based security. This approach enables the host to limit the program's capabilities on a program-by-program basis, rather than allowing it to call any system call with the user's full permissions.

However, it's important to note that just having a mechanism for sandboxing does not make a system secure by itself. The host can still choose to put all the capabilities into the sandbox, which would not improve the security of the system. Nonetheless, having the option of capability-based security gives the host the opportunity to create a more secure system.

Another important design principle behind WASI is its interposition[9]. Interposition in the context of WASI interfaces is the ability for a WebAssembly instance to implement a given

[9] Interposition is sometimes referred to as "virtualization", however we use "interposition" here because the word "virtualization" has several related meanings.

WASI interface, and for a consumer WebAssembly instance to be able to use this implementation transparently. This can be used to adapt or attenuate the functionality of a WASI API without changing the code using it.

In WASI, we envision interposition will primarily be configured through the mechanisms in the module linking link-time virtualization. Imports are resolved when a module is instantiated, which may happen during the runtime of a larger logical application, so we can support interposition of WASI APIs without defining them in terms of explicit dynamic dispatch mechanisms.

Compatibility with existing applications and libraries, as well as existing host platforms, is important, but will sometimes conflict with overall API cleanliness, safety, performance, or portability. Where practical, WASI seeks to keep the WASI API itself free of compatibility concerns, and provides compatibility through libraries, such as WASI libc, and tools. This way, applications which do not require compatibility for compatibility' sake are not burdened by it.

Portability is important to WASI; however, the meaning of portability will be specific to each API.

WASI's modular nature means that engines do not need to implement every API in WASI, so we do not need to exclude APIs just because some host environments cannot implement them.

Taking a detailed look at the file system is a worthwhile exercise as it is not shared among different modules. Instead of the host providing the file system, a compatibility layer is used and the module virtualizes its own file system. The files to be accessed are stored in the linear memory of the wasm module, which eliminates the global shared mutable state problem that the traditional file system introduces.

The file system in wasm modules is not shared among different modules, and a compatibility layer is used to virtualize it. Rather than the host providing the file system, the module virtualizes its own file system, with files stored in the linear memory of the wasm module. This avoids the global shared mutable state problem introduced by traditional file systems. While files are treated in the source code as they would be on a native system, they use WASI I/O types for portability. However, these virtualizations can cause performance inefficiencies and large file sizes. To achieve both portability and efficiency, the developer can use the WASI I/O API and pass I/O types instead of files. This allows the code to be run anywhere, as every

system can represent these basic primitive types. As a result, the per-module file system overhead and global shared mutable state issues are eliminated.

Implementation

As previously mentioned, WASI is a relatively new technology and is still undergoing development. As a result, certain features such as file locking, file change monitoring, scalable event-based I/O, and crash recovery are not yet available. Networking is also a critical area of concern for WASI, with only a few primitives currently available including sock_recv(), sock_send(), sock_close(), poll_oneoff(), and more recently, sock_accept(). Consequently, creating new listeners or outgoing connections is currently not possible. The limitations stem from the fact that network namespaces are useful for isolating resources for different processes, whereas WASI's nano process model takes isolation a step further by defining the capabilities of guest Wasm modules loaded in as third-party libraries. This allows for the restriction of a module to make network calls to only a certain host, while another module can only call another host.

Unlike BSD sockets[10], WASI sockets require capability handles to create sockets and perform domain name lookups. On top of capability handles, WASI Socket implementations should implement deny-by-default firewalling.

Use cases

When WASI was announced, a set of high-level goals were established:

- Define a collection of portable, modular, runtime-independent, and WebAssembly-native APIs which can be used by WebAssembly code to interact with the outside world. These APIs preserve the essential sandboxed nature of WebAssembly through a Capability-based API design.
- Specify and implement incrementally. Start with a Minimum Viable Product, then adding additional features, prioritized by feedback and experience.

[10] also called Berkeley sockets

- Supplement API designs with documentation and tests, and, when feasible, reference implementations which can be shared between wasm engines.
- Make a great platform:
    - Work with WebAssembly tool and library authors to help them provide WASI support for their users.
    - When being WebAssembly-native means the platform is not directly compatible with existing applications written for other platforms, design to enable compatibility to be provided by tools and libraries.
    - Allow the overall API to evolve over time; to make changes to API modules that have been standardized, build implementations of them using libraries on top of new API modules to provide compatibility.

All the above open several opportunities and use cases for developers who want to make use of WASI:

- Command Line Tools: WASI can be used to write command line tools that run in the browser, such as file compression utilities, text editors, and system utilities. WASI provides access to low-level system calls, such as file I/O, networking, and process management, making it possible to write command line tools that run in the browser with the same functionality as native command line tools.
- Game Development: WASI can be used to write games that run in the browser as well as on the operating system, providing access to low-level system calls for graphics, audio, and input. This enables developers to write high-performance games that run in the browser, without having to use proprietary technologies like Flash or Silverlight.
- Scientific Computing: WASI can be used to write scientific applications, providing access to low-level system calls for numerical computing, matrix operations, and parallel processing. This enables researchers to run scientific simulations and computations, without having to worry about platform compatibility or security issues.
- Web Services: WASI can be used to write web services that run in the browser, providing access to low-level system calls for networking, process management, and file I/O. This enables developers to write server-side code that runs in the browser, without having to worry about deploying and maintaining a server.
- Embedded Systems: WASI can be used to write applications for embedded systems, such as Internet of Things (IoT) devices, providing a web-based runtime environment that is portable, secure, and energy-efficient.

These are just a few examples of the many potential use cases of WASI. The objective is to demonstrate the versatility and practicality of WASI as a technology for system programming on the web.

Development tools

There are a few development tools for Wasm and WASI out there. One of the most popular for Wasm is Emscripten, an LLVM[11]/Clang[12]-based compiler that compiles C and C++ source code to WebAssembly. Emscripten allows applications and libraries written in C or C++ to be compiled ahead of time and run efficiently in web browsers, typically at speeds comparable to or faster than interpreted or dynamically compiled JavaScript. It even emulates an entire POSIX operating system, enabling programmers to use functions from the C standard library (libc).

Another popular tool for Wasm and WASI is Wasmtime, a Bytecode Alliance project that is a standalone wasm-only optimizing runtime for WebAssembly and WASI. It runs WebAssembly code outside of the Web and can be used both as a command-line utility or as a library embedded in a larger application.

Wasmtime strives to be a highly configurable and embeddable runtime to run on any scale of application, although many features are still under development, as we could expect.

Wasmtime is just a runtime and does not supply any build tool. For compiling, we could use either Emscripten or the C compiler provided in the WASI-SDK, which includes a build of WASI Libc in its sysroot.

Another viable option for WASI runtime is the WASI API embedded in the latest versions of Node. It's still in the early stages of development, thus it could not be suitable for a production application, but neither is WASI. The WASI class provides the WASI system call API and additional convenience methods for working with WASI-based applications. Each WASI instance represents a distinct sandbox environment. For security purposes, each WASI instance

[11] LLVM is a set of compiler and toolchain technologies that can be used to develop a front end for any programming language and a back end for any instruction set architecture. LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes.
LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization.
[12] Clang is a compiler front end for C and C-like languages, written in C++. It is a subproject of LLVM; thus, it is free and open-source software.

must have its command-line arguments, environment variables, and sandbox directory structure configured explicitly.

Runtime

Wasmtime is a runtime environment for WebAssembly that includes support for WASI. Wasmtime provides a secure and efficient way to run WebAssembly modules on a variety of platforms, including desktop and mobile devices, cloud services, and embedded systems.

Here is how the Wasmtime runtime for WASI works:

1. Loading the Wasm module: Wasmtime loads the WebAssembly module into memory and validates it to ensure that it is well-formed and valid.
2. Initializing the WASI environment: Wasmtime initializes the WASI environment by creating a WASI instance, which provides access to low-level system calls for process management, file I/O, and networking.
3. Running the Wasm module: Wasmtime runs the WebAssembly module by executing its instructions in a sandboxed environment. The WASI instance is used to interact with the underlying operating system, by making system calls to perform tasks such as reading and writing files, creating, and managing processes, and opening network sockets.
4. Interacting with the Wasm module: Wasmtime provides APIs for interacting with the WebAssembly module from the host environment, such as passing arguments and returning values, and for accessing its memory and exports.
5. Managing the runtime: Wasmtime provides APIs for managing the runtime environment, such as starting and stopping the execution of the WebAssembly module, and for monitoring its resource usage and performance.

The Wasmtime runtime for WASI is designed to be lightweight, efficient, and portable, and to provide a secure and sandboxed environment for executing WebAssembly modules outside of the browser. It is being actively developed and maintained as an open source project and is available for use in a wide range of applications and use cases.

During the execution of the module, an interesting aspect to be explained in a more detailed manner is the threading model. WASI provides a threading model that allows WebAssembly modules to create and manage threads in a secure and portable manner. The threading model is

designed to be lightweight, efficient, and platform-independent, and to provide a secure and sandboxed environment for executing threaded WebAssembly modules.

The WASI threading model is based on the concept of "threads" and "synchronization primitives". A thread is a lightweight execution context that shares the same memory space as the other threads in the same WebAssembly module and can run in parallel with them. Synchronization primitives are objects that are used to coordinate the access to shared resources between threads, such as locks, semaphores, and condition variables.

WASI threading model provides a set of system calls that can be used by WebAssembly modules to create and manage threads and synchronization primitives, such as wasi_snapshot_preview1::sched_yield, wasi_snapshot_preview1::mutex_*, wasi_snapshot_preview1::condvar_*, and wasi_snapshot_preview1::sem_*.

WASI threading model is designed to be compatible with the threading models of different programming languages and runtime environments, such as Rust, C, and Java. It is also designed to be portable across different operating systems and hardware platforms, and to provide a consistent and predictable behavior for threaded WebAssembly modules, regardless of the environment in which they are executed.

WASI threading model is implemented in the Wasmtime runtime environment.

In Wasmtime, the implementation of the WASI threading model is based on the native threading primitives of the host environment, such as POSIX threads (pthreads) on Unix-like systems and Windows threads on Windows systems. When a WebAssembly module calls a threading-related system call provided by the WASI API, Wasmtime maps the call to the corresponding native threading primitive and performs the requested operation.

For example, when a WebAssembly module calls the wasi_snapshot_preview1::sched_yield system call to yield the current thread and allow another thread to run, Wasmtime calls the native pthread_yield or Sleep function to perform the operation, depending on the host operating system.

Wasmtime also provides a set of APIs that allow WebAssembly modules to interact with the threading model from the host environment. These APIs include functions for creating and joining threads, creating and managing synchronization primitives, and setting thread-local data.

Overall, the implementation of the WASI threading model in Wasmtime is designed to be efficient, secure, and portable, and to provide a seamless integration with the native threading primitives of the host environment. It allows WebAssembly modules to create and manage threads in a sandboxed and predictable manner, and to run in parallel with other threads in the same module or in the host environment.

Hello, World

It is quite hard to pick up these concepts without a practical demonstration. Let us jump right into it. All the following operations have been performed on a Virtual Machine running Ubuntu 22.04 LTS.

First things first, we need to set-up our environment, following the subsequent steps:

1. Install a stable version WASI-SDK, which will also include the clang compiler. If any error occurs, check the dependencies, including cmake and ninja.
2. Install a runtime, in this case Wasmtime.
3. Install a recent version of Node, v19.2.0 will be used.

To introduce the capability system and demonstrate how it works, it's more effective to provide a simple demo application than a "Hello, World!" program. A basic C program called "demo.c" will be used to showcase how to compile and run programs and configure simple sandboxes. This program performs a file copy using standard POSIX APIs, without any knowledge of WASI, WebAssembly, or sandboxing.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

int main(int argc, char **argv) {
  ssize_t n, m;
  char buf[BUFSIZ];

  if (argc != 3) {
    fprintf(stderr, "usage: %s <from> <to>\n", argv[0]);
    exit(1);
  }

  int in = open(argv[1], O_RDONLY);
  if (in < 0) {
    fprintf(stderr, "error opening input %s: %s\n", argv[1], strerror(errno));
    exit(1);
  }

  int out = open(argv[2], O_WRONLY | O_CREAT, 0660);
  if (out < 0) {
    fprintf(stderr, "error opening output %s: %s\n", argv[2], strerror(errno));
    exit(1);
  }

  while ((n = read(in, buf, BUFSIZ)) > 0) {
    char *ptr = buf;
    while (n > 0) {
      m = write(out, ptr, (size_t)n);
      if (m < 0) {
        fprintf(stderr, "write error: %s\n", strerror(errno));
        exit(1);
      }
      n -= m;
      ptr += m;
    }
  }

  if (n < 0) {
    fprintf(stderr, "read error: %s\n", strerror(errno));
    exit(1);
  }
  return EXIT_SUCCESS;
}
```

Let us compile to WASM using the clang compiler included in the WASI-SDK:

```
$ clang demo.c -o demo.wasm
```

We can now execute our artifact using Wasmtime but let us first create a test file to be copied.

```
$ echo hello world > test.txt
$ wasmtime demo.wasm test.txt /tmp/somewhere.txt
error opening input test.txt: Capabilities insufficient
```

Now we can see the sandboxing in action. This program is attempting to access a file by the name of test.txt, however it has not been given the capability to do so.

Let us try again, but giving it the required capabilities:

```
$ wasmtime --dir=. --dir=/tmp demo.wasm test.txt /tmp/somewhere.txt
$ cat /tmp/somewhere.txt
hello world
```

Now it runs as expected: the “--dir” option instructs Wasmtime to preopen a directory and make it available to the program as a capability which can be used to open files inside that directory. As a brief aside, note that we used the path “.” to grant the program access to the current directory. This is needed because the mapping from paths to associated capabilities is performed by libc, so it's part of the WebAssembly program, and we do not expose the actual current working directory to the WebAssembly program. So, we always must use “.” to refer to the current directory. This may seem vulnerable to privilege escalation, but if the program attempts to access the parent directory using “..”, the sandbox will deny it.

We can note the error message was “Capabilities insufficient”, rather than Unix access controls ("Permission denied"). Even if the user running Wasmtime had write access to /etc/passwd, WASI programs do not have the capability to access files outside of the directories they've been granted. This is true when resolving symbolic links as well.

What if we wanted to run this WASM program inside a Node server application? We can certainly do that. Here is a simple JS module instantiating a WASI object and running it.

We can notice it is also possible to map directories to sandbox paths, in this case, “.” was mapped to “/sandbox”.

The following snippet was saved to “index.mjs”:

```
import { readFile } from 'node:fs/promises';
import { WASI } from 'wasi';
import { argv, env } from 'node:process';

const wasi = new WASI({
args: ["node --experimental-wasi-unstable-preview1 index.mjs",
       "/sandbox/test.txt", "/sandbox/test.copy"],
  env,
  preopens: {
    '/sandbox': '.'
  }
});

// Some WASI binaries require:
// const importObject = { wasi_unstable: wasi.wasiImport };
const importObject = { wasi_snapshot_preview1: wasi.wasiImport };

const wasm = await WebAssembly.compile(
  await readFile(new URL('./demo.wasm', import.meta.url))
);
const instance = await WebAssembly.instantiate(wasm, importObject);

wasi.start(instance);
```

Let us run this Node module:

```
$ node --experimental-wasi-unstable-preview1 index.mjs
$ cat test.copy
(node:4908) ExperimentalWarning: WASI is an experimental feature and might
change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
hello world
```

Node will warn us that WASI is still in an experimental stage of development, which means it is not safe to use this code in production. Although, Node will still run our application successfully.

Throughout Node documentation are indications of a section's stability. Some APIs are so proven and so relied upon that they are unlikely to ever change at all. Others are brand new and experimental, or known to be hazardous. The stability indices are as follows:

0. Deprecated. The feature may emit warnings. Backward compatibility is not guaranteed.

1. Experimental. The feature is not subject to semantic versioning rules. Non-backward compatible changes or removal may occur in any future release. Use of the feature is not recommended in production environments.
2. Stable. Compatibility with the NPM[13] ecosystem is a high priority.
3. Legacy. Although this feature is unlikely to be removed and is still covered by semantic versioning guarantees, it is no longer actively maintained, and other alternatives are available.

Performance

As an example, a simple, yet computationally heavy, program has been used. The same program has been written in C and JavaScript, compiled to native (elf[14]) executable and to Wasm, then various statistics have been measured.

The assignment was the calculation of a square matrix determinant, using the LU decomposition[15] algorithm. Here is a code snippet of the two functions that have been implemented. The first one takes in 4 parameters: the matrix, its size, the tolerance to detect failure when the matrix is degenerate and an integer array with a n + 1 length which will contain column indexes where the permutation matrix has “1”. The second one takes as an input the manipulated matrix, the integer array, and the size, and returns the determinant.

[13] NPM stands for "Node Package Manager". It is a package manager for the JavaScript programming language and is primarily used to manage dependencies in Node.js projects. NPM provides a command-line interface for developers to install, share, and manage packages of code that can be used in Node.js projects.
[14] The Executable and Linkable Format (ELF) is a file format used for executables, object code, shared libraries, and core dumps. ELF is used on most Unix-like operating systems. An ELF file consists of a header and one or more sections, each containing program code, data, or other information.
[15] In numerical analysis and linear algebra, lower–upper (LU) decomposition or factorization factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. It is also a key step in computing the determinant of a matrix.

```c
int lu_decompose(double **a, int n, double tol, int *p)
{
  int i, j, k, imax;
  double maxA, *ptr, absA;

  for (i = 0; i <= n; i++)
    p[i] = i; // Unit permutation matrix, p[n] initialized with n

  for (i = 0; i < n; i++) {
    maxA = 0.0;
    imax = i;

    for (k = i; k < n; k++)
      if ((absA = fabs(a[k][i])) > maxA)
        maxA = absA;
        imax = k;
      }

    if (maxA < tol)
      return 0; // Failure: matrix is degenerate

    if (imax != i){
      // Pivoting p
      j = p[i];
      p[i] = p[imax];
      p[imax] = j;

      // Pivoting rows of a
      ptr = a[i];
      a[i] = a[imax];
      a[imax] = ptr;

      // Counting pivots starting from n (for determinant)
      p[n]++;
    }

    for (j = i + 1; j < n; j++){
      a[j][i] /= a[i][i];

      for (k = i + 1; k < n; k++)
        a[j][k] -= a[j][i] * a[i][k];
    }
  }

  return 1; // decomposition done
}
```

```c
double lu_determinant(double **a, int *p, int n)
{
  double det = a[0][0];

  for (int i = 1; i < n; i++)
    det *= a[i][i];

  return (p[n] - n) % 2 == 0 ? det : -det;
}
```

Javascript version is a simple translation of this code, therefore it is less significant to show.

Let us first compare the size of the artifacts: ELF binary is only 16.4 KB, while Wasm is 31 KB, almost double the size. JS size is a little out of context, cause it's just-in-time compiled, and we can only measure the source code size, which is 2 KB in this case.

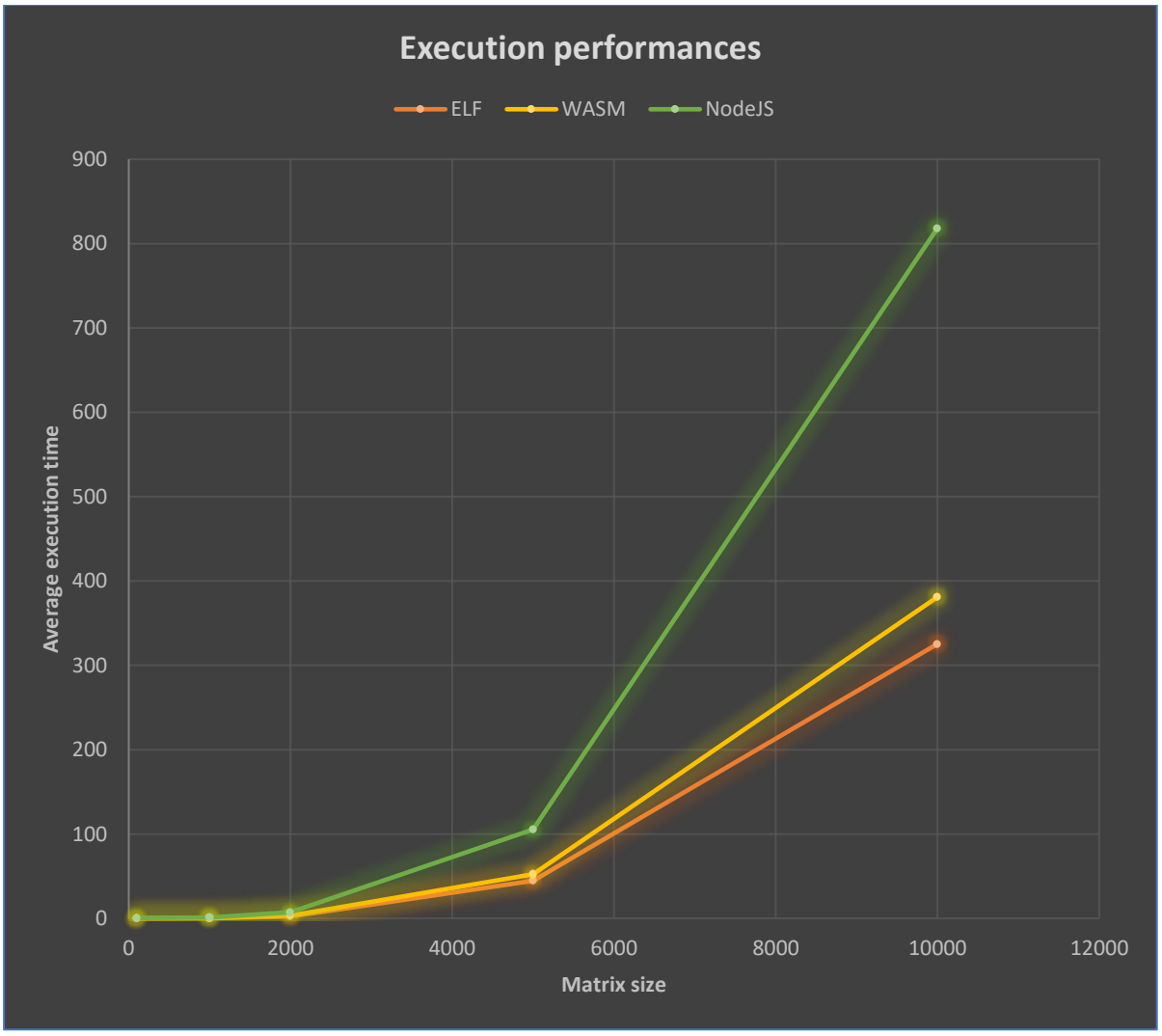
Every script takes one command line parameter as an input, which dictates the dimensions of the square matrix of which the determinant will be calculated. This allows to create a better structure of benchmark tests[16].

The most meaningful parameter to be compared is the execution time of the task based on input, that is, in this case, the size of the matrix.

Here are the data, both in a tabular and in a graphical view, to get a better idea of the big picture.

		Technology		
		ELF	WASM	NodeJS
Matrix size	100	0,001	0,067	0,402
	1000	0,275	0,338	0,922
	2000	2,324	3,369	7,186
	5000	44,728	52,372	105,418
	10000	325,396	381,045	818,37

[16] The following benchmark tests have been performed 3 times and the results extracted are the average values. For each of them, the input matrix was randomly generated and filled with doubles from 0 to 1.
The results of degenerate matrixes have not been considered.



In the first row we can notice a substantial execution time difference due to the startup of the virtual machines for WASM and Node, while ELF can run directly on the operating system.

In the next rows this percentual difference gets flatter, stabilizing the execution time ratio of WASM and ELF between 1.17 and 1.45, while the Node-ELF ratio swings between 2.35 and 3.35, thanks to the optimizations inside the V8 engine[17].

Two other interesting parameters to observe are CPU and memory usage.

For what concerns CPU usage, it is up to task manager, in this case, launching the 3 tasks simultaneously, they each use around 33% of CPU, as they have the same scheduling priority.

Regarding RAM usage, we need to reason about data size first. Since a double occupies 8 bytes of memory, letting N the matrix row/column count, the minimum RAM usage per test will be

[17] V8 is Google's open-source high-performance JavaScript and WebAssembly engine, written in C++. It is used in Chrome and in Node.js, among others.

$8 \times N^2$. At runtime, Elf takes up less than 1 KB other than the matrix, thus it will be used as a reference for other measurements.

Across the tests, RAM usage was linear with N, but it can be approximated to a direct proportionality, being the constant term < 1% for N > 100.

We can evince that Node uses ~140% memory compared to Elf and WASM uses only ~104.6%.

3. Project: WASI for image manipulation

Overview

In this chapter, we present a demo project that leverages the power of WASI and Node.js to create a web application for image processing. The application consists of a simple front-end made with HTML/CSS/JS/React and a Node/Express server that creates a new WASI instance for each incoming request. The server receives an image and a processing task as input, performs the processing on the image using the STB image library, and returns the processed image to the client.

The demo project showcases the benefits of using WASI for building web applications, including the ability to create portable and secure applications that can run on different platforms and environments. By using a Node server with WASI, we can take advantage of the performance benefits of WebAssembly while still providing a familiar programming environment for developers.

In the following sections, we will provide a detailed overview of the technologies used in the demo project, including WebAssembly, WASI, and the STB image library. We will then describe the design and implementation of the web application, including the front-end and server-side components. Finally, we will present the results of our evaluation, including performance benchmarks and a discussion of the trade-offs between performance and portability in a WASI-based application.

Overall, this demo project provides a proof-of-concept for using WASI to build web applications and demonstrates the potential for leveraging the power of WebAssembly in combination with existing web technologies to create high-performance, portable applications.

Background

As software stack for this demo project, we have Node.js with the Express.js framework, WASI, of course, with the STB C library for image processing. The front-end will be powered by React, for its ease of use and time-to-market.

Even though Node does not really need an introduction, a short one will be provided to let the reader in the key concepts and the reasons behind this choice.

Node.js, often simply called Node, is an open-source, cross-platform, runtime environment that allows developers to execute JavaScript code outside of a web browser. It was created by Ryan Dahl in 2009 and has since gained widespread popularity among developers for its ability to build fast, scalable, and real-time applications.

Node is built on top of the V8 JavaScript engine, which is the same engine used by Google Chrome. It allows developers to use JavaScript on the server-side, which was previously only possible on the client-side within web browsers.

Node has many built-in modules, including modules for file input/output (I/O), networking, and encryption, among others. It also has many third-party modules available through the Node Package Manager (npm), which is the largest ecosystem of open-source libraries in the world.

One of the key features of Node is its non-blocking I/O model, which allows it to handle many concurrent connections without blocking the event loop. This makes it particularly well-suited for building real-time applications, such as chat applications or online games.

Node is also often used as a backend technology for web applications, particularly with frameworks such as Express.js, which provides a lightweight and flexible way to build web applications and APIs.

Overall, Node is a powerful and versatile technology that has revolutionized the way that web applications are built and deployed. Its popularity is a testament to the growing importance of JavaScript as a programming language for both front-end and back-end development.

For performance intensive tasks, we will use C and the STB library. The reasons behind these choices are performance and simplicity. With its limited syntax and raw memory management, the C programming language is suitable for a high-performance small task like image manipulation.

STB is a collection of single-file libraries written in C that provide a range of useful functionality for graphics, audio, and other multimedia applications.

Although STB might sound like “Set-Top Boxes” or something, they are just the initials of the name of the main maintainer, Sean T. Barrett. This was not chosen out of egomania, but as a moderately sane way of namespacing the filenames and source function names.

The libraries are designed to be lightweight, easy to integrate, and portable across a range of platforms, making them an ideal choice for developers who want to quickly add multimedia features to their applications.

The STB libraries cover a wide range of functionality, including image loading and manipulation, font rendering, audio playback, and 3D graphics. They are designed to be used in a variety of contexts, from desktop and mobile applications to video games and other interactive multimedia projects. One of the key features of STB is its simplicity. Each library is contained within a single C file, making it easy to integrate into existing projects or to use as a standalone library. The idea behind single-header file libraries is that they're easy to distribute and deploy because all the code is contained in a single file. By default, the .h files in here act as their own header files, i.e., they declare the functions contained in the file but don't actually result in any code getting compiled.

One of the key features is the capability of outlining an image as an array of unsigned characters[18], each representing the value of the corresponding channel. For instance, if an image has 3 channels (RGB), in the STB schematization, every group of 3 elements corresponds to a pixel.

STB includes a range of libraries that are specifically designed for image processing, making it a popular choice for developers who need to work with images in their applications. These libraries provide functionality for loading and saving images in JPG or PNG format, as well as for manipulating and transforming images using a range of filters and effects. There is no intention to add any more image types: as stb_image use has grown; it has become important to focus on security of the codebase. Adding new image formats increases the amount of code that needs to be secured. The libraries are designed to be lightweight and fast, making them well-suited for real-time image processing applications such as video games or interactive multimedia projects.

STB libraries are distributed in the public domain, which means anyone can do anything with them and has no legal obligation.

[18] Corresponding to the uint8_t typedef in the stdint.h library.

Design and Implementation

The goal of this paragraph is to show how WASI can be integrated with established Web technologies in a practical manner. To achieve this, we will walk through the development of an application step-by-step. Additionally, we have included a diagram of the application we will be building, which illustrates the positioning of each technology.

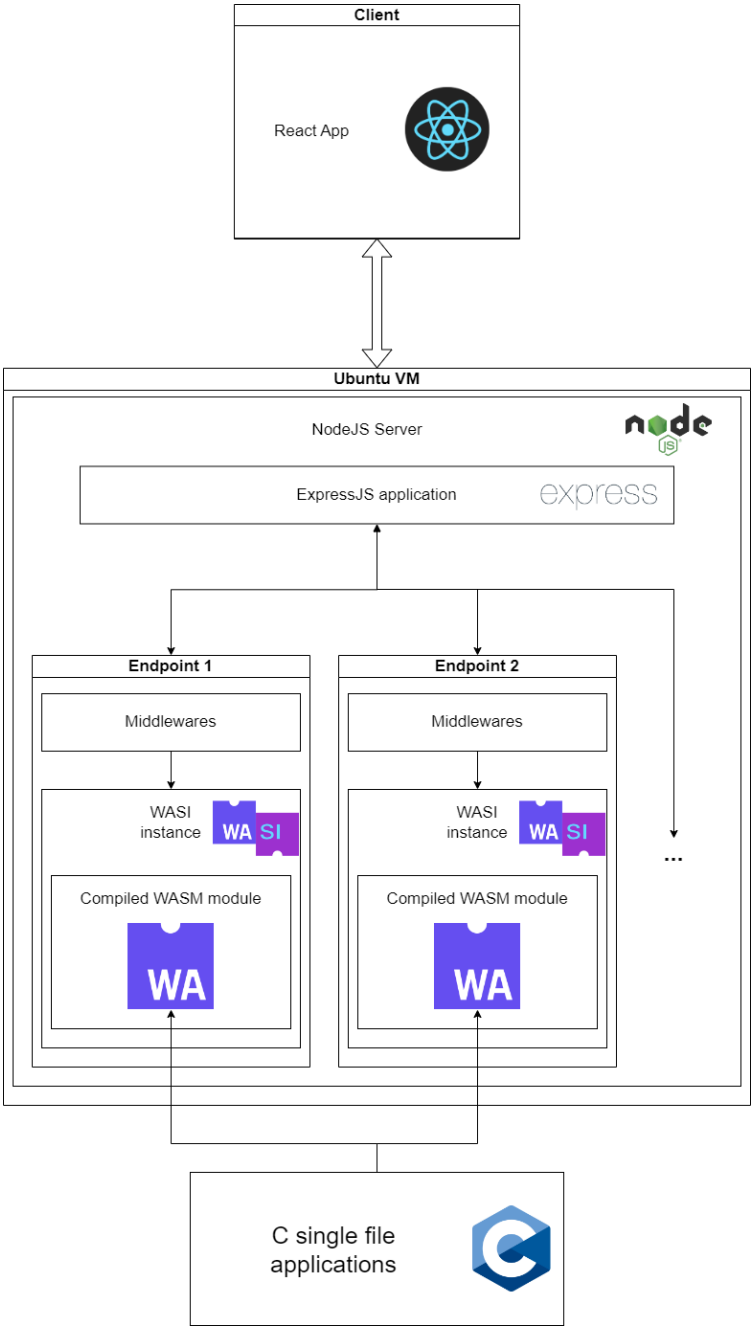


Figure 1: Application diagram

To begin, let us prepare our environment. We performed all the following operations on an Ubuntu LTS Virtual Box VM, where Node, NPM, and Yarn were already installed. Once we navigate to the project directory, we can initialize it, which will automatically create the package.json file. Then, we can proceed to initialize the client application.

```
# init the server
yarn init
# init the client
yarn create react-app client
# test the creation of the client
cd client && yarn start
```

Let us check on the browser window that will be opened, the result should look like this:

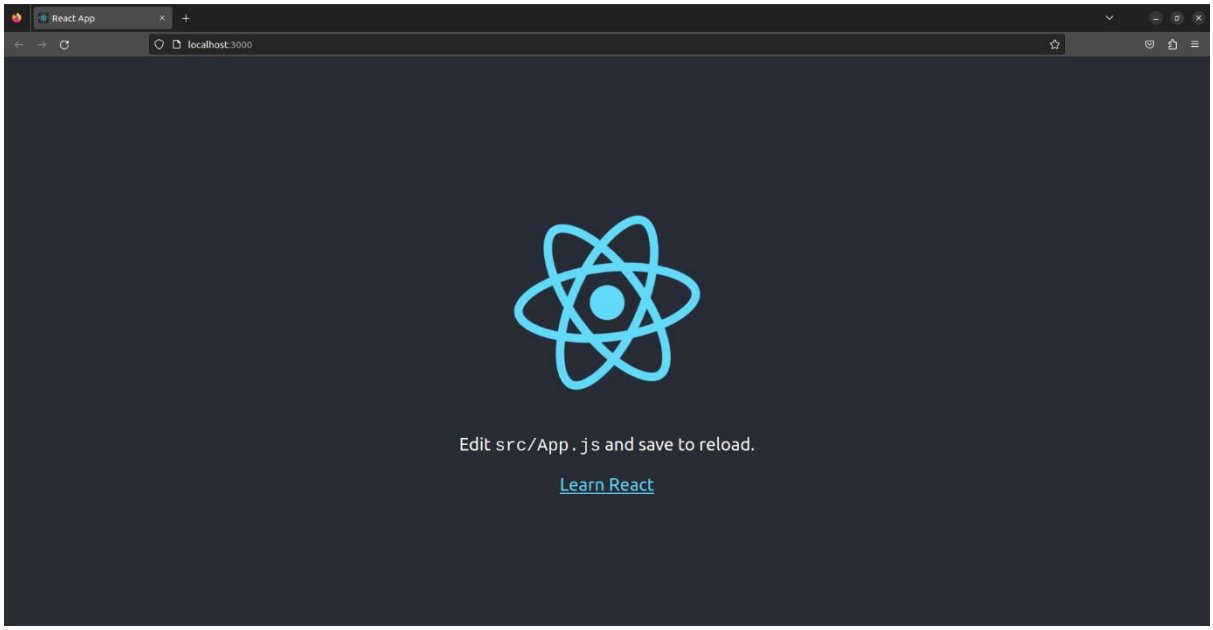


Figure 2: yarn start output

Let us return to parent directory and create some useful folders

- “c” will contain the .c and .h files including the STB library
- “bin” will contain the WASM artifacts
- “requests” will contain the endpoints request callbacks

```
cd ..
mkdir c bin requests
```

Let us create the server root file, the "mjs" extension stands for JS module, then open the folder with a suitable text editor, in this case Visual Studio Code.

```
touch index.mjs
code .
```

Insert some boilerplate code for our express application:

```
import express from "express";
import bodyParser from "body-parser";
import cors from "cors";
import multer from "multer";

import { tmpFolder } from "./requests/utils.mjs";

const port = 5000;
const app = express();

// Middlewares
app.use(cors());
app.use(multer({ dest: tmpFolder }).single("image"));
app.use(bodyParser.json({ limit: "5mb" }));
app.use(bodyParser.urlencoded({ extended: true, limit: "5mb" }));

// Serve React client
app.use(express.static("client/build"));

app.listen(port);
```

If you have seen a Node/Express application before, you should be quite familiar with this type of structure. An interesting note is about middlewares:

- cors allows cross-origin requests, this might not be needed in production, but in a testing environment it can be comfortable for testing front-end and back-end at the same time.
- multer helps handling file uploads, saving the uploaded file to a temporary file on the server.
- bodyParser converts JSON formatted text to an actual JS object.

It is important to note that more middlewares might introduce a delay per request.

Before testing our server application starter, we need to compile an optimized build for the client, which will be served at port 5000.

```
cd client
yarn build
cd ..
node server.mjs
```

If we open a browser tab and connect to localhost:5000, we should see something similar to Figure 2.

The approach in this section will be modular, to create a structure which presents the least redundancy possible with reusable pure functions. This applies both for Javascript and C code.

Starting with Javascript, we will have a utilities file including all the functions which will be used and reused across the application, named “utils.mjs” inside the requests folder. We will create one function per endpoint, so that it will be easy to debug in case of problems.

Here is the documented code for utils (imports excluded):

```
export const tmpFolder = "/tmp/";

/**
 * Middleware for validating input image (if any).
 * @param {Request} req: Express request
 * @param {Response} res: Express response
 * @param {import('express').NextFunction} next: Express next function
 */
export const validateImage = (req, res, next) => {
  const imageFile = req.file;

  // Check if there's a file in the request body
  if (!imageFile) {
    return res.status(400).json({ error: "No file uploaded." });
  }

  // Check if the uploaded file is a JPEG or PNG image
  if (
    imageFile.mimetype !== "image/jpeg" &&
    imageFile.mimetype !== "image/png"
  ) {
    return res
      .status(400)
      .json({
           error: "Invalid file type. Please upload a JPEG or PNG image."}
       );
  }

  next();
};
```

```
/**
 * Wrapper function for running WASI with given arguments
 * @param {String} filename: WASM file name without extension
 * @param  {...String} wasmArgs: WASM arguments (including input and output file paths)
 */
export const runWasi = async (filename, ...wasmArgs) => {
  const wasi = new WASI({
    args: [argv[0], ...wasmArgs],
    env,
    preopens: {
      "/tmp": "/tmp",
    },
  });

  const importObject = { wasi_snapshot_preview1: wasi.wasiImport };
  const wasm = await WebAssembly.compile(
    await readFile(new URL(`../bin/${filename}.wasm`, import.meta.url))
  );
  const instance = await WebAssembly.instantiate(wasm, importObject);
  wasi.start(instance);
};

/**
 * Wrapper function for runWasi function, which handles the tempfile and arguments from request.
 * @param {Request} req: request object from express
 * @param {Response} res: response object from express
 * @param {String} filename: WASM file name without extension
 * @param {...String} otherArgs: additional args for WASI instance, other than input and output paths
 */
export const handleRun = async (req, res, filename, ...otherArgs) => {
  const imagePath = req.file.path;
  const output = tmp.fileSync({
    mode: 0o644,
    postfix: req.file.mimetype === "image/jpeg" ? ".jpg" : ".png",
  }).name;
  await runWasi(filename, imagePath, output, ...otherArgs);

  // Read the uploaded file from disk and send it back to the client
  fs.readFile(output, (err, data) => {
    if (err) res.status(500).json({ error: "Error reading file." });
    res.contentType(req.file.mimetype);
    res.send(data);
    fs.unlink(output, (err) => {
      if (err) console.error(`Error deleting file ${output}:`, err);
    });
  });
};
```

This library will reduce the development time and code for each endpoint, which will look like the following:

```
import { handleRun } from "./utils.mjs";

export const toGrayScale = async (req, res) => {
  await handleRun(req, res, "gray_scale");
};
```

Exceptionally clean, all the dirty work has been done in utils. This function is self-explanatory, it will be connected to an endpoint for black and white (gray scale) image conversion.

A similar approach will be applied to the C part of the project: the creation of a user library will enhance the production of complex tasks.

First, as it lives inside the “c” folder, we need to clone the STB library here, copy the needed files and remove the repository folder.

```
cd c
git clone git@github.com:nothings/stb.git
mv stb stb_image
mkdir stb
cp stb_image/*.h stb_image/*.c stb
rm -r stb_image
```

Now we can create our base library files: utils.h for common utility functions, Image.h and Image.c. Image will contain a structure to abstract the image type and a collection of basic functions for loading, creating, saving, and freeing RAM. Here is the header file:

```
enum allocation_type {
  NO_ALLOCATION, SELF_ALLOCATED, _ALLOCATED
};

typedef struct {
  int width;
  int height;
  int channels;
  size_t size;
  uint8_t *data;
  enum allocation_type allocation_;
} Image;

void Image_load(Image *img, const char *fname);
void Image_create(Image *img, int width, int height, int channels, bool
zeroed);
void Image_save(const Image *img, const char *fname);
void Image_free(Image *img);
```

The implementation of these functions is not that significant as long as they work properly.

Image will also contain manipulation functions like the following, which takes an Image pointer as input and outputs the gray scale version.

```
void Image_to_gray(const Image *orig, Image *gray)
{
  ON_ERROR_EXIT(
      !(orig->allocation_ != NO_ALLOCATION && orig->channels >= 3
      "The input image must have at least 3 channels."
  );
  int channels = orig->channels == 4 ? 2 : 1;
  Image_create(gray, orig->width, orig->height, channels, false);
  ON_ERROR_EXIT(gray->data == NULL, "Error in creating the image");

  for (
       unsigned char *p = orig->data, *pg = gray->data;
       p != orig->data + orig->size;
       p += orig->channels, pg += gray->channels
  ) {
    *pg = (uint8_t)((*p + *(p + 1) + *(p + 2)) / 3.0);
    if (orig->channels == 4)
    {
      *(pg + 1) = *(p + 3);
    }
  }
}
```

ON_ERROR_EXIT is a simple macro for exiting with status one in case the condition is verified.

With this library at our disposal, we can now create a simple main file for handling a gray scale conversion.

```c
#include "Image.h"
#include "utils.h"

int main(int argc, char **argv)
{
  if (argc != 3)
  {
    fprintf(stderr, "Usage: %s <input> <output>\n", argv[0]);
    exit(1);
  }

  Image img_input, img_output;

  Image_load(&img_input, argv[1]);
  ON_ERROR_EXIT(img_input.data == NULL, "Error in loading the image");

  // Convert the image to gray
  Image_to_gray(&img_input, &img_output);

  // Save image
  Image_save(&img_output, argv[2]);

  // Release memory
  Image_free(&img_input);
  Image_free(&img_output);
}
```

Let us compile this file named gray_scale.c and run it by converting an image of the sky to gray scale:

```
clang gray_scale.c Image.c -o gray_scale.wasm -std=c17 -Wall -lm -O3
wasmtime ./gray_scale.wasm --dir . sky.jpg sky_gray.jpg
xdg-open sky_gray.jpg
```

The last line allows us to verify the output.

After copying the wasm artifact to bin folder, we must connect the endpoint "/gray-scale" to the callback function toGrayScale, and we can do that in a single line, after importing the function in our index.mjs.

```
app.post("/gray-scale", validateImage, toGrayScale);
```

All we need to do is creating a simple client interface with React and use the fetch API to interface with our backend. We can use a functional component, as we do not need an application-wide state.

```
import React, { useState } from "react";

export default function ImageUploader() {
  const [selectedFile, setSelectedFile] = useState(null);
  const [imageSrc, setImageSrc] = useState(null);

  const handleFileChange = (event) => {
    setSelectedFile(event.target.files[0]);
  };

  const handleConvertToGrayScale = () => {
    const formData = new FormData();
    formData.append("image", selectedFile);

    fetch("http://localhost:5000/gray-scale", {
      method: "POST",
      body: formData,
    })
      .then((response) => response.blob())
      .then((blob) => {
        const objectURL = URL.createObjectURL(blob);
        setImageSrc(objectURL);
      })
      .catch((error) => {
        console.error("Error:", error);
      });
  };

  return (
    <div className="table">
      <label htmlFor="file-input">Choose a .jpg or .png file:</label>
      <input
        type="file"
        id="file-input"
        name="file"
        accept=".jpg,.png"
        onChange={handleFileChange}
      />
      <button onClick={handleConvertToGrayScale}>To Gray Scale</button>
      {imageSrc && <img src={imageSrc} alt="Converted to Gray Scale" />}
    </div>
  );
}
```

After inserting this component in App.js, we can now create a script for compiling our React build and serving our application.

```
cd client
yarn build
cd ..
nodemon --experimental-wasi-unstable-preview1 index.mjs
```

If we open a browser tab and connect to localhost:5000, we can upload a file and get back from the server the gray scale version.

This is just a start, but it is easy to expand the functionalities of this application, because the structure is suitable for being expanded.

The only redundancy we have is the creation of a main function for each endpoint instead of using the same C file, but this has its upsides too:

- We introduce less overhead in the arguments management, which makes the structure more error safe.
- Having a microservice[19]-like architecture in the Node server, it would be a contradiction creating a convergence into the same wasm module.
- Once a service is working, it cannot be affected by the next ones.

To make our application more interesting, we can now easily implement a set of useful functions for image manipulation, like:

- Resize to reduce the quality of the image and the space on disk. This action can be implemented by using the STB library and simply changing the width and the height of the inputted image.
- Adjust brightness in a range of 50 to 150%, where a value of 100 causes no change in the brightness. This effect can be obtained by amplifying the value of each channel but fixing a limit to 255: always remember we are working on unsigned chars, and it is necessary to avoid overflow.
- Adjust contrast in a range of -100 to 100, where a value of 0 represents no change in contrast, negative values decrease the contrast, and positive values increase the contrast. The function loops through each pixel in the image and for each pixel component (e.g. R, G, B), it adjusts the pixel value using the scaling factor that is calculated based on

[19] Microservice architecture is a software development approach that structures an application as a collection of small, independent services, each of which is focused on a specific business capability. These services communicate with each other over lightweight protocols, such as HTTP or message queues, and can be developed, deployed, and scaled independently of each other. The microservice architecture has gained popularity in recent years due to its ability to improve flexibility, scalability, and maintainability of complex applications.

the contrast value. The adjusted pixel value is then clamped to the valid range of [0, 255] and set as the new pixel value. The result is an image with adjusted contrast.

- Rotate the image by 90°, which can be performed multiple times to obtain the right rotation. A simple algorithm that uses transformation coordinates.
- Blur and sharpening effect in a fixed value, not to make things too complicated. We can apply a Gaussian blur effect to the image using a kernel with a given radius (in this case, 2.0). The sharpening effect works by enhancing the edges of an image to make it appear sharper. It does this by performing a high-pass filter on the image, which amplifies the high-frequency components and suppresses the low-frequency components. This is done by subtracting a blurred version of the image from the original, resulting in an image that emphasizes the edges. The amount of sharpening can be controlled by adjusting the strength of the high-pass filter.
- Cropping the image, with visual feedback before performing the action.
- Format conversion: JPG to PNG and vice versa, which is supported directly in the STB library.
- Sepia effect, by applying for each pixel a filter based on a combination of the channels with fixed coefficients.

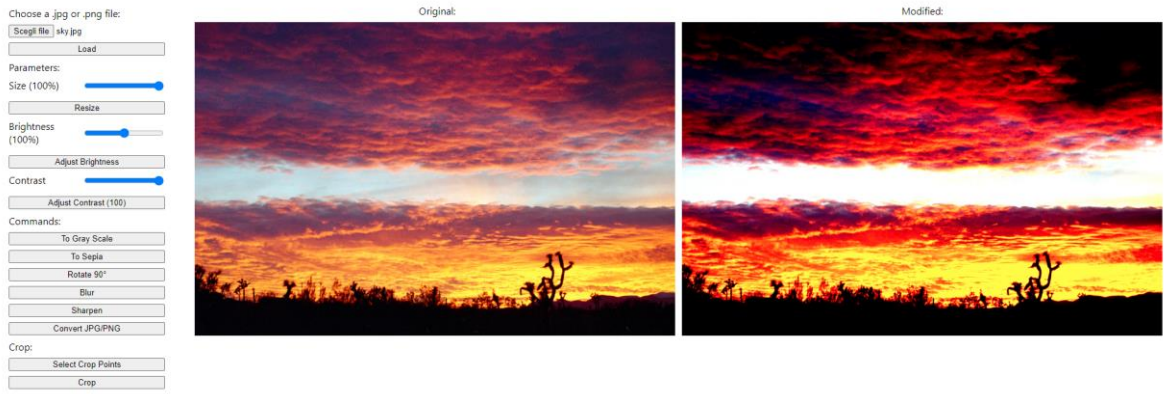
In the end, we will have an interesting palette of commands for image manipulation, each connected to a different endpoint and capable of processing different arguments.

On the font-end, the communication with each endpoint will be entrusted to a different button connected with a specific callback function.

For example, the blurring command takes no argument as an input, while the cropping one takes 4. Being more flexible, a check on the arguments can be performed in the JS code, to avoid instancing WASI if not needed and sending a proper error message to the client.

Results and Evaluation

Here is an example image of the result on the client:



The client can upload an image in a JPG or PNG format and apply any of the changes. The result of the chosen action will be displayed on the right, the client will be able to compare the 2 images and possibly pick the transformed image as the base image for the next action.

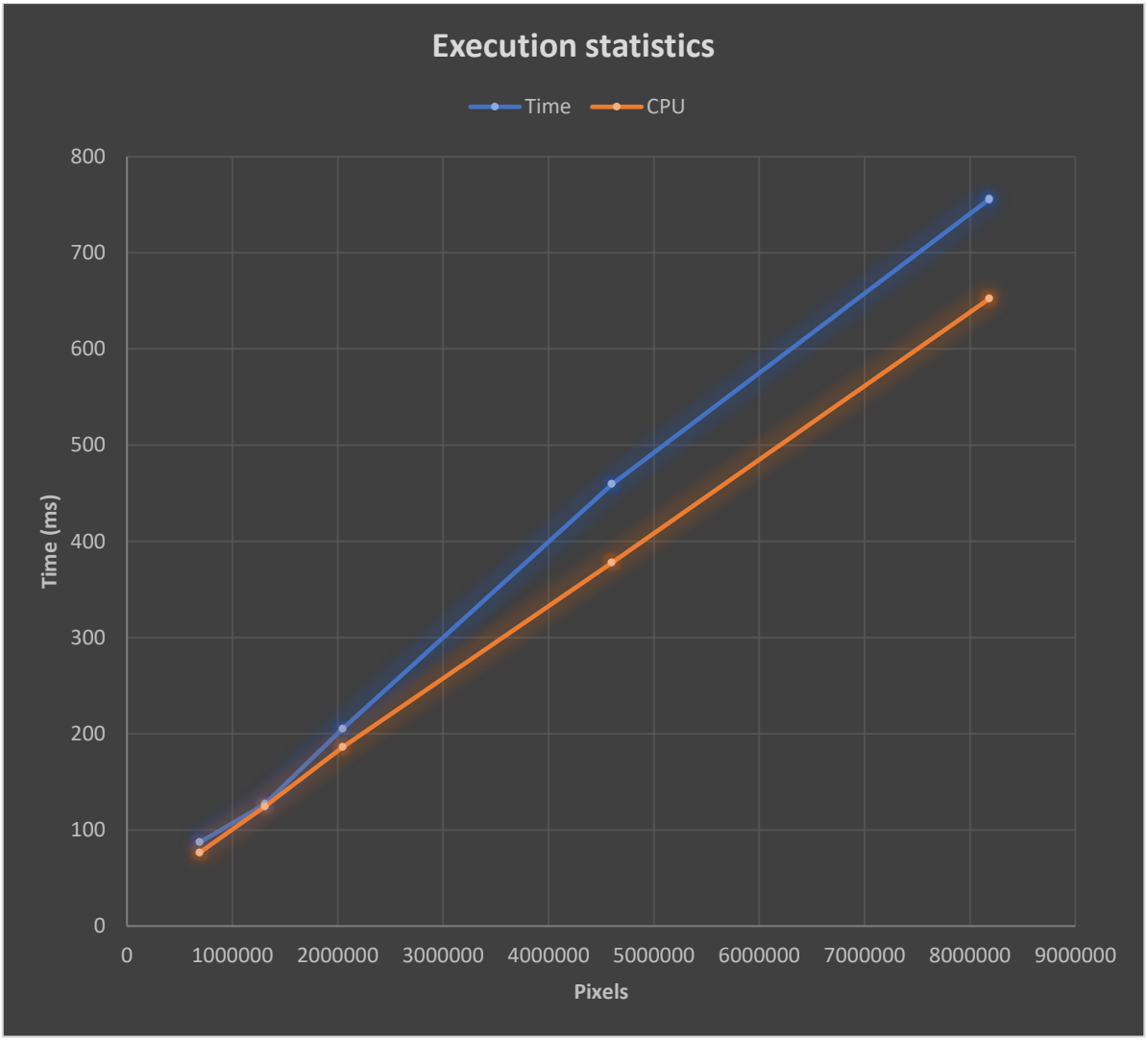
To run some significant benchmark tests, 5 images were used, with different sizes among each other. The parameters considered for the tests were CPU and RAM usage and execution time. Each of images was taken as an input for each function and the average value of every statistic was then considered.

All the images were 3 channels JPG's and the number of pixels, calculated as width × height × channels, was used as the studied variable.

To no one's surprise, the RAM usage was around 2 × width × height × channels + a fixed offset.

This is because two copies of the same image are loaded into the RAM during the execution.

Execution time and CPU usage in terms of milliseconds go hand in hand and the relationship with input image pixels is remarkably close to a direct proportionality, meaning the overhead for starting the WASI instance is negligible.



These results can be improved with parallelization of the tasks, but some of the main features of this approach are simplicity and reduces use of resources.

Conclusions

As we wrap up this thesis on the WebAssembly System Interface (WASI) and its potential applications, we cannot help but feel excited about the possibilities this technology opens up.

WASI is a specification that allows WebAssembly modules to interact with host operating systems in a platform-independent way. Some of its major features include a standardized system call interface, sandboxed execution environment, and support for multiple programming languages. It aims to provide a common runtime environment for WebAssembly modules to run on any device or platform, making it a promising technology for building cross-platform applications.

WASI's design and implementation ensure that security is a fundamental aspect of the runtime environment. The protection ring model, which allows for fine-grained control of system access, is an essential part of WASI's security architecture. Additionally, WASI's portability is a key feature, as it enables code to run securely across different architectures and operating systems without modification. This means that developers can focus on writing secure code without worrying about the specifics of the underlying platform.

But some cross-platform frameworks already exist, how is this new?

With WASI, we are no longer talking about portable code, WASI aims to make binaries portable, which would mean compiling once, running everywhere in an analogous way to what Java and the Java Virtual Machine do, but with some major differences. WASI and JVM are two different technologies used for different purposes. WASI is an interface specification that defines a standard way for WebAssembly code to interact with a host system, such as an operating system, without relying on specific system details. On the other hand, JVM is a virtual machine that is used to execute Java code. While both technologies provide portability, WASI is designed to provide system-level portability for WebAssembly modules, while the JVM provides language-level portability for Java code. Additionally, the JVM is specifically designed for executing Java code, whereas WASI is a general-purpose interface that can be used by any language that compiles to WebAssembly.

One of the advantages of using WASI is its flexibility in terms of programming language support. As a WebAssembly-based standard, any language that can compile to WebAssembly can target WASI, including C, C++, Rust, and even higher-level languages. This allows developers to work with the language they are most comfortable with while taking advantage

of the security and portability benefits of WASI. Additionally, the interoperability between different programming languages is also enhanced using WASI, as it provides a standard ABI (application binary interface) that allows code written in different languages to seamlessly interact with each other.

From microservices to image processing, we have seen how WASI can be applied to various use cases, offering a new way of developing software for the web. In this concluding chapter, we have seen how WASI can be placed in a real-world application to make best use of it.

In the previous chapter, we have tested how close to native speed WASI is able to run and we could observe the small overhead its instantiation brings to the table in terms of CPU and RAM usage, and execution time, compared to an ELF executable.

As with any relatively recent technology, WASI is still in its early stages and is likely to continue evolving and expanding in the future. While it has already demonstrated its usefulness in a variety of contexts, including edge computing and cloud computing, there is still much to explore and discover about the potential of WASI. As more developers begin to experiment with the technology and push its boundaries, we can expect to see even more powerful and innovative applications of WASI in the years to come. With the support of the open-source community and the ongoing efforts of organizations such as the Wasmtime project, the future of WASI looks bright and full of exciting possibilities.

4. Sitography

Articles

- Clark L., *Standardizing WASI: A system interface to run WebAssembly outside the web,* https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, visited on 09/02/2023.
- Salinas Gardón D., *Webassembly vs. JavaScript: How Do They Compare*, https://snipcart.com/blog/webassembly-vs-javascript, visited on 10/02/2023.
- During S., *WASI: secure capability-based networking*, https://blog.jdriven.com/2022/08/wasi-capability-based-networking/, visited on 21/02/2023

Sites

- https://en.wikipedia.org/wiki/POSIX, visited on 12/02/2023
- https://en.wikipedia.org/wiki/Moore%27s_law, visited on 12/02/2023
- https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#Computational_analysis, visited on 14/02/2023
- https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts, visited on 12/02/2023
- https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-capabilities.md, visited on 12/02/2023
- https://github.com/webassembly/wasi, visited on 12/02/2023
- https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, visited on 16/02/2023
- https://en.wikipedia.org/wiki/LU_decomposition, visited on 16/02/2023
- https://v8.dev/, visited on 17/02/2023
- https://nodejs.org/api/documentation.html#stability-index, visited on 20/02/2023
- https://nodejs.org/api/wasi.html, visited on 20/02/2023
- https://github.com/WebAssembly/wasi-sdk, visited on 20/02/2023
- https://github.com/WebAssembly/wasi-libc, visited on 21/02/2023
- https://webassembly.org/docs/use-cases/, visited on 21/02/2023