Dynamic IoT Applications and Isomorphic IoT Systems Using WebAssembly

1st Kentaro Kuribayashi
Pepabo R&D Institute
GMO Pepabo, Inc.
Tokyo, Japan
antipop@pepabo.com

2nd Yusuke Miyake
Pepabo R&D Institute
GMO Pepabo, Inc.
Fukuoka, Japan
miyakey@pepabo.com

3rd Kenji Rikitake
Pepabo R&D Institute
GMO Pepabo, Inc.
Kenji Rikitake Professional Engineer's Office
Tokyo, Japan
kenji.rikitake@acm.org

4th Kiyofumi Tanaka
School of Information Science
Japan Advanced Institute of Science and Technology
Ishikawa, Japan
kiyofumi@jaist.ac.jp

5th Yoichi Shinoda
School of Information Science
Japan Advanced Institute of Science and Technology
Ishikawa, Japan
shinoda@jaist.ac.jp

2023 IEEE 9th World Forum on Internet of Things (WF-IoT) | 979-8-3503-1161-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/WF-IOT58464.2023.10539584

***Abstract*—With the proliferation of Internet of Things (IoT) devices, developers need to respond to diverse and rapidly changing user requirements quickly. Thus, a method for rapidly and frequently updating IoT devices is required. Moreover, owing to the development of cloud/edge computing, an approach to enable the efficient development and maintenance of multi-layer IoT systems is necessary. In this study, we propose a method for building dynamically updatable IoT devices using WebAssembly (Wasm), where the IoT device consists of a combination of the primary programming language implementing the application and the Wasm runtime. Furthermore, we propose isomorphic IoT systems that use the same Wasm binary that is built from a common code base in each layer of the systems. We showed a specific use case of compiling machine learning models for image recognition and classification into Wasm binaries and built an isomorphic architecture where the inference process is executed using the same Wasm binary in each layer of the IoT system. We performed a quantitative evaluation to confirm the effectiveness of the proposed method. Although the proposed method introduces an overhead that is caused by calling Wasm functions from the application, the impact thereof is limited. We also measured the performance of the proposed method using the widely used image recognition and classification models ResNet-50 and MobileNetV2. We confirmed that the proposed method is practical in the current situation and offers promise for the future.**

***Index Terms*—IoT, IoT system, Dynamic IoT application, isomorphic IoT System, WebAssembly, Wasm**

I. Introduction

The Internet of Things (IoT) is transforming the manner in which we live and conduct business owing to the proliferation of smart devices with sensors and actuators, thereby enabling real-time data collection and control. IoT devices are used in various applications, including smart homes, wearable devices, and smart cities. These IoT devices generate considerable amounts of data, the analysis of which has led to solving various problems and creating new business opportunities. Further development is expected with the introduction of 5G networks, which will enable higher data rates and low-latency communication.

Meanwhile, the rapid proliferation of IoT devices has led to development challenges. Developers need to develop and update IoT devices quickly and frequently to meet the diverse and rapidly changing needs of users. The functionality of IoT devices must change dynamically for developers to update IoT devices rapidly. Moreover, with the development of cloud computing and edge computing, IoT applications are built not only as devices but also as layered systems that are composed of different platforms and architectures [1]. Moreover, IoT devices may consist of different hardware. Thus, in the development of such IoT systems, the use of different technologies in layers that are composed of different platforms and architectures leads to efficiency and maintainability problems.

We propose two methods to solve the aforementioned problems using WebAssembly [2] (Wasm). First, we present dynamic IoT applications that allow on-demand partial updating of IoT device behavior without restarting the device using Wasm. In the proposed method, an IoT device consists of primary language and Wasm implementations, and the Wasm implementation part can be dynamically updated to build a dynamic IoT application. This enables developers to respond quickly and frequently to diverse and rapidly changing user requirements. Second, we propose isomorphic IoT systems that use the same Wasm binary from a common code base across IoT system layers, which represents the isomorphism of the running code between the layers. The proposed method can use the same Wasm binary throughout the IoT system, which can be composed of different platforms and architectures, by leveraging the portability of Wasm. The use of a common code base improves the development efficiency and maintainability of IoT systems.

We implement an IoT device using Raspberry Pi 4 [3] as hardware with Nerves [4], which is a platform for developing

IoT devices in the Elixir programming language [5], to build the dynamic IoT application. Moreover, the Wasm runtime will run on Elixir, which is the primary IoT device implementation language. We develop Wasmtube [6] to relay the processing between the Wasm module that runs on the Wasm runtime and Elixir. This enables the Elixir code to call functions in the Wasm module to perform processing and reflect the results in the behavior of the IoT devices. Furthermore, Wasmtube detects the replacement of the Wasm binary and restarts the Wasm runtime based on the new binary, which enables the dynamic updating of IoT devices without restarting the entire device.

We compile machine learning models into a Wasm binary and demonstrate that the identical Wasm binary that is built from a common code base can operate in multiple layers as a use case for the isomorphic IoT systems. First, we use Apache TVM [7] to compile a machine learning model, which is distributed in the ONNX [8] format and can perform image recognition and classification, into a Wasm binary. Subsequently, the Wasm binary is deployed to each application in the multiple layers of the IoT system. Each application is implemented using Wasmtube, which runs the Wasm runtime on Elixir. The machine learning model that is realized by the identical Wasm binary runs on that Wasm runtime. The Wasm module is run from an Elixir application with image content as an argument to perform image recognition and classification processing. This isomorphic architecture enables a common code base across platforms and devices for a practical use case of inferential processing using machine learning models. Moreover, each layer will have a common code base based on Wasm, which can facilitate workload-aware processing offloading [9].

We evaluate the proposed methods from two perspectives to demonstrate their effectiveness. First, we measure the overhead that is introduced by the method when the IoT device is composed of the primary language in Elixir and a Wasm module that runs on top of the Wasm runtime. The experimental results confirm that an overhead of approximately 200 μs is introduced when calling Wasm functions from the application compared to the process being completed within the application. However, this time is short compared to the processing time of the entire IoT device. Furthermore, a longer processing time in the Wasm module itself results in a relatively smaller overhead. The overhead is small enough and is acceptable to show the effectiveness of the proposed method. Second, we confirm the feasibility of using the same Wasm binary across IoT system layers by applying machine learning models that are compiled from ResNet-50 [10] and MobileNetV2 [11] to the Wasm binaries. We performed image recognition and classification on an IoT device, a cloud server, and a local machine. We confirmed that MobileNetV2 running on Wasm runtime runs in approximately 0.6 s, even on the lowest-performing Raspberry Pi 4. The effectiveness of the proposed method will be improved further if hardware acceleration, such as GPUs, can be easily exploited by Wasm in the future.

The main contributions of this paper are as follows:

1) We propose dynamic IoT applications and isomorphic IoT systems using Wasm.
2) We implement the proposed methods based on a practical use case of IoT systems, namely the execution of machine learning models.
3) We quantitatively evaluate the actual implementations to confirm the effectiveness of the proposed methods.

The remainder of this paper is organized as follows: Section II summarizes the background and related studies and contextualizes this study within the flow of previous works. The proposed methods and the implementations are described in Section III. Section IV presents empirical evaluations of the proposed methods and implementations, and Section V concludes the paper and outlines the future directions.

II. Background and Related Works

We contextualize this study within the flow of previous works by summarizing the background in Section II-A and related works in Section II-B.

A. Background

The method of distributing new firmware images to devices over the air has been used extensively for updating IoT devices. However, this method requires the device to be rebooted, which means that completing the update is time-consuming [12]. Developers would not be able to respond quickly and frequently to user requests with this approach. Therefore, studies have been conducted on methods that partially change the code that constitutes an IoT device without requiring rebooting; for example, by using dynamic links to replace code references or using the dynamic nature of scripting to update the running code [13]. Regarding recent developments in cloud/edge computing, we need to look at the IoT system as a multiple-layered system that includes the edge and cloud tiers, rather than just focusing on IoT devices [1]. Thus, it is necessary to establish a dynamic update method for IoT devices that is compatible with a method for the efficient development and maintenance of multi-layer IoT systems.

B. Related Works

Several studies have attempted to solve the above problem with Wasm [2], which is a binary format for describing instructions for stack-based VMs that was originally developed to accelerate processing on a web browser. At present, Wasm is used outside of the web browser owing to its portable and secure specification and execution environment. [14] expressed interest in using Wasm as a means of dynamically updating IoT devices without rebooting being necessary. [15] suggested that the dynamic updating of IoT devices is possible by embedding a Wasm runtime as a lightweight container in an IoT device and running an application on top of the Wasm runtime. [15] demonstrated an implementation that embeds a Wasm runtime into an application that runs on ESP32 [16], a popular microcontroller that is widely used for IoT device implementation, and dynamically rewrites the Wasm binary that runs on top to modify the device behavior. This

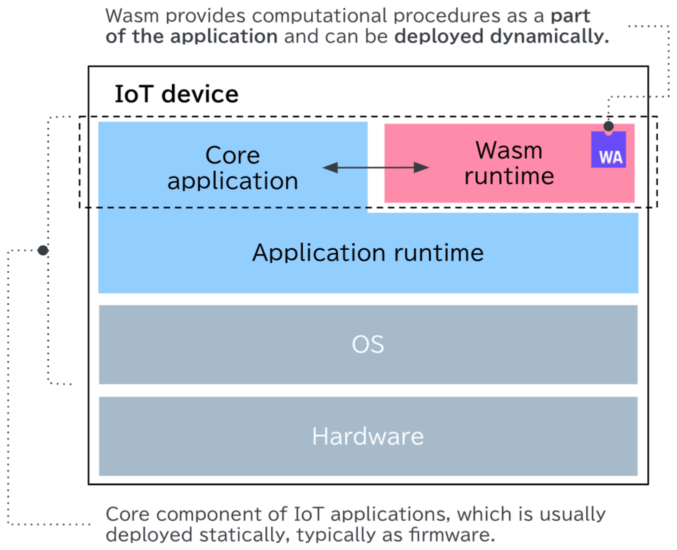

Wasm provides computational procedures as a part of the application and can be deployed dynamically.
IoT device
Core application
Wasm runtime
WA
Application runtime
OS
Hardware
Core component of IoT applications, which is usually deployed statically, typically as firmware.

Fig. 1: Overview of the proposed method for dynamic IoT applications using Wasm.

implementation can only provide interaction with the device via a limited set of primitives, integers, and floats and does not provide a general procedure for binary updates. In the following section, we propose a method for dynamically updating IoT devices using Wasm to address these issues.

The concept of isomorphic IoT systems using a common code base across multiple layers of IoT systems has been proposed [17]. With the advancements in cloud/edge computing, current IoT applications consist of IoT devices as well as multi-layer IoT systems that are composed of different platforms, architectures, and hardware. Thus, IoT systems are complex and developers are required to be proficient in various technologies [18]. The burden on developers can be reduced by using a common code base to implement each layer of IoT systems. Hence, [17], [18] focused on the portability of Wasm and proposed its use to build applications throughout IoT systems. However, the isomorphism that was proposed in these studies is limited as each layer uses the Wasm runtime in the same manner, and it does not extend to building and executing applications with the same Wasm binaries. Moreover, the proposals in these studies only consider the conceptual level and do not provide concrete implementations. In the following section, we present and describe the implementation of a method to build isomorphic IoT systems at the level of executing Wasm binaries based on practical use cases.

III. Proposed Methods and Implementations

We describe the proposed methods in Section III-A and their implementation in Section III-B.

A. Proposed Methods

1) Dynamic IoT Applications Using Wasm: First, we propose a Wasm-based method that enables developers to update IoT devices dynamically. Figure 1 presents an overview of the scheme. The proposed method assumes an IoT device with a common configuration in which the operating system, application runtime, and applications run on the hardware. These are typically built as a single piece of firmware that implements an IoT device. In the proposed method, the application that implements the IoT device is divided into two parts: the core application part and the part that runs on the Wasm runtime. The part that is implemented by Wasm is executed by a dedicated Wasm runtime.

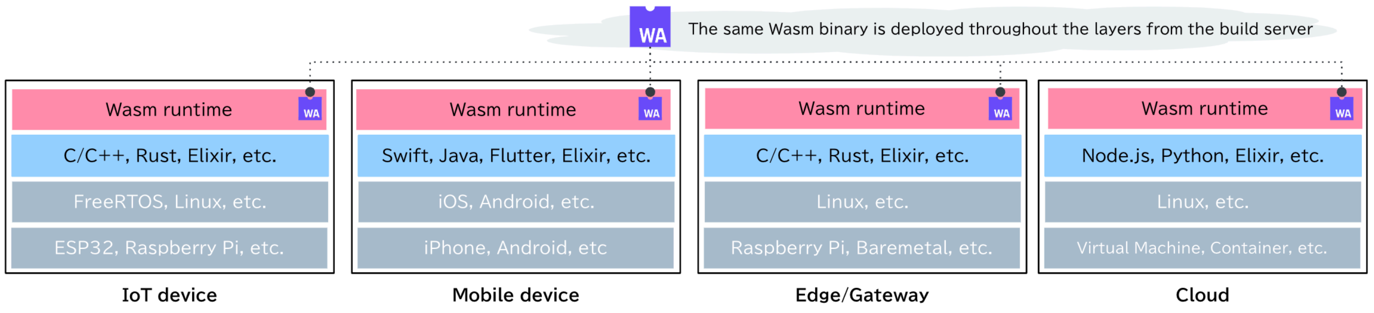
Wasm includes a system call abstraction layer known as WASI [19]. However, it is currently not sufficiently generic to cover various IoT devices [14]. Thus, in the proposed method, the core part of the application is responsible for accessing the lower layers through the system interface, following which the application is integrated with the Wasm runtime. This enables dynamic updating of the application behavior by replacing the Wasm binary without updating and restarting the entire application, while also enabling the implementation of a general-purpose IoT device. Developers can quickly respond to diverse and rapidly changing user requirements using the proposed method.

2) Isomorphic IoT Systems Using Wasm: Furthermore, we propose a Wasm-based method that enables the use of a common code base across multiple layers of IoT systems. Figure 2 presents an overview of the scheme. The proposed method assumes that the applications at each layer of IoT systems include a Wasm runtime, similar to the dynamically updatable IoT device scheme discussed above. As mentioned in the previous section, Wasm is a platform-agnostic execution format. The same Wasm binary can be deployed and executed on any layer of different platforms and devices by leveraging this fact. In the proposed scheme, the same Wasm binary that is built from a common code base is deployed on each layer.

The code isomorphism in the proposed method represents not only the fact that each layer has its own Wasm runtime as part of the application execution environment but also the fact that it deploys and executes the same Wasm binary built from a common code base. This enables a common code base to be used across different platforms and devices, thereby improving the efficiency and maintainability of the IoT system development.

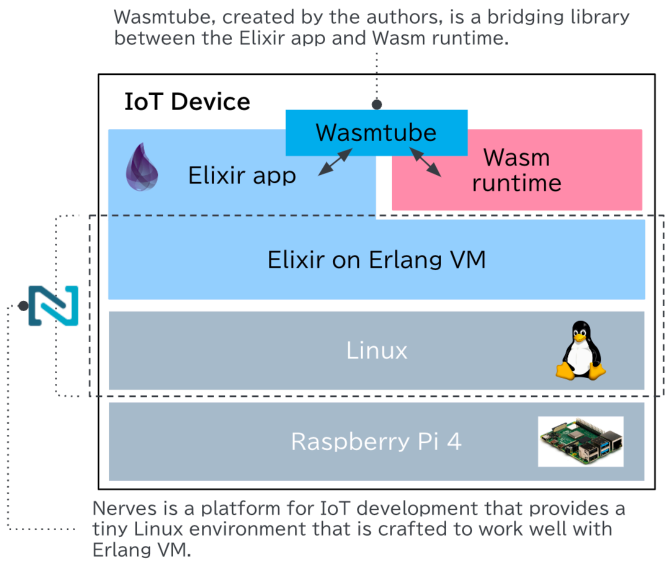
B. Implementations

1) Dynamic IoT Applications Using Wasm: We implemented the proposed method as shown in Figure 3 to build the proposed dynamic IoT applications. We used Nerves [4], which is an IoT development platform based on the Elixir programming language [5], and Raspberry Pi 4 [3] as the hardware for implementing the IoT devices. The Wasm runtime runs on top of Elixir, which is the primary language for implementing IoT devices. It executes the Wasm modules that implement dynamically updatable processes in IoT devices. We developed Wasmtube [6] to relay processing between Wasm modules that run on the Wasm runtime and Elixir, allowing the Elixir code to call functions in the Wasm modules. Subsequently, the IoT device that runs with Elixir can then reflect the results from the Wasm modules. Furthermore, when



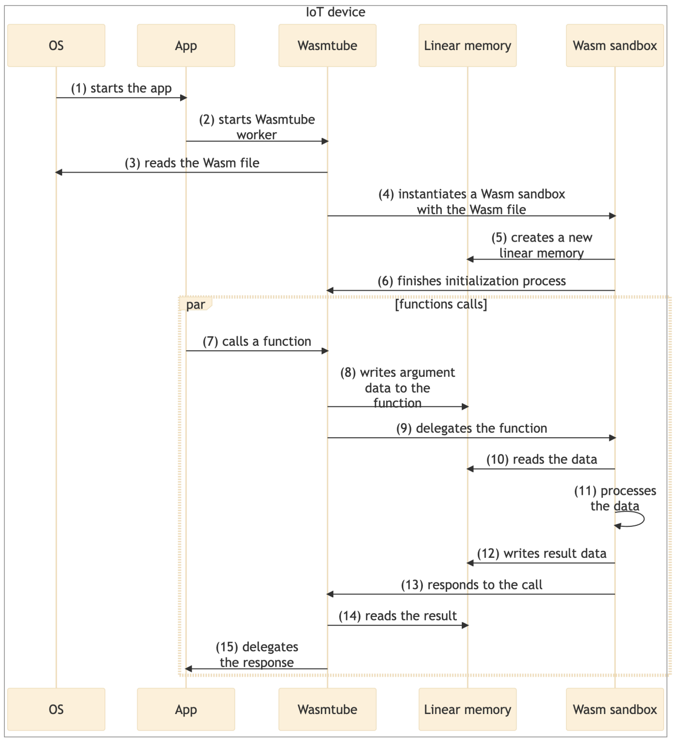
The same Wasm binary is deployed throughout the layers from the build server
WA
Wasm runtime
C/C++, Rust, Elixir, etc.
FreeRTOS, Linux, etc.
ESP32, Raspberry Pi, etc.
IoT device
Wasm runtime
Swift, Java, Flutter, Elixir, etc.
iOS, Android, etc.
iPhone, Android, etc
Mobile device
Wasm runtime
C/C++, Rust, Elixir, etc.
Linux, etc.
Raspberry Pi, Baremetal, etc.
Edge/Gateway
Wasm runtime
Node.js, Python, Elixir, etc.
Linux, etc.
Virtual Machine, Container, etc.
Cloud

Fig. 2: Overview of the proposed method for isomorphic IoT systems using Wasm.



Wasmtube, created by the authors, is a bridging library between the Elixir app and Wasm runtime.
IoT Device
Wasmtube
Elixir app
Wasm runtime
Elixir on Erlang VM
Linux
Raspberry Pi 4
Nerves is a platform for IoT development that provides a tiny Linux environment that is crafted to work well with Erlang VM.

Fig. 3: Implementation of the proposed method for dynamic IoT applications using Wasm.



IoT device
OS
App
Wasmtube
Linear memory
Wasm sandbox
(1) starts the app
(2) starts Wasmtube worker
(3) reads the Wasm file
(4) instantiates a Wasm sandbox with the Wasm file
(5) creates a new linear memory
(6) finishes initialization process
par
[functions calls]
(7) calls a function
(8) writes argument data to the function
(9) delegates the function
(10) reads the data
(11) processes the data
(12) writes result data
(13) responds to the call
(14) reads the result
(15) delegates the response
OS
App
Wasmtube
Linear memory
Wasm sandbox

Fig. 4: Sequence diagram of Wasmtube working with Wasm.

Wasmtube detects that the Wasm binary has been replaced, it can dynamically update the IoT device by restarting the Wasm runtime based on the new binary.

We describe the operation of Wasmtube in implementing the proposed method in two steps. First, Figure 4 illustrates how Wasmtube initializes Wasm and forwards function calls within an Elixir application. The application launches Wasmtube, which reads the specified Wasm binary （Figure 4 (1)–(3)）. The Wasm runtime creates a sandbox as an isolated environment for executing the specified Wasm binary as well as a dedicated linear memory area. Following initialization, an interface is made available to the application to access the Wasm sandbox （Figure 4 (4)–(6)）. The application calls the function that is provided by Wasm through Wasmtube. Wasmtube writes the arguments to the called function as a binary sequence into linear memory in the Wasm sandbox. Wasmtube expects the arguments to be passed as Elixir data types that can be encoded into JSON-encoded strings or as image binaries and their sizes. Thereafter, Wasmtube calls the specified Wasm function of the application （Figure 4 (7)–(9)）. It first reads the argument data from linear memory. Following processing based on this data, the result is written to linear memory. When the function completes its execution, the process returns to Wasmtube, which reads the result of the Wasm processing from linear memory and returns it to the application （Figure 4 (10)-(15)）. The application continues processing based on the result.

Figure 5 depicts the manner in which Wasmtube reinitializes Wasm and updates the behavior of the IoT device based on the newly deployed Wasm binary in an application. Developers deploy a new Wasm binary to the operating system to update the behavior of the IoT device （Figure 5 (1)）. Once the Wasm binary has been updated, the Linux `inotify` API notifies Wasmtube of the update without polling. Wasmtube reloads the updated Wasm binary based on the notification （Figure 5 (2)-(3)）. Wasmtube discards the existing Wasm sandbox and initializes a new sandbox based on the updated Wasm

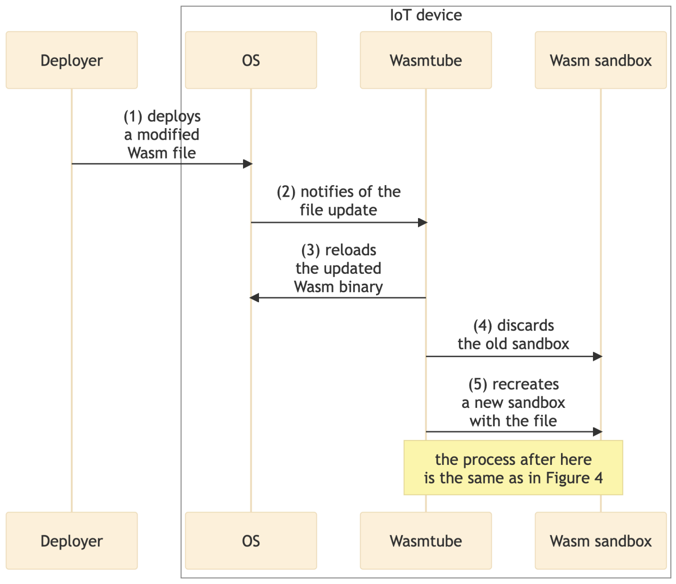

IoT device
Deployer
OS
Wasmtube
Wasm sandbox
(1) deploys a modified Wasm file
(2) notifies of the file update
(3) reloads the updated Wasm binary
(4) discards the old sandbox
(5) recreates a new sandbox with the file
the process after here is the same as in Figure 4
Deployer
OS
Wasmtube
Wasm sandbox

Fig. 5: Sequence diagram of Wasmtube handling file updating.

binary （Figure 5 (4)–(5)）. The initialization process and subsequent function calls are the same as in Figure 4. Although the figure depicts the deployment of the Wasm binary from outside the IoT device, it is also possible for the IoT device to obtain a new Wasm binary over the network. Regardless of how it is accomplished, replacing the Wasm binary allows the IoT device to update its behavior dynamically.

2) Isomorphic IoT Systems Using Wasm: We chose an IoT system that performs image recognition and classification tasks using machine learning models as a real-world example to demonstrate the use case of the isomorphic IoT systems. In this implementation, the image recognition and classification tasks were performed on an application that was implemented using the aforementioned Wasmtube, with the machine learning model compiled into the same Wasm binary. The implementation is illustrated in Figure 6.

We describe the pipeline for compiling machine learning models into a Wasm binary. The image recognition and classification models that are distributed in the ONNX [8] format [1] are compiled into a single Wasm binary using Apache TVM [7] (hereafter referred to as TVM). TVM is a framework for compiling machine learning models into various forms that are suitable for different execution environments. As TVM also supports compilation into Wasm, machine learning models can be compiled into a single Wasm binary. First, we used a script that is provided by TVM to extract the machine learning model and parameters from a file that is written in the ONNX format. Thereafter, we used the Rust programming language to implement a function that reads the models and parameters using the libraries that are provided by TVM, and performs recognition and classification on the images that are passed as arguments. Subsequently, the function that was implemented by Rust was compiled into a Wasm binary. The code that was used to compile the image recognition and classification models into Wasm binaries using the above method is available in a GitHub repository[2].

The generated Wasm binaries are deployed on the operating system of each layer that constitutes the IoT system. We used Livebook [20], which allows the Elixir code to be executed in a browser, as the application that runs Elixir on each layer. In each application, the Wasm runtime runs on top of the Elixir application using Wasmtube, as mentioned previously. The functions that are provided by Wasm are called from the Elixir application via Wasmtube, with the image contents as arguments. The application continues processing based on the results of processing by Wasm. Figure 7 depicts the result of the image recognition and classification task execution by the application that was implemented on the IoT device as described above. In the same manner, the machine learning model that was realized by the same Wasm binary was executed by the application on the other layers. The proposed method makes it possible to implement inferential processing with machine learning models using a common code base across different platforms and devices.

IV. Evaluation

We present the evaluation of the proposed methods: the dynamic IoT applications in Section IV-A and the isomorphic IoT systems in Section IV-B.

A. Dynamic IoT Applications Using Wasm

In the proposed method, the application consists of a combination of the core application and Wasm to enable the dynamic updating of IoT devices. Nerves used in the implementation of the proposed method described in Section III-B1 provides a firmware update method [21]. This method requires a restart of the entire operating system to complete the application update. In the experimental environment shown in Table I, preliminary experiments confirmed that it takes more than 10 seconds for the OS to restart and for application services to become available. In contrast, the proposed method requires only a restart of the Wasm runtime, which takes approximately 1.4 seconds for the rereading of the Wasm binary of the machine learning model used in the following experiment and restarting the runtime. This means that the proposed method of not having to reboot the operating system is much faster. However, it inevitably introduces an overhead of function calls between the application and Wasm. We performed an experiment to determine the amount of overhead that is introduced and examine the effectiveness of the proposed method, and we discuss its general applicability.

1) Experimental Methodology: To measure the overhead that is introduced by the proposed method, we determined the difference between implementing an IoT device when using only the primary language comprising the application

[1]onnx/models: A collection of pre-trained, state-of-the-art models in the ONNX format https://github.com/onnx/models

[2]https://github.com/kentaro/wasm-standalone-builder

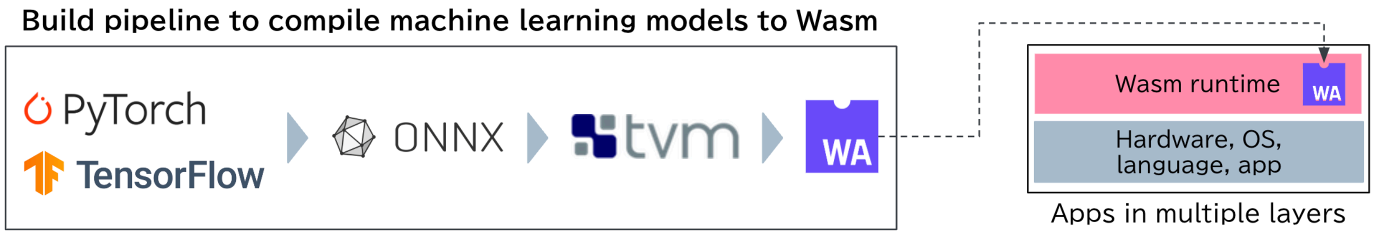
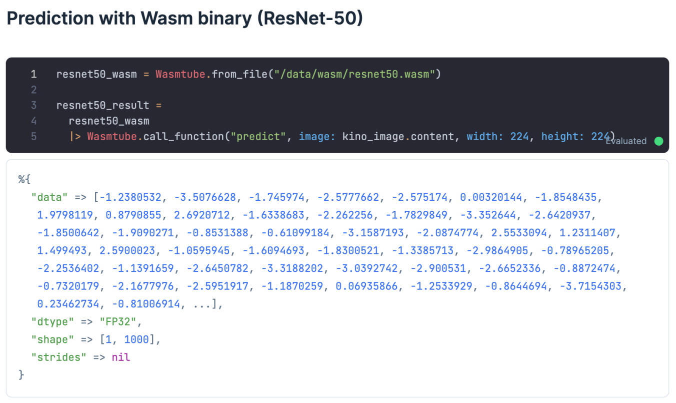

Build pipeline to compile machine learning models to Wasm
PyTorch
TensorFlow
ONNX
tvm
WA
Wasm runtime
WA
Hardware, OS, language, app
Apps in multiple layers

Fig. 6: Use case of the proposed method for isomorphic IoT systems using Wasm.


Prediction with Wasm binary (ResNet-50)

```
resnet50_wasm = Wasmtube.from_file("/data/wasm/resnet50.wasm")

resnet50_result =
  resnet50_wasm
  |> Wasmtube.call_function("predict", image: kino_image.content, width: 224, height: 224)
```

Evaluated

```
%{
  "data" => [-1.2380532, -3.5076628, -1.745974, -2.5777662, -2.575174, 0.00320144, -1.8548435,
   1.9798119, 0.8790855, 2.6920712, -1.6338683, -2.262256, -1.7829849, -3.352644, -2.6420937,
   -1.8500642, -1.9090271, -0.8531388, -0.61099184, -3.1587193, -2.0874774, 2.5533094, 1.2311407,
   1.499493, 2.5900023, -1.0595945, -1.6094693, -1.8300521, -1.3385713, -2.9864905, -0.78965205,
   -2.2536402, -1.1391659, -2.6450782, -3.3188202, -3.0392742, -2.900531, -2.6652336, -0.8872474,
   -0.7320179, -2.1677976, -2.5951917, -1.1870259, 0.06935866, -1.2533929, -0.8644694, -3.7154303,
   0.23462734, -0.81006914, ...],
  "dtype" => "FP32",
  "shape" => [1, 1000],
  "strides" => nil
}
```

Fig. 7: Example of function call from Elixir app to Wasm.

TABLE I: Experimental environment

Machine	CPU	Architecture
Raspberry Pi 4 Model B	Cortex-A72 1.5 GHz	ARMv8-A
Memory	OS	
4GB	Linux (Nerves)	

and when using the Wasm runtime. Table I presents the experimental environment. The experimental setup assumed an IoT device that collects sensor data once per second, statistically processes the data once per minute, and sends the data over the network to the upper layers. We used a function that calculates the mean and median using an array of 60 integer-type numbers as arguments as the workload for measuring the overhead. Subsequently, we compared the case in which the above calculation is performed only by an application that is implemented in Elixir and that in which the calculation is performed by a function call that is implemented in Wasm. Moreover, we measured a Wasm function that only returns a fixed value without performing the aforementioned computation to confirm the overhead of the Wasm function call itself. The overhead introduced by the proposed method could be confirmed through this experiment.

2) Experiment: Table II shows the results of the experiments that we performed using the above method[3]. When the above computations were performed using only applications implemented in Elixir (the row in the table with the name

[3]Further details regarding the experiment are available at https://github.com/kentaro/wfiot2023

TABLE II: Results of performance comparison experiment

Name	IPS*	Average	Deviation	Median	99th%
Elixir	32.24K	31.01μs	$\pm$ 65.48%	31.17μs	43.26μs
Wasm	3.96K	252.48μs	$\pm$ 22.50%	240.26μs	399.59μs
Wasm (Noop)	4.05K	246.67μs	$\pm$ 15.16%	233.78μs	392.25μs

*Iterations per second.

Elixir), the average time was 31.01 μs and the median was 31.17 μs. When the computations that were implemented in Wasm were performed using function calls (the row in the table with the name Wasm), the average time was 252.48 μs and the median was 240.26 μs. When calling the Wasm function that simply returns a fixed value without any computation (the row in the table with the name Wasm (Noop)), the mean was 246.67 μs and the median was 233.78 μs. This demonstrates that the calculation process in the primary language that constitutes the application by itself is faster than the calculation process that is implemented in Wasm with function calls. A comparison of the two results for Wasm function calls reveals that the difference was small. Thus, the overhead was not due to the Elixir–Wasm speed difference, but due to the overhead of the Elixir–Wasm function calls.

3) Discussion: In the previous experiment, we confirmed the overhead that is introduced by the proposed method. This overhead is dependent on the process of calling Wasm functions from Elixir, and not on the performance difference between Elixir, which is the primary language that implements the application, and the Wasm runtime. Thus, we can assume that the overhead is an almost constant value regardless of the task that is performed. That is, as a percentage of the total processing time, the overhead decreases as the task that is processed in Wasm becomes more complex and time-consuming. Therefore, the proposed method of dynamically updating IoT devices using Wasm is sufficiently effective if the performance requirements of the IoT devices can tolerate an overhead of approximately 200 μs.

A study that evaluated the performance of Wasm inside a browser reported a performance degradation of approximately 1.5 times when executing code on the Wasm runtime compared to native code [22]. Future reports are anticipated on the performance comparisons when Wasm is executed on a standalone runtime, as in this study [14]. Wasmtube developed by the authors uses Wasmtime [23], which is a well-known Wasm runtime, via an Elixir extension library known as Wasmex [24]. However, many Wasm runtimes are currently available, as outlined in [25], and more performance-enhancing Wasm runtimes that are specialized for IoT devices will become available in the future.

We discuss the general applicability of the proposed method. The implementation of the proposed method uses Raspberry Pi 4, which is classified as a high-end family of hardware for IoT devices [26]. [15] integrated code running on the ESP32, which is classified as mid-range by [26], with Wasm3 [27] to build a dynamically updatable IoT device. However, the implementation that was presented in the study only supports integers and floats in the process of Wasm function calls; thus, it is unlikely to be applicable for performing inference processing with machine learning models as in our study, which demonstrates the potential for broader applications of this work.

B. Isomorphic IoT Systems Using Wasm

We implemented the proposed method based on the practical use case of performing inference processing with machine learning models on an IoT system that consists of multiple layers of different platforms and devices while using a common code base. In the implemented applications, we measured the execution speed of the inference processes using these machine learning models. In this section, we discuss the current feasibility of the proposed method and its future prospects.

1) Experimental Methodology: We measured the performance of the inference process using the machine learning models that were compiled in the Wasm binaries, as outlined in Section III-B2, on several environments that are typically used in multi-layer IoT systems. Table III presents our experimental environment. We used an e2-medium instance provided by the Google Cloud Compute Engine as a cloud layer. This widely used instance type claims to offer "day-to-day computing at a lower cost."[4] Experiments were also performed on a local iMac for comparison. We used ResNet-50 [10] and MobileNetV2 [11] as machine learning models for image recognition and classification. ResNet-50 is a widely used model for image recognition and classification. However, it is a large model with approximately 25 million parameters and is usually run on a GPU. Therefore, in the experiment, we also used MobileNetV2, which achieves high performance even on relatively low-power devices such as mobile phones. In the experiment, we compiled these machine learning models into Wasm and ran them on the Wasm runtime, whereby images were passed from the Elixir application to Wasm via Wasmtube for inference processing and performance measurements.

2) Experiment: We conducted the experiment using the method described above. We first confirmed that both ResNet-50 and MobileNetV2 could perform the inference task correctly at all levels. Specifically, we uploaded images and performed inference on the images using the models that were compiled into Wasm to confirm that the correct results were obtained. Table IV presents the results of the performance that were obtained by invoking the Wasm functions from the application[5]. MobileNetV2 outperformed ResNet-50 at all layers, with the average execution time of ResNet-50 exceeding 1 s at the device and cloud layers, whereas the average execution time of MobileNetV2 was 588.69 ms, even at the most inefficient device layer.

3) Discussion: In the previous experiment, we demonstrated that inference processing can indeed be performed using machine learning models that are compiled into Wasm, which runs image recognition and classification models in each layer of an IoT system. The results confirmed that the inference process at the device layer required more than 3 s for ResNet-50, whereas only approximately 0.6 s was required for MobileNetV2. Thus, if MobileNetV2 is adopted as the machine learning model, the proposed method is still effective for building isomorphic IoT systems using the same Wasm binary that is built from a common code base, provided that a processing speed of approximately 1 FPS is acceptable.

As using hardware acceleration such as GPUs with Wasm, which is oriented toward portable interfaces, is difficult, inference processing was performed by the CPU in this experimental environment. Therefore, as we confirmed previously, it is difficult to perform inference processing using relatively large models such as ResNet-50 at the device layer with practical performance. A specification known as wasi-nn [28] is currently being developed to define a common interface from Wasm to hardware such as GPUs that can accelerate the execution of machine learning models. If such a specification becomes available for the hardware that is used in IoT devices, it is expected that the effectiveness of the proposed method will increase in the future.

Furthermore, a more thorough implementation of isomorphism can be considered. In the implementation of the isomorphic IoT systems of the proposed method, we used Elixir to implement each layer of the application. However, as explained in Section III-B2, we used Rust for compiling the machine learning models into Wasm binaries. If machine learning models can be implemented as Wasm binaries using Elixir, isomorphism can be realized at the level of Wasm as well as its implementation language. Firefly [29] is being developed as a compiler to compile applications that are written in Elixir to Wasm. In the future, this compiler could be used to compile machine learning models that are distributed in the ONNX format into Wasm binaries. If this can be achieved, it is expected that the degree of isomorphism in the implementation of the proposed method can be increased, at the level of Wasm binaries as well as its implementation language.

V. Conclusions and Further Research

We proposed the use of Wasm to solve problems that arise in the development of IoT devices with the proliferation of such devices and IoT systems owing to the evolution of

[4]https://cloud.google.com/compute/docs/machine-resource

[5]Further details regarding the experiment are available at https://github.com/kentaro/wfiot2023

TABLE III: Experimental environment

Layer	Machine	CPU	Architecture	Memory	OS
Device	Raspberry Pi 4 Model B	Cortex-A72 1.5 GHz	ARMv8-A	4GB	Linux (Nerves)
Cloud	Google Cloud Compute Engine e2-medium	2 vCPU (Intel Broadwell)	x86/64	4GB	Debian GNU/Linux 11
Local	iMac	Apple M1	arm64	16GB	macOS 13.2

TABLE IV: Results of performance comparison experiment with machine learning models

Layer	Model	IPS*	Average	Deviation	Median	99th%
Device	ResNet-50	0.30	3304.31 ms	± 0.14%	3303.16 ms	3319.68 ms
	MobileNetV2	1.70	588.69 ms	± 0.22%	588.92 ms	590.93 ms
Cloud	ResNet-50	0.89	1120.75 ms	± 20.77%	1065.05 ms	2224.26 ms
	MobileNetV2	5.93	168.55 ms	± 1.31%	168.19 ms	178.07 ms
Local	ResNet-50	3.39	295.32 ms	± 9.49%	286.29 ms	472.07 ms
	MobileNetV2	17.12	58.41 ms	± 18.37%	56.75 ms	100.44 ms

*Iterations per second.

cloud/edge computing. First, we proposed a method to realize dynamically updatable IoT devices using Wasm. Second, we proposed isomorphic IoT systems that use the same Wasm binary that is built from a common code base in different layers of the IoT system. We implemented these proposals and demonstrated their application in practical use cases. We performed quantitative evaluations of the proposed methods to confirm their current effectiveness and to discuss future prospects.

There are three possible directions for the development of this research. First, an MLOps [30] flow can be constructed using the proposed method. We believe that the construction of IoT systems using machine learning techniques as isomorphic IoT systems, which operate on a common code base that is compiled in Wasm, will be highly effective in efficiently realizing MLOps. Second, processing can be offloaded across layers in IoT systems using the proposed method [9]. As each layer has a common code base using Wasm, it is expected to be easier to delegate processing according to the workload. Third, the proposed method can be applied to federated learning [31]. The use of a machine learning model based on Wasm, which can be commonly executed in different local environments, is expected to be effective as a basis for improving the efficiency of federated learning.

References

[1] S. Bansal and D. Kumar, "Iot ecosystem: A survey on devices, gateways, operating systems, middleware and communication," *Int. J. Wireless Inf. Networks*, vol. 27, no. 3, pp. 340–364, Sep. 2020.

[2] WebAssembly. https://webassembly.org/. Accessed: 2023-6-14.

[3] The Raspberry Pi Foundation. Teach, Learn, and Make with Raspberry Pi. https://www.raspberrypi.org/. Accessed: 2023-6-14.

[4] Nerves Project. https://nerves-project.org/. Accessed: 2023-6-14.

[5] The Elixir programming language. https://elixir-lang.org/. Accessed: 2023-6-14.

[6] K. Kuribayashi. Wasmtube: A bridging library which allows you to communicate between Elixir and Wasm. https://github.com/kentaro/wasmtube. Accessed: 2023-6-14.

[7] Apache TVM. https://tvm.apache.org/. Accessed: 2023-6-14.

[8] ONNX. https://onnx.ai/. Accessed: 2023-6-14.

[9] B. Wang, C. Wang, W. Huang, Y. Song, and X. Qin, "A Survey and Taxonomy on Task Offloading for Edge-Cloud Computing," *IEEE Access*, vol. 8, pp. 186 080–186 101, 2020.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778.

[11] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun. 2018, pp. 4510–4520.

[12] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. D. Poorter, "Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles," *IEEE Commun. Mag.*, vol. 58, no. 2, pp. 35–41, Feb. 2020.

[13] P. Ruckebusch, S. Giannoulis, I. Moerman, J. Hoebeke, and E. De Poorter, "Modelling the energy consumption for over-the-air software updates in LPWAN networks: SigFox, LoRa and IEEE 802.15.4g," *Internet of Things*, vol. 3-4, pp. 104–119.

[14] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "WebAssembly Modules as Lightweight Containers for Liquid IoT Applications," in *Web Engineering*. Springer International Publishing, 2021, pp. 328–336.

[15] I. Koren, "A Standalone WebAssembly Development Environment for the Internet of Things," in *Web Engineering*. Springer International Publishing, 2021, pp. 353–360.

[16] ESP32 Wi-Fi & Bluetooth MCU | Espressif Systems. https://www.espressif.com/en/products/socs/esp32. Accessed: 2023-6-14.

[17] T. Mikkonen, C. Pautasso, and A. Taivalsaari, "Isomorphic Internet of Things Architectures With Web Technologies," *Computer*, vol. 54, no. 7, pp. 69–78, Jul. 2021.

[18] A. Taivalsaari, T. Mikkonen, and C. Pautasso, "Towards Seamless IoT Device-Edge-Cloud Continuum:," in *ICWE 2021 Workshops*, M. Bakaev, I.-Y. Ko, M. Mrissa, C. Pautasso, and A. Srivastava, Eds. Cham: Springer International Publishing, 2022, pp. 82–98.

[19] WASI — The WebAssembly System Interface. https://wasi.dev/. Accessed: 2023-6-14.

[20] "Home - livebook.Dev," https://livebook.dev/, accessed: 2023-6-14.

[21] The Nerves Project Authors, "ssh_subsystem_fwup: Erlang SSH Subsystem for Nerves firmware updates," https://github.com/nerves-project/ssh_subsystem_fwup, accessed: 2023-6-14.

[22] 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*, 2019, pp. 107–120.

[23] Wasmtime. https://wasmtime.dev/. Accessed: 2023-6-14.

[24] P. Tessenow, "Wasmex: Execute WebAssembly / WASM from Elixir," https://github.com/tessi/wasmex, accessed: 2023-6-14.

[25] S. Akinyemi. Awesome WebAssembly Runtimes: A list of webassemby runtimes. https://github.com/appcypher/awesome-wasm-runtimes. Accessed: 2023-6-14.

[26] M. O. Ojo, S. Giordano, G. Procissi, and I. N. Seitanidis, "A Review of Low-End, Middle-End, and High-End Iot Devices," *IEEE Access*, vol. 6, pp. 70 528–70 554, 2018.

[27] Wasm3: A fast WebAssembly interpreter, and the most universal WASM runtime. https://github.com/wasm3/wasm3. Accessed: 2023-6-14.

[28] "wasi-nn: Neural Network proposal for WASI," https://github.com/WebAssembly/wasi-nn, accessed: 2023-6-14.

[29] Firefly: An alternative BEAM implementation, designed for WebAssembly. https://github.com/GetFirefly/firefly. Accessed: 2023-6-14.

[30] G. Symeonidis, E. Nerantzis, A. Kazakis, and G. A. Papakostas, "MLOps - Definitions, Tools and Challenges," in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2022, pp. 0453–0460.

[31] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated Learning: Challenges, Methods, and Future Directions," *IEEE Signal Process. Mag.*, vol. 37, no. 3, pp. 50–60, May 2020.