

WAWoT: Towards Flexible and Efficient Web of Things Services via WebAssembly on Resource-Constrained IoT Devices

Borui Li , Member, IEEE, Hongchang Fan , Yi Gao , Member, IEEE, and Wei Dong 

Abstract—Web of Things (WoT) is an emerging concept to connect IoT devices to the web using standard interfaces. This provides interoperability between different IoT platforms and enables seamless integration with web and cloud services. However, running sophisticated web services directly on resource-constrained IoT devices is challenging due to limitations in memory, computation, and energy. This paper proposes WAWoT, a Wasm-based framework for flexible and efficient Web of Things services. WAWoT allows flexible WoT service development using annotations and automatic partitioning. It also enables dynamic service migration using WebAssembly modules to adapt placement between IoT devices and web clients. We also introduce an ahead-of-time compiler optimized for low memory usage through techniques like streamed compilation and trimming. For energy efficiency, we use optimizations like bulk instruction writing and direct I/O accessing. Safety is ensured through compile-time and run-time analyses to guarantee sandboxed execution. Evaluations demonstrate WAWoT exhibits better flexibility than existing WoT development approaches. Furthermore, WAWoT can also reduce RAM usage by 84.9x and energy consumption by 1.9x-4.9x over existing WebAssembly runtimes. Overall, it enables efficient, safe, and flexible WoT services on constrained IoT devices.

Index Terms—WebAssembly, Internet of Things, web of things, ahead of time compilation.

I. INTRODUCTION

THE Internet of Things (IoT) suffers from an interoperability problem [1]. According to [2], there exists over 800 ontologies defined by various IoT standards and projects.

Received 17 February 2024; revised 2 September 2024; accepted 8 November 2024. Date of publication 18 November 2024; date of current version 12 February 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62302096, Grant 62272407, and Grant 62072396, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20230813, in part by the Zhishan Young Scholar Program of Southeast University Grant 3209002402A2, in part by the “Pioneer” and “Leading Goose” R&D Program of Zhejiang under Grant 2023C01033, in part by the National Youth Talent Support Program, and in part by the Information Technology Center and State Key Lab of CAD&CG, Zhejiang University. Recommended for acceptance by J. Palsberg. (Corresponding authors: Borui Li; Wei Dong.)

Borui Li is with the School of Computer Science and Engineering, Southeast University, Nanjing 210000, China (e-mail: libr@seu.edu.cn).

Hongchang Fan, Yi Gao, and Wei Dong are with the College of Computer Science, Zhejiang University, Hangzhou 310000, China (e-mail: dongw@zju.edu.cn).

Digital Object Identifier 10.1109/TC.2024.3500385

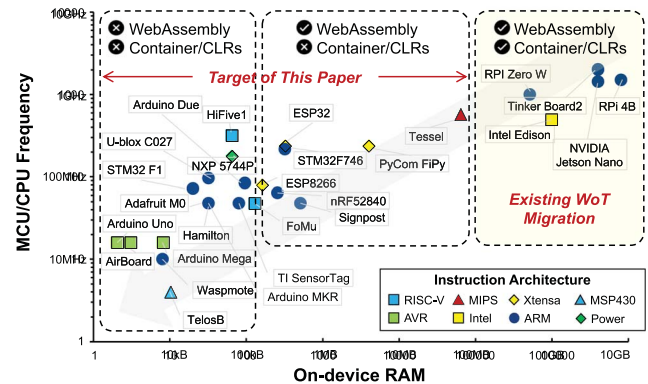


Fig. 1. Hardware performance of existing IoT devices and their migration applicability of Web of Things.

This lack of universal data representation schema impedes the human-to-machine (H2M) and machine-to-machine (M2M) interaction, further creating data and service silos within different IoT ecosystems. To counter this fragmentation problem, the World Wide Web Consortium (W3C) proposes Web of Things (WoT) architecture [3] that uses and extends standardized web technologies to achieve interoperability in the IoT domain. WoT paradigm is widely adopted in smart city [4] and Industry 4.0 [5], etc.

A common approach to developing a WoT service is to clearly partition the functions run on IoT devices and web clients (e.g., browsers) and separately develop the codebases for both sides. However, this process raises a significant *flexibility* issue. The rigid pre-definition of service interfaces and their fixed placement require additional efforts from developers and cannot adapt to diverse working condition changes such as network fluctuation and device disconnection. This inflexibility upraises development difficulty and also leads to suboptimal service performance.

Existing works shed light on solving the flexibility problem of WoT via service migration [6], [7], [8], [9]. To smoothen the migration, a consistent execution environment is necessary. In cloud computing, the service migration is done with virtual machines (VMs) [10]. While in the realm of WoT, existing works leverage containers or common language runtimes (CLRs) such as ThingsJS [8] and NodeWoT [9] for flexible migration. Fig. 1 shows our investigation of the applicability of existing WoT

migration techniques on commercial-off-the-shelf IoT devices. Existing works mainly target migrating WoT services for high-end IoT devices because both CLRs or containers require massive computing resources and incur much run-time overhead. We also observed a large gap between the low-end devices and flexible WoT services.

WebAssembly (Wasm) [11] offers a potential solution. It is an emerging lightweight and portable bytecode format with near-native efficiency. As shown in Fig. 1, WebAssembly extends the support to a subset of resource-constrained devices. Moreover, WebAssembly is recognized as the fourth standard language for the web after HTML, CSS, and JavaScript, which means it is supported by most mainstream browsers. These advantages of WebAssembly make it well-suited for the WoT ecosystem, despite not being its original design focus.

Therefore, this paper aims to investigate constructing a WoT framework with WebAssembly that facilitates enhanced interoperability between both high-end and low-end IoT devices and web platforms. However, existing works [12], [13], [14] do not fully explore the potential of WebAssembly within the context of resource-constrained WoT, leaving the following several key questions unsolved.

How can we enable flexible WoT services with Web Assembly? The meaning of flexibility is two-fold. First, developers can create a WoT service flexibly without paying attention to how to decouple the service into different devices. Second, we should facilitate seamless migration with WebAssembly to adapt to the dynamic environment.

How can we execute WoT services on resource-constrained IoT devices efficiently? We can observe in Fig. 1 that WebAssembly is not yet capable of supporting all kinds of resource-constrained IoT devices. Executing WebAssembly needs a dedicated runtime to translate bytecode into native instruction. Hence, it is necessary to design a WebAssembly runtime with high resource and energy efficiency to extend the WoT interoperability to a wider array of devices.

How can we guarantee the secure execution of WoT services on the device? The WoT architecture makes IoT devices more vulnerable to network-based attacks than standalone applications. Furthermore, low-end IoT devices generally do not have memory management units or CPU privilege levels, which makes secure execution harder.

To tackle all the challenges above, we present **WAWoT**, a *Wasm*-based *Web of Things* system for resource-constrained IoT devices. With WAWoT, users could create WoT services in various high-level languages and compile them into WebAssembly modules. Then the application logic to be executed on the IoT device is disseminated via wired or wireless. Upon receiving the module, WAWoT performs on-device Ahead-of-Time (AOT) compilation, checks the sandbox guarantees and executes the module. To be more specific, WAWoT proposes the corresponding solutions to the three challenges.

Solution 1: Different from traditional separated development, WAWoT presents a complete development workflow with *code annotation* for developers to create WoT services

in an integrated manner. Developers can focus on the overall application logic without struggling with detailed service fragmentation. Furthermore, based on the `import` function mechanism, WAWoT facilitates *WebAssembly-based lightweight run-time WoT service migration* to make full use of the web resources.

Solution 2: To cope with the limited resources, WAWoT reduces the compile-time memory footprint by *streamed look-back compilation*, and optimizes the run-time memory layout by *post-compile memory trimming* and *constants remapping* to support complex applications. To reduce energy consumption, WAWoT adopts the *bulk instruction writing* and the *I/O direct accessing* approach to reduce the energy consumption of both compile- and run-time.

Solution 3: To provide a safe and deterministic integrated execution environment on the resource-constrained devices, WAWoT introduces memory safety and control-flow integrity checks. To minimize the overhead during execution, WAWoT moves most of the checks *to the AOT compilation stage* and carefully *selects the instructions being checked*.

We implement WAWoT and evaluate its performance extensively. Results show that: (1) WAWoT achieves programming and execution flexibility with real-world WoT applications. (2) Compared with the state-of-the-art WebAssembly interpreter [15], [16] and AOT runtime [17], WAWoT reduces $13.6\times$, $31.9\times$, and $84.8\times$ less RAM, respectively. (3) The energy optimization approaches of WAWoT achieve $1.2\times \sim 4.9\times$ power reduction. (4) WAWoT only incurs 19.1% average run-time overhead for ensuring the sandboxed execution of WebAssembly.

Contributions of this paper are summarized as follows:

- We propose WAWoT, a holistic WoT framework based on WebAssembly, which can reduce the development and deployment overhead of WoT services.
- We introduce an efficient AOT runtime of WebAssembly designed for resource-constrained IoT devices, which leverages stream look-back compilation to reduce the memory footprint, several optimizations for energy optimization, and dynamic instruction checking for minimized security overhead.
- Experimental results show that WAWoT achieves flexible WoT programming and execution while reducing the memory footprint by up to $84.8\times$.

II. BACKGROUND

In this section, we briefly introduce several technical backgrounds of WAWoT.

Web of Things and WebAssembly. The Web of Things (WoT) aims to overcome the IoT interoperability problem by extending web technologies to physical devices, bringing uniform interfaces and standards. WoT abstracts real-world entities into Web Things (WT). Each WT exposes capabilities through a Thing Description (TD) which conforms to JSON-LD format. Specifically, a TD specifies: (1) properties as readable/writable state variables, (2) actions to invoke functions on the WT, and (3) events that trigger effects under defined conditions. Overall,

WoT provides a standardized way to represent IoT devices as Web Things accessible over common web protocols using machine-readable Thing Descriptions. This enables seamless integration of heterogeneous physical objects into web-based applications.

WebAssembly is the fourth web standard language, which naturally fits the WoT scenario. Using WebAssembly in WoT also has the following advantages: (1) It is more efficient than JavaScript, the de facto programming language for WoT. (2) The language portability (e.g., C/C++, Rust, Go) of WebAssembly facilitates a better use of existing code bases. (3) The platform portability of WebAssembly enables cross-architecture migration, which can further enhance the flexibility of WoT.

Preliminaries of WebAssembly. WebAssembly owns a series of properties that contribute to efficiency and safety. Among these properties, import mechanism, linear memory, and control instructions play essential roles in our design.

Import mechanism. WebAssembly's import mechanism enables host environments to provide objects like memory, tables, and functions to a WebAssembly module. This allows efficient data exchange and function sharing between the host and WebAssembly. Importing Memory facilitates access to mutable storage inside WebAssembly. Importing functions extends WebAssembly capabilities by calling external code.

Linear memory. To achieve the isolated execution of a WebAssembly module by design, WebAssembly organizes its main storage in a linear memory region. Linear memory is a mutable continuous array of bytes containing global/local variables and constants. At runtime, a WebAssembly module can only access its linear memory. The linear memory is separated from the code space and the stack which prevents various dangerous behavior such as arbitrary jumps.

Control instructions. The control-flow instructions of WebAssembly are classified by the function-type (i.e., FUNC, CALL, RETURN and CALL_INDIRECT) and the block-type (i.e., BLOCK, LOOP, IF, ELSE, END, BR, BR_IF and BR_TABLE). Note that WebAssembly discards the direct jump instruction with structured control-flow instructions to lower the compilation and validation overhead [11]. Both types can consume input and produce output, and the signature of each function/block is explicitly recorded in the WebAssembly module. The block-type instructions can be nested, but the function-type instructions can not. Another distinct design of the control instruction of WebAssembly is its BR and BR_IF operation. These instructions denote jumping outwards to the upper level of the nested block and their operand means the depth to jump outward.

Interpreted, JIT, and AOT bytecode execution. There are mainly three approaches to running the cross-platform bytecode (such as WebAssembly) on a certain platform: interpreter, Just-in-Time (JIT), and Ahead-of-Time (AOT) compilation-based execution. The interpreter repeatedly reads one instruction of the bytecode and translates it to machine code which is easy to implement and requires few resources. JIT compilation means to compile the code while running, and AOT means to compile the code before running it. Interpreted execution is generally

TABLE I
COMPARISON OF THE WEBASSEMBLY RUNTIME

Runtime	Stars Jul. 2024	Linux-compatible devices			IoT devices	
		Interp.	JIT	AOT	Interp.	AOT
Wasmer	18.3k	✗	✓	✓	✗	✗
Wasmtime	14.9k	✗	✓	✗	✗	✗
Lucet	4.1k	✗	✗	✓	✗	✗
Wasm3	7.1k	✓	✗	✗	✓	✗
WAMR	4.1k	✓	✓	✓	✓	✗
WAC	465	✓	✗	✗	✓	✗
Wizard	351	✓	✓	✗	✗	✗
WARDuino	75	✗	✗	✗	✓	✗

slower than JIT or AOT-based execution because the latter can apply various runtime optimizations to the binary code, which interpreters typically cannot perform.

We survey the existing WebAssembly runtimes that are publicly available and their underlying translation technologies in Table I. Note that JIT compilation is impractical on IoT devices because they can not execute code from RAM. We can see from the table that none of the existing runtimes support AOT compilation on constrained devices. The AOT compilation-based runtimes on Linux-compatible devices can scarcely be ported to IoT devices due to the usage of a heavy-weight code generation framework (e.g., LLVM). WAWoT bridges the gap between efficient AOT compilation of WebAssembly and IoT devices.

III. SYSTEM OVERVIEW OF WAWoT

In this section, we present the design considerations and introduce the building blocks of WAWoT.

An overview of the WAWoT workflow is shown in Fig. 2. As an example to demonstrate the usage of WAWoT, we implement a face recognition WoT service. This service, depicted in Fig. 3, captures an image from a camera, transforms it to grayscale, and recognizes any faces in the image.

Development phase. Our central principle is to make developing Web of Things services as if the services completely run on the web client. Developers do not need to pay much attention to the interface fragmentation. Hence, WAWoT presents APIs to write codes concerning IoT peripherals. Moreover, WAWoT also provides a straightforward development process for WoT services.

Based on our investigation of applications from popular IoT forums, we found that the peripherals generally communicate with the MCU using the following protocols: Analog, Digital, UART, I2C, PWM, and SPI. Hence, WAWoT focuses on providing APIs for the above protocols, which will further enable all kinds of sensor manipulation. Moreover, to facilitate the duty-cycling on IoT devices, WAWoT also provides `WAWoT_sleep()` API to set the device to low-power mode. On the other hand, benefiting from the multi-language support of WebAssembly, the WoT services could be compiled from several high-level languages such as C, Rust, and Go. Hence, the IoT-related APIs that WAWoT provides also support various languages. For example, to use our C API to read data from a

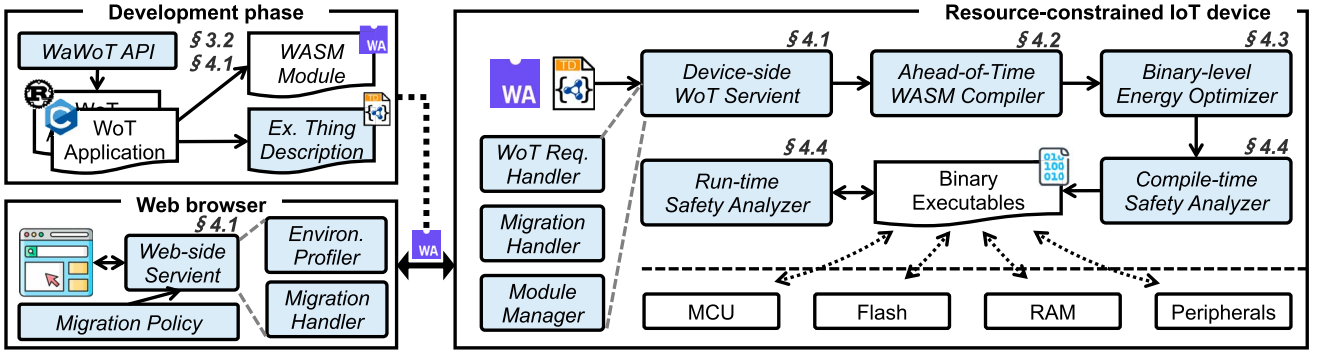


Fig. 2. Overview of WAWoT system.

```

1 #include "face_recognition.h"
2 #include "wawot.h"
3 using namespace std;
4
5 vector<uint8_t> Cam(int w, int d)#[Action,IoT]{/*Impl.*/}
6 vector<uint8_t> grayScale(vector<uint8_t> raw){/*Impl.*/}
7 string eigenFace(vector<uint8_t> img){/*Impl.*/}
8 string faceRecog()#[Action]{
9     vector<uint8_t> raw_data = Cam(640, 480);
10    vector<uint8_t> img_data = grayScale(raw_data);
11    return eigenFace(img_data);
12 }

```

Fig. 3. The annotated code of face recognition WoT service.

digital sensor, developers should first declare WAWoT's API as an external function as follows:

```

1 extern int WaWoT_readDigital(int pinNum);

```

Based on the APIs, developers could write the entire service in one piece of code.

The code is then compiled into a WebAssembly module that can then be imported into a web page. Furthermore, an extended Thing Description (exTD) file which is used for WoT service discovery is also automatically generated, as illustrated in Fig. 4. To achieve this, WAWoT provides several additional code annotations: `#[Property]`, `#[Action]`, `#[Event]`, and `#[IoT]`. The first three label the exposed property, action, and event of the service, following the Thing Description model of the WoT standard [3]. For instance, line 8 in Fig. 3 exposes a `faceRecog` WoT action, allowing any user of the service to invoke the entire recognition process. Moreover, the `#[IoT]` tags functions that must run on the IoT device itself, like those interacting with peripherals (e.g., line 5 in Fig. 3). WAWoT further extends the TD standard with more fields that are necessary to make offloading decisions, such as the on-device execution time and transmission size.

Execution phase. Flexibility is one of the design goals of WAWoT. As a standard Web language and a platform-agnostic binary format, WebAssembly is a perfect bridge for Web of Things and flexibility. Hence, between IoT devices and web browsers, WAWoT achieves flexible computation offloading mechanism with the help of the *device-side* and the *web-side* WoT servient (§IV-A). A servient in WoT refers to a software entity that provides or consumes WoT services. Once a web

```

1 {
2   "@context": "https://www.w3.org/2022/wot/td/v1.1",
3   "title": "WaWoT-FaceRecog",
4   "actions": {
5     "faceRecog": {
6       "type": "string",
7       "migratable": true, "localExec": 589.3,
8       "transSize": 921600, "outputSize": 20,
9       "forms": [{ "href": "http://<IoT_ip>/faceRecog" }]
10    },
11    "Cam": {
12      "type": "string",
13      "migratable": false, "localExec": 30,
14      "transSize": 0, "outputSize": 921600,
15      "forms": [{ "href": "http://<IoT_ip>/Cam" }]
16    },
17  },
18 }

```

Fig. 4. The exTD file of face recognition WoT service.

client invokes a WoT service, these device-side and web-side servients will cooperatively decide where to place the computation that exhibits better performance.

WAWoT also values the efficiency and safety of WoT services, especially on resource-constrained IoT devices. Therefore, for computation efficiency, WAWoT proposes an *Ahead-of-Time compiler* (§IV-B) which transforms the WebAssembly bytecode into native assembly in the AOT manner to maximize the execution speed while minimizing memory requirement. Also, the WebAssembly module is optimized for energy efficiency by a *binary-level energy optimizer* (§IV-C) on the IoT device. For security, WAWoT leverages a *run-time analyzer* (§IV-D) to ensure the WebAssembly module is executed properly. Moreover, to further optimize the run-time efficiency, WAWoT moves a large fraction of the checks to the *compile-time safety analyzer* (§IV-D).

IV. WAWoT DESIGN

In this section, we first present a flexible programming and deployment model for WoT services, elaborate on the lightweight methods we employ in our AOT compiler to reduce the memory usage, and the energy optimizations on the IoT device. Then we describe the safety checks of WAWoT under WoT scenarios.

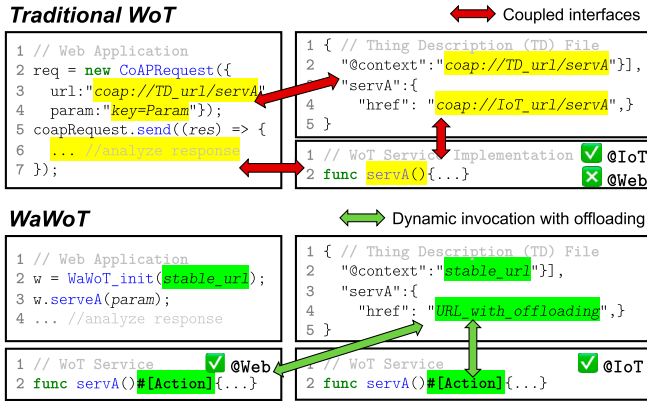


Fig. 5. Coupled interface design of traditional WoT development and WAWoT's flexible service model.

A. Flexible WoT Service Model

Before we dive into WAWoT's service model, we outline the limitations of traditional WoT frameworks [3], [9] with an example. As shown in Fig. 5, these frameworks necessitate developers to pre-define all interfaces in device-side code and Thing Description. Developers should explicitly invoke the interfaces on the web side with pre-defined parameters. This tight coupling of interfaces leads to a need for re-development across all three sides if any part is modified. To enable flexible development of WoT services, WAWoT offers developers a set of placement-agnostic annotations to describe crucial service aspects. During run-time, WAWoT utilizes dynamic WoT service offloading to adjust service placement for optimized performance.

Annotation-based WoT description. WAWoT empowers developers to holistically create a WoT application that is distributed across the IoT device, Thing Description, and the web. Designing such an annotation system necessitates gathering adequate information to (1) determine the application logic placement, and (2) generate the thing description (TD) file. Hence, our annotation system consists of two categories: location-aware and TD-aware annotations.

Within location-aware annotations, functions are inherently considered offloadable without specific labels. Additionally, we offer annotations `# [IoT]` and `# [Web]` to designate functions for execution on the IoT device or the web. For example, functions interacting with sensors or peripherals should be labeled `# [IoT]`.

Concerning TD-aware annotations, the WoT specification defines three interface types: property, action, and event. Hence, WAWoT provides corresponding annotations: `# [Property]` for exposing properties of the WoT service that clients can monitor or control (e.g., declaring a sensor reading value as a property), `# [Action]` to declare actions clients can invoke on the service (e.g., an action to initialize or calibrate a sensor), and `# [Event]` for annotating asynchronously emitted events (e.g., declaring a motion detection event).

These annotations are used to generate an *extended Thing Description* (exTD) file automatically. The exTD file contains metadata like the service title and exposed URIs for the declared properties, actions, and events conforming to the WoT

standard [3]. In addition, the exTD file also contains the information that is needed for service offloading decisions. The *migratable* field denotes if it could be migrated, which can be directly extracted by the annotation. The *localExec* field denotes execution time on the IoT devices (in ms), which is estimated during the AOT compilation (§IV-B). The *trans-Size* stands for the additional data transmission size if this function is offloaded to the web client, and the *outputSize* denotes the output data size. These two data sizes are estimated via a taint analyzer built upon Valgrind. These numeric data are necessary to make offloading decisions, which is fundamentally the tradeoff between networking overhead and migration gain.

Moreover, with these annotations, developers can focus on expressing the overall application logic in only one piece of code rather than having to separate IoT devices and web clients rigidly. The actual computation and data placement will be decided during execution with the help of our WebAssembly-based WoT service migration technique.

WebAssembly-based WoT service migration. With the help of exTD files, WAWoT can determine the best endpoint for execution based on current conditions. Once decide to offload functions to the web client, our web-side and device-side WoT servient will handle the migration automatically.

To be more specific, WoT services reside on IoT devices as WebAssembly modules. The modules contain portable WebAssembly bytecode that can execute across platforms. Web of Things requests are generally initiated by the web client. With WAWoT, when the web page requests the WoT service, it first discovers the exTD file which contains the provided services and migration information. An environment profiler in the web-side servient leverages network probes to detect the current network condition and decides if offloading to the web exhibits better performance with the information in exTD files.

Once decided to offload, an offloading handler on the web client fetches the WebAssembly module and starts local execution. The data of each module is also migrated with the WebAssembly module [14]. There emerges a question: how to handle the "IoT-only" functions that are called during the sandboxed WebAssembly execution on the client? These functions need to be executed on the IoT device itself. The import mechanism of WebAssembly provides a lightweight approach to intercept the execution flow in the sandbox and invoke the remote functions. WAWoT imports the implementation of invoking the I/O interfaces and data retrieving via the network for each IoT-related API we presented in §III.

Customizable WoT service migration policy. By default, WAWoT employs a simple yet effective latency-optimized migration policy. An identical WoT service starts from the raw data collection on the IoT device and ends by demonstrating the data on the web. Hence, WAWoT's default offloading policy separates the WoT service into two parts that run on IoT and web sequentially. With the help of the exTD file, the migration policy could be formulated as:

$$\arg \min_{c \in [0, C]} \left(\sum_{f=0}^c t_f^{IoT} + \frac{S_c}{N} + \sum_{f=c+1}^C t_f^{Web} \right), \quad (1)$$

where c is the cutting point and C is the number of all functions. t_f^{IoT} and t_f^{Web} are the execution time of function f on the IoT device or the web. S_c is the output size of cutting point c (with protocol overhead considered) and N denotes the measured bandwidth of current network environment.

Furthermore, WAWoT also supports customization of the migration policy. Calculating when and what to migrate is generally entangled with other building blocks in existing migration systems. Towards this, WAWoT adopts a decoupled design of migration policy and the web-side WoT servient. We also provide a code template with well-defined inputs and outputs for users to easily customize it.

B. Lightweight WebAssembly Runtime

Reducing the memory footprint of WAWoT as much as possible is the key to realizing efficient WebAssembly execution on resource-constrained devices. Therefore, we present our approaches to making WAWoT lightweight below. We propose the streamed look-back compilation, post-compile memory trimming, and constant values remapping to reduce the RAM usage *during* the AOT compilation, *after* the compilation, and *during* the run-time.

Streamed look-back compilation. Existing AOT runtimes of WebAssembly [17] exploit complicated code generation frameworks such as LLVM or Cranelift to translate the bytecode to native instructions. These frameworks lead to an unacceptable RAM footprint and can hardly port to restricted devices. Hence, we build our own AOT compiler by replacing each bytecode with a set of native instructions.

Unfortunately, directly loading the whole WebAssembly module into RAM to perform AOT compilation fails sometimes because loading the module occupies most of the available RAM. Inspired by stream processing approaches [18], WAWoT adopts a streamed compilation to read and translate the bytecode instruction-by-instruction.

However, this streamed compilation of WebAssembly also faces non-trivial problems. As mentioned in the background, WebAssembly is designed as a stack machine. It abandons simple jump instructions, and instead provides completely structured control-flow instructions (e.g., BLOCK and LOOP). The implications of these WebAssembly characteristics on WAWoT are two-fold.

First, WAWoT needs to convert structured control-flow instructions to jump-based ones because resource-constrained devices commonly only support direct jump. Nevertheless, the jump instructions require an exact address to redirect to, whereas the address may not be available under the streamed compilation approach. The reason is that the jump destination may not yet be compiled when the streamed compilation translates a control-flow instruction. Towards this problem, WAWoT upgrades the streamed compilation with a look-back. The whole process is described in Alg. 1. The look-back means WAWoT labels the undetermined branch targets with a pseudo-address during the first-pass streamed compilation (line #3–#9) and substitutes them when the compilation of all the bytecode is finished (line #10–#13).

Algorithm 1: Streamed look-back compilation

Input: Set of WebAssembly instructions $\mathcal{L}_w$

Output: Set of native instructions $\mathcal{L}_n$

```

1 EMIT( $l_w$ ) emits the native code set of  $l_w \in \mathcal{L}_w$ ;
2 Stack depth of  $l_w$ :  $d_{lw}$ , Look-back code buffer:  $\mathcal{L}_{lb}$ ;
3 foreach  $l_w \in \mathcal{L}_w$  do
4   if  $l_w$  is a control-flow instruction then
5      $\mathcal{L}_{lb} \leftarrow \mathcal{L}_{lb} \cup \{< l_w, d_{lw} >\}$ ;
6   else
7      $\mathcal{L}_n \leftarrow \mathcal{L}_n \cup \text{EMIT}(l_w)$ ;
8   end
9 end
10 foreach  $l_w \in \mathcal{L}_{lb}$  do
11   Get jump destination  $dest$  and stack depth  $d_{dest}$ ;
12    $\mathcal{L}_n \leftarrow \mathcal{L}_n \cup \text{EMIT}(l_w, dest, d_{lw}, d_{dest})$ ;
13 end

```

Second, the stack state is critical to the correctness of WebAssembly. When jumping to a decided destination, the stack should be also restored to the entrance state of the destination. However, the native jump instructions do not monitor or manage the stack. Hence, WAWoT records the stack depth of each BLOCK and LOOP entrances during the first pass of the streamed compilation (line #5 in Alg. 1). During the look-back stage, WAWoT restores the stack to the proper depth, i.e., the entrance depth plus the return size of the target block/loop (line #11–#12). We minimize the stack restoration costs when discarding multiple stack values by directly modifying SP register (5 cycles on AVR ISA) instead of emitting multiple pops (4 cycles).

Post-compile memory trimming. WebAssembly uses linear memory to store the run-time data, as we described in the background. Intuitively, we could allocate a buffer in the heap as the linear memory of WebAssembly. However, this naive approach suffers from frequent linear memory shortage because the heap only takes up a small portion of the RAM. Fortunately, we find that the majority of data in the .data and .bss sections are no longer needed after AOT compilation, e.g., the data structure recording the block depth. We can trim off the unnecessary data before we execute the WebAssembly module to reserve more space for the linear memory—but how? There is still some essential data that is intertwined with these useless data.

To address this challenge, WAWoT first differentiates between useful and useless data via a new section named .wa as shown in Fig. 6. Data structures that are helpful after AOT compilation are assigned to the .wa section using attribute specifiers. To modify the memory layout at run-time, we found that the configuration of the heap (e.g., start address and size) is stored in several global variables of malloc(). Hence, we instrument the malloc() for those variables and assign them to the .wa section, and assign the new configurations of the heap after the .data and .bss are trimmed off.

Constants remapping. Constant data is commonly used in IoT applications to save fixed calculating parameters (e.g., for FFT and sinewave) [19], [20]. Most of these data never change

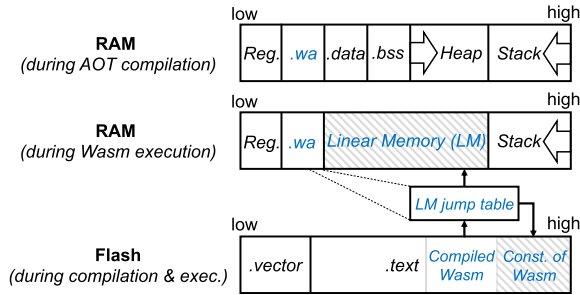


Fig. 6. The memory layout of WAWoT during compilation and execution.

throughout the execution of the application but occupy a large amount of precious RAM. Hence, our basic idea is to reduce the RAM usage by moving constants, which were originally located in the RAM, to the relatively abundant Flash space. This design leads to the question that the constants are logically located in the linear memory but physically distributed in RAM and Flash. Therefore, WAWoT leverages a linear memory (LM) jump table, which is located in the `.wa` section, that maps the logical and physical address, as shown in the Flash illustration of Fig. 6. Each time the WebAssembly module intends to access linear memory, WAWoT takes over the accessing and refers to the LM jump table for the exact address. Undoubtedly, this approach will introduce run-time overhead because accessing the Flash is slower than directly loading data from RAM. We will evaluate it in §VI-C.

C. Energy Optimization on IoT Devices

In this subsection, we push a step further towards optimizing the energy consumption of both the compile-time and execution-time.

Bulk instruction writing at compile-time. During AOT compilation, WAWoT reads the bytecode, performs translation, and writes the generated code back to the Flash for execution. During our preliminary implementation, it turned out that writing each translated instruction instantly to the Flash introduces a lot of overhead.

Under our further investigation, we found that the Flash offers random-access read operations but does not offer the random-access rewrite due to its electrical characteristics [21]. In most writing situations, the whole block needs to be erased first and then written with new data, even if there is only one byte to write. Furthermore, the Flash will not store information reliably if it has been erased too many times. This writing characteristic of Flash requires us to reconsider how to save the translated instructions.

To address this problem, WAWoT adopts bulk instruction writing for translated assembly. We retain a buffer in RAM to temporarily save the compiled native instructions. Once the buffer is full, we perform a bulk write of the buffer data to Flash. The buffer size is set to the same as the block size to minimize overhead brought by the bulk write. We also apply this approach to the writing of constant data to Flash. Note that during execution time, we do not employ bulk write because

postponing some writes will lead to stale data read by the application.

I/O direct accessing at run-time. There are two key distinctions between IoT applications and those that operate on PCs or servers: peripheral accessing and duty-cycling. WebAssembly does not support these two operations at the binary level. To facilitate applications with these features, WAWoT provides programming APIs for developers which we described in §III.

The only challenge that remains now is how WAWoT supports and optimizes the energy consumption of the APIs at run-time. The simple answer to this question is that we implement each API in our runtime and import these implementations to the application via the `IMPORT` mechanism of WebAssembly. However, this approach is only suitable for the web but resource-wasting for the AOTed module. This is because breaking out from sandboxed execution needs to go through several safety checks and call the imported functions which are implemented in another binary. Our system has to switch to the new binary and invoke it, which is inefficient.

By taking advantage of our AOT compilation, WAWoT has the opportunity to modify the translated assembly. Hence, during AOT compilation, WAWoT replaces the instructions that call the imported function with the assemblies that directly access the peripheral (or fall into sleep mode) via a binary rewriting technique. This direct access is facilitated by the I/O direct accessing mechanism, which uses the registers to interact with the peripherals and could further reduce the accessing energy consumption.

D. Two-Phase Sandbox Checks

WAWoT performs sandbox checks at both the compile phase and execution phase. During the first pass of the streamed look-back compilation, WAWoT validates the received WebAssembly module to guarantee it is well-formed. However, this validation can only guarantee the static safety properties, and WAWoT also applies several checks during execution. Specifically, WAWoT protects the control-flow integrity (CFI) by restricting execution to either the compiled assembly of the bytecode following the behavior defined by the WebAssembly or the code offered by WAWoT, and ensures memory safety by prohibiting illegal read and write for both the linear memory and the stack.

Compile-time verification. Validating a WebAssembly module consists of three targets [22]: the overall module environment, the control flow, and the variable instructions.

First, WAWoT checks the sanity of the module in terms of instruction format, entry points, function types, indirect function calls, and linear memories (CV-O1~O5) to ensure it conforms to the WebAssembly convention.

Second, WAWoT guarantees the control-flow validity and checks if the functions or blocks are unclosed (CV-C1), ensures the callee function (or inner block) could not affect the stack of caller function (or outer block) (CV-C2), and protects the correctness (CV-C3) and type-safety (CV-C4) of the stack state before and after executing a function/block. Different from

TABLE II
SANDBOX CHECKS FOR COMPILE-TIME VERIFICATION AND EXECUTION-TIME MONITORING IN WAWoT

Number	Description
Compile-time verification. (CV-O for overall verifications, CV-C for control-flow verifications. CV-V for variable verifications.)	
CV-O1	A correct number of operands and types should be applied for each instruction.
CV-O2	The function index of the start component $\leq$ maximum function index.
CV-O3	For each function, its type index $\leq$ maximum number of signatures listed in the type section.
CV-O4	A linear memory region is compulsory for a valid WebAssembly module.
CV-O5	The size of the data segment $<$ minimum declared length of the linear memory.
CV-C1	Each FUNC must be terminated with RETURN; LOOP/BLOCK must be terminated with END; IF must be terminated with END or ELSE.
CV-C2	Individually examine the stack for each function/loop to prevent the outer stack from being modified by the instruction of an inner layer.
CV-C3	The stack depth of the entrance of a function/block + the size of the return value = the stack depth of the exit of a function/block.
CV-C4	The return type of BLOCK, IF, and LOOP is the same as the declaration.
CV-C5	The branch index of a BR/BR_IF $\leq$ the block depth of current instruction.
CV-V1	The access index of variable instructions $< =$ the total number of the local/global variables.
CV-V2	The SET_LOCAL and SET_GLOBAL instructions could not manipulate the immutable variables.
CV-V3	The imported variable could not be modified by SET_LOCAL and SET_GLOBAL.
Execution-time monitoring. (EM-M for memory safety checks, EM-C for control-flow integrity checks.)	
EM-M1	Current stack depth + the number of additional stacks used by next instruction $\leq$ maximum stack depth.
EM-M2	The memory addressing index $\leq$ size of existing linear memory.
EM-M3	The imported neural network graph must pass the format checks (e.g., using the magic number).
EM-C1	The signature of the callee function must be identical to the signature specified in the CALL_INDIRECT.
EM-C2	The index of the callee function $\leq$ the length of the TABLE in the CALL_INDIRECT.

functions, blocks in WebAssembly could be nested with each other and jumped out via BR. Thus, WAWoT checks the validity of each branch instruction like BR and BR_IF (CV-C5). During this verification, WAWoT also obtains the necessary information of the exact jump destination of the block-type instructions.

Finally, for variable instructions, WAWoT only verifies the variable instructions (e.g., GET_LOCAL and SET_GLOBAL) that manipulate the global and local variables that reside in the stack rather than linear memory. Because different from the memory instructions, the index that variable instructions access is explicitly expressed by the operand, which can be verified statically. Hence, WAWoT statically checks if the index is illegal (CV-V1), and whether the instructions are modifying variables that are set to immutable (CV-V2/V3). These variable instructions checks also contribute to preventing third-party programs from modifying the generated executable code. If a malicious program could access outside of its linear memory, it may further modify the generated executable code, take full control of the IoT device, and lead to unpredictable results.

Execution-time monitoring. During execution time, WAWoT monitors two key aspects: memory safety and control-flow integrity.

WebAssembly operates as a stack-based virtual machine, where the majority of instructions work directly with the stack. Therefore, maintaining stack integrity is crucial for ensuring memory safety. Therefore, we check the quantity and type of the operands that reside on the top of the stack to guarantee the valid execution of each simple instruction. Moreover, WAWoT also checks for potential stack overflow by evaluating whether the top of the stack will exceed the maximum stack depth after executing the next instruction (EM-M1). Prior lightweight compilation approaches of WebAssembly [23] insert a guard page at the end of the stack and take advantage of the OS-level memory protection for lightweight overflow check. However, this approach is only suitable for high-end devices that facilitate

memory management units and have sufficient OS support for memory protection. Hence, on resource-constrained devices, WAWoT adopts the code instrumentation approach and emits additional checking instructions during compilation. Moreover, to reduce the run-time overhead, we optimize EM-M1 by only checking the instructions that will grow the stack, rather than instrumenting every instruction. We will discuss the performance of this optimization in §VI-F.

The memory instructions such as the I32.LOAD and F64.STORE directly manipulate the linear memory, which is the key protected area of WAWoT. The exact address to be accessed can not be obtained at the AOT compilation phase even using symbolic execution approaches, which are hardly implementable on IoT devices. Therefore, WAWoT uses a run-time memory access checker and a linear memory jump table to ensure the memory is legally accessed (EM-M2).

As the integration of AI and IoT tightens [24], a new proposal of WebAssembly named wasi-nn is widely adopted to extend the support of neural network (NN) operations. For NN model inference, WebAssembly modules take the saved model from widely-used frameworks such as ONNX or PyTorch and compute with it. The input model may contain executable scripts such as TorchScript. Hence, checking whether the input model has a legal format is vital against code injection threats (EM-M3).

For control-flow integrity, the CALL_INDIRECT of WebAssembly is widely used in WAWoT to invoke the function provided by our runtime such as the sensor manipulation APIs. Hence, we check whether the index (EM-C1) and the signature (EM-C2) of the indirect call are valid.

V. IMPLEMENTATION

In this section, we introduce the implementation details of WAWoT and its compatibility with official WoT specifications.

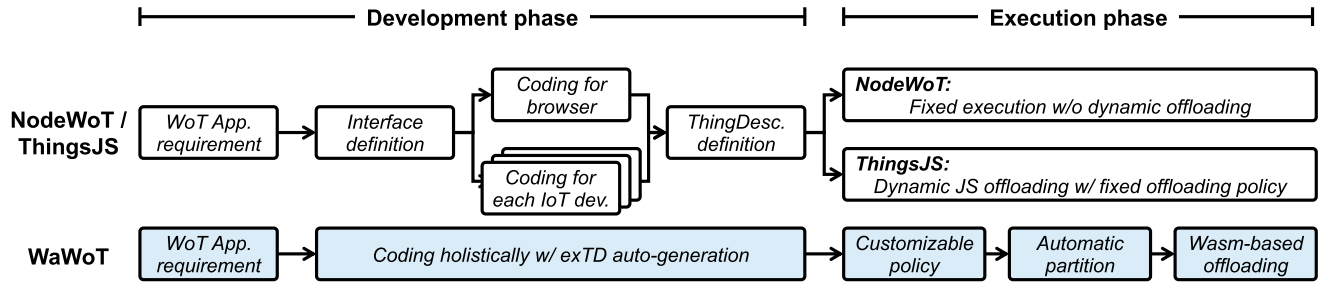


Fig. 7. Development and execution process comparison between WAWoT and existing approaches.

We have implemented the building blocks of WAWoT illustrated in Fig. 2.

Specifically, we implement the client-side components on the Atmel ATmega128 micro-controller with 4KB RAM and 128KB Flash, which is widely used in sensing platforms [19], [25]. The implementation of WAWoT includes over 5000 lines of C code. The core code occupies 27KB of the 128KB program memory. In the device-side WoT servient, there are three sub-modules: WoT request handler, migration handler, and module manager. WoT request handler contains a lightweight network stack to parse the RESTful requests of WoT and invokes the corresponding WebAssembly module. The migration handler interacts with the web-side counterpart to perform the WebAssembly migration. The module manager communicates with the cloud server periodically to check if there is a WebAssembly module to load. The communication period is set to five minutes empirically considering the tradeoff between reprogramming timeliness and battery lifetime of the device because we notice the software update occurs only once per day on average in an actively developed in-the-wild deployment of IoT devices [26], [27].

The web-side servient is implemented using JavaScript with standard HTML-WebAssembly APIs. It has two sub-modules: environment profiler and migration handler. The profiler profiles the network connection between web clients and IoT devices using the piggybacking technique to reduce the overhead. With the information provided in the exTD file and environment profiler, the migration handler makes the decision and conducts migration with the IoT-side handler. Similar to [14], WAWoT also allows users to customize the offloading policy, for which we reserve the interface and calculation. We also implement the WAWoT APIs for the developers using C/C++ and Rust languages.

VI. EVALUATION

In this section, we evaluate WAWoT to answer the following questions: (1) How does WAWoT supports flexible WoT application development? (2) How does the lightweight WebAssembly runtime perform on resource-constrained IoT devices? (3) How does the execution efficiency compare with existing WoT solutions? (4) What is the performance of WAWoT's energy optimizations? (5) What is the result after we apply the overhead mitigation approaches that WAWoT designs for the sandbox checks?

A. Flexibility of WAWoT

We measure the flexibility of WAWoT in two dimensions: programming and execution. Programming flexibility denotes the adaptability of the developed WoT application to environmental changes without requiring extensive redevelopment efforts. Execution flexibility refers to the ability of the WoT application to automatically handle environmental changes for improved performance during execution.

Methodology. We employ a dual-method approach, incorporating both qualitative and quantitative methods to assess the flexibility of WAWoT. We first qualitatively illustrate the development process of WAWoT and compare it with baselines to highlight its superiority. For quantitative evaluation, we implement two representative WoT applications using all the approaches: (1) *Seizure Onset Detection*. Utilizing polyphase wavelet decomposition on EEG signals to detect seizure onset [28], sourced from Wishbone [29]. (2) *Face Recognition*. Implementing a face recognition service with SeetaFaceEngine, as exemplified in §III. Additionally, we gauge WAWoT's performance under diverse network conditions by employing 802.15.4-based transmission [30] for seizure onset detection and BLE for face recognition.

We use NodeWoT [9] and ThingsJS [8] as baselines. NodeWoT serves as the official reference WoT implementation in the W3C specification [3], while ThingsJS is a state-of-the-art self-adaptable middleware for WoT applications.

Qualitative results. Fig. 7 illustrates the comparison of the overall development process between WAWoT and baselines. In traditional methods, developers using baselines must initially define WoT interfaces and implement them for each device during the development phase. WAWoT enhances flexibility by allowing developers to declare interaction interfaces easily using flexible annotation, leaving detailed implementation to the programming framework to handle. In the execution phase, NodeWoT lacks support for dynamic offloading, and ThingsJS only accommodates a fixed offloading policy. In contrast, WAWoT automatically partitions and offloads the Wasm module between the Web and IoT devices, offering customizable offloading policy support. Unlike the separated development and less-flexible execution of NodeWoT and ThingsJS, WAWoT prioritizes flexibility by designing a holistic development mechanism with annotations and a dynamic deployment process with Wasm.

TABLE III
LINES OF CODE DIFFERENCE ($\Delta\text{LoC}\%$) BETWEEN THE BASIC VERSION AND THE OFFLOADED VERSION OF TWO REAL-WORLD APPLICATIONS

$\Delta\text{LoC}\%$	Seizure onset detection			Face recognition		
	Web	TD	Thing	Web	TD	Thing
NodeWoT	26.1+ $\diamond$	32.2	34.1	33.3+ $\diamond$	40.0	26.8
ThingsJS+ $\clubsuit$	0+ $\clubsuit$	12.9	0	0+ $\clubsuit$	25.0	0
WAWoT	None			None		

- $\diamond$ Device-side code needs to be re-implemented on the web client. We only count ΔLoC of the web page and use + to denote the possible re-implementation for a fair comparison.
- $\clubsuit$ ThingsJS does not include a web-side implementation for using offloaded logic on the web. Hence, we use WAWoT's web-side adaptation APIs in the implementation to be more conservative and fair on the ΔLoC . Without this enhancement, ThingsJS falls back to the same performance with NodeWoT for TD modification.

Quantitative results. We now use the performance of seizure onset detection and face recognition application to illustrate the flexibility of WAWoT.

Programming flexibility. In assessing programming flexibility, we simulate developers' re-developing efforts under the following circumstances: they adjust the placement of each computation part of a WoT application when the environment changes. We use the lines of code difference percentage ($\Delta\text{LoC}\%$) between two code versions as the primary metric. Initial implementation involves basic versions of two applications with all data processing logic placed on IoT devices. Subsequently, modifications are made to offload part of the logic to the web. $\Delta\text{LoC}\%$ is recorded for the web page, Thing Description (TD), and IoT devices, with results presented in Table III.

Results indicate that NodeWoT requires developers to modify 26.1%~40.0% of code on all three devices to adapt to a new environment. With ThingsJS, IoT device code can remain unmodified due to its offloading mechanism, but the TD file requires updates for new WoT service URLs. With WAWoT's web-side library and automatic extended TD file generation (§IV-A), the application can adapt effortlessly to various situations *without any* re-development efforts.

Execution flexibility. We then evaluate the runtime efficiency with and without WAWoT by comparing it against NodeWoT, ThingsJS, and based Wasm-based approach (i.e., WAWoT without dynamic offloading).

The seizure onset detection involves seven levels of consecutive wavelet decomposition operations on EEG signals, each doubling the data size. We denote various scenarios with "cutting points," such as #2 indicating the first 2 wavelet decomposition operations assigned to IoT devices. Results are presented in Fig. 8, where star icons signify the cutting points selected by WAWoT and ThingsJS. Both WAWoT and ThingsJS identify optimized cutting points, but WAWoT outperforms ThingsJS by 60.3% due to WAWoT's faster execution than JS on IoT devices.

In face recognition, involving three stages (capturing a picture via camera, RGB-to-grayscale conversion, and face recognition), we evaluate the WoT service performance using

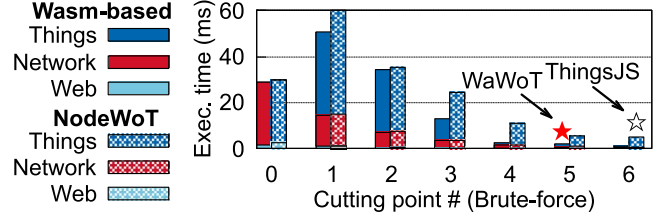


Fig. 8. Performance of different cutting points on seizure onset detection. WAWoT achieves 60.3% acceleration compared with SOTA [8].

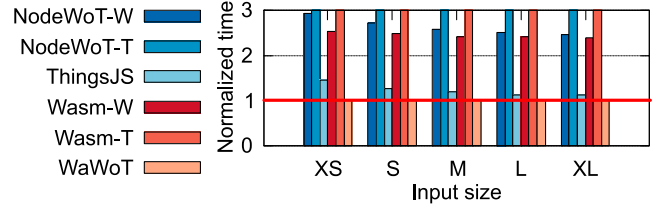


Fig. 9. Normalized performance of different inputs on face recognition. Because NodeWoT does not migrate WoT services automatically, we use -W and -T to denote manually placing all the computation on the web or the IoT device. Similarly, we use Wasm-W/-T to denote implementing WAWoT without service migration.

pictures of different sizes. Fig. 9 shows the results, demonstrating that migrating all computations to the web or keeping them entirely on the IoT device is suboptimal. WAWoT strategically places the RGB-to-grayscale stage on the IoT device, transmitting the grayscaled vectors to the web client for final recognition. This decision accelerates the execution by 18.1% on average for saving network transmission time, showcasing WAWoT's ability to adapt flexibly and optimize WoT service placement.

B. Runtime Memory Footprint

Micro-benchmarks. To make our experiment closer to reality, we use six comprehensive benchmarks from real-world IoT applications to evaluate the performance of WAWoT. The benchmarks are: (1) *BSrch*. It uses binary search to obtain certain sensing data. (2) *FFT*. It performs Fast Fourier Transform on input samples, which is a CPU-intensive application. (3) *LEC*. It performs lossless data compression on sensor data [31]. (4) *Outlier*. It obtains a set of sensor readings, saves them in the buffer, and finds the potential outlier, which is an I/O-intensive application. (5) *HeatC*. It uses an 8x8 heat sensor to track an object. It is divided into a calibration and a detection stage [19]. This benchmark performs the calibration stage. (6) *HeatD*. This benchmark is the detection stage of the object-tracking application using a heat sensor. All of the benchmarks are compiled using `wasi-sdk-12.0` at optimization level `-Os`.

Baselines. We use the following existing works to illustrate how WAWoT achieves lightweight, efficient, and power-saving execution of WebAssembly: (1) *WAMR* [17]: the state-of-the-art WebAssembly AOT runtime on Linux-compatible devices with minimum memory footprint among those listed in Table I.

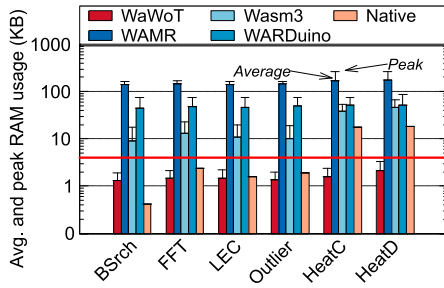


Fig. 10. Memory footprint comparison. The red line indicates the RAM upper bound.

(2) *Wasm3* [15]: the most commonly used WebAssembly interpreter on resource-constrained IoT devices. (3) *WARDuino* [16]: a WebAssembly interpreter for prevalent IoT devices such as ESP32 with the support for IoT hardware interactions. (4) *Native*: to directly execute the application compiled from C language.

To evaluate the performance of our lightweight methods described in §IV-B, we measure the RAM usage of the benchmarks with different WebAssembly runtimes. Because WAWoT is the only WebAssembly AOT runtime that could execute on resource-constrained IoT devices as far as we know, we measure the RAM usage of WAMR, Wasm3, and WARDuino on compatible IoT devices and subtract the RAM usage of other services on that device (e.g., the OS). Note that we only illustrate the RAM usage of the compilation or preparation stage of the baselines except the native because it does not have a translation/compilation stage on the device.

Memory footprint comparison. Fig. 10 shows the average and peak memory consumption of each benchmark. The red line denotes the RAM limit (4KB) on ATmega128 MCU. We can see from the figure that only WAWoT could achieve successful execution of all the benchmarks on this MCU. Even using native execution, *HeatC* and *HeatD* are not executable because they have a large amount of constant data that exceeds the RAM limit. The constant remapping moves the constants to the Flash and makes them executable. The peak RAM consumption of WAMR, Wasm3, and WARDuino are $84.9\times$, $13.6\times$, and $31.9\times$ of WAWoT's, respectively. To be more specific, we have the following three observations.

(1) Compared with the baselines, WAWoT is more suitable for executing WebAssembly on resource-constrained devices. From constrained to powerful, IoT devices are grouped into C0, C1, and C2 classes according to RFC7228 [32] proposed by the Internet Engineering Task Force (IETF). To further illustrate the detailed RAM footprint and compare it with the constraints of each group of IoT devices, we depict the RAM consumption of each approach throughout the compilation and execution stage in Fig. 11 with the memory upper bound of C1 and C2 devices. We can see that WAWoT is the only WebAssembly runtime to execute on C0, C1, and C2 devices. Although both are AOT runtimes, WAMR uses more memory than WAWoT because it uses the heavy-weight LLVM as the compilation backend, which even exceeds the upper bound for C2 devices.

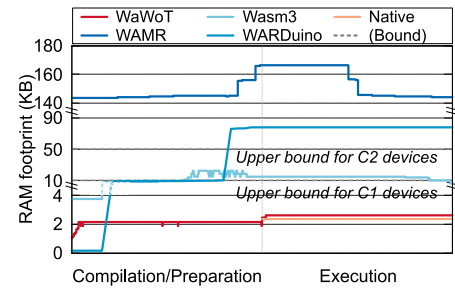


Fig. 11. RAM footprint timeline of FFT benchmark of the compilation and execution stage.

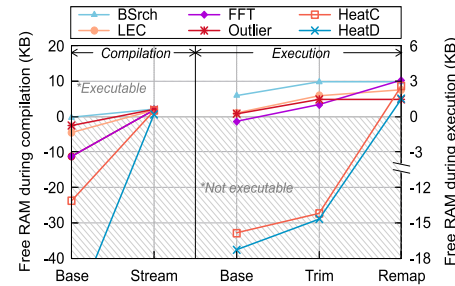


Fig. 12. Breakdown of the RAM gain of each lightweight method of WAWoT.

WARDuino and Wasm3 exhibit a lower RAM footprint than WAMR, but their run-time efficiency is lower than the AOT runtimes because their interpreted execution introduces higher overhead than directly translating the bytecode into assemblies.

(2) The second observation is all three lightweight methods of WAWoT contributes to the final low memory footprint. We depict the free available RAM during the execution of each benchmark with or without our lightweight methods in Fig. 12. We use “Base” to represent the execution without any optimizations, “Stream” for the streamed look-back compilation, “Trim” for post-compile memory trimming, and “Remap” for constants remapping. We can see that none of the benchmarks are compilable without the streamed look-back compilation. Furthermore, our memory trimming technique facilitates the execution of *FFT*, and constant remapping moves the constants of *HeatC* and *HeatD* to the Flash to make them executable.

(3) We also find in Fig. 12 that compared with the native approach, WAWoT introduces 254 bytes run-time RAM overhead. This extra RAM usage is introduced by the size of .wa segment, which saves the necessary data for the execution of WAWoT. We consider this overhead to be acceptable, and the overhead can be counteracted by our constant remapping technique.

C. Effectiveness of Constant Remapping

We evaluate the effectiveness and overhead of constant remapping also with the micro-benchmarks. Results are shown in Table IV. Benchmarks that do not have explicit constants are ignored for space.

TABLE IV
EFFECTIVENESS AND OVERHEAD OF CONSTANT REMAPPING

Bchmrk.	Const. size (B)	Const. remap	Size in Flash (B)	Size in RAM (B)	Perf. overhead
FFT	2,075	No	0	2,331	0
		Yes	2,075	256	+15.89%
LEC	512	No	0	1,601	0
		Yes	512	1,089	+13.07%
HeatC	16,798	No	N/E ¹	N/E ¹	N/E ¹
		Yes	16,798	768	N/A ²
HeatD	16,186	No	N/E ¹	N/E ¹	N/E ¹
		Yes	16,186	1856	N/A ²

¹ N/E: Not executable for its requirement exceeds the available RAM.

² N/A: Not available due to its comparing method is not executable.

The most noticeable benefit of constant remapping is that the *HeatC* and *HeatD* are executable only with this optimization. With remapping, WAWoT moves constants the two benchmarks to Flash during AOT compilation to facilitate its execution. This technique could also be enabled for executable applications (e.g., *FFT* and *LEC*). If we enable remapping for these applications, it could save more RAM space. For example, the RAM usage of *LEC* drops from 1,601 bytes to 1,089 bytes, and the remaining RAM usage is used by non-constant variables, stacks, etc. Nevertheless, reading data from Flash instead of RAM will inevitably slow down the memory manipulating instructions and results in run-time overhead (15.89% for *FFT* and 13.07% for *LEC*).

We can see from Table IV that this overhead is not only related to the size of the constant data but also the accessing times of the constants. For example, the constants in *FFT* are used as the trigonometry table, which is partly read by the application. In contrast, the *LEC* accesses each element in the constant array per execution.

D. Energy Reduction With WAWoT

To give an intuitive illustration of our energy optimization, we build a battery lifetime model considering the power of the MCU, networking, and other peripherals on ATmega128 MCU based IoT prototyping board, proposed in [26]. We evaluate the energy saving against different WebAssembly module update frequencies. Upon each update, the IoT device receives a new WebAssembly module via the network, then WAWoT compiles the module and starts the module for data collection and processing. Our energy model considers all the above procedures, and we use a widely used 2200mAh NiMH battery as the power supply.

We show how the device lifetime varies and the multiplier between lifetime with and without our energy optimization against the update frequency in Fig. 13. Our energy optimization approaches reduce overall energy use and result in a battery lifetime increase of $1.9\times$ to $4.9\times$. As we expected, the benefit of our energy-saving methods is larger when the WebAssembly module is frequently updated. For example, our optimization technique achieves $3.1\times$ longer lifetime when the update interval is 1 day and $1.9\times$ when updates every seven days for

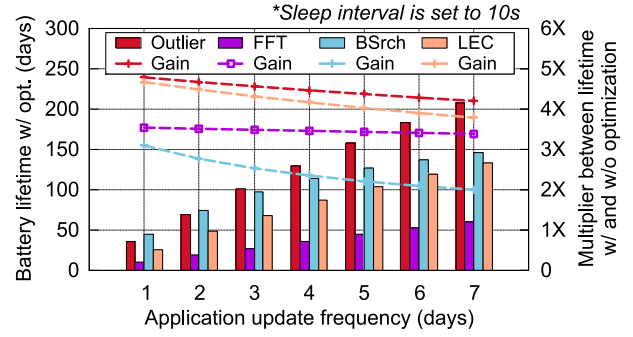


Fig. 13. Node lifetime variation and lifetime gain with energy optimization of WAWoT against update frequency.

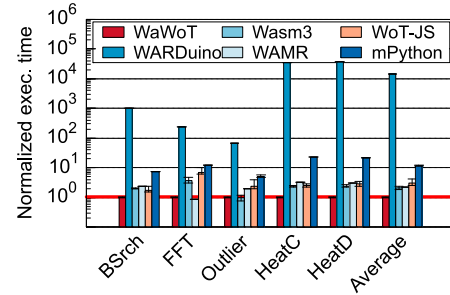


Fig. 14. Execution time comparison of different WoT runtimes (normalized to WAWoT).

BSrch benchmark. This is because the bulk writing technique of WAWoT could reduce the compile-time energy consumption for writing the Flash to a large extent, which finally turns out to have larger benefits when the module is more frequently updated.

E. Runtime Execution Efficiency

We then evaluate the runtime execution efficiency of our WebAssembly-based WoT services architecture. Besides the baseline WebAssembly runtimes, we also compare WAWoT with the prominent WoT runtime such as JavaScript used by NodeWoT and ThingsJS (denoted as WoT-JS), as well as microPython (denoted with mPython) in Fig. 14. A red line denotes the performance of WAWoT.

We can see from the figure that the WebAssembly-based approaches perform better than WoT runtimes in general. Benefiting from the compact binary instruction format, WebAssembly has the potential to be faster than scripting languages. On average, microPython and JavaScript are $10.6\times$ and $2.0\times$ slower than WAWoT. This result strengthens our basic idea that using WebAssembly in WoT services could achieve better efficiency without sacrificing the interoperability of the web pages. Moreover, when we compare the execution time among WebAssembly runtimes, we can see that WAWoT shows its superior efficiency. We give this credit to the AOT compilation we conduct on the IoT devices, which makes WAWoT accelerates for 55.5% and 51.9% against WAMR and Wasm3. WARDuino does not perform well in this experiment mainly because it is designed to support on-device debugging and IoT

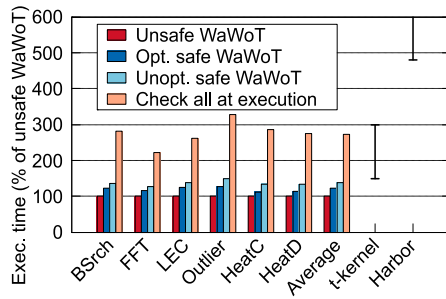


Fig. 15. Overhead reduction brought by the optimizations for sandbox checks of WAWoT.

hardware interaction, which is not optimized solely for execution time.

F. Sandbox Check Optimization

WAWoT optimizes the overhead of sandbox checks by: (1) Moving some of the execution-time checks to compile-time, i.e., EM-C1/C2 in §IV-D. (2) Only examining the instructions that will grow the stack instead of all the instructions when checking the potential stack overflow. Furthermore, we also include the comparison against two software-based safety protection approaches, t-kernel [33] and Harbor [34], to depict the run-time overhead introduced by the sandbox checks of WAWoT.

Hence, to answer the initial question, we compare the execution overhead of “check all at execution”, “unoptimized runtime”, which do not include our optimization to stack overflow checks, and the optimized sandbox runtime with the unsafe version of WAWoT (i.e., canceling all the checks in §IV-D) in Fig. 15.

We observe a 19.1% average overhead of our optimized runtime compared with the unsafe version, which performs better compared with the overhead of existing software-based protection approaches, t-kernel (50%-150%) and Harbor (380%-690%). This performance gain is mainly achieved by our faster memory protection checks. For example, Harbor uses a per-block safety check which uses 65 cycles, while our memory access check (EM-1) uses only 33 cycles. Moreover, WAWoT achieves 89.0% average overhead reduction compared with performing checks mostly at run-time and 12.9% reduction than the unoptimized safe runtime.

We also investigated the compilation time overhead with or without our sandbox check optimization techniques. The results are shown in Fig. 16. We can see from the figure that our optimization approaches also facilitate faster AOT compilation compared to checking all at execution or the unoptimized ones. On average, our optimization results in a 12% reduction in AOT compilation time compared to unoptimized ones and 32.3% compared with checking all at execution. This improvement seems to be counter-intuitive because our optimization approaches move several checks to compilation time, which is supposed to increase the AOT compilation time. According to our further investigation, we found that the additional time of

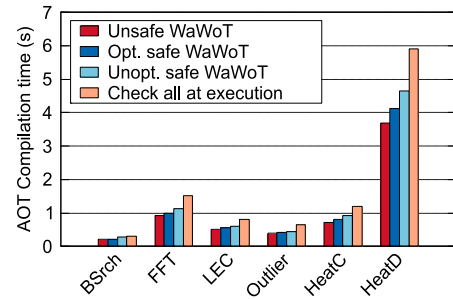


Fig. 16. Compilation time overhead w/ and w/o the optimizations for sandbox checks.

the unoptimized compilation is mainly used for issuing the instructions that are needed for runtime checks. Issuing additional instructions needs to write more data to the Flash during AOT compilation, which is a time-consuming job. On the contrary, moving the run-time checks to compilation time only requires us to add a few additional data structures, and call the checking functions whose overhead is negligible.

VII. DISCUSSION

Compatibility with WoT specification. We analyze if our implementation aligns with the official specification.

Thing Description. In WAWoT, we extend the Thing Description file to enhance support for runtime migration. This extension doesn’t discontinue support for traditional TD files. Both our client- and web-side servients can revert to no-migration mode as the exTD file doesn’t override any fields in the traditional Thing Description.

Affordances. WAWoT’s annotation system is designed to encompass the three Affordances types specified by W3C: properties, actions, and events.

Servients and Protocol Bindings. Our web-side servient, possessing abundant resources, supports commonly used protocol bindings like HTTP, CoAP, and MQTT. However, in our current implementation, the device-side servient, equipped with only 4KB RAM, supports CoAP protocol binding based on TCPlp [30] due to its optimized performance for resource-constrained devices. Notably, the device-side servient is not limited to a specific protocol stack; the server-side binding implementation can be easily adapted for IoT devices with abundant resources.

Compatibility with IoT standards. WAWoT is not designed to replace the original WoT or IoT standards, but to serve as a complementary add-on for existing WoT platforms to improve flexibility and efficiency.

The oneM2M standard is designed to unify vertical IoT applications via a common middleware. The architecture of oneM2M is composed of three layers: application, services, and network layer. The services layer of oneM2M holds significant importance because it provides commonly needed functions for IoT applications.

Integrating WAWoT with oneM2M is *easy to implement* and *beneficial*. Thanks to the modular design of oneM2M, we can simply add WAWoT runtime and a migration handler to

the common services functions of oneM2M. During development, applications can be developed via WAWoT APIs with placement annotations (see §IV-A of our paper) and deployed in the form of WebAssembly. Applications can take advantage of WAWoT runtime to execute efficiently and the migration handler manages the flexible offloading of application entities. Furthermore, existing common services of oneM2M such as registration and service discovery can, in turn, enhance the efficiency of online migration and management of WebAssembly modules.

Compile WebAssembly to native on cloud or device?

Storing the WebAssembly modules on the server and compiling them to the native assembly of target IoT devices on-demand is a viable alternative to WAWoT's on-device AOT compilation. Performing AOT compilation on the cloud could reduce the runtime overhead of the IoT device. Nevertheless, if we do so, the system portability will be greatly reduced and the complexity will be greatly increased. This is because it may require compiling and saving binaries for heterogeneous IoT devices on the cloud and dispatching each binary for its target device during run-time.

This portability brought by WAWoT's approach is necessary for two types of applications: (1) Applications that need ad-hoc or peer-to-peer computation migration, such as vehicle-to-everything (V2X) computing, mobile agent computing, and personal area network (PAN). We use V2X computing as an example, which means that cars could communicate with the roadside units (RSU) and migrate computing tasks such as navigation planning to the RSUs with tight timeliness requirements. Without WAWoT's portability, each migration involves a device-to-cloud communication for downloading AOT compiled modules, whose latency is unpredictable due to the complicated network environment. Additionally, storing numerous versions of binary files, even if they are small, on end devices poses storage pressure on limited resources. (2) Applications with bi-directional security requirements, i.e., the application does not trust the computing device and vice versa. The smart contract application [35], [36], [37] is the most suitable case. The sandboxed execution facilitated by WebAssembly and WAWoT is necessary for these applications, and compiling to native code on the cloud enlarges the attack surface.

VIII. RELATED WORK

Systems extending IoT/WoT flexibility via computation migration can be classified as VM-based [10], native-based [38], [39], or CLR-based [6], [7], [8], [14], [40].

VM solutions migrate full VM images, requiring substantial resources which is unavailable on IoT devices. Recent native methods like EdgeProg [38] and TinyLink 2.0 [39] propose DSLs to partition service logic across devices and automatically generate native executables on the corresponding platform. However, native approaches struggle to support fine-grained state migration due to different memory layouts and instruction sets.

Acknowledging the limitations in the aforementioned categories, WAWoT opts for a CLR-based solution to achieve adaptable and efficient WoT services. Prior efforts [6], [8] utilize JavaScript for WoT service migration due to its status as a standard Web language, ensuring better interoperability with web pages. Notably, ThingsJS [8] stands as the most relevant work in this context, offering offloading capability for each WoT agent to enable more flexible execution. However, in comparison to our approach, ThingsJS exhibits lesser flexibility primarily due to two reasons. Firstly, it lacks WoT-targeted annotation and automatic Thing Description generation, essential for achieving simplified and more flexible WoT development. Secondly, ThingsJS operates at a JS-file level for migration granularity, coarser than the function-level granularity of WAWoT. Additionally, JS-based approaches often surpass the resource limits of IoT devices.

WebAssembly emerges as a new standard web language with enhanced execution efficiency. Thus, in our paper, we construct flexible WoT services using WebAssembly. WiProg [14] shed light on the possibility of offloading WebAssembly between devices. Nevertheless, WAWoT distinctively targets WoT scenarios, employing various automation techniques to reduce development overhead. Specifically, WAWoT considers the WoT specification when designing annotations, proposes an extension to the official Thing Description file for flexible WoT offloading, and automates exTD file generation. There are also pioneer works on in-place fast compilation of WebAssembly [23]. However, WAWoT accommodates more resource-constrained devices due to our lightweight device-side WebAssembly runtime, which is orthogonal to the existing works.

IX. CONCLUSION

In this paper, we propose WAWoT, a flexible and efficient WebAssembly-based Web of Things framework. For flexibility, we advocate an annotation-based WoT service description and automatic WoT service migration mechanism. For efficiency, we leverage Ahead-of-Time compilation on resource-constrained IoT devices with a series of lightweight approaches. We also reduce energy consumption by optimizing peripheral accessing and duty-cycling. Moreover, WAWoT achieves efficient sandbox execution by moving most of the run-time safety checks to compile-time. Results show that WAWoT reduces the RAM usage by $84.9\times$ and the energy consumption by $1.9\times$ to $4.9\times$.

ACKNOWLEDGMENT

The authors thank the editor and the reviewers for their useful feedback that improved this paper.

REFERENCES

- [1] M. Noura et al., "Interoperability in Internet of Things: Taxonomies and open challenges," *Mobile Netw. Appl.*, vol. 24, pp. 796–809, Jun. 2019.
- [2] M. Noura et al., "Automatic knowledge extraction to build semantic web of things applications," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8447–8454, Oct. 2019.

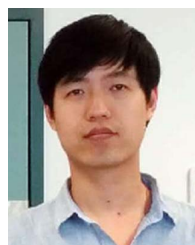
- [3] "Web of things standard." W3C. 2022. [Online]. Available: <https://www.w3.org/WoT>
- [4] D. Ibaseta et al., "Monitoring and control of energy consumption in buildings using wot: A novel approach for smart retrofit," *Sustain. Cities Soc.*, vol. 65, 2021, Art. no. 102637.
- [5] P. Patel et al., "From raw data to smart manufacturing: AI and semantic web of things for industry 4.0," *IEEE Intell. Syst.*, vol. 33, no. 4, pp. 79–86, Jul./Aug. 2018.
- [6] C. Aguzzi et al., "From cloud to edge: Seamless software migration at the era of the web of things," *IEEE Access*, vol. 8, pp. 228118–228135, 2020.
- [7] M. Vögler et al., "Optimizing elastic IoT application deployments," *IEEE Trans. Services Comput.*, vol. 11, no. 5, pp. 879–892, Sep./Oct. 2016.
- [8] J. Gascon-Samson et al., "ThingsJS: Towards a flexible and self-adaptable middleware for dynamic and heterogeneous IoT environments," in *Proc. ACM MA4IoT*, 2017, pp. 11–16.
- [9] E. ThingWeb. (2023) "Nodewot project." [Online]. Available: <https://github.com/eclipse-thingweb/node-wot>
- [10] K. Tsakalozos et al., "Live VM migration under time-constraints in share-nothing IaaS-clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2285–2298, Aug. 2017.
- [11] A. Haas et al., "Bringing the web up to speed with WebAssembly," in *Proc. ACM PLDI*, 2017, pp. 185–200.
- [12] K. Liu et al., "Demystifying web-based mobile extended reality accelerated by WebAssembly," in *Proc. ACM Internet Meas. Conf.*, New York, NY, USA, Oct. 2023, pp. 145–153.
- [13] S. Shillaker and P. Pietzuch, "FAASM: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX ATC*, 2020, pp. 419–433.
- [14] B. Li et al., "WiProg: A WebAssembly-based approach to integrated IoT programming," in *Proc. IEEE INFOCOM*, 2021, pp. 1–10.
- [15] "Wasm3," Wasm3 Labs. 2022. [Online]. Available: <https://github.com/wasm3/wasm3>
- [16] T. Lauwaerts et al., "WARDuino: An embedded webassembly virtual machine," *J. Comput. Lang.*, Feb. 2024, Art. no. 101268.
- [17] "Webassembly micro runtime," Bytecode Alliance. 2022. [Online]. Available: <https://github.com/bytecodealliance/wasm-micro-runtime>
- [18] B. Li et al., "S2: A small delta and small memory differencing algorithm for reprogramming resource-constrained IoT devices," in *Proc. IEEE INFOCOM Demos Posters*, 2021, pp. 1–2.
- [19] N. Reijers and C.-S. Shih, "CapeVM: A safe and fast virtual machine for resource-constrained Internet-of-Things devices," in *Proc. ACM SenSys*, 2018, pp. 250–263.
- [20] X. Fan et al., "Tensor-based confident information coverage reliability of hybrid internet of things," *IEEE Trans. Mobile Comput.*, vol. 23, no. 3, pp. 2171–2185, Mar. 2024.
- [21] H. Li et al., "Flash-optimized temporal indexing for time-series data storage on sensor platforms," *ACM Trans. Sensor Netw. (TOSN)*, vol. 10, no. 4, pp. 1–30, 2014.
- [22] B. L. Titzer, "A fast in-place interpreter for WebAssembly," in *Proc. ACM OOPSLA*, Oct. 2022, pp. 646–672.
- [23] B. L. Titzer, "Whose baseline compiler is it anyway?" in *Proc. IEEE/ACM CGO*, Mar. 2024, pp. 207–220.
- [24] Q. He et al., "Pyramid: Enabling hierarchical neural networks with edge computing," in *Proc. ACM WWW*, 2022, pp. 1860–1870.
- [25] P. Dutta et al., "Design of a wireless sensor network platform for detecting rare, random, and ephemeral events," in *Proc. IEEE IPSN*, 2005, pp. 497–502.
- [26] W. Dong et al., "Optimizing relocatable code for efficient software update in networked embedded systems," *ACM Trans. Sensor Netw. (TOSN)*, vol. 11, no. 2, 2015, Art. no. 22.
- [27] Y. Liu et al., "Does wireless sensor network scale? A measurement study on GreenOrbs," in *Proc. IEEE INFOCOM*, 2011, pp. 873–881.
- [28] A. Shoeb et al., "Detecting seizure onset in the ambulatory setting: Demonstrating feasibility," in *Proc. IEEE EMBS*. Piscataway, NJ, USA: IEEE Press, 2006, pp. 3546–3550.
- [29] R. Newton et al., "Wishbone: Profile-based partitioning for sensor network applications," in *Proc. USENIX NSDI*, 2009, pp. 395–408.
- [30] S. Kumar et al., "Performant TCP for low-power wireless networks," in *Proc. USENIX NSDI*, 2020, pp. 911–932.
- [31] F. Marcelloni and M. Vecchio, "An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks," *Comput. J.*, vol. 52, no. 8, pp. 969–987, 2009.
- [32] (2022) "RFC7228: Terminology for constrained-node networks," IETF. 2022. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc7228.html>
- [33] L. Gu and J. A. Stankovic, "T-Kernel: Providing reliable OS support to wireless sensor networks," in *Proc. ACM SenSys*, 2006, pp. 1–14.
- [34] R. Kumar et al., "Harbor: Software-based memory protection for sensor nodes," in *Proc. ACM/IEEE IPSN*, 2007, pp. 340–349.
- [35] N. He et al., "{EOSAFE}: Security analysis of {EOSIO} smart contracts," in *Proc. USENIX Secur.*, 2021, pp. 1271–1288.
- [36] Z. Yang et al., "Seraph: Enabling cross-platform security analysis for EVM and WASM smart contracts," in *Proc. IEEE/ACM ICSE Companion*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 21–24.
- [37] B. Jiang et al., "WANA: Symbolic execution of WASM bytecode for extensible smart contract vulnerability detection," in *Proc. IEEE QRS*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 926–937.
- [38] B. Li and W. Dong, "EdgeProg: Edge-centric programming for IoT applications," in *Proc. IEEE ICDCS*, 2020, pp. 212–222.
- [39] G. Guan et al., "TinyLink 2.0: Integrating device, cloud, and client development for IoT applications," in *Proc. ACM MobiCom*, 2020, pp. 1–13.
- [40] H.-J. Jeong et al., "Seamless offloading of web app computations from mobile device to edge clouds via HTML5 web worker migration," in *Proc. ACM SoCC*, 2019, pp. 38–49.



Borui Li (Member, IEEE) received the Ph.D. degree in computer science from Zhejiang University, China, in 2022. Currently, he is an Assistant Professor with the Southeast University. His research interests include IoT system, edge computing, and WebAssembly. He is a member of the ACM.



Hongchang Fan received the B.S. and M.Sc. degrees from the College of Computer Science, Zhejiang University, China, in 2019 and 2022, respectively. His research interests include the Internet of Things and wireless sensor networks.



Yi Gao (Member, IEEE) received the B.S. and Ph.D. degrees from Zhejiang University, China, in 2009 and 2014, respectively. From 2015 to 2016, he visited McGill University as a Visiting Scholar. Currently, he is a Full Professor with Zhejiang University. His research interests include the Internet of Things and intelligent edge computing. He is a member of the ACM.



Wei Dong received the B.S. and Ph.D. degrees from the College of Computer Science, Zhejiang University, in 2005 and 2011, respectively. Currently, he is a Full Professor with the College of Computer Science, Zhejiang University. He has published over 100 papers in prestigious conferences and journals, including NSDI, MobiCom, ToN, and TC. His research interests include Internet of Things and sensor networks, wireless, and mobile computing.