2023 3rd International Conference on Intelligent Technologies (CONIT)
Karnataka, India. June 23-25, 2023

2023 3rd International Conference on Intelligent Technologies (CONIT) | 979-8-3503-3860-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/CONIT59222.2023.10205816

WebAssembly Beyond the Web: A Review for the Edge-Cloud Continuum

Sangeeta Kakati[†] and Mats Brorsson[†]

[†]Interdisciplinary Center for Security, Reliability, and Trust (SnT)
[†]University of Luxembourg, Luxembourg
E-mail:{sangeeta.kakati, mats.brorsson}@uni.lu

***Abstract*—The cloud computing environment has changed over the past years, transitioning from a centralized architecture including big data centers to a dispersed and heterogeneous architecture that incorporates edge followed by device and processing units. This transformation calls for a cross-platform, interoperable solution, a feature that WebAssembly (Wasm) offers. Wasm can be used as a compact and effective representation of server-less functions or micro-services deployment at the cloud edge. In heterogeneous edge settings, where various hardware and software systems might be employed, this is especially crucial. Developers can create applications that can operate on any Wasm-compatible device without spending time worrying about platform-specific challenges by using a common runtime environment.**

In this survey, we indicate the main challenges and opportunities for Wasm runtimes in the edge-cloud continuum, such as performance optimisation, security, and interoperability with other programming languages and platforms. We provide a comprehensive overview of the current landscape of Wasm outside the web, including possible standardization efforts and best practices for using these runtimes, thus serving as a valuable resource for researchers and practitioners in the field.

***Index Terms*—WebAssembly, Cloud computing, Edge computing, IoT, Heterogeneity, Runtimes**

I. Introduction

WebAssembly (Wasm) was initially designed to run within web browsers. A majority of the web browsers use just-in-time (JIT) compilers to compile WebAssembly modules to native code at run-time. As a consequence of its rising demand in the recent years, Wasm has moreover paved its way towards offering edge-cloud functionalities. This ideally aids in reducing the size of the deployed code, making it more efficient and faster to execute. Also, it can help to mitigate the security risks associated with running code in shared hosting environments. It has the ability to provide software developer-friendly solutions to enable applications to run across the wide range of accelerator hardware. One potential application of Wasm in the edge-cloud continuum is in the deployment of lightweight, portable applications that can run on a variety of devices and platforms. Developers can create applications that can be compiled to run on multiple architectures, including CPUs and GPUs. The escalation of mobile and pervasive computing has given way to the era of ubiquitous computing, which means that, as intended, an increasing number of heterogeneous devices are being incorporated into our immediate vicinity.

WebAssembly runtimes are software environments that can execute Wasm binaries. These runtimes can be implemented in various ways, including browser engines or standalone runtimes that can be embedded in applications and can provide an execution environment for Wasm codes; with features such as memory management, security, and performance optimizations. They are also suitable for embedded systems, where code size and performance are critical. Embedded Wasm runtimes are typically used for IoT devices, microcontrollers, and other use cases that require low memory overhead and efficient execution. For instance, the Wasm micro runtime (WAMR) is a popular embedded runtime for Wasm which is designed to be highly optimized for embedded systems, with a small binary size and low memory overhead.

Each runtime has its own strengths and weaknesses, which is discussed thoroughly in this paper, depending on the use case and programming language. WebAssembly is gaining attention as a versatile technology that can be used for performance-critical tasks in various contexts, including cloud/edge computing, media processing, machine learning, and IoT. As Wasm adoption continues to grow, we aim to provide a comprehensive analysis of the prevalent state-of-the art and the exploration of runtimes which is likely to increase, making them a key area of focus for developers and researchers alike.

Apart from contextually comprehending WebAssembly runtimes within the broader landscape, the motivation of this work can be listed as follows:

- Summarising the latest research and development: With the increasing demand for edge computing, the use of Wasm outside the web has become an emerging topic of research. This review paper provides an overview of the current state-of-the-art, including the most prominent Wasm runtimes presently accessible and how they are being used in the edge-cloud continuum.
- Comparison of different runtimes: This paper can provide a detailed overview of the progress in Wasm runtimes, including their features, performance, and limitations, to help researchers and developers choose the most suitable runtime for their specific use case.
- Identification of research gaps: This survey can identify gaps in the research on Wasm runtimes and highlight areas that require further investigation. It can help guide

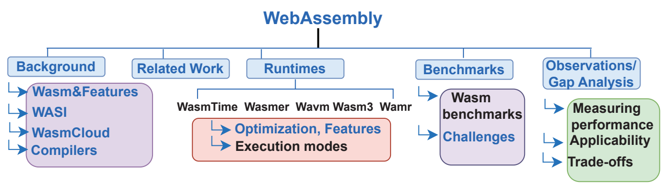

Fig. 1: Study overview

future research in the field and facilitate the development of new and innovative WebAssembly runtime technologies.

- Usage in the cloud-edge environment: WebAssembly is still a relatively new technology, many developers may not be familiar with its capabilities and potential use cases. This review paper can help to raise awareness of Wasm and its potential applications outside the web, thus, bridging the gap between academia and industry.

There is a need for researchers to have a reliable destination in order to find all the most significant advancements in using the WebAssembly runtimes for the edge-cloud environments, considering that WebAssembly is still developing in its infancy, and is growing in popularity. The goal of this paper is to present a concise rational explanation for using each of the existing WebAssembly runtimes. To be more specific, the contribution of this paper are as follows:

1) To the best of our knowledge, this is the first exploratory and comprehensive study on WebAssembly runtimes.
2) We present a thorough analysis of all the runtimes that are presently in use and are being actively developed and improved.
3) Researchers and those eager to make significant progress with WebAssembly can use this survey as a starting point and one-stop resource.

We have organized this survey according to Fig. 1.

II. Background

WebAssembly is a binary encoding format. The binaries are made to process promptly and be small. It can be represented in WAT, a human-readable, precise text format. In contrast to most native systems, Wasm globals, locals, function arguments, and instruction results are all typed. Before being run, binaries are formally type-checked. There are four basic types: single and double precision floats, as well as 32- and 64-bit integers (i32, i64) (f32, f64). There are no arrays or designated pointers. Thus, during compilation, source-level formats are simplified to these primitive types.

A. *WebAssembly and its Features*

Modules written in WebAssembly are run in a host environment. WebAssembly programs cannot, for instance, execute I/O or access the network without the host environment. Instead, this capability is offered by the host through functions that the Wasm module can import (as discussed in the subsequent section II-B). An overview of the WebAssembly paradigm is shown in Fig. 2.

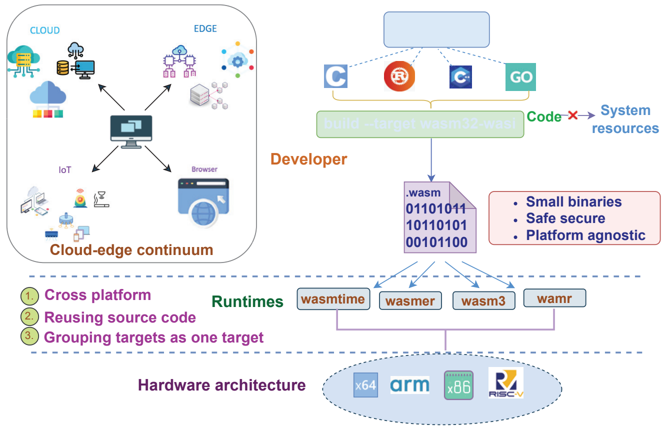

Fig. 2: Depiction of WebAssembly and WASI

B. *WebAssembly System Interface: WASI*

The WebAssembly System Interface (WASI), which is a standard specifying a POSIX-like interface, allows Wasm to communicate with the underlying operating system. It is suitable for restricted settings, like IoT and peripheral devices, because WASI has been intended to be concise and portable, enabling the platforms to swiftly carry out the specifications. Currently, it provides a collection of 46 functions that let applications communicate with files, networks, and many other operating system features. POSIX calls are smoothly converted to WASI calls by well-known compilers for languages like C and Rust. Additionally, WASI adheres to the idea of capability-based security, a security model in which the Wasm runtime must give access to each resource (such as a socket or file) in order to create a sandbox.

C. *Compilers*

There are several compilation infrastructures that can be used to compile codes to Wasm, including, *LLVM* (Low-Level Virtual Machine), which is a widely used open-source compiler infrastructure, *Emscripten,* which is a LLVM-based toolchain that compiles C/C++ and other languages to Wasm, *Binaryen* provides a set of tools to optimize and transform Wasm code, and *Rust*(systems programming language) includes its own Wasm backend, making it easy to compile rust code to Wasm. Among all these, LLVM is the most popular that supports multiple programming languages and architectures. To compile codes to WebAssembly, it uses the LLVM WebAssembly backend, which generates WebAssembly binary code from its intermediate representation (IR). The overview of the workflow can be described as:

- The front-end of the supported language is used to generate the LLVM IR. The front-end is responsible for parsing the code, generating the IR, and performing some language-specific optimizations.
- The LLVM optimization passes is used to optimize the intermediate representation.
- The Wasm backend is utilized to generate the required binary code from the optimized LLVM IR.

- Optionally, it also provides *Wasm-opt*, a tool from the Binaryen project, to further optimize the Wasm code. Wasm-opt performs a set of Wasm-specific optimizations that can improve the performance and size of a code.

D. WasmCloud

WasmCloud is a distributed framework that enables the creation of portable business logic that can be executed anywhere, from the edge to the cloud. It intends to eliminate redundant boilerplate from the developer experience and reintroduce the experience of creating distributed apps by making security the default setting. An overview of WasmCloud is illustrated in Fig. 3. It can run on any platform that supports a Wasm runtime with WASI support, including Linux, MacOS, Windows, and even embedded devices.

The runtimes are a key component of WasmCloud, as it is responsible for executing Wasm modules. A brief working of WasmCloud can be interpreted as:

WebAssembly modules: Developers can choose to code with any of the computer programming language that can be compiled to WebAssembly, such as Rust or C++. These modules are lightweight and can be easily shared between different applications.

Hosts: WasmCloud hosts are runtime environments that can load and execute Wasm modules. These hosts can run on different platforms, such as linux, windows, or kubernetes.

Actors: Actors are instances of Wasm modules that are loaded into hosts. Actors perform business logic and can call or be called by capability providers and also communicate with other actors.

Messaging: Actors communicate with each other using a pub/sub messaging protocol. This protocol is designed to be lightweight and efficient, making it ideal for distributed systems.

Capability providers: Capability providers are natively compiled software modules running on the host that provide specific capabilities to actors. For example, a capability provider could provide access to a database or a messaging system.

Control plane: The WasmCloud control plane is responsible for managing the lifecycle of actors, hosts, and capability providers. It also provides security and authentication mechanisms to ensure that only authorized actors can communicate with each other.

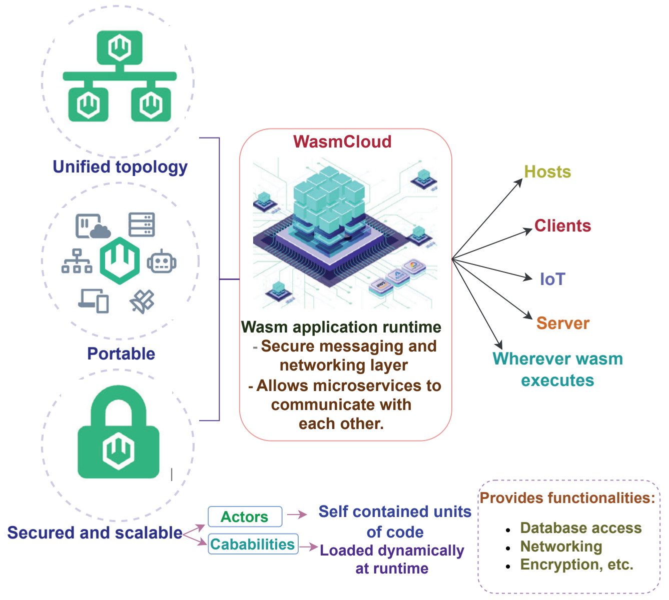

Fig. 3: Characterisation and representation of WasmCloud

III. Related work

Although considerable progress has been made in WebAssembly in the recent years, there has been yet no systematic research providing a cumulative view and usage of standard Wasm runtimes. In an early study, Jangda et al. make an effort to comprehend how effectively Wasm binaries work [1]. The research, however, is restricted to web browsers and thus does not offer enough information about standalone runtimes. Similarly, a study conducted by Yan et al. [2], aims to comprehend how Wasm and JavaScript apps perform differently. The paper contrasted web browser execution efficiency, for instance, in Chrome, Firefox, and Edge, WebAssembly consumes noticeably more memory than Javascript. The reason is evident since JavaScript has garbage collection, which uses dynamic monitoring of memory allocations to decide when to release memory that is no longer in use, whereas Wasm uses a linear memory model and does not immediately release memory. Similarly Wang narrowly concentrates on web applications as opposed to a wide range of applications in non-web domains [3]. Wang focuses less on the emerging application of Wasm outside the web and instead, in a nutshell, considers a single goal i.e., WebAssembly vs Javascript. In another studym Wen and Weber present an OS to execute Wasm applications in edge environments and contrasted their results to native execution in Linux [4]. They claim to achieve better execution speed when compared to native also including the security features of Wasm.

Continuing in the WebAssembly continuum, Wang leverages the gap of the previous works by including a more detailed analysis of the standard Wasm runtimes [5]. The five most common independent Wasm runtimes are covered in this research. Additionally, a well-researched investigation is followed resulting in a new benchmark suite namely, WABench, which includes tools from established benchmarks as well as whole applications from different areas. The idea is to investigate performance efficiency of the standalone Wasm runtimes. Wang examines how JIT compilers, AOT compilation, and Wasm compiler optimizations would affect the performance. According to the findings made, wasmtime, wavm, and wasmer, which are JIT compilation-based runtimes, performs better than wasm3 and wamr, which are interpretation-based runtimes. This work can be considered as a baseline for doing performance analysis since they have integrated a wide range of benchmarks to a single stop although it lacks multi-architecture support as it targets only x86-based architectures. In a nutshell, the paper made the following observations: The performance slowdown of interpretation-based runtimes is stable for both short- and long-running benchmarks. For the JetStream, MiBench, and PolyBench benchmark suites, the Cranelift JIT offers promising outcomes. In all the benchmarks, WAVM uses large percentage of the memory, and Wasm runtimes generally use more memory than native ones for the whole applications.

Lehmann and Pradel describes the challenges of analyzing Wasm programs, including the lack of debug information and the difficulty of understanding the control flow of the program [6]. To address this, they present a design and implementation of framework named Wasabi, which provides a number of features for dynamic analysis of Wasm programs. It includes support for tracing and profiling, as well as features for analyzing control flow and memory usage. Spies and Mock analyse the performance and usability of Wasm in three non-web environments: desktop, server, and IoT devices [7]. The authors conclude that Wasm has potential for use in non-web environments, particularly in resource-constrained environments where it can provide a lightweight alternative to native code.

Lehmann et al. examine the degree of exploitability of vulnerabilities in Wasm binaries and contrasts it to native code [8]. The use of linear memory by the applications in Wasm is thoroughly examined in terms of security. In another work, Szewczyk et al. examine the performance of Wasm and its bounds-checked memory access safety mechanism [9]. Their study extends four popular Wasm runtimes with modern bounds checking mechanisms, evaluates their performance compared to native compiled code on different instruction set architectures, and demonstrates that performance-oriented runtimes can achieve execution times within 20-35% of native.

While recent studies have shown that Wasm performs well overall, Ménéntrey et al. claim to be the first to evaluate ”two” set of architectures for WebAssembly [10]. Their objective is to demonstrate that Wasm can execute code on a range of devices for comparable tasks without experiencing substantial performance overheads. They chose WAMR in AOT as the ”only” Wasm runtime because of its portability across OS systems and restricted environments and compact size. The benchmarks are first compiled using clang into Wasm format, and then they are compiled once more beforehand into a native format using the wamr compiler (i.e., wamrc). However, it is hypothesized that Wasm is inherently slower than native due to the growing register pressure, the larger code, the additional branch lines, and in some instances, it is quicker due to lower number of cache misses. Subsequently, they came to the conclusion that some tasks are not optimally optimised when first written in WebAssembly and further compiled again to native code. Notwithstanding, it is the only work till now to make a useful distinction of cross architecture support in WebAssembly. A summary of the related literature is shown in Table I.

Synthesis: The primary focus of this literature review is on WebAssembly runtimes utilized beyond the web environment. Our analysis will concentrate on the latest runtime technologies employed in current research and development of Wasm.

TABLE I: Literature studies in WebAssembly

Research objective	Addressing WebAssembly
Analysing performance of WebAssembly [1].	Analysis of Wasm compiled Unix applications inside the browser.
Performance of Wasm applications vs javascript [2].	Comparison of WebAssembly JIT compilers and javascript
WebAssembly study in the web [3].	JIT optimization of Wasm and javascript in chrome
Analysing performance and portability of WebAssembly for non web environments [7].	Current state of WebAssembly and its system interface
Identifying performance gap by analysing cranelift and llvm [11].	Proposal for designing WebAssembly runtimes
Construction of a benchmark suite (wabench) [5].	Evaluation of WebAssembly runtimes in wabench
Vulnerabilities of Wasm vs native node [8].	Security analysis of Wasm binaries
WebAssembly as a universal interface for the cloud-edge continuum [10].	Benchmarking Wasm(PolyBench) in x86 and arm.
An overview of implementation of Wasm in the browsers [12].	Discussion on the development, design and implementation experiences of WebAssembly.
Vulnerabilities in Wasm binaries [13].	Exploitation of WebAssembly with insecure source languages.
Proof of soundness in the type system of Wasm [14].	Implementation of type checker and interpreter.
Analyzing and understanding WebAssembly codes at runtime [6].	A dynamic analysis approach to examine the behavior of Wasm modules during runtime, with a focus on security analysis.
Investigate existing memory safety operations in WebAssembly [15].	Enhance Wasm for improving memory safety.
Feasibility of using Wasm to bring seamless cloud-IoT integration [16].	WebAssembly runtime for resource constrained devices.
Introducing dynamic linking capabilities to the Wasm platform [17].	A dynamic linking mechanism that allows Wasm modules to be linked together at runtime (wasmtime).
Framework for developing and verifying cryptographic web applications using WebAssembly [18].	Verification and implementation of cryptographic protocols in Wasm.
Analysis of Wasm vs native in the spec cpu benchmarks [19].	Performance measure of applications compiled to Wasm in firefox and chrome.
Use of Wasm runtimes in serverless edge computing [20].	A prototype for Wasm execution using the Fastly edge cloud platform and the wasmtime WebAssembly runtime.
Design of an OS to run Wasm modules natively [4].	Implementing a lightweight OS kernel that can directly run Wasm modules in edge devices.

IV. WebAssembly runtimes outside the web

When WebAssembly is used outside the web, it typically requires a runtime environment that is capable of executing its modules. These runtime environments are often referred to as WebAssembly engines or Wasm interpreters. It allows developers to write code in any language that compiles to WASM, which can then be executed in a variety of environments, including web browsers, servers, and standalone runtimes. WebAssembly runtimes are the software components responsible for loading, executing, and managing Wasm modules. The WebAssembly System Interface (WASI) was designed to provide a secure and portable interface between Wasm modules and the underlying host system. WASI is supported by several desktop runtimes, including wasmtime and wasmer. In this literature review, we explored some of the popular Wasm runtimes and their performance characteristics.

Wasmtime: Wasmtime is an open-source standalone WebAssembly runtime developed by Mozilla. It has a fast and efficient Wasm execution engine and supports AOT and JIT compilation. The engine is optimized for runtime performance, memory usage, and security. It can be used as a standalone runtime or embedded in other applications. Wasmtime also includes support for WASI, allowing it to run Wasm modules in a sandboxed environment. Some of the significant wasmtime characteristics are:

- Compact: An easy-to-use standalone runtime that can be expanded as requirements might change. It can be applied to both large and small chips, servers and almost any program can embed it.
- Fast: Perform high-resolution realtime machine coding; compatible with Cranelift.
- Complement: Widespread developer and user community support, compliant with Wasm test suite standards.

Wasmer: Wasmer is another popular WebAssembly runtime like wasmtime, it supports multiple languages and includes support for WASI. It is designed to be highly modular and configurable, with support for pluggable compilers and sandboxing technologies. Some of the key wasmer features are:

- Pluggability: Able to operate with different compilation frameworks.
- Efficiency: Able to run Wasm in a fully sandboxed environment at near native speed.
- Complement: Widespread developer and user community support, complying with Wasm test suite standards.

Lucet: Lucet was announceed on March 28, 2019. It is a native WebAssembly compiler and runtime. Developers who want to safely execute untrusted Wasm programs inside their application have the option of using this less well-known compiler/runtime. The maintenance has been currently shifted to wasmtime.

Wavm: With a small amount of architecture-specific assembly and LLVM IR generation code, the WAVM runtime is primarily created in portable C/C++. WAVM has undergone considerable testing and is compatible with X86-64 versions of Windows, MacOS, and Linux. Although it has not been routinely tested on other platforms, it is intended to operate on any POSIX-compatible system. While AArch64 support is still under development, addressing Wasm stack overflow as well as partially out-of-bounds storage are two among the frequently tracked issues of this runtime.

Wasm3: Wasm3 acknowledges the following as the rationale for choosing a ”slow interpreter” as opposed to a ”fast JIT”: When speed is not the most important factor to consider; with the interpreter approach, runtime executable size, memory utilization, and startup latency can be improved. Security and portability are much simpler to manage. While wasm3 has several advantages, there are also some limitations that should be considered:

- Limited memory management: Wasm3 currently does not support dynamic memory allocation, which means that memory must be allocated statically at compile-time. This can lead to memory wastage and limit the scalability of applications.
- Performance optimizations: Wasm3 is designed to be lightweight and portable, which means that it does not include many of the performance optimizations that are available in other Wasm runtimes. This can lead to slower performance in benchmarks and applications.
- Platform support: While it is designed to be portable, it currently only supports a limited number of platforms, thus limiting the types of applications that can be run on wasm3.

  Nonetheless, it is a promising runtime for WebAssembly, but it currently has some limitations that should be considered before using it for production applications.

Wamr: WebAssembly is also well-suited for embedded systems, where code size and performance are critical. WAMR is a lightweight and efficient Wasm runtime that is designed to be easily embedded in resource-constrained environments, such as IoT devices, microcontrollers, and other edge devices. WAMR differs from other Wasm runtimes in its:

- Performance: It is designed to be highly efficient and optimized for low-power devices. It has a small memory footprint and fast startup time.
- iwasm VM core: Offers just-in-time, ahead of time compilation, and Wasm interpretation.
- Size: Wamr is lightweight and has a small binary size. This makes it ideal for running on devices with limited memory and storage. It has a file size of 209 KB in AOT, 230 KB in interpretation based and 41mb in JIT based.
- Portability: The runtime is portable and can be easily adapted to run on different platforms and architectures.
- Customizability: WAMR is designed to be highly customizable, allowing developers to tailor the runtime to their specific needs.
- Fast: It can run at near-native speed (AOT)

TABLE II: Runtime features in WebAssembly

Runtime	Non web standards	Language	Compilation mode
Lucet	WASI	Rust	AOT
wasm3	WASI, Custom	C	Interpreted
wasmEdge	WASI	C++	AOT, Interpreted
wasmer	WASI, Wasm-c-api	Rust, C++	JIT, AOT
wasmtime	WASI, Wasm-c-api	Rust	JIT
wasmVM	N/A	C++	Interpreted
WAVM	WASI	C++, Python	JIT
WAMR	Wasm-c-api	C	Interpreted, AOT, JIT

A. OPTIMIZATION AND FEATURES

WebAssembly runtimes are becoming increasingly popular due to their ability to run high-performance code in and outside the web environments. Notwithstanding, there are several ways to optimize the performance of Wasm runtimes, including:

1) Memory management: Optimizing memory management can help reduce overhead and improve performance. This includes minimizing the use of dynamic memory allocation and using efficient memory allocation strategies.
2) Hardware acceleration: Wasm runtimes can take advantage of hardware acceleration characteristics such as

Single Instruction Multiple Data instructions and GPU acceleration to improve performance.

3) Ahead-of-time (AOT) compilation: WebAssembly can be compiled ahead of time to native code, which can improve startup time and reduce runtime overhead.
4) Just-in-time (JIT) compilation: JIT compilation involves compiling Wasm code at runtime, which can result in significant performance improvements compared to interpretation.
5) Code optimization: Optimizing the Wasm code itself can result in significant performance improvements. This includes techniques such as loop unrolling, function inlining, and instruction scheduling.
6) Profiling and benchmarking: Profiling and benchmarking can help identify performance bottlenecks and guide optimization efforts.
7) Caching: Caching frequently used data and code can improve performance by reducing the amount of time spent loading data and compiling code.

B. EXECUTION MODES

A Wasm module can typically be run in one of three compilation modes: Interpreter, Just-In-Time (JIT), or Ahead-of-Time (AOT). The decision depends on the user's preference for resource usage, execution speed, etc. All three options are supported by the WAMR runtime,

- **AOT**, which facilitates quick startup, exceptionally small footprint, and results in almost native speed.
- **Interpreter**, which offers small memory usage, little footprint, and comparatively slow. The two interpreters present in WAMR are:
  - Classic Interpreter: Currently, it is required to enable source debugging.
  - The Fast Interpreter: Precompiles the Wasm opcode to internal opcode, running about two times as quickly as the Classic Interpreter (CI), but using a bit more memory than the former.
- **JIT**, it allows to run Wasm at almost native speed while maintaining a platform-neutral distribution. However, compilation is expensive during implementation. Furthermore, the two JIT layers supported by WAMR are:
  - LLVM JIT, has longer compilation time and faster execution.
  - Fast JIT: A JIT engine that is lightweight, has a tiny footprint, starts up quickly, and better performance.

V. BENCHMARKS

In the current state of the art of experimenting WebAssembly runtimes, the benchmarks used are predominantly based on PolyBench and a few others are dicussed in section V-A. The ongoing experiments of WebAssembly developed by the Bytecode Alliance, a community of organizations and individuals working together to advance the adoption and development of WebAssembly and related technologies, has, however, achieved significant improvement and addition, also offering a benchmark suite called **Sightglass**.
These benchmarks are designed to evaluate a variety of performance characteristics of a Wasm runtime, such as startup time, memory consumption, and execution speed. They cover a range of use cases, from concise mathematical computations to more complex tasks like image processing and cryptography, and are written in a variety of computer languages.

A. State-of-the-Art Benchmarks

After analyzing the most recent literatures in this field, we could draw the conclusion that PolyBench, MiBench, JetStream2, and an addition of whole applications are in the status quo of benchmarking WebAssembly. Below is a thorough explanation and analysis of the benchmarks:

- **PolyBenchC:** This collection of benchmarks contains relatively small scientific computing kernels. The fact that PolyBenchC is made to test low-level performance characteristics like cache usage and memory access patterns that might not be directly applicable to Wasm is one of the major problems with utilizing it to evaluate Wasm. Since Wasm is a greater level of abstraction than C, some differences in performance are probably to be expected.
- **MiBench:** It includes benchmarks for tasks such as digital signal processing, network routing, and security algorithms. These tasks are important in embedded systems, but they may not be representative of the types of workloads that Wasm is typically used for. However, it is important to keep in mind that benchmarks only provide a limited view of the overall performance of a system and should be used in conjunction with other performance analysis techniques.
- **JetStream2:** The JetStream2 benchmark contains several tests that are specifically created to test the performance of JavaScript in web applications and is mainly intended to evaluate the performance of JavaScript engines in web browsers. Furthermore, there are no tests specially created to assess the performance of Wasm modules outside of a web browser context. Therefore, it might not be the best option for assessing Wasm's performance in non-browser contexts.
- **Whole applications:** This benchmark includes a set of whole applications, for instance, face detection, mnist, bzip2, whitedb, espeak and gnu chess. While this benchmark can be used to evaluate the performance of Wasm, it is important to note that it may not be the best choice in all cases. The WholeApplication benchmark typically involves running a complete application, which can be time-consuming and may not be feasible in all contexts.

Ultimately, the choice of benchmark will depend on the specific use case and requirements of the system being evaluated. It may be necessary to use a combination of different benchmarks to get a comprehensive view of the performance of Wasm. Notwithstanding, there are other benchmark suites that performs well in AOT mode evaluation of WebAssembly, such as the Sightglass benchmark. These benchmarks are designed keeping in mind the most recent advances in Wasm runtimes and are likely to be more relevant and accurate for evaluating the performance.

VI. Observation and Gap Analysis

Following a thorough review of all the benchmarks and current runtimes, the results and inferences to be paid particular attention are as follows:

1) **Measuring WebAssembly Runtime Performance:** How do we deal with the runtime startup and teardown cost while benchmarking WebAssembly?
   a) Benchmark size: The size of the benchmark can affect the startup and teardown cost. Large benchmarks may take longer to execute and have additional/overall time costs, whereas small benchmarks might not accurately reflect the runtime performance. Even though we might only need a portion of the real execution time, the benchmark is measuring a large percentage of setup time.
   b) Instrumentation overhead: The process of measuring the startup and teardown cost can itself introduce overhead that affects the runtime performance of the code.
   c) Cold start vs. warm start: The startup cost of Wasm code can be significantly higher than the runtime cost due to the time required to load the code and initialize the runtime.

   Below we provide some rationals in order to overcome the inference VI :
   a) Identifying a benchmark that accurately reflects the intended use case of the WebAssembly code.
   b) The benchmark code is ought to be instrumented to record startup and shutdown durations. Performance timers or adding custom measurements, for instance, measuring the time of an ”empty” main program and subtracting it from the other measurements.
   c) Calculating the average startup and teardown times for multiple runs of the benchmark to get a more accurate measurement.
   d) Repeating the benchmarking process with different Wasm runtimes and compilers to compare their startup and teardown times.
   e) Using profiling tools to identify performance bottlenecks and optimizing the Wasm code as needed.
   f) To measure the cold start time: Measuring the startup time for the first iteration of the benchmark; and to measure the warm start time, measuring the startup time for subsequent iterations after the Wasm runtimes has been initialized, might introduce beneficial results.

   Also, it is worth noting that the exact tools and techniques used to measure the startup and teardown cost of WebAssembly benchmarking can vary depending on the specific use case and platform being used.

2) **Runtime Applicability**
   We initially started by examining the runtimes that have recently been addressed in the literature and are essentially under constant development. We whereupon offered thorough review to investigating the runtimes wasmtime, wasmer, and wamr. PolyBench, MiBench, JetStream2, and Sightglass were used as benchmark references for the conclusions drawn. Furthermore, there are a number of queries and concerns that have to be considered when utilizing these runtimes, including:
   a) The variations in execution times between wasmer and wasmtime, despite the fact that both runtimes aim to provide almost identical performance benefits.
   b) Wamr offering faster processing times in AOT than wasmer and wasmtime.
   c) How to determine the usability of WAMR in devices with limited resources when wamr in AOT mode eliminates portability.
   d) Although JIT runtimes can hold compiled code, their performance is still slower than AOT.
   e) The measured time in benchmarks, i.e., the need to measure the wall clock time (benchmarks with dominant sub-parts of code) or, time-stamping (measuring start and finish time of section of interest).

   The following are some beneficial initiatives that could potentially be considered for addressing the aforementioned concerns:
   a) Since mean values are subject to outliers, taking multiple measurements to calculate the median would result in precise measurement of a benchmark.
   b) Evaluating the efficiency of a runtime where some codes must be measured in steady-state due to the requirement to warm-up caches and hardware structures, and can only be measured with timestamps after code warmup.
   c) Making sure that the workload of the benchmark is sufficient (i.e., execution time $>$1second) would also aid in exact evaluation of a runtime.

3) **Trade-off Between Different Execution Modes of the Runtimes:**
   When AOT compilation can be used by the WebAssembly runtimes for its:
   a) Faster startup time: AOT compilation reduces the startup time of Wasm code and can be especially important for applications where fast startup times are critical, such as games or interactive applications.
   b) Better performance: AOT-compiled code can potentially run faster than JIT-compiled code, as there is no overhead associated with JIT compilation at

runtime.

However, this compilation can result to:

a) Limited portability: The code is less portable than JIT-compiled code, as it is compiled for a specific target platform and cannot be easily run on other platforms without recompilation.
b) Longer build times: It can take longer than JIT compilation, as the entire codebase needs to be compiled before the application can be run.

To summarize, it is important to carefully design benchmarks that accurately reflect the intended use case, and to use tools that can accurately measure the startup and teardown cost of the code. It is also necessary to run benchmarks on multiple runtimes and compilers to get a complete picture of the performance characteristics of the code.

VII. Conclusion

This research has demonstrated how WebAssembly can be used as a compact and effective runtime environment for serverless tasks or microservices deployment at the cloud-edge. Wasm modules can be utilized for assembling the function's code and dependencies, allowing for simple and swift deployment to the edge for low-latency processing. Runtimes play a critical role in the performance and efficiency of Wasm modules. The runtimes reviewed in this research are considered for different aspects of WASM performance, such as code size, runtime performance, memory usage, and security. None of the recent related works have considered the size of the binaries, especially [10], where wamr is the only runtime considered and only wamr has a significant difference in the binaries when compared to its interpretation, fast interpretation, jit and aot based compilations. Another emerging aspect while evaluating WebAssembly is to consider the startup time, which has not been addressed significantly in the literature. In this paper, in contrast to previous research, we have provided an exclusive state-of-the-art of WebAssembly and its performance evaluation in terms of the various runtimes introduced. Beginning with the introduction of WebAssembly and then proceeding through its runtimes, we have highlighted the current challenges and a set of viable remedies. Developers and academicians can use this article as a resource whilst discovering WebAssembly, and selecting a suitable runtime for their specific use cases, and to understand the constraints it imposes.

Acknowledgment

This work has been partly funded by the Luxembourg National Research Fund (FNR) under contract number 16327771 and has been supported by Proximus Luxembourg SA.

References

[1] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of webassembly vs. native code.," in *USENIX Annual Technical Conference*, pp. 107–120, 2019.

[2] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the performance of webassembly applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, pp. 533–549, 2021.

[3] W. Wang, "Empowering web applications with webassembly: are we there yet?," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1301–1305, IEEE, 2021.

[4] E. Wen and G. Weber, "Wasmachine: Bring the edge up to speed with a webassembly os," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pp. 353–360, IEEE, 2020.

[5] W. Wang, "How far we've come–a characterization study of standalone webassembly runtimes," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 228–241, IEEE, 2022.

[6] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1045–1058, 2019.

[7] B. Spies and M. Mock, "An evaluation of webassembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, pp. 1–10, IEEE, 2021.

[8] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *Proceedings of the 29th USENIX Conference on Security Symposium*, pp. 217–234, 2020.

[9] R. Szewczyk, K. Stonehouse, A. Barbalace, and T. Spink, "Leaps and bounds: Analyzing webassembly's performance with a focus on bounds checking," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 256–268, IEEE, 2022.

[10] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Webassembly as a common layer for the cloud-edge continuum," in *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, pp. 3–8, 2022.

[11] Z. Wang, J. Wang, Z. Wang, and Y. Hu, "Characterization and implication of edge webassembly runtimes," in *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pp. 71–80, IEEE, 2021.

[12] X. Qiao, P. Ren, S. Dustdar, L. Liu, H. Ma, and J. Chen, "Web ar: A promising future for mobile augmented reality—state of the art, challenges, and insights," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 651–666, 2019.

[13] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021*, pp. 2696–2708, 2021.

[14] C. Watt, "Mechanising and verifying the webassembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs*, pp. 53–65, 2018.

[15] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, "Position paper: Progressive memory safety for webassembly," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 1–8, 2019.

[16] B. Li, H. Fan, Y. Gao, and W. Dong, "Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pp. 261–272, 2022.

[17] N. Mäkitalo, V. Bankowski, P. Daubaris, R. Mikkola, O. Beletski, and T. Mikkonen, "Bringing webassembly up to speed with dynamic linking," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 1727–1735, 2021.

[18] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in webassembly," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1256–1274, IEEE, 2019.

[19] A. Jangda, B. Powers, A. Guha, and E. Berger, "Mind the gap: Analyzing the performance of webassembly vs. native code," *arXiv preprint arXiv:1901.09056*, 2019.

[20] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 140–149, IEEE, 2022.