2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 979-8-3503-9566-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/CCGrid59990.2024.00046

A Cross-Architecture Evaluation of WebAssembly in the Cloud-Edge 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*—As cloud-to-edge computing becomes increasingly prevalent, the need for an application framework capable of dynamically utilizing the entire spectrum of resources has grown. As developers increasingly seek to deploy applications across heterogeneous computing environments, this research aims to introduce how WebAssembly(Wasm) emerges as a versatile ally, seamlessly executing on different architectures, allowing developers to craft applications without being bothered by the underlying hardware platform.**

WebAssembly binaries need a runtime to execute. In this paper, we present a thorough performance analysis of the two most prominent WebAssembly runtimes employing an extensive array of instrumented benchmarks to ensure precise and reliable results. We focus on investigating WebAssembly's performance characteristics while considering important metrics like execution speed and startup time. Unprecedentedly, we extended the evaluation to four diverse sets of architectures: two server-class architectures (X86_64, ARM64) and two embedded boards (Nvidia Jetson Nano with ARM64, StarFive VisionFive2 with RISCV64), marking the first-ever cross-architecture analysis of WebAssembly runtimes. This novel evaluation empowers us to offer valuable insights into the performance traits and considerations of WebAssembly. By scrutinizing architecture-specific results, we shed light on Wasm's potential to address the requirements of a cross-architecture cloud-to-edge application framework and reshape the landscape of modern application frameworks.

***Index Terms*—WebAssembly, Heterogeneity, Cloud-Edge continuum, Cross-architecture, Runtimes.**

I. Introduction

The cloud landscape is evolving, and we increasingly see a development to include not only data center servers, but also regional or on-premise servers as well as mobile edge stations and even end-user devices connecting to the edge. Collectively we call this the *cloud-edge continuum* [1]. This evolving infrastructure is characterized by a great deal of heterogeneity.

There are a multitude of architectures and different performance characteristics, making application development and deployment difficult. We need to ensure that software can run on as many different nodes as possible without the need to adapt them to each and every hardware platform specifically. We also need to make sure that they run on the nodes that provide the best fulfillment of requirements without wasting resources (service placement and rescheduling). The latter problem is a highly active research problem dealt by several researchers [2]–[4].

For dealing with heterogeneous hardware we propose the use of WebAssembly (Wasm) to achieve architecture independence. The growing use of WebAssembly outside of the browser ecosystem indicates that this strategy is popular with developers. WebAssembly is a quick and effective bytecode format that was introduced in 2017. Wasm has an appealing execution format for executing code outside of the browser due to its near-native speed, quick startup time, full portability, and sandboxed execution. Additionally, the availability of programming languages with developing support for compiling to Wasm proves to be a major benefit. Figure 1 illustrates the place of Wasm in the cloud-edge continuum.

Knowing the performance characteristics of WebAssembly runtimes is essential given the rise in edge computing demand and the diversity of hardware architectures. A WebAssembly program needs a natively compiled *Runtime* to execute and this runtime is essential for the effective execution of WebAssembly modules, and has a big impact on the user experience and overall performance of the application.

In this paper, we present a comprehensive performance evaluation and contribution in the field of WebAssembly across a range of diverse set of benchmarks and architectures and two state-of-the-art WebAssembly runtimes: Wasmtime[1] and WAMR[2], both of these are projects within the Bytecode Alliance[3]. Our evaluation focuses on crucial performance metrics including execution speed, startup time, and other pertinent characteristics, offering a better understanding of Wasm's capabilities and valuable insights for developers, runtime implementers, and researchers alike.

In particular, these are our main contributions:

- A key novelty of our work lies in its methodology, which surpasses existing studies by conducting the first-ever cross-architecture evaluation of Wasm benchmarks on the Wasmtime and WAMR runtimes. We use two server

[1]https://github.com/bytecodealliance/wasmtime
[2]https://github.com/bytecodealliance/wasm-micro-runtime
[3]https://bytecodealliance.org/

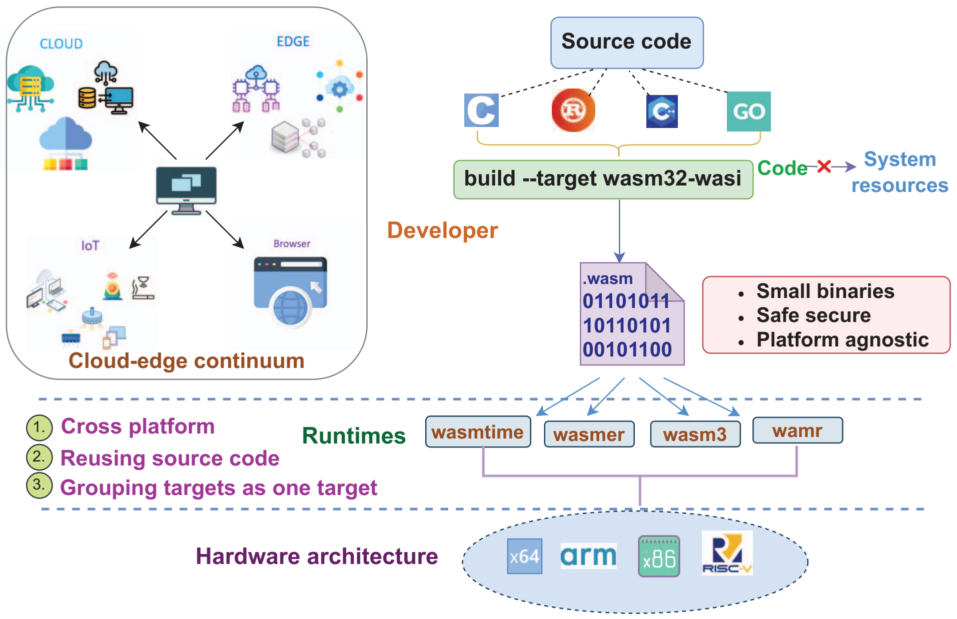


Fig. 1: WebAssembly and its use in the cloud-edge continuum.

platforms on X86_64 and ARM64 and two embedded platforms on ARM64 and RISCV64.

- We introduce methodological enhancements through detailed instrumentation of runtimes and benchmarks. This instrumentation facilitates precise measurement of startup times and execution durations for specific program segments, we term it as the *specific regions of interest*. This allowed us to provide a more detailed and granular analysis than previously reported research in this field.
- With in-depth analysis of the results, our work advances the state of the art in Wasm performance evaluation. In the context of benchmarking, we contributed a calibrated and instrumented suite to the existing WaBench benchmarks. This suite expands the scope of future research, ensuring precise and reliable performance evaluations.

The rest of the paper is organized as follows: Sections 2 and 3 provide an overview of the background and related work. Section 4 describes the experimental setup, including the benchmark suite, runtime, and architectures used in our evaluation. Section 5 presents the results and analysis of the performance evaluation, highlighting key findings and trends. Section 6 discusses the implications of our findings and potential future research directions. Finally, Section 7 concludes the paper and summarizes the contributions of our work.

II. Background

WebAssembly is a binary compilation target, meaning it is represented as compact and efficient binary code that can be transmitted and loaded quickly over the internet. A Wasm *module* can be thought of as a self-contained unit that encapsulates functions, global variables, and memory. The functions are represented as bytecode instructions for a simple stack-based virtual machine. Executing a Wasm binary requires a *runtime* that will interpret the Wasm bytecode instructions or compile them into the machine code of the host that executes the code.

Wasm modules can be compiled from various source languages, and they can import and export functionalities, allowing them to interact with other modules or the host environment, but in a very controlled manner. 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. The final important part of the interface between the host and the WebAssembly module is the *linear memory* which is accessible from both the host and the WebAssembly code.

The WebAssembly System Interface (WASI)[4], which is a standardized system interface for WebAssembly, provides a set of APIs to interact with the host environment in a platform-independent way. It allows WebAssembly modules to perform tasks like I/O operations, file system access, and other system-level operations. It is reliable for constrained environments such as IoT and peripherals because WASI was designed to be lightweight and portable, allowing platforms to implement specifications quickly. It presently provides an inclusion of 46 functions which allows applications to communicate with files, networks, and other features of the operating system. In addition, WASI complies with capability-based security, a model where the runtime must allow explicit access to every resource needed, thus creating a secure sandbox.

[4]https://wasi.dev/

WebAssembly can be generated from many different languages, although Rust, C, C++, and Go are the most stable ones used in production[5]. Rust and Go have their own native WebAssembly backends. For C, C++ you can choose from LLVM-based tools such as clang and Emscripten or Binaryen. LLVM is also the base for several other languages targeting WebAssembly.

In the field of service placement, real-time algorithms, and multi-objective optimization, the landscape of computing environments is evolving rapidly. This environment is marked by increasing heterogeneity, with a wide array of computing hardware, including CPUs of diverse architectures like X86, ARM, RISC-V, as well as accelerators such as GPUs, TPUs, and FPGAs. Adapting applications and algorithms to this diverse ecosystem presents significant challenges.

In light of these complexities, WebAssembly emerges as a compelling solution. Wasm modules are schedulable and offer architecture-independent targeting. This feature, in particular, holds significant promise in managing diverse and complex computing environments. In this context, as we embrace the heterogeneous landscape of CPUs and accelerators, one critical inquiry pertains to the runtime performance of applications and algorithms across these various architectures. How does WebAssembly facilitate efficient execution, and what are the implications for real-time algorithms and multiobjective optimization across this spectrum of hardware?

Our research aims to provide valuable insights into the practicality and effectiveness of WebAssembly in addressing the challenges posed by service placement, real-time algorithms in an environment characterized by heterogeneity, constraint-driven configurations, and ever-evolving computing hardware.

III. Related work

In this section, we provide a comprehensive overview of previous research and studies that are closely aligned with our current investigation. WebAssembly has gained significant attention in recent years as a cross-architecture runtime environment. Several WebAssembly runtimes, including Wasmtime, WAMR, Wasmer[6], Wasm3, and others, have been developed to facilitate the execution of Wasm programs across a wide range of hardware and software configurations.

WebAssembly's ability to address constraints related to quality of service, hardware availability, and networking configurations has been a subject of investigation. The research work of Hilbig et al. [5] has explored an update on the usage of WebAssembly in the real world, highlighting vulnerabilities in insecure source languages and a growing diverse ecosystem.

Another study by Chadha et al. [6] explores the utilization of WebAssembly as a distribution format for MPI-based HPC applications, introducing MPIWasm to facilitate high-performance execution of Wasm code. The study demonstrates competitive native application performance, coupled with a significant reduction in binary size compared to statically-linked binaries for standardized benchmarks.

[5]https://github.com/appcypher/awesome-wasm-langs
[6]https://github.com/wasmerio/wasmer

The employment of common WebAssembly runtimes is rarely examined in related studies in a comprehensive manner. Some previous research in [7], [8], and [9] aims to understand how the Wasm and JavaScript apps perform differently. It focuses on web applications as opposed to the multitude of applications in non-web environments.

Bosshard et al. [10] explore the execution of Wasm in various serverless contexts. It compares server-side Wasm runtime options, such as Wasmer, Wasmtime, and Lucet, and investigates running it within Openwhisk and AWS Lambda platforms. Experiments performed by Spies et al. [11] and Wang et al. [12] demonstrated Wasm's usage using the benchmark PolyBench. Despite being an effective compiler benchmark suite, PolyBench appears to be unable to reflect applications in a wider range of non-web domains. Yet another study by Watt et al. [13] presented a proof of soundness in the type system of WebAssembly by implementing a type checker and interpreter.

A previous investigation by Hockley et al. [14] examined the use of WebAssembly as a sandboxed environment for general-purpose runtime scripting. They conducted a comparison of execution times between Wasm and native mode using micro and macro-benchmarks. However, the experiments are conducted on specific hardware configurations, which might not fully represent the performance on a broader range of devices and architectures.

Kjorveziroski et al. [15] evaluate WebAssembly's impact on serverless computing using a benchmarking suite and compare Wasm runtimes with container runtimes. WebAssembly shows better results in most tests, with Wasmtime being the fastest runtime.

Continuing the exploration of WebAssembly, Wang et al. [16] addressed the gap in previous research by conducting a detailed analysis of standard Wasm runtimes, covering five widely used Wasm runtimes. They introduced a benchmark suite named WaBench, which includes tools from established benchmark suites and whole applications from various domains. This allowed them to assess the performance efficiency of standalone Wasm runtimes and investigate the impact of JIT compilers, AOT compilation, and Wasm compiler optimizations.

In the context of WebAssembly performance, Jangda et al. [17] analyzed applications compiled into WebAssembly using SPEC CPU benchmarks in Firefox and Chrome. Gadepalli et al. in [18] introduced Sledge, an WebAssembly-based serverless framework for the Edge which was optimized for serverless workloads. They offered optimized scheduling policies, efficient work distribution, and a lightweight function isolation model using WebAssembly-based software fault isolation.

Investigations by Stievenart et al. [19] focuses the security implications of compiling C programs to WebAssembly, focusing on buffer overflow vulnerabilities. The study finds that, in some cases, the generated WebAssembly lacks security measures like stack canaries, allowing the program to continue executing after a buffer overflow, in contrast to X86 code that crashes. Thus, highlighting the need for additional precautions when compiling C programs to WebAssembly to ensure security, and encouraging further investigation in this area.

In another study, Lehmann et al. [20] proposed an analysis named Wasabi to examine the functioning of Wasm modules, particularly for security analysis. TWINE, a WebAssembly trusted runtime, was introduced by Ménétrey et al. [21], and evaluated using various benchmarks and real-world applications, including a secure version of SQLite. The study showed that while there are some performance overheads, the additional security guarantees and full compatibility with standard WebAssembly largely compensate for them.

Li et al. [22] explored the feasibility of using WebAssembly for seamless cloud-IoT integration, focusing on WebAssembly runtime for resource-constrained devices. Breitfelder et al. [23] address the security concerns associated with the increased usage of Wasm in back-end technology. The authors introduce WasmA, a static analysis framework for WebAssembly, designed to provide essential information to client analyses efficiently. The evaluation demonstrates WasmA's performance, generality, and extensibility, making it competitive with existing tools.

Lehmann et al. [24] examined the vulnerabilities of wasm binaries compared to native code, with a particular focus on the use of linear memory in wasm applications for security analysis. Another work by [25] introduces a serverless framework, aWsm, based on Wasm, designed to enable efficient serverless computing at the Edge. It addresses the limitations of existing serverless solutions for IoT data processing, offering near-native speed, a small memory footprint, and fast invocation times. The study outlines design details, performance challenges, and highlights aWsm's potential with low startup times and a compact memory footprint for selected MiBench microbenchmarks.

Ménétrey et al. [26] claimed to be the first work to investigate two distinct sets of architectures for WebAssembly, aiming to validate Wasm code for comparable tasks omitting the notable performance overheads. Gackstatter et al. [27] explored the use of WebAssembly runtimes in serverless edge computing, developing a prototype for Wasm execution using the Fastly edge-cloud platform.

Additionally, Wen et al. [28] designed an operating system capable of running Wasm modules natively, specifically targeting lightweight OS kernels for edge devices. Kakati et al. [29] highlights a WebAssembly survey as the changing landscape of cloud computing, with a shift towards edge computing, and emphasizes the need for a cross-platform and interoperable solution.

The position paper by Wallentowitz et al. [30] explores the potential of using WebAssembly as an application virtual machine in Embedded Systems. It highlights the strong ecosystem of WebAssembly and discusses the challenges related to its efficient utilization in such systems. The paper includes a real-world case study demonstrating its viability and outlines open issues and future work.

Although the above list of related work is quite extensive, we find a lack of studies investigating the performance and characterization of WebAssembly runtimes across multiple architectures and for a wider set of benchmarks.

IV. Methodology

In this section we describe the details of our experiments and how they were made. The objectives are to quantitatively and qualitatively analyze the performance of WebAssembly programs on modern runtimes and different architectures and to assess the suitability of using WebAssembly as the execution vehicle for distributed applications in the cloud-edge computing continuum.

A. Benchmarks

We aim to focus on the behavior of Wasm applications when executed on diverse hardware architectures and runtime environments. For this context, we use the WaBench benchmark suite assembled by Wang [16]. We have instrumented these benchmarks with timestamps to be able to measure the time to execute a specific *region of interest*. The purpose of this is to specifically measure execution speed when the application initialization and startup procedures have been completed. We have also instrumented the benchmarks and the runtimes to measure the accurate startup time. This time is defined as the time from giving the Linux shell command to starting the benchmark and to the very beginning of the `main` function. The instrumented benchmarks can be found on github[7].

Also, to mitigate the impact of variations in system load and resource availability, we conducted multiple repetitions of each experiment, adjusting the number of repetitions based on the execution time of the tests. This approach ensures statistical validity and reliability in our results. We report the median value of the observed metrics in our experiments which helps to minimize the influence of outliers and provides a representative measure of central tendency. To maintain the integrity of our experiments, we conducted tests on an undisturbed machine, minimizing external interference, and ensuring a stable testing environment.

Our experiments were conducted without the use of CPU-pinning or freezing clock frequency. This decision was guided by our tests, which did not reveal appreciable differences when employing these techniques.

WaBench consists of over 50 benchmarks in total. They are all written in C or C++ and were compiled with the clang/clang++ compiler version 16 with optimization `-O2` and using WASI SDK version 20. We have made a selection of six benchmarks which we feel are representative to illustrate the differences between runtimes and architectures. The results of all the benchmarks can be found online[8].

B. Performance Metrics Utilized

The execution of a Wasm benchmark program can be divided into a few phases, as illustrated in Figure 2.

The *runtime* phase is the time from when we call the runtime from the Linux shell and until the main function in

[7]https://github.com/matsbror/wabench
[8]https://github.com/sangeeta1998/wasm_execution

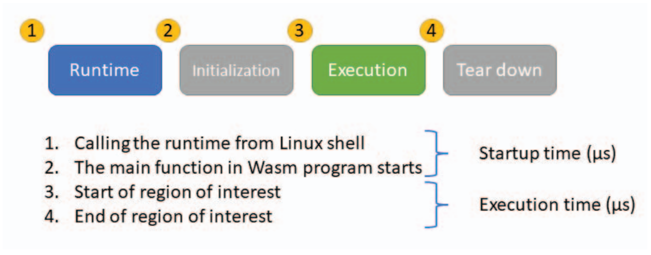

1
2
3
4
Runtime
Initialization
Execution
Tear down
1. Calling the runtime from Linux shell
2. The main function in Wasm program starts
3. Start of region of interest
4. End of region of interest
Startup time (µs)
Execution time (µs)

Fig. 2: Timestamps and regions of interest.

the benchmark starts to execute. This time includes the time it takes for the operating system to start the runtime process, the time to read the Wasm file from disk, instantiation of the WebAssembly module, possibly compilation of WebAssembly to native code (unless the runtime is an interpreter), and starting to execute the Wasm code.

The *initialization* phase is in the benchmark when the internal data structures of the benchmark are configured or generated. This phase is generally not representative of the benchmark and we therefore disregard it from our measurements.

The *execution phase* is our region of interest from which we derive the execution time. Finally, the *tear down* phase is when the benchmark prints out statistics or results that are generally not part of the actual benchmark. This phase is also disregarded.

As mentioned before, we have instrumented the code to take four timestamps: ts_1 when we call the runtime from the shell, ts_2 when the main function in the benchmark starts executing, ts_3 at the beginning of the region-of-interest, and ts_4 at the end of the region-of-interest. We collect these timestamps with a microsecond resolution and take the difference of ts_2-ts_1 to be the startup time and the difference between ts_4-ts_3 to be the execution time.

Since we use different architectures and platforms with quite different performance profiles, but we still would like to be able to compare them, we also take these timestamps for each benchmark running them natively on each platform. We can then compare the relative startup and execution times, as well as the absolute.

C. Experimental Setup

To investigate the performance and resource characteristics, we conducted experiments utilizing four different hardware platforms with three different instruction set architectures. Table I provides some details on the platforms used.

The AWS instances represents powerful cloud datacenter server instances. Both the Neoverse-N1 CPU and the Intel Xeon 8275CL CPU are powerful processors with advanced parallel pipelines using dynamic instruction scheduling, branch prediction, and cache prefetching mechanisms. In contrast the Nvidia Jetson and the StarFive VisionFive2 boards represent the embedded domain more likely to be found at the edge. The processors employed on these boards exhibit a more modest performance profile. First of all, the clock frequency is significantly lower. Secondly, the cores themselves are simpler and with smaller amount of cache memory. Still, on paper, the Nvidia Jetson with its ARM Cortex-A57 should provide better performance due to its four-way superscalar, out-of-order core architecture and 50% higher clock frequency compared to the VisionFive2 board.

D. WebAssembly Runtimes Used

There are now quite a few different WebAssembly runtimes to choose from to run Wasm programs outside the browser. We considered and tested runtimes such as Wasmer, WasmEdge, Wasm3, and Wazero in addition to the two runtime projects harbored in the Bytecode Alliance: Wasmtime and WAMR (WebAssembly Micro Runtime). In the end, we chose **Wasmtime** and **WAMR** because they were consistently the best performing runtimes available on all platforms we wanted to study and because they both also are tightly connected to the WebAssembly community group defining the standardization of WebAssembly.

1) Runtime 1: Wasmtime

Wasmtime is a WebAssembly runtime implemented in the Rust programming language, emphasizing standards compliance, efficiency, and high performance. It is built upon the Cranelift JIT compiler, also a Bytecode Alliance project, and supports the WASI standard to allow WebAssembly programs to interact with the environment. Wasmtime also puts great emphasis on safety and security, thanks to Rust's safety guarantees and a rigorous process for introducing new features.

In all our experiments, we use Wasmtime, which was built from a fork of the official repository(15 November 2023), with default configuration using the command: `cargo build --release`. The resulting executable is about 37 MB, but there are ways to reduce the size significantly if we restrict functionality and disable logging, for instance, here[9].

2) Runtime 2: WAMR – WebAssembly Micro Runtime

Our second runtime selection was the WAMR – WebAssembly Micro Runtime. The goal of WAMR, besides being standard compliant, is to be modular and that we can build versions targeting different devices, like IoT devices which requires small footprint and low memory usage as well as servers with more resources. WAMR is implemented in C, and we have built four versions with different characteristics according to the description below.

Interpreter: An interpreting runtime designed for small devices such as in IoT. Executable size: 293 kB.

LLVM JIT: A just-in-time compiler based on LLVM compiles the Wasm bytecode before starting execution. Executable size: 50 MB. This is the default execution mode we have used in our experiments as it is of similar complexity as that of Wasmtime.

Fast JIT: A very fast JIT compiler but not as high-quality code as with LLVM. Executable size: 663 kB.

[9]https://docs.wasmtime.dev/examples-minimal.html

TABLE I: Hardware Architectures Used in Experiments

Platform	CPU Model	Clock frequency	Threads per Core	Cores per Socket	Caches
ARM AWS c6g.metal	Neoverse-N1	3 GHz	1	64	L1d: 64 KiB, L1i: 64 KiB, L2: 1 MiB, L3: 32 MiB
X86_64 AWS c5.metal	Intel Xeon 8275CL	3 GHz	2	24	L1d: 32 KiB, L1i: 32 KiB, L2: 1 MiB, L3: 71.5 MiB
Nvidia Jetson Nano	ARM Cortex-A57	1.5 GHz	2	4	L1d: 32 KiB, L1i: 48 KiB, L2: 2048 KiB
StarFive VisionFive2	SiFive U74 Core	1 GHz	1	4	L1d: 32 KiB, L1i: 32 KiB, L2: 2048 KiB

Multi-tier JIT:This version starts with the Fast JIT and in parallel, other threads compile the Wasm bytecode with the LLVM JIT which will take over when they are ready. Executable size 50 MB.

The WAMR executable is called `iwasm` which is why we interchangeably use this term to denote WAMR in the result section below.

V. Results

We have organized our analysis of the results as follows: First, we analyze the performance in the region of interest and compare it to the native execution time on each platform. This removes any overhead due to JIT compilation or the instantiation of the WebAssembly module. Next, we analyze the startup time which explicitly consider these overheads. We make a special study of the WAMR execution modes mentioned above and finally study the performance when ahead-of-time (AOT) compilation of the WebAssembly files is used instead of interpretation or JIT compilation.

A. Execution speed

Figure 3 shows the execution time relative to native execution for the six selected benchmarks; figure (a) shows it for Wasmtime and (b) for WAMR. The execution time for each platform is shown as bars going from left to right: aarch64 (AWS ARM64), jetson (Nvidia Jetson), riscv64 (VisionFive2) and X86_64 (AWS X86_64).

Some things can be immediately observed in Figure 3. The execution time for several benchmarks on both runtimes and on many hardware platforms is near native. In fact, looking at all 51 benchmarks that we ran, more than 50% of them run within a factor of 2x with Wasmtime from the native execution time, and almost 50% do the same on WAMR. This reinforces the claim that, from a performance perspective, WebAssembly is a suitable execution vehicle for cloud-based applications to achieve architecture independence. Table II also shows that Wasm modules are also resource effective with a binary size which is just a small fraction of the corresponding native executable size.

Figure 4 shows big differences in execution time between the runtimes and platforms (note the log scale on the y-axis). Not surprisingly, the embedded platforms (jetson and riscv64) are comparatively slower, in particular riscv64 with its, not only slower, but also less advanced processor.

Except for the benchmarks: MB-cjpeg and WA-bzip2, there is no big difference between the two runtimes. Both use similar technology: JIT compilation. WAMR uses LLVM which is a bit more mature, particularly for the riscv64 architecture than

TABLE II: Size of WebAssembly modules compared to statically linked native executables.

Benchmark	Native size (X86_64)	Wasm size
JS2-tsf	1.5 MB	414 kB
MB-cjpeg	1.2 MB	152 kB
PB-3mm	897 kB	48 kB
PB-correlation	897 MB	47 kB
PB-jacobi-2d	897 MB	47 kB
WA-bzip2	1.1 MB	142 kB
WA-mnist	915 kB	56 kB

Cranelift, which is the compiler used in Wasmtime. Cranelift has three characteristics which differ from LLVM: i) it is simple and easy to contribute to, ii) it is fast, and iii) it emphasizes security needed for Wasm execution. Cranelift is significantly faster than the LLVM JIT which affects startup times.

MB-cjpeg and WA-bzip2 are the shortest running benchmarks, and therefore, any difference in absolute execution time shows up larger relative to native.

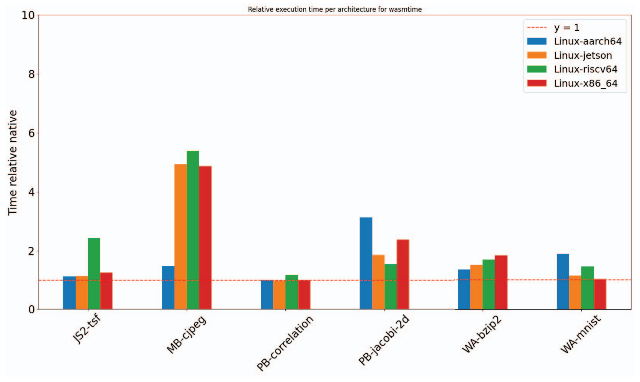
The PB-correlation for WAMR is interesting, as it shows a faster execution time than the native execution for aarch64 and X86_64. Using performance counter analysis tools such as perf revealed that the total number of instructions executed by WAMR is larger than in native execution, but the total cycles of native execution was much higher off-setting the advantage of less instructions. We have not yet found the root cause of this difference which needs further investigation.

B. Startup times

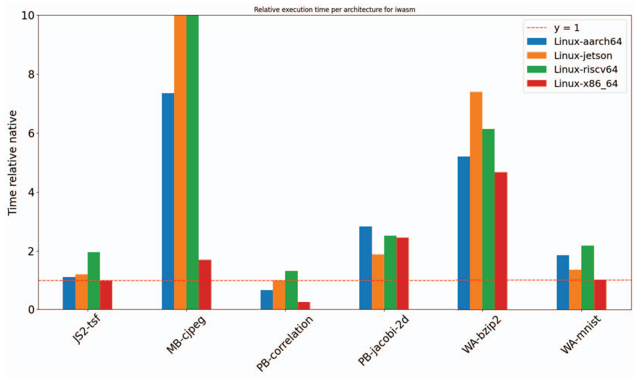
Next, we investigate startup times. Figure 5 shows the absolute startup times for Wasmtime (a) and WAMR (b). The first striking observation is that the startup times for WAMR with the LLVM JIT compiler is about 10 times longer than for Wasmtime with the much faster Cranelift compiler. We will see later that the fast JIT compiler in WAMR effectively mitigates this deficiency.

Another observation is that startup times of wasmtime on the server architectures are a few tens of milliseconds. Comparing that with regular FaaS startup times or the time to get a docker container up and running, the use of Wasmtime is a factor of ten better.

Finally we observe that the embedded platforms, in particular RISCV64, are significantly slower. This is in line with the 10x performance difference we saw for RISCV64 compared to X86_64 in Figure 4. The difference between server architectures and the Jetson platform can mostly be attributed to the difference in clock speed while we are not fully confident about the larger difference to the RISCV64

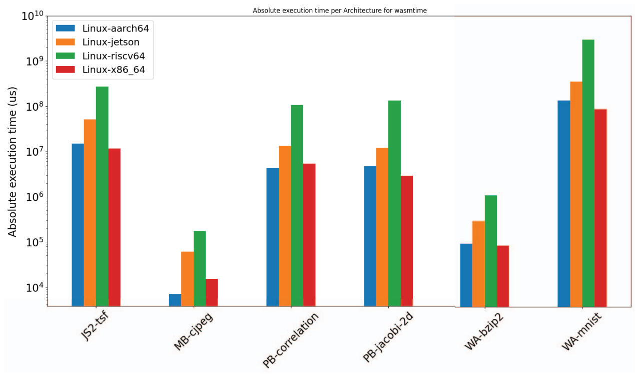


(a) Wasmtime

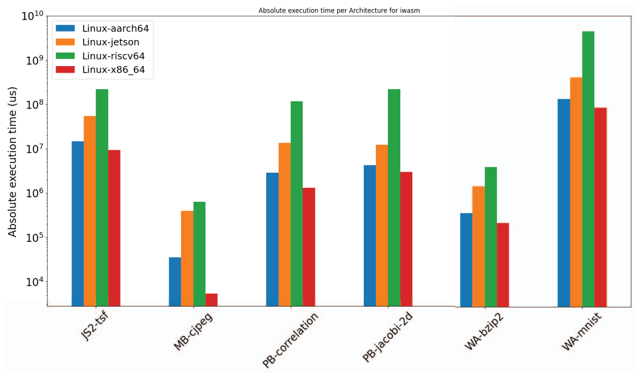


(b) WAMR (LLVM JIT)

Fig. 3: Relative execution time per architecture.



(a) Wasmtime



(b) WAMR (LLVM JIT)

Fig. 4: Absolute execution time per architecture.

platform. We know that the SiFive core used is not as advanced as the ARM Cortex-A57 but exactly how that plays out in performance is outside the scope of this paper.

C. WAMR execution modes

As explained in section IV-D2, WAMR can be built to use one or more different execution modes[10]. Figure 6 shows the performance impact of these execution modes in relation to native execution and Wasmtime on X86_64 only, as the fast JIT is only available on this platform. In summary, the *interp* mode is interpretation, which is obviously slower, but with a very small footprint of less than 300 kB. The *llvm-jit* uses the LLVM compiler as JIT, which has a long startup time, but generally provides better performance than the Cranelift JIT compiler of Wasmtime. The *fast-jit* mode is using a fast light weight JIT compiler similar to Cranelift, but with a significantly smaller memory footprint. It does not achieve the same performance as LLVM-jit, but as can be seen in Figure 7 the startup time begins to approach the one for interpretation mode.

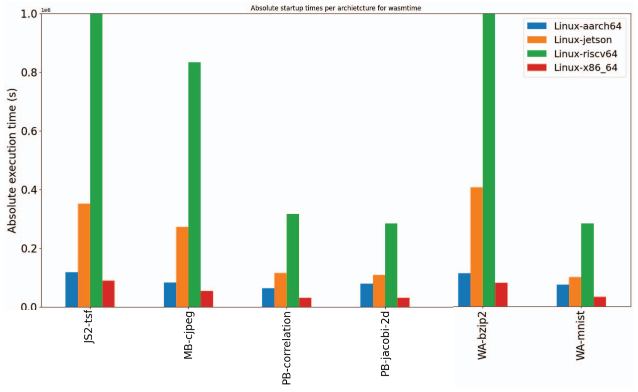
[10]https://bytecodealliance.github.io/wamr.dev/blog/introduction-to-wamr-running-modes/

Finally, WAMR also has a *multi-tier* mode which starts with the fast jit and in a parallel thread uses the LLVM jit to improve the result of the fast jit. It achieves in general similar performance as the LLVM-jit with the startup time of the fast jot.

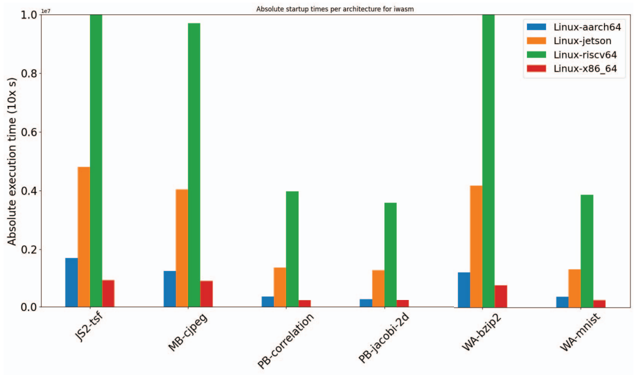
D. Ahead-of-time compilation

Both Wasmtime and WAMR provide ahead-of-time (AOT) compilation. Figure 8 shows the relative execution time on (a) ARM64 on AWS, (b) RISCV64, and on (c) X86_64 for both runtimes. Wasmtime uses the Cranelift compiler to build an AOT-image which can be executed by Wasmtime. WAMR has a separate command to perform the AOT-compilation using the LLVM compiler (which is the same as used in LLVM JIT). We observe that the relative performance is even closer to native execution, and, apparently, the resulting startup times become very low. Typically less than 10 μs.

Ahead-of-time compilation may seem compelling, but a significant drawback is that we then lose the ability to dynamically free ourselves from the architectures of the computing infrastructure. Once again the executable image has become architecture dependent. Still, the scenario is considerably better



(a) Wasmtime



(b) WAMR (LLVM JIT)

Fig. 5: Absolute startup times per architecture. In (a) the y-scale is in seconds while in (b) the y-scale is in 10x seconds.

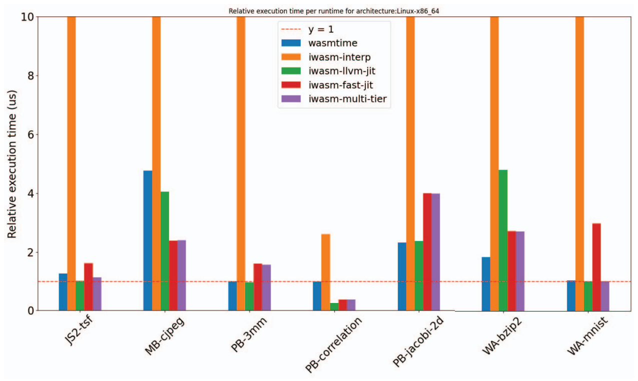


Fig. 6: Relative execution time for different WAMR execution modes on X86_64.

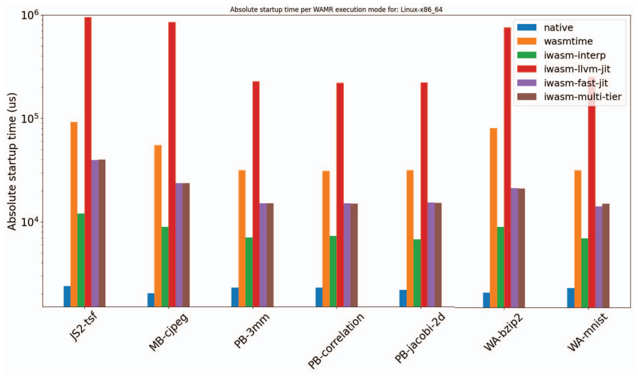


Fig. 7: Startup time for different WAMR execution modes on X86_64.

than having to re-compile the source code, which is dependent not only on the processor architecture but also on the runtime environment, OS, and so on, which WebAssembly does not require.

E. Wasmtime: A Performant Runtime

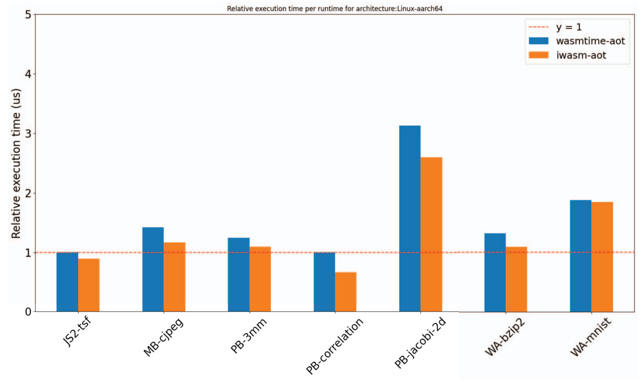
As evident from the execution time shown in Section V which in most cases provides near native execution, Wasmtime emerges as the standout performer, consistently delivering optimal results across all architectures. Figure 5 also illustrates the absolute startup times of the Wasmtime runtime which predominantly fall within the range of 30-150 ms. Notably, specific applications on X86_64 achieve remarkably low startup times, as fast as 1-2 ms. The X86_64 architecture consistently exhibits the lowest startup time across all selected benchmarks followed by the AARCH64 AWS and ARM Jetson Nano, while RISCV consistently portrays the highest startup time.

F. WebAssembly Micro Runtime Insights

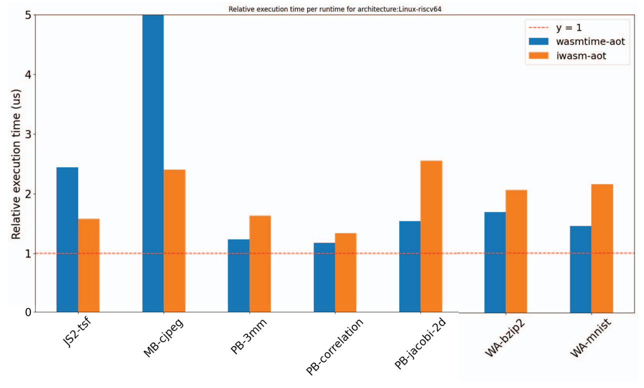
As illustrated in Figure 5, for AWS instances, almost all benchmarks experience consistent low (sub 10 ms) startup times. Larger binary files can lead to higher startup times. X86_64 maintains the lowest startup times, showcasing superior performance, followed by the AARCH64 AWS, ARM Jetson Nano, and RISCV. The iWasm startup times are consistently one-magnitude higher than Wasmtime across all architectures. The impact of architecture variation is more pronounced in Wasmtime, where the difference in startup times between X86_64 and AARCH64 AWS is more significant. In contrast, iWasm exhibits a more consistent difference between these architectures.

G. Runtime Selection Considerations

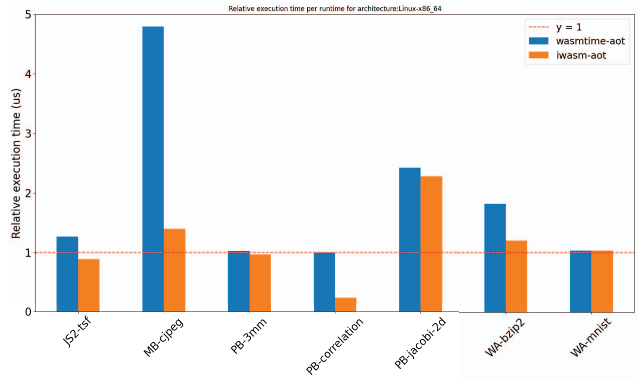
Choosing the appropriate runtime depends on specific deployment requirements. For scenarios involving the migration of Wasm modules across different architectures, WAMR (interpretation) on ARM/RISCV with low memory usage is a viable choice. However, for greater compatibility, feature support, and generally good startup times, Wasmtime stands out as the preferred runtime. Its extensive platform support, feature-rich nature, and competitive startup times make it a compelling choice for a wide range of applications.



(a) AWS ARM



(b) RISCV64



(c) AWS x86_64

Fig. 8: Relative execution when doing ahead of time compilation of the WebAssembly bytecode.

VI. Discussion

Our experiments observed significant differences in performance among various benchmarks and hardware architectures. Differences in hardware architectures played a crucial role in shaping our observations. For instance, ARM Cortex-A57's out-of-order pipeline design and support for SIMD extensions and virtualization proved advantageous, contributing to competitive performance. On the other hand, the RISCV architectures exhibited a dual-issue, in-order execution pipeline. Although promising, the performance of RISCV platforms lagged behind, likely due to the compiler's optimization and instruction alignment challenges. The unique features and configurations of the SiFive U74 core used in the VisionFive2 board, such as the L2 cache and error correction capabilities, contributed to the performance characteristics observed.

It is important to note that among the tested runtimes, Wasmtime emerged as the sole runtime capable of consistently delivering high performance across various hardware architectures. This presents an opportunity to further enhance the other runtimes for compatibility and competitiveness.

A. Future Research Opportunities

The observed performance characteristics brings the way for future investigations. Questions regarding the significant difference in Wasmtime startup time between architectures, the reasons behind the relatively slower performance of RISCV, and strategies for runtime executable size reduction will be integral to our ongoing research. Future investigations must be done into the underlying reasons for this discrepancy. Certain benchmarks displayed relatively superior performance on embedded architectures. Understanding the factors contributing to this phenomenon is crucial to utilize the full potential of WebAssembly in resource-constrained environments. Future research questions can be investigated in terms of

Startup Time Discrepancies: What are the underlying reasons for the significant differences in Wasmtime startup times across various architectures? How can we optimize startup performance to ensure consistency on diverse platforms?

RISCV Performance Challenges: How can we enhance the performance of WebAssembly runtimes on RISCV platforms to bridge the performance gap ensuring performance parity across diverse hardware?

Runtime Executable Size Reduction Strategies: In addressing concerns about the size of runtime executables, what approaches can be explored to reduce their size while maintaining optimal performance and functionality?

Benchmark Performance on Embedded Architectures: What factors contribute to the superior performance of certain benchmarks on embedded architectures? How can future research focus on these factors to fully leverage the potential of WebAssembly in resource-constrained environments?

VII. Conclusion

WebAssembly, as a versatile compilation target, has emerged as a pivotal ally for developers navigating the diverse landscape of cloud-to-edge computing. In this paper, we have conducted an extensive performance analysis of WebAssembly runtimes, pushing the boundaries of cross-architecture evaluation. Our objective was to empower developers, enabling them to craft applications seamlessly across diverse hardware platforms without the worrying about hardware-specific considerations. Our research embarks on a *first-of-its-kind* study that spans across four distinct platforms: AWS servers using X86_64 and ARM64, and two embedded platforms. Nvidia Jetson with ARM64 and VisionFive2 with RISCV64.

By subjecting WebAssembly runtimes to a range of carefully annotated benchmarks, we explored the performance characteristics that challenge conventional assumptions. The findings provide valuable insights on the compatibility, efficiency, and deployment considerations of WebAssembly on varied platforms.

In the area of benchmarking, we contribute a calibrated and instrumented programs to the existing WaBench benchmarks suite, providing a valuable resource for the research community. This suite extends the comprehensiveness of future studies, ensuring precise and reliable evaluations for WebAssembly runtimes. We navigated the challenges of runtime selection, benchmark suitability, and cross-architecture optimization, providing a roadmap for developers seeking to use the full potential of WebAssembly in their applications. As cloud-to-edge computing continues to evolve, the congizance presented in this paper will be instrumental in shaping the future of application development, where developers can truly write once and deploy anywhere.

Acknowledgment

This research has been partly funded by the Luxembourg National Research Fund (FNR) under contract number 16327771 and has been supported by Proximus Luxembourg SA. For the purpose of open access, and in fulfillment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

References

[1] P. Gkonis, A. Giannopoulos, P. Trakadas, X. Masip-Bruin, and F. D'Andria, "A survey on iot-edge-cloud continuum systems: Status, challenges, use cases, and open issues," *Future Internet*, vol. 15, no. 12, p. 383, 2023.

[2] G. P. Mattia and R. Beraldi, "Leveraging Reinforcement Learning for online scheduling of real-time tasks in the Edge/Fog-to-Cloud computing continuum," in *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, pp. 1–9, Nov. 2021. ISSN: 2643-7929.

[3] S. Nastic, T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, D. Vii, and Y. Xiong, "Polaris Scheduler: Edge Sensitive and SLO Aware Workload Scheduling in Cloud-Edge-IoT Clusters," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pp. 206–216, Sept. 2021. ISSN: 2159-6190.

[4] A. Orive, A. Agirre, H.-L. Truong, I. Sarachaga, and M. Marcos, "Quality of Service Aware Orchestration for Cloud–Edge Continuum Applications," *Sensors*, vol. 22, p. 1755, Jan. 2022. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

[5] 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.

[6] M. Chadha, N. Krueger, J. John, A. Jindal, M. Gerndt, and S. Benedict, "Exploring the use of webassembly in hpc," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 92–106, 2023.

[7] 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.

[8] 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.

[9] 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.

[10] B. Bosshard, "On the use of web assembly in a serverless context," in *Agile Processes in Software Engineering and Extreme Programming–Workshops*, p. 141, 2020.

[11] 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.

[12] 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.

[13] 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.

[14] D. Hockley and C. Williamson, "Benchmarking runtime scripting performance in webassembly," 2022.

[15] V. Kjorveziroski and S. Filiposka, "Webassembly as an enabler for next generation serverless computing," *Journal of Grid Computing*, vol. 21, no. 3, p. 34, 2023.

[16] 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.

[17] 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.

[18] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, pp. 265–279, 2020.

[19] Q. Stiévenart, C. De Roover, and M. Ghafari, "The security risk of lacking compiler protection in webassembly," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pp. 132–139, IEEE, 2021.

[20] 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.

[21] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An embedded trusted runtime for webassembly," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 205–216, IEEE, 2021.

[22] 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.

[23] F. Breitfelder, T. Roth, L. Baumgärtner, and M. Mezini, "Wasma: A static webassembly analysis framework for everyone," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 753–757, IEEE, 2023.

[24] 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.

[25] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, "Challenges and opportunities for efficient serverless computing at the edge," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pp. 261–2615, IEEE, 2019.

[26] 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.

[27] 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.

[28] 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.

[29] S. Kakati and M. Brorsson, "Webassembly beyond the web: A review for the edge-cloud continuum," in *2023 3rd International Conference on Intelligent Technologies (CONIT)*, pp. 1–8, IEEE, 2023.

[30] S. Wallentowitz, B. Kersting, and D. M. Dumitriu, "Potential of webassembly for embedded systems," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–4, IEEE, 2022.