WebAssembly potentials: A performance analysis on desktop environment and opportunities for discussions to its application on CPS environment

João Lourenço Souza Junior
Atlantic Institute
Fortaleza, Brazil
joao_lourenco@atlantico.com.br

Davi Cedraz S. de Oliveira
Atlantic Institute
Fortaleza, Brazil
davi_cedraz@atlantico.com.br

Victor Nobrega Praxedes
Atlantic Institute
Fortaleza, Brazil
victor_praxedes@atlantico.com.br

Dennys da Silva Simiao
Atlantic Institute
Fortaleza, Brazil
dennys_simiao@atlantico.com.br

***Abstract*—The Web started as a simple document-sharing network and today has evolved to become a consolidated and ubiquitous platform for creation and application distribution. To explore its demands, web browser vendors have been working on new technologies like WebAssembly, a new type of machine language for a conceptual machine instead of a real physical machine, supported by the modern web browsers, providing new features and greater performance for web applications. At the other end, embedded devices have also evolved along with applications. However, there are still semantic heterogeneity, maintainability, and development issues inherent to the vast number of devices and services that operates in the numerous domains of Cyber-Physical Systems (CPS). The overall objective of this work is to study the WebAssembly technology through a performance analysis in a desktop environment, presenting empirical comparisons between the execution of a program compiled in native machine code and the same program compiled in WebAssembly, to verify its flexibility to compile code written in different languages for web applications and maintain similar performance to their native applications counterpart. We also point out the opportunities and challenges to potentially apply WebAssembly as a semantic abstraction layer for embedded devices in CPS development.**

***Index Terms*—WebAssembly, Internet of Things, Cyber-Physical Systems**

I. Introduction

The current Web scenario has been helping society to redefine the way organizations and individuals automate actions, optimize processes, communicate, and effectively collaborate with each other [1]. All of these applications can stay connected and all of this started as a simple document sharing network. Today it has evolved to become a consolidated and ubiquitous platform for creating and distributing applications, being accessible across a wide range of operating systems and devices (e.g., desktops, Smartphones, Tablets, among others).

Based on this evolution, several emerging technology fields are bringing new technologies to innovate in this scenario, browsers providers started working on new technologies to properly integrate applications into their software [2]. In April 2015 the World Wide Web Consortium (W3C), formed the WebAssembly Working Group, where engineers from the top four web browsers (Chrome, Firefox, Safari, and Edge) designed low-level portable code so collaborative called WebAssembly (abbreviated to Wasm).

The WebAssembly offers a compact representation, validation, and compilation, in an efficient way, with low-level security, and without execution overhead. This technology is a new type of machine language that can be run in modern web browsers, providing new features and increased performance for web aplications. The idea is not to write using Wasm syntax directly, but rather, to be an effective compilation target for low-level languages like C, C++, Rust, etc. It is also designed to run alongside JavaScript, allowing both work together. The WebAssembly is not linked to a specific programming model, being platform-independent and, from its formal semantic definition, its compiled code is written in different languages to run applications on the web with similar performance to native applications[1] [3]. But, the Wasm specification doesn't make any web specific assumptions or provide web specific features, and it can be employed in other environments [4].

Therefore, the main objective of this work is to validate the performance equivalence of the use of WebAssembly in comparison to the use of native compiled platforms and applications in a desktop environment, presenting empirical quantitative comparisons, focusing on performance measures of algorithmic execution. In addition, we also raised the discussion of the potential use of Wasm as a semantic abstraction layer to develop embedded devices in CPS development.

The work will be based on bibliographical and exploratory research, carried out from the knowledge obtained through scientific articles, books, and documentation referent to the technologies that involve WebAssembly. A proof of concept

[1]Native applications refer to applications that are compiled for an operating system and specific hardware.

will be drawn up and subjected to a set of tests that will produce outputs to make the above comparisons and analyzes.

The remainder of this paper is organized as follows. Section II presents a deep theoretical foundation and literature review to explore the state of the art of WebAssembly. Section III presents the experiment conducted in this work, defining the process of execution, collection, and analysis of the performance comparison experiment by running WebAssembly on a desktop environment. In Section IV, we argue in details the opportunities of using WebAssembly in the context of Cyber-Physical Systems and IoT environments. Section V presents the related work. Section VI presents the threats to validity. Finally, Section VII concludes this paper and points out directions for future work.

II. WebAssembly

The WebAssembly is formally defined as a secure, portable, and low-level code format, like Assembly, designed to efficient execution and compact representation. Its primary objective is to enable high-performance applications on the Web, however, it doesn't make any specific assumptions about the Web or provides specific Web features, therefore it can be used in other environments. Furthermore, its specification defines the concrete representations (binary format and the text format), structure, execution, and validation of this language [4]. The Wasm code can be generated in two formats, binary and textual, both based on a common structure, described in the form of an abstract syntax. The Binary format is a linear encoding of this abstract syntax, it is defined by a grammar of attributes whose only terminal symbols are bytes. In the textual format, the Wasm code is represented as a large S-Expression, consisting of a large list of instructions [3].

A. Instruction Set Architecture - ISA

The process of transforming a piece of code written in a high-level programming language to machine code is defined from a specific instruction set architecture (ISA), and the most common type of ISA are: reduced instruction set computer (RISC), complex instruction set computer (CISC), and stack-based. Wasm is a little bit different from these architectures, it is a virtual stack-based ISA, that is, a machine language for a conceptual machine instead of a real physical machine and, as such, has many use cases and can be incorporated in many different environments.

In general, when a Wasm application is available, it is loaded and executed by the web browser, and not directly on the target machine, the application does not need to know the architecture (ARM, x86 ...) of this machine. After the generation of the binary file, the browser can download this file and then decode the WebAssembly to the native machine code, this decoding is very efficient due to the similarity between Wasm and Assembly.

B. Compile

As Wasm is a virtual ISA that isn't linked to any specific programming language, but currently there exists a set of languages that can be compiled to Wasm, however, the languages C, C++, and Rust have tool-chains in more advanced stages of development. Therefore, the experiments in this work were implemented in these languages.

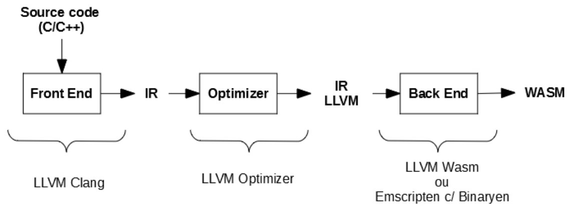

Fig. 1: WebAssembly compile process

The Figure 1 illustrates the compile process of C/C++ to Wasm binary, which consists of transforming the source code into an intermediate representation (IR) using the Clang front end from LLVM [5], so the IR can be optimized by LLVM, and finally, a back end LLVM generate the WebAssemly code. The back end is still evolving, in constant development, resulting in complications and instability at the moment.

C. Execution

As shown in Figure 2, the WebAssembly execution process involves decoding the Wasm code into native machine code, validation, instantiation and program invocation [4].

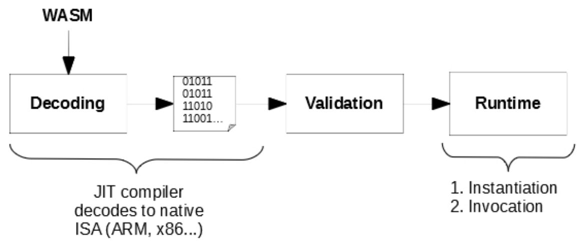

Fig. 2: WebAssembly execution process

For the Web environment, the Wasm reuses the infrastructure of the mechanisms of JavaScript execution, especially JIT compilers. However, the WebAssembly System Interface (WASI) specification and API provide support to run Wasm code outside the web browsers [6].

Although Wasm is mostly used in browsers, it was designed for any sandbox environment, not just for the web [2]. In fact, there is an effort on the part of the community to apply WebAssembly beyond browsers, since this technology provides speed, scalability, and security to run the same code on any operating system or computer architecture [3]. The Wasm3 project, for example, aims at creating a WebAssembly interpreter, allowing the execution of Wasm files on a wide variety of devices, including microcontrollers [7]. This is a goal similar to the WebAssembly Micro Runtime project (WAMR), an independent micro runtime for WebAssembly [8]. Both projects based on the WASI specification.

III. Experiment: Performance Comparison

The experiment aims to analyze the performance of a digital image processing application in the native environment versus web browsers. Since this is demanding in computational terms, it is a relevant experiment to analyze the performance of applications compiled for Wasm under the quantitative aspect. Its result provided the inputs to make empirical performance comparisons and an analysis of the impact of using WebAssembly in the development of multi-platform applications.

A. *Proof of Concept Development*

The proof of concept consists of a concept program to reproduce frames from a video highlighting the detected faces using the Haar object detection algorithm feature-based cascade classifiers, an approach based on machine learning where the function of cascade is trained from many positive and negative images, providing the classifier with the ability to detect objects in other images. OpenCV already contains many pre-trained classifiers for faces, eyes, smiles, etc. [9].

B. *Test Scenarios*

A set of tests were applied to the concept program, which produced the necessary data to accomplish the objectives of this work. These tests were designed based on a systematic approach to performance measurement, where objectives, test data, metrics, parameters, and expected results were defined.

1) Objective: Measure and collect the execution times of the face detection program algorithm and their respective subroutines, image pre-processing, face marking and image post-treatment, in the native execution environment and WebAssembly, through the system Windows 10 Pro 64-bit operating system and Google Chrome web browsers version 75.0.3770.142 64-bit and Mozilla Firefox version 68.0.1 64-bit. Logically, considering the same computational resources for all test scenarios, an x86 architecture computer with the following configurations: Intel (R) Core (TM) i7-8550U 1.80GHz processor/CPU; 16GB DDR RAM memory; 100GB SSD hard drive and Windows 10 Pro 64-bit operating system.

2) Test Data: The test data set consists of three videos: DATASET_1[2], DATASET_2[3] and DATASET_3[4]. The selection criteria were: be a public accessible video, that is, without restrictions for reproduction and be available in MPEG-4 format with a standard resolution of 1920px by 1080px.

3) Metrics: The performance times of each phase of the face detection process were considered as performance indicators, and also the total frame reproduction time with the face detected or not. Where the time is inversely proportional to the performance, that is, less execution time means better performance.

[2]https://www.youtube.com/watch?v=EWUdGRAwUpY

[3]https://www.youtube.com/watch?v=vQtLX6pW5eA

[4]https://www.youtube.com/watch?v=RuL5jVqc4Tg

4) Parameters: During the development of the proof of concept and the experiments execution, were identified parameters that influenced the performance of the program:

- **Video resolution**: The execution times proved to be directly proportional to the size of the pixels matrix that composes the processed videos frames. The different resolutions applied in the tests are classified by LOW, MEDIUM, and HIGH with the pixel matrix 480px x 320px, 768px x 480px, and 1280px x 720px respectivaly;
- **Test data**: Each video of the test data described in Section III-B2 produced different execution times, even when compared with the same resolutions;
- **Execution environment**: Different times were identified for the different execution environments.

5) Expected results: According to the parameters defined in the previous section III-B3, tests were performed per execution environment, test data and resolution of the videos, generating a total of 27 different test scenarios following the metrics defined in Section III-B3.

C. *Results Analysis*

From the data obtained with the execution of all test scenarios and the experiences attained during the research and development process, we have elaborated quantitative analyzes. The execution of 27 test scenarios produced a significant measurement database with the amount of 38771 measurements. Based on these data, the analysis process was carried out, which is divided into three stages: data standardization, statistical analysis and comparative analysis.

1) Standardization of results: The test results were loaded and unified by test data (DATASET _1, DATASET _2 and DATASET _3). Then it was identified that the most common number of faces marked in a video frame was 1 (one), so a filter was applied to consider only the results where only 1 (one) face was detected.

2) Statistical analysis: With the result of the standardized tests, the mean and median of each metric per parameter were calculated. It was observed that the mean and the median showed similar values for all metrics, but, since the mean is more sensitive to unusual measurement points, the calculation of the median was adopted in all performance comparisons.

3) Comparative analysis: Based on the statistical analysis demonstrated in section before, the data were grouped and organized into graphs for better visualization of the information. And all the metrics presented in Section III-B3 were analyzed.

Pre-processing: In the performance comparison of the execution of the pre-processing of video frames, we have noticed that the version of WebAsembly that runs on Chrome and Firefox got better performance than the native version, since the best execution times occurred in the version running on Chrome, reaching 3.0 times faster than the native version and 2.0 times faster than running on Firefox.

Face detection: is the main functionality of the proof of concept developed and, as expected, it was the routine that demanded the longest execution time in all execution environments. Unlike what occurred in the pre-processing, the native version was 3.32 and 4.31 times more performance than the WebAssembly code executed in the Firefox and Chrome web browser respectively.

Marking of detected faces and **Post-processing**: The analysis of the execution times of these routines allowed us to identify that they have equivalent performance regardless of the variation of the parameters. However, it is worth noting that the execution times of the module with WebAssembly were registered in the order of nanoseconds.

Overall performance: The last comparative analysis was made with the total processing time measurements, that is, the execution time needed to capture the original video frame, process and reproduce the copy. With the data analysis shown in Figure 3, it became clear to us that the biggest bottleneck to the execution time of the program was the execution of the face detection algorithm provided by OpenCV. Therefore, similarly to the analysis of the face detection process, the native version performed better than the version of the concept program ported to WebAssembly, being 3.42 times faster than running on Firefox and 4.19 times faster that running on Chrome.

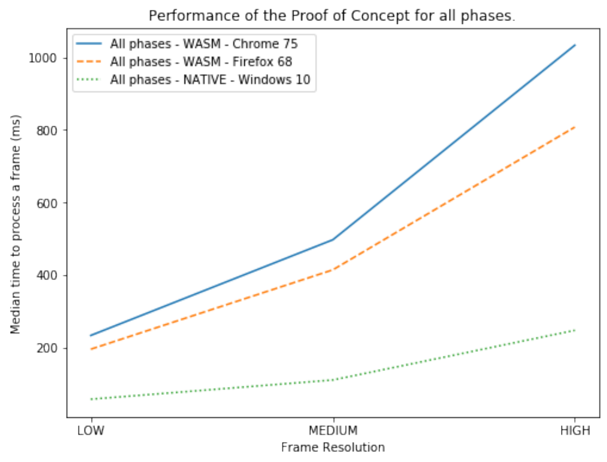

Fig. 3: Performance comparison of Proof of Concept Program - WebAssembly (Chrome and Firefox) vs Native (Windows)

The results showed that the WebAssembly version of proof of concept achieved similar performance to the native version when considering the following performance metrics: pre-processing, face marking, and post-processing. However, the face detection routine was significantly better in the version compiled for the native machine. This difference can be attributed to the fact that implementation of the *Haar feature-based cascade classifiers* algorithm by OpenCV contains features that haven't yet been optimized for Wasm.

It is important to note that the face detection routine provided by OpenCV is a computer vision library with thousands of C++ code lines with many features, which until the launch of WebAssembly would not possible to see on a web application. So, based on the results obtained we can conclude that WebAssembly has the real potential to bring better performance to web applications. In addition the Wasm bring to the web a niche of computational solutions that were restricted to the Desktop environment. And we could prove that WebAssembly provides the capacity to maintains the mains base code of an application independent of the target ISA.

IV. WebAssembly at CPS environment

In Cyber-Physical Systems, the devices assist users in several distinct domains and it requires a strong interoperability [10]. It is essential that devices of different architectures, communication protocols, and firmware semantic composition to be able to work together in the same ecosystem. The technological heterogeneity issue is still challenging due to the large number of devices and protocols, mostly for ubiquitous systems [11]. Web-services based approach (WoT, Web of Things) and semantic web technologies (SWoT, Semantic Web of Things) are now widely accepted as possible solutions, it has been leveraged to enrich devices and services with semantic annotations used to qualify them [12].

In this context, one technology that could solve some of those problems is WebAssembly. Its use in IoT devices is justified because it seeks to guarantee semantic interoperability, in addition to providing a safer environment associated with optimized code, low memory usage, portability, and performance equivalent to native applications. Figure 4 contains a diagram that depicts the conceptual idea of a WebAssembly application on an IoT device. The Wasm Binary, an application compiled on WebAssembly, is executed through a platform-independent Wasm Runtime. The WASI for Embedded Systems is a platform-specific implementation that provides a Hardware Abstraction Layer (HAL), that is, it makes the hardware transparent to the upper layers. Finally, the firmware layer, the one closest to the hardware, is responsible for providing basic functionalities such as OTA interface for updating the Wasm application via network.

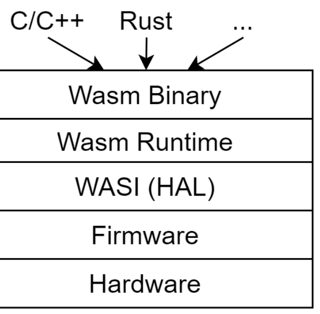

Fig. 4: WebAssembly as Hardware Abstraction Layer.

Based on our research on very recent developments and the experiment carried out in this work, we propose the three main facts that we believe corroborate with the wide application of Wasm in the CPS context:

- The WAMR and Wasm3 interpreter allows us to write the same firmware for different devices and different architectures [13]. By reducing or almost eliminating the semantic divergence in the source codes embedded in the devices, it is possible to isolate the business layer from the communication and sensing modules, contributing to its scale with much more security and flexibility.

- WebAssembly was designed to have portability, especially to the non-Web context, that means that it does not require any Web APIs and can be used as a portable binary format on many platforms, bringing benefits in tooling, performance, and language-agnosticism (since it supports C/C ++ level semantics).
- It has numerous unexplored applications for the non-web context [14], in the CPS ecosystem, it can be used as a HAL (Hardware Abstraction Layer) [15] and theoretically as a TEE (Trusted Execution Environment) which can facilitate OTA (Over the Air) firmware updates on devices, possibly becoming the future of Cyber-Physical Systems as well.

Mazhelis, Luoma, and Warma defines the IoT ecosystem and its devices as an ever-changing field [16]. We understand that it is due to a set of dynamic technologies that may change all the time in the way they are connected, communicated, and can be programmed. Therefore, supported by the latent need to build interoperable IoT devices in very dynamic and heterogeneous environments, Wasm as bytecode in an IoT architecture can be used by developers to improve standardization, virtualization, context-awareness, and many others Semantic Web of Things aspects [17]. We presented in this section some opportunities for using Wasm in a CPS context. Although the development of new SWoT approaches and Wasm novelty projects is rising, many studies need to be carried out before applying it in real production environments.

V. Related Work

In the last years, several proposals present WebAssembly as a quiet revolution of the Web [14] and the future of the web development [18], mainly due to its performance, flexibility, portability, and others aspects [3]. However, few of them have been truly investigated the use of Wasm as an embedded bytecode; Hardware Abstraction Layer; Trusted Execution Environment or system interface to present it as a Web Semantic Technologies to improve interoperability between devices, with different architectures.

Putra [19] concluded on his work that WebAssembly modules can be used to support embedded system-based biomedical sensors. He points out that Wasm has limitations to compensate with its portability, but can be applied for achieving safety and effectiveness on biomedical sensor devices development. The comparison performed by the author shows superiority for WebAssembly between native implementation on flashing operation time, managing to be almost 10 seconds faster; the Wasm module file size can be smaller than native file; and also a superior efficiency in the WebAssembly module usage than in the native-only usage. It was also shown that all WebAssembly modules execute longer than the native implementation. However, according to him, since the Wasm implementation may result in significantly smaller file size and shorter installation time, the use of the Wasm module was more sustainable during development.

Jacobsson and Willén [13] proposed the use of WebAssembly as a virtual machine standard, not only for web browsers but also for embedded systems. To demonstrate the usage of their WebAssembly system, they connect the Texas Instruments CC26x2R LaunchPad and their BLE stack SimpleLink to a photoplethysmogram (PPG) sensor connected to a Bluetooth Low Energy device. An advantage pointed out by them is that WebAssembly implementation is platform-agnostic, so they also have it working on Contiki-OS as well. Using Wasm, they developed a software to extract the heart rate and implements a simple signal processing application to receive the raw sensor data. Due to the effectiveness of WebAssembly, they claim it has enabled to run exactly the same code in all parts of the complete wearable sensor system. They also express that Wasm finally allows them to use one single programming language and environment to program all different parts of a system.

VI. Threats to Validity

This research can be affected by different factors, either by the results of other studies or internal factors that can weaken or invalidate our findings. Based on that, we gathered these factors to discuss in this section.

It is well known that, in software engineering, there is no "silver bullet", so it is in the vast amount of domains of Web and either Cyber-Physical Systems development there is no semantic technology that can cover all problems. Rourke [2] presents many WebAssembly limitations as not having, at the time of this work, important features like garbage collectors, thread support, or a defined standardization process. In fact, WebAssembly can be slower compared to native code (10% to 55%) and has a substantial performance gap [20] due to missing optimizations, code generation issues, and others inherent to the WebAssembly platform.

We are well aware that the main threat to the validity of this work is the discussion of the opportunities for applying WebAssembly on embedded devices, although we haven't applied yet an experiment or apply Wasm directly in CPS environment. The results obtained in the experiment carried out in this work, for the purpose of performance comparison, was applied in web test scenarios and enables us to infer WebAssembly's performance in heavy tasks such as image processing. If we look at the demands in the context of edge computing, the demand for such processing is getting closer and closer to that carried out on IoT devices [21], but the threat that the experiment carried out in this work was not applied to an embedded device still holds, therefore, we still cannot determine the differences in the results obtained.

To the best of our knowledge, specifically of WebAssembly use in Cyber-Physical Systems, even with some limitations, Wasm has code compiler performance similar to native code and its general usage can be more efficient than in the native-only usage. However, we must be aware that its performance problems can be a key deterrent in IoT projects where this attribute is a requirement, resulting in a major trade-off between performance and portability.

VII. Conclusions and Future Work

In this paper, we investigate the WebAssembly technology by conducting an experiment and compared the execution of algorithms' performance in both their native environment and web browsers. In addition, we discussed about the hypothesis that a Wasm program could have similar performance of a program compiled and executed on a native machine.

We first elaborate test scenarios in order to obtain a database of relevant performance measurements. With this data, statistical analyzes were carried out and concluded that the hypothesis raised in this work could not be proved or refuted. That is, because the results from the proof of concept showed we achieved a similar performance to their native version, considering the following performance metrics: pre-processing, marking of faces and post processing. Based on the obtained results, we can conclude that the Wasm has potential to bring better performance to web applications, in addition, Wasm is capable of taking not only computational power, but a niche of computational solutions that were restricted to the Desktop environment, such as complex applications running on IoT devices in a Cyber-Physical Systems (CPS) context.

By conducting several bibliographical researches to understand how Wasm can be applied and how it can contribute to the context of CPS, we have finding several works that corroborate with this hypothesis, and which also highlight the limitations of this technology. We concluded that non-web Wasm environment and its tools like WebAssembly Micro Runtime and Wasm3 interpreter allows us to write the same firmware for different devices and different architectures, which leads us to a new hypothesis: by compiling a high-level language to machine code for different processor architectures (x64, x86, ARM) it makes Wasm binary files much smaller in size (in Kb) and easier to execute, maintain and debug.

In this way, a future work would be to perform a formal experiment to evaluate the pros and cons in therms of interoperability, portability and performance, in comparison to the native development of a CPS embedded on a microcontroller. We plan to: understand what caused the Haar feature-based cascade classifiers run slower with WebAssembly in order to determine points of improvement and its limitations; apply the proposed use of Wasm3 and WASI as a Hardware Abstraction Layer to an CPS system; and test the performance of a larger set of algorithms that allows us to get better indication of Wasm performance, not only on the web but also on embedded devices.

Acknowledgment

We acknowledge that this work is a partial result of the studies performed on Cyber-Physical Systems innovation platform, supported by non-profit private company Atlantic Institute.

References

[1] Y. Dwivedi, M. Williams, A. Mitra, S. Niranjan, and V. Weerakkody, "Understanding advances in web technologies: evolution from web 2.0 to web 3.0," 2011.

[2] M. Rourke, *Learn WebAssembly: build web applications with native performance using Wasm and C/C++*, 2018, oCLC: 1059521746.

[3] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web Up to Speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 185–200. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062363

[4] W3C. (2019) WebAssembly Core Specification. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[5] LLVM, "Mirror of official llvm git repository located at http://llvm.org/git/llvm. updated every five minutes.: llvm-mirror/llvm," 2019, original-date: 2012-01-27T23:49:56Z. [Online]. Available: https://github.com/llvm-mirror/llvm

[6] "WASI," 2020, original-date: 2019-04-02T18:23:05Z. [Online]. Available: https://github.com/WebAssembly/WASI

[7] "wasm3," 2020, original-date: 2019-10-01T17:06:03Z. [Online]. Available: https://github.com/wasm3/wasm3

[8] "Wasm micro runtime," 2020, original-date: 2019-05-02T21:32:09Z. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime

[9] OpenCV. (2019) Face detection using haar cascades. [Online]. Available: https://docs.opencv.org/4.1.0/d7/d8b/tutorial_py_face_detection.html

[10] M. Ganzha, M. Paprzycki, W. Pawłowski, P. Szmeja, and K. Wasielewska, "Towards semantic interoperability between internet of things platforms," in *Integration, interconnection, and interoperability of iot systems*. Springer, 2018, pp. 103–127.

[11] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng, "Iot middleware: A survey on issues and enabling technologies," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 1–20, 2016.

[12] A. J. Jara, A. C. Olivieri, Y. Bocchi, M. Jung, W. Kastner, and A. F. Skarmeta, "Semantic web of things: an analysis of the application semantics for the iot moving towards the iot convergence," *International Journal of Web and Grid Services*, vol. 10, no. 2-3, pp. 244–272, 2014.

[13] M. Jacobsson and J. Willén, "Virtual machine execution for wearables based on webassembly," in *EAI International Conference on Body Area Networks*. Springer, 2018, pp. 381–389.

[14] A. Moller, "Technical Perspective: WebAssembly: A Quiet Revolution of the Web," *Commun. ACM*, vol. 61, no. 12, pp. 106–106, Nov. 2018. [Online]. Available: http://doi.acm.org/10.1145/3282508

[15] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn, "A hardware abstraction layer in java," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 4, pp. 1–40, 2011.

[16] O. Mazhelis, E. Luoma, and H. Warma, "Defining an internet-of-things ecosystem," in *Internet of Things, Smart Spaces, and Next Generation Networking*. Springer, 2012, pp. 1–14.

[17] A. Rhayem, M. B. A. Mhiri, and F. Gargouri, "Semantic web technologies for the internet of things: Systematic literature review," *Internet of Things*, p. 100206, 2020.

[18] M. Tyagi, M. Sharma, and P. Sharma, "The future of the web," *Available at SSRN 3563679*, 2020.

[19] R. P. Putra, "Implementation and evaluation of webassembly modules on embedded system-based basic biomedical sensors," 2019.

[20] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: analyzing the performance of webassembly vs. native code," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 107–120.

[21] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.