Bringing WebAssembly to Resource-constrained IoT Devices for Seamless Device-Cloud Integration

Borui Li, Hongchang Fan, Yi Gao and Wei Dong*
College of Computer Science, Zhejiang University, and
Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China
{libr,fanhc,gaoy,dongw}@zju.edu.cn

ABSTRACT

Recent years have witnessed the progressive integration between IoT (Internet of Things) devices and the cloud server, which promotes the efficiency and interoperability of IoT applications. WebAssembly, known for its performance and portability, is considered a promising technology to bridge the heterogeneity between devices and the server. Nevertheless, resource-constrained devices, which are commonly deployed in the wild, have difficulty participating in this device-cloud integration because they can hardly run WebAssembly efficiently.

Hence, we propose WAIT, a *lightweight* WebAssembly runtime on resource-constrained IoT devices for device-cloud integrated applications. WAIT is the first work to enable the Ahead-of-Time (AOT) compilation of WebAssembly on resource-constrained devices by leveraging several approaches to reduce memory usage. Moreover, WAIT introduces various safety checks at compile-time to guarantee the sandbox execution of WebAssembly and optimizes energy consumption for IoT devices. Results show that WAIT achieves 84.8× lower RAM usage compared with the state-of-the-art WebAssembly AOT runtime, and reduces energy consumption by 1.2×~4.9× while guaranteeing the sandboxed execution of WebAssembly modules.

CCS CONCEPTS

• **Computer systems organization** → **Embedded and cyber-physical systems**; • **Networks** → *Cloud computing*; *Network components*; • **Information systems** → *Computing platforms.*

KEYWORDS

WebAssembly, Internet of Things, Ahead-of-Time Compilation

ACM Reference Format:
Borui Li, Hongchang Fan, Yi Gao and Wei Dong*. 2022. Bringing WebAssembly to Resource-constrained IoT Devices for Seamless Device-Cloud Integration. In *The 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22), June 25–July 1, 2022, Portland, OR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3498361.3538922

1 INTRODUCTION

Nowadays, the Internet of Things (IoT) technology penetrates various scenarios of our daily lives and work, such as city-scale sensing [1, 16], wild monitoring [4, 37], and smart healthcare [45, 53]. The prevalent cloud computing further extends the ability of IoT applications. The cloud server aggregates the sensing data from multiple sources to generate knowledge and facilitates the convenient interaction between IoT devices and human beings, which finally leads to the device-cloud integration.

Recent advances of computation offloading [32, 33] tighten the integration by supporting not only data exchange, but also the portability and interoperability of computing tasks between the device and the cloud. To smoothen the offloading process, a consistent execution environment that empowers seamless migration of the code between heterogeneous instruction set architectures (ISAs) is necessary. Researchers address this issue via virtual machines (VMs) [44, 48] and common language runtimes (CLRs) such as C# [11], Java [29] and JavaScript [36].

Nevertheless, many IoT devices only have *constrained computing resources* because they are generally deployed in the wild and expected to last for months or longer with battery power or even energy-harvesting. For example, the Atmel ATmega128 microcontroller (MCU), commonly used in industrial automation and wild sensing, is only equipped with 4KB RAM and 128KB Flash while achieving 16.43mW average active power. Existing efforts towards the consistent execution on heterogeneous platforms either require OS-level support (e.g., Hypervisor for VMs) or incur much overhead (e.g., over 100× for JavaScript engine), which require massive computing resources. *WebAssembly* [21] seems to be a promising technology to achieve seamless device-cloud integration on resource-constrained devices by its portability and efficiency. WebAssembly is a bytecode format designed to run on a wide range of platforms. It also serves as a common compilation target of various high-level languages (e.g., Rust, C++) and exhibits better runtime performance than other CLRs [33].

However, supporting device-cloud integration with WebAssembly on resource-constrained devices still faces non-trivial challenges.

Challenge 1: how can we support the device-cloud integrated application and execute it efficiently on resource-constrained devices? Executing WebAssembly needs a dedicated runtime to translate bytecode into native instruction. Some attempts are targeting low-end devices by building an interpreter of WebAssembly, such as `wasm3` [50], but they run more than ten times slower than the native code. Ahead-of-time (AOT) translation could accelerate the

* Corresponding author.

MobiSys '22, June 25–July 1, 2022, Portland, OR, USA

ACM ISBN 978-1-4503-9185-6/22/06…$15.00
https://doi.org/10.1145/3498361.3538922

execution by translating the bytecode to native at load time. Nevertheless, almost all AOT approaches of WebAssembly found in the literature are not feasible due to the constrained resources.

Challenge 2: how can we guarantee the sandboxed execution of WebAssembly? The device-cloud integrated computing scheme makes the IoT devices more vulnerable to network-based attacks than standalone applications. Furthermore, the low-power, low-cost IoT devices generally do not have memory management unit or CPU privileged levels, which makes sandboxed execution harder.

Challenge 3: how can we optimize the energy consumption of the on-device execution? Peripheral accessing, duty-cycling, and Flash accessing are unique features of IoT applications compared with traditional desktop applications. However, the core specification of WebAssembly does not include support for the above features so far.

Therefore, we present **WAIT**, a lightweight WebAssembly runtime on resource-constrained IoT devices for device-cloud integrated applications. With WAIT, users could write device-cloud integrated applications in various high-level languages and compile them to WebAssembly modules. Then the application logic to be executed on the IoT device is disseminated via wired or wireless. Upon receiving the module, WAIT performs on-device AOT compilation, checks the sandbox guarantees, and executes the module. To be more specific, WAIT proposes the corresponding solutions to the three challenges.

Solution 1: WAIT advocates a loading agent to regularly communicate with the cloud server and load the WebAssembly module if available. A lightweight AOT compiler and a series of memory optimizations lie inside the agent to correctly and efficiently execute WebAssembly on resource-constrained devices. To cope with the limited resources, WAIT 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.

Solution 2: In order to provide a safe and deterministic integrated execution environment on the resource-constrained devices, WAIT introduces memory safety and control-flow integrity checks. To minimize the overhead during execution, WAIT moves most of the checks *to the AOT compilation stage* and carefully *selects the instructions being checked.*

Solution 3: For developers, WAIT provides *IoT-related APIs* for peripheral-accessing and duty-cycling to facilitate complete IoT programming. Moreover, WAIT adopts the *bulk instruction writing* and the *I/O direct accessing* approach to reduce the energy consumption of both compile- and run-time.

We implement WAIT and evaluate its performance extensively. Results show that: (1) The state-of-the-art WebAssembly interpreter [50] and AOT runtime [9] consume 13.6× and 84.8× more RAM than WAIT, respectively. (2) The energy optimization approaches of WAIT achieve 1.2×~4.9× power reduction. (3) WAIT only incurs 19.1% average run-time overhead for ensuring the sandboxed execution of WebAssembly. WAIT is open-source on https://github.com/liborui/WAIT.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation of WAIT. Section 3 presents its overview. Section 4 and Section 5 describe the design and implementation details. Section 6 evaluates the performance extensively. Section 7 discusses some important issues and Section 8 presents the related work. Finally, Section 9 concludes the paper.

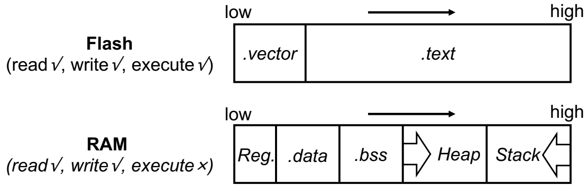


Figure 1: Flash and RAM layout of constrained IoT devices.

2 BACKGROUND

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

Linear memory. To achieve the isolated execution of a WebAssembly module by design, WebAssembly organizes its main storage in a linear memory region. The linear memory is a mutable continuous array of bytes containing global/local variables and constants. At runtime, a WebAssembly module can only access its own 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 because it is considered harmful [13]. 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 can be nested, but the function-type 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.

System architecture of resource-constrained IoT devices. Generally, resource-constrained IoT devices adopt the Harvard architecture for their simple circuit design. As shown in Figure 1, the storage on resource-constrained devices is organized in two parts, the RAM and the Flash, The data saved in the Flash is readable, writable and executable, hence the binary instructions (i.e., the `.text` segment) reside in the Flash. The data used or generated at run-time is stored in the RAM. The static and global variables are saved `in.data`, uninitialized data is saved in `.bss`, and the dynamically allocated data (e.g., using `malloc`) is saved in the heap. However, the RAM data is not executable even if it is assembly code. The RAM is commonly much smaller than the Flash for cost reasons, which motivates us to pay more attention to the efficient usage of the RAM in the design of WAIT.

Interpreted, JIT, and AOT bytecode execution. There are mainly three approaches to run 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

Table 1: Comparison of the WebAssembly runtime

Runtime	Stars	Linux-compatible devices			IoT devices	
		Interp.	JIT	AOT	Interp.	AOT
Wasmer [51]	10.1k	✘	✓	✓	✘	✘
Wasmtime [8]	5.6k	✘	✓	✘	✘	✘
Lucet [7]	3.9k	✘	✘	✓	✘	✘
Wasm3 [50]	3.9k	✓	✘	✘	✓	✘
WAMR [9]	2.2k	✓	✓	✓	✓	✘
WAC [28]	378	✓	✘	✘	✓	✘

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. The interpreted execution is generally slower than those who use JIT or AOT because the runtime based on JIT or AOT could perform various optimizations to the binary but the interpreter can hardly do so.

We survey the existing WebAssembly runtimes that are publicly available and their underlying translation technologies in Table 1. 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 the IoT devices due to the usage of a heavy-weight code generation framework (e.g., LLVM). WAIT bridges the gap between efficient AOT compilation of WebAssembly and IoT devices.

3 SYSTEM OVERVIEW OF WAIT

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

3.1 Design goals

During the design of WAIT system, we take the characteristic of the IoT device (limited memory and energy) and safety concerns brought by the device-cloud integration into consideration. Thus, the design and implementation of WAIT must satisfy the following goals:

- **Memory efficiency.** The first and foremost goal of WAIT is to get WebAssembly executed. Compared with the GB-level RAM on Linux-compatible devices, the memory on the IoT device shrinks to KB-level, which makes most of the existing approaches for executing WebAssembly unavailable. Hence, WAIT must minimize the memory footprint both at compile-time and run-time.
- **Execution safety.** Execution safety is paramount for the device-cloud integrated computing scheme. This is because the integration brings a broader attack surface in addition to a performance improvement due to execution of code transmitted over a network.
- **Energy efficiency.** The IoT device is generally powered by batteries, especially those which are deployed in the wild. A subtle increase in energy consumption could result in days of lifetime difference. Therefore, we recognize energy efficiency as one of the design goals.

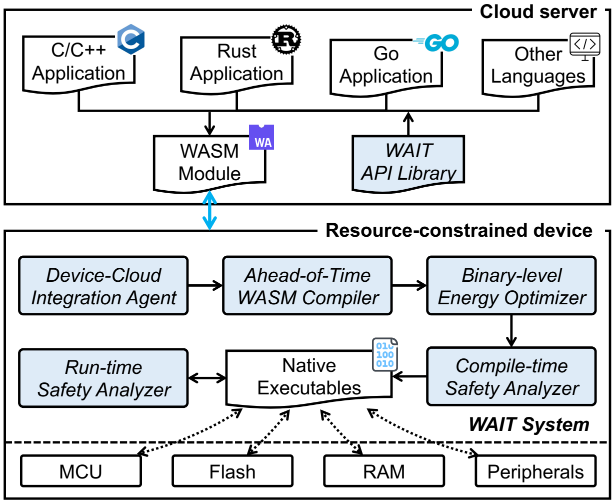


Figure 2: Overview of WAIT system.

3.2 System workflow of WAIT

Figure 2 illustrates the overall workflow of WAIT. Developers could write a device-cloud integrated application in various languages such as C, C++ and Rust. WAIT provides an *API library* including IoT-related APIs for developers to manipulate peripherals or fall into low-power mode. Then, the application is compiled to a WebAssembly module that could run on both the cloud server or the resource-constrained IoT device facilitated by WAIT. On the IoT device, an *Ahead-of-Time compiler* (Section 4.1) transforms the WebAssembly bytecode into native assembly. Then, the module is checked for safey and optimized for energy by a *compile-time safety analyzer* (Section 4.2) and a *binary-level energy optimizer* (Section 4.3), respectively. Finally, the module runs on the constrained IoT device and a small fraction of the checks are left to the *run-time analyzer* (Section 4.2). With respect to the coordination between IoT devices and the cloud, a *device-cloud integration agent* residing on IoT device is responsible for receiving the WebAssembly module via network and saving it to Flash (Section 4.4).

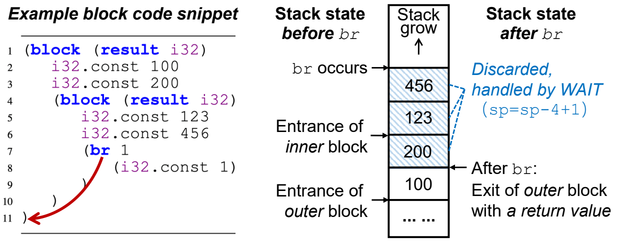
4 WAIT DESIGN

In this section, we first present the lightweight methods we employ in our AOT compiler to reduce the memory usage of both the compile and execution stages. Then we describe the safety checks and energy optimizations of WAIT.

4.1 Lightweight WebAssembly runtime

As we stated in Section 3.1, reducing the memory footprint of WAIT as much as possible is the key to realizing efficient WebAssembly execution on resource-constrained devices. Therefore, we present our approaches to making WAIT 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 [9, 51] exploit complicated code generation frameworks such as LLVM or Cranelift [6] to translate the bytecode to native instructions. These frameworks lead to an unacceptable



Example block code snippet

```
1  (block (result i32)
2     i32.const 100
3     i32.const 200
4     (block (result i32)
5        i32.const 123
6        i32.const 456
7        (br 1
8           (i32.const 1)
9        )
10    )
11 )
```

Figure 3: The stack state before and after the BR instruction. BR m means jumping out m levels of blocks. Here $m = 1$ means the first outer block, i.e., the block from line 1 to 11.

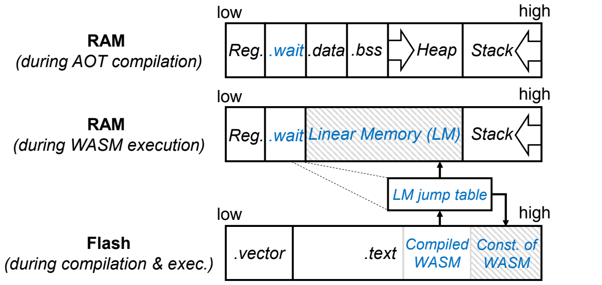


Figure 4: The memory layout of WAIT during compilation and execution.

RAM footprint and can hardly port to restricted devices. Hence, we build our own AOT compiler by replacing each bytecode with a succession 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 [34], WAIT 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 we 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 WAIT are two-fold.

First, WAIT 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, WAIT upgrades the streamed compilation with a look-back. The look-back means WAIT labels the undetermined branch targets with a pseudo-address during the streamed compilation and substitutes them when the compilation of all the bytecode is finished. For example, WAIT labels each BLOCK and LOOP using a universal index throughout the WebAssembly module and temporarily substitutes the target of each branch instruction with the index according to the branch depth. After the complete pass of the streamed compilation, WAIT then fills the physical address instead of the temporary index.

Second, the stack state is critical to the execution correctness of WebAssembly, but the native jump instructions on the IoT device will not keep an eye on the stack. For example, as shown in the code snippet in Figure 3, the shaded stack frame should be discarded after executing the BR (line 7), while the native jmp assembly does nothing with the stack. Hence, WAIT leverages *lightweight stack restoring* to take care of the stack before and after the block-type instructions (e.g., BR). During the first pass of the steamed compilation, WAIT records the stack depth of each BLOCK and LOOP entrances. Then during the look-back compilation stage, WAIT restores the stack to the proper depth (i.e., the entrance depth plus the return size of the target block/loop). We further minimize the restoration costs (i.e., CPU cycles) by selectively emitting different instructions under different stack states. Taking the ATMega128 MCU as an example, WAIT emits pop when discarding only one value (4 cycles per pop) and modifies the SP register directly for correct stack depth when discarding multiple values (5 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 is 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, WAIT needs to answer two questions: (1) How can we differentiate between useful and useless data? (2) how can we change the memory layout at run-time to use the spare space considering the layout is generally ascertained at the linking phase of compilation. For the first question, WAIT leverages a new section named .wait as shown in Figure 4. WAIT's data structures, which are helpful after AOT compilation, are assigned to the .wait section using attribute specifiers. For the second question, we found that the parameters of the heap (e.g., start address and size) are stored in several global variables of malloc(). Hence, we instrument the malloc() for those variables, assign them to the .wait section, and assign the new parameters 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) [43]. Most of these data never change throughout the execution of the application but occupies a large amount of precious RAM. Hence, our basic idea is to reduce the RAM usage by moving constants, which originally located in the RAM, to the relatively abundant Flash space. This design leads to a question that the constants are logically located in the linear memory but physically distributed in RAM and Flash. Therefore, WAIT leverages a linear memory (LM) jump table, which is located in the .wait section, that maps the logical and physical address, as shown in the Flash illustration of Figure 4. Each time the WebAssembly module intends to access linear memory, WAIT takes over the accessing and refers to the LM jump table for exact address. Undoubtedly, this approach will introduce run-time overhead because accessing the

Table 2: Sandbox checks of WAIT

Number	Description
***Compile-time checks**. (CC for control-flow integrity checks, CM for memory safety checks.)*	
CC-1	The number of elements in the top of the value stack ≥ the number of the operand of the next instruction will consume.
CC-2	The type of the elements should be the same to the operand type of next instruction.
CC-3	Each FUNC must be terminated with RETURN; LOOP/BLOCK must be terminated with END; IF must be terminated with END or ELSE.
CC-4	For each function, its type index ≤ maximum number of signatures listed in the type section.
CC-5	Individually examine the stack for each function/loop to prevent the outer stack from being modified by the instruction of a inner layer.
CC-6	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.
CC-7	The function index of the start component ≤ maximum function index.
CC-8	The branch index of a BR/BR_IF ≤ the block depth of current instruction.
CM-1	A linear memory region is compulsory for a valid WebAssembly module.
CM-2	The size of the data segment < minimum declared length of the linear memory.
CM-3	The access index of variable instructions <= the total number of the local/global variables.
CM-4	The SET_LOCAL and SET_GLOBAL instruction could not manipulate the immutable variables.
CM-5	The imported variable could not be modified by SET_LOCAL and SET_GLOBAL.
***Execution-time checks**. (EC for control-flow integrity checks, EM for memory safety checks.)*	
EC-1	Current stack depth + number of additional stack use by next instruction ≤ maximum stack depth
EC-2	The signature of the callee function must be identical to the signature specified in the CALL_INDIRECT.
EC-3	The index of the callee function ≤ the length of the TABLE in the CALL_INDIRECT.
EM-1	The memory addressing index ≤ size of existing linear memory.

Flash is slower than directly load data from RAM. We will evaluate this overhead in Section 6.5.

4.2 Two-phase sandbox checks

WAIT performs sandbox checks at both the compile phase and execution phase.

The high-level goals of WAIT's sandbox checks are to guarantee control-flow integrity (CFI) and memory safety during the execution of the WebAssembly module. Specifically, WAIT protects the CFI by restricting execution to either the compiled assembly of the bytecode following the behavior defined by the WebAssembly or the code offered by WAIT, and ensures memory safety by prohibiting illegal read and write for both the linear memory and the stack. We list the complete list of checks that WAIT performs in Table 2, classified by the stage the constraints are checked.

Protecting control-flow integrity. We show the completeness of our protection of CFI by grouping the checks to the corresponding WebAssembly instructions. The CFI could be affected by both the control-flow instructions and the other simple instructions.

Simple instructions. We start with simple instructions. WebAssembly is a stack-based virtual machine. Hence most of the instructions interact with the stack. Therefore, we check the quantity (CC-1) and type (CC-2) of the operands that reside on the top of the stack to guarantee the valid execution of each simple instruction. Moreover, WAIT 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 (EC-1). Intuitively, each instruction that interacts with the stack should go through EC-1. However, we found that only a small portion of the bytecodes grow the stack. These are the instructions that push variable onto the stack and perform function calls. Hence, to reduce the run-time overhead, we optimize EC-1 by only checking these instructions that will grow the stack. We will discuss the performance of this optimization in Section 6.3.

Common checks for both function- and block-type instructions. The following checks are shared by both types. We start with sanity checks that ensure each building block of the control flow is correctly organized. For example, the fundamental control flow constraint would be broken if there are unclosed functions or blocks (CC-3). Moreover, the other sanity check enforces an explicit and valid function signature (CC-4), which also provides necessary signature information for the following checks. Considering the stack state of control instructions, WAIT ensures the callee function (or inner block) could not affect the stack of caller function (or outer block) (CC-5). With the aid of the signature information, we also check the correctness of the stack state before and after executing a function/block (CC-6).

Function-type instructions. To start the execution, a WebAssembly module must have a start component and the start function index defined in the module must be valid (CC-7). WebAssembly supports CALL_INDIRECT to invoke the function provided by the WAIT runtime. Hence, we should check whether the index (EC-2) and the signature (EC-3) of the indirect call are valid.

Block-type instructions. Different from functions, blocks in WebAssembly could be nested with each other and jumped out via BR. Thus, WAIT checks the validity of each branch instruction like BR and BR_IF (CC-8).

Ensuring memory safety. Figure 4 illustrates the memory layout of WAIT. The WebAssembly module is allowed to access only the linear memory region (including the Flash region that stores the constants). Any reads and writes to other regions is illegal because the access could compromise the whole device. We categorize the instructions that could access memory as the variable instructions and memory instructions.

Basic sanity check. Two sanity checks are performed to ensure basic memory safety. First, the module is required to contain a default linear memory (CM-1). Second, the size of the data segment of WebAssembly can not exceed the minimum length of the linear memory (CM-2).

Memory instructions. The memory instructions (e.g., I32.LOAD and F64.STORE) directly manipulate the linear memory, which is the key protected area of WAIT. The exact address to be accessed can not be obtained at AOT-compilation phase even using symbolic execution approaches, which are hardly implementable on IoT devices. Therefore, WAIT uses a run-time memory access checker and a linear memory jump table to ensure the memory is legally accessed (EM-1).

Variable instructions. The variable instructions (e.g., `GET_LOCAL` and `SET_GLOBAL`) manipulate the global and local variables that reside in the stack rather than linear memory. Different from the memory instructions, the index that variable instructions access is explicitly expressed by the operand. Hence, WAIT statically checks if the index is illegal (CM-3), and whether the instructions are modifying variables that are set to immutable (CM-4 and CM-5). 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 the full control of the IoT device, and leads to unpredictable results. CM-3 and CM-4 can prevent the illegal linear memory access, which prevents the potential malicious program from modifying key data and code.

4.3 Energy optimization of the compilation and execution phase

The above lightweight runtime enables basic execution of WebAssembly, and the two-phase sandbox checks ensure execution safety. 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, WAIT 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 [35]. 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, WAIT 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.

IoT-related APIs and I/O direct accessing at run-time. There are two prominent differences that distinguish IoT applications and the ones run 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 important features, WAIT provides programming APIs for developers and efficient run-time supports for these features.

API design. The primary design consideration of WAIT's peripheral-accessing APIs is not to implement APIs for every kind of peripherals. This will add additional overhead to store nearly identical implementations, such as providing both temperature read and light read APIs even if they share the same underlying protocol to communicate with MCU. Based on our investigation of applications from popular IoT forums [2, 12], we found that the peripherals generally communicate with the MCU using the following protocols: Analog, Digital, UART, I2C, PWM and SPI. Hence, WAIT 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, WAIT also provides `WAIT_sleep()` API to set the device to low-power mode. On the other hand, benefited by the multi-language support of WebAssembly, the cloud-device applications could be compiled from several high-level languages such as C, Rust and Go. Hence, the IoT-related APIs that WAIT provides also supports various languages. For example, to use our Rust API to read data from a digital sensor, developers should firstly declare WAIT's API as an external function as follows:

```rust
#[wasm_bindgen]
extern {
   pub fn WAIT_readDigital(pinNum: i32) -> (val: i32);
}
```

Similarly, the WAIT C API to read the digital sensor is:

```c
extern int WAIT_readDigital(int pinNum);
```

Then, developers could use the declared function in their application akin to other system calls, and these APIs will be processed during our AOT compilation.

I/O direct accessing. The only challenge that remains now is how WAIT supports the above 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 introduces extra invocation cost at run-time because calling an external function needs to break out from our sandboxed execution, go through several safety checks and finally call the imported functions. By taking advantage of our AOT compilation, WAIT has the opportunity to modify the translated assembly. Hence, during AOT compilation, WAIT replaces the instructions that call the imported function with the assemblies that directly access the peripheral (or fall into sleep mode) via 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.

4.4 Device-cloud Integration Agent

Besides the building blocks we described above, we also implemented the device-cloud integration agent who communicates with the cloud server periodically to check if there is a WebAssembly module to load.

Because the resource-constrained devices lack multi-thread support, the agent is implemented as an interrupt service routine (ISR) and invoked by a pre-set timer. The communication period is set to

```
case I32Add: // Generate the assembly for Wasm 32-bit integer add
{
    emit_x_POP_32bit(R22); // Pop operands from the Wasm's stack
    emit_x_POP_32bit(R18);
    emit_ADD(R22, R18); // Perform the 32-bit add
    emit_ADC(R23, R19); // ADD means "add without carry"
    emit_ADC(R24, R20); // ADC means "add with carry"
    emit_ADC(R25, R21);
    emit_x_PUSH_32bit(R22); // Push back the result to the stack
    ts.stack_top--;
    break;
}
```

Figure 5: Code snippet of generating the native assembly for WebAssembly instruction I32.Add.

```
#define OPCODE_ADC 0x1C00 // Opcode of ADC on ATMega128
#define emit_ADC(dest, src) emit_op2arg(OPCODE_ADC, dest, src)
// Emit (write) instructions to the flash
void emit(u16 opcode) {
    emit_raw_word(opcode);
}
// Generate the binary format of the instructions with 2 args
// May vary across different platforms
void emit_op2arg(uint16_t opcode, uint8_t dest, uint8_t src) {
    return emit(((opcode) + ((dest) << 4) + srcReg2Bin(src)));
}
```

Figure 6: Code snippet of the conceptual assemblies and the emit functions.

five minutes empirically considering the tradeoff between reprogramming timeliness and battery lifetime of the device because we notice software updates occur only once per day on average in an actively developed in-the-wild deployment of IoT devices [14, 37].

5 IMPLEMENTATION

In this section, we introduce the implementation details of WAIT.

Implementation details of WAIT. We have implemented WAIT for the Atmel ATMega128 micro-controller with 4KB RAM and 128KB Flash, which is widely used in sensing platforms [15, 43]. The implementation of WAIT includes over 5000 lines of C code. The core code occupies 27KB of the 128KB program memory.

Platform portability of WAIT.

During the implementation of WAIT, we take the porting simplicity into full consideration and try our best to ease porting overhead. We take the implementation of generating native assembly for WebAssembly instruction I32Add as an example. The code snippet of the generation is shown in Figure 5. We abstract the platform-dependent native assemblies by *conceptual assemblies* using C macros in the implementation of each bytecode. Functions beginning with emit_ prefix are the macros of conceptual assemblies. We select these conceptual assemblies by investigating the common assembly instructions across multiple ISAs (e.g., AVR, MSP). For example, the definition of emit_ADC and the related functions are shown in Figure 6. With these conceptual assemblies, to port WAIT to a new platform (e.g., MSP), developers should (1) Modify the OPCODE_ADC according to MSP's instruction datasheet [47]. (2) Modify the function that generates the binary format of the instructions (emit_op2arg() in Figure 6). (3) Because of the Ahead-of-Time compilation mechanism, WAIT puts a constraint on the platform being ported to that it needs to support executing code from the writable region. Hence, the developer should also declare the memory layout for the new platform in the linking script of WAIT that is used for placing the .wait section. If the target platform leverages a different op-code width (e.g., 64-bit), developers are required to re-construct the conceptual assemblies to cope with the width.

The conceptual assemblies act as a bridge between our implementation and various platforms, which could reduce the porting overhead of WAIT.

6 EVALUATION

In this section, we test WAIT to answer the following questions:

- How do the lightweight methods of WAIT reduce the RAM usage on resource-constrained IoT devices?
- What is the result after we apply the overhead mitigation approaches that WAIT designs for the sandbox checks?
- What is the performance of WAIT's energy optimizations?

6.1 Methodology

Benchmarks. To make our experiment closer to reality, we use six comprehensive benchmarks from real-world IoT applications to evaluate the performance of WAIT. The benchmarks are:

- *BSrch.* It uses binary search to obtain certain sensing data.
- *FFT.* It performs Fast Fourier Transform on input samples, which is a CPU-intensive application.
- *LEC.* It performs lossless data compression on sensor data [39].
- *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.
- *HeatC.* It uses an 8x8 heat sensor to track an object. It is divided into a calibration and a detection stage [43]. This benchmark performs the calibration stage.
- *HeatD.* This benchmark is the detection stage of the object tracking application using 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 WAIT achieves lightweight, efficient and power-saving execution of WebAssembly:

- *WAMR [9]:* the state-of-the-art WebAssembly AOT runtime on Linux-compatible devices with minimum memory footprint among those listed in Table 1.
- *Wasm3 [50]:* the most commonly used WebAssembly interpreter on resource-constrained IoT devices.
- *Native:* to directly execute the application compiled from C language.

Furthermore, we also include the comparison against two software-based safety protection approaches, t-kernel [19] and Harbor [30], to depict the run-time overhead introduced by the sandbox checks of WAIT.

6.2 Runtime memory footprint

In order to evaluate the performance of our lightweight methods described in Section 4.1, we measure the RAM usage of the benchmarks with different WebAssembly runtimes. Because WAIT is the

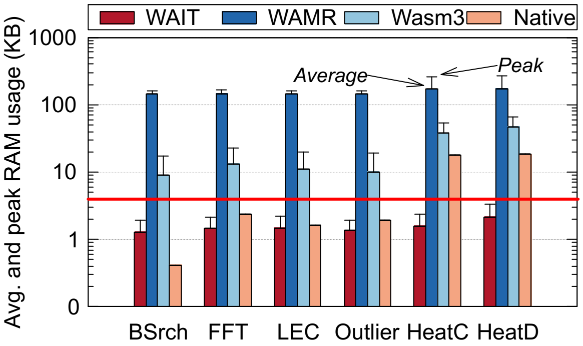


Figure 7: Memory footprint comparison with baselines. The red line indicates the RAM upper bound.

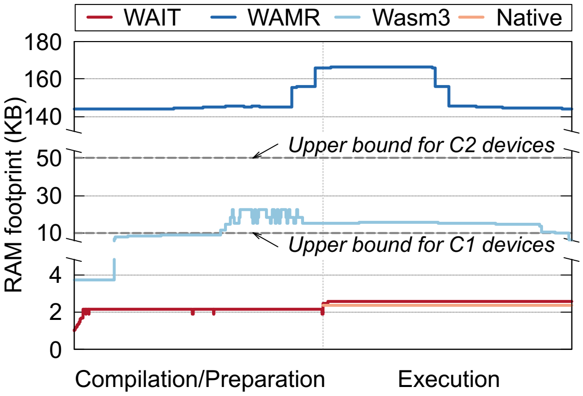


Figure 8: RAM footprint timeline of FFT benchmark throughout the compilation and execution stage.

only WebAssembly AOT runtime that could execute on resource-constrained IoT devices as far as we know, we measure the RAM usage of WAMR and Wasm3 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.

Figure 7 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 WAIT could achieve successful execution of all the benchmarks on this MCU. Even using native execution, *HeatC* and *HeatD* is not executable because they have a large amount of constant data which exceeds the RAM limit. The constant remapping moves the constants to the Flash and makes them executable. The peak RAM consumption of WAMR and Wasm3 are 84.9× and 13.6× of WAIT's, respectively. To be more specific, we have the following three observations.

(1) Compared with the baselines, WAIT 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 [25] 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 Figure 8 with the memory upper bound of C1 and C2 devices. We can see that WAIT is the only WebAssembly runtime to execute on C0, C1 and C2 devices. Although both are AOT runtimes, WAMR uses more memory than WAIT because it uses the heavy-weight LLVM as the compilation backend, which even exceeds the upper bound for C2 devices. Wasm3 is only suitable to execute on C2 devices, but its run-time efficiency is lower than the AOT runtimes because its interpreted execution introduces higher overhead than directly translating the bytecode into assemblies.

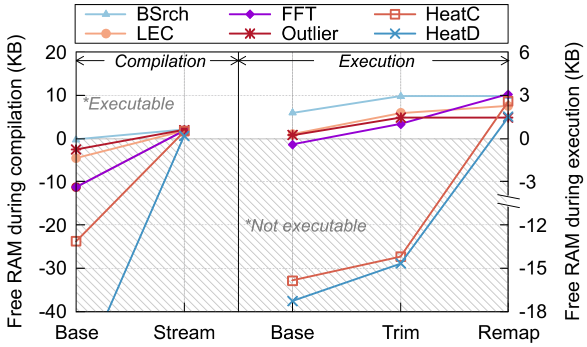


Figure 9: Breakdown of the RAM gain of each lightweight method of WAIT.

(2) The second observation is each of the three lightweight methods of WAIT 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 Figure 9. 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 Figure 8 that compared with the native approach, WAIT introduces 254 bytes run-time RAM overhead. This extra RAM usage is introduced by the size of `.wait` segment, which saves the necessary data for the execution of WAIT. We consider this overhead is acceptable, and the overhead can be counteracted by our constant remapping technique.

6.3 Sandbox check optimization

We now move on to the second question: how do our overhead mitigation approaches perform in our sandbox checks?

To be more specific, WAIT optimizes the overhead of sandbox checks by:

- Moving some of the execution-time checks to compile-time, i.e., EC-2, EC-3 in Table 2.
- Only examining the instructions that will grow the stack instead of all the instructions when checking the potential stack overflow.

Hence, towards answering 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 WAIT (i.e., canceling all the checks in Table 2) in Figure 10.

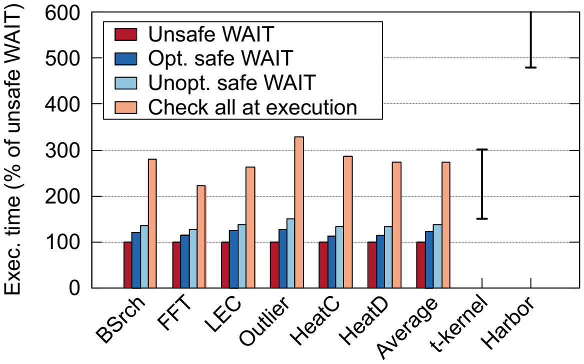


Figure 10: Overhead reduction brought by the optimizations for sandbox checks of WAIT.

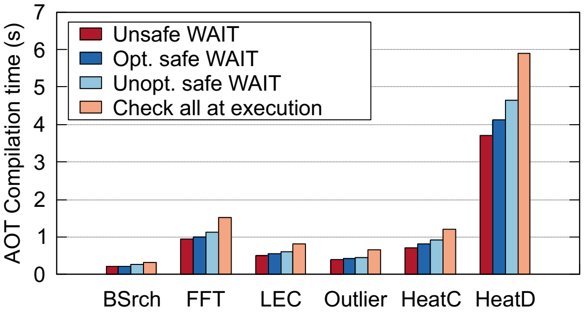


Figure 11: Compilation time overhead with and without the optimizations for sandbox checks of WAIT.

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, WAIT 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 Figure 11. 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 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.

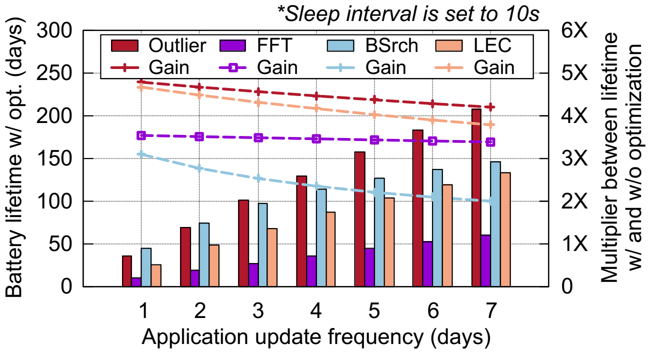


Figure 12: Node lifetime variation and lifetime gain with energy optimization of WAIT against update frequency.

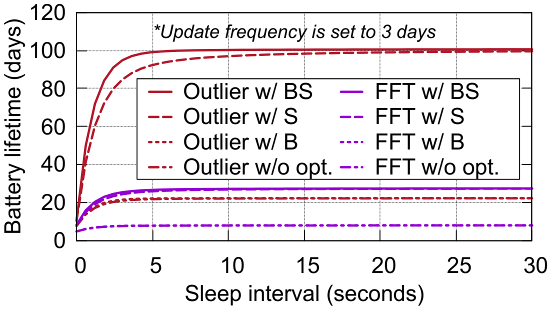


Figure 13: Node lifetime against the sleep interval. We use B for bulk writing optimization and S for I/O direct accessing optimization.

6.4 Energy reduction of WAIT

Now we reach the last question: how much energy could be saved by the optimizations of WAIT? In order to give an intuitive illustration of our energy optimization, we build a battery lifetime model for ATMega128 MCU based IoT prototyping board, including the MCU, networking and other peripherals, proposed in [14]. We evaluate the energy saving against different WebAssembly module update frequencies and sleep intervals of the IoT device. Upon each update, the IoT device receives a new WebAssembly module via the network, then WAIT compiles the module, and starts the module for data collection and processing. Our energy model takes all the above procedures into consideration, and we use a widely-used 2200mAh NiMH battery as the power supply.

We first show how the device lifetime varies and the multiplier between lifetime with and without our energy optimization against the update frequency in Figure 12. Our energy optimization approaches reduce overall energy use and result in a battery lifetime increase of 1.9 to 4.9×. 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× longer lifetime when the update interval is 1 day and 1.9× when updates every seven days for *BSrch* benchmark. This is because the bulk writing technique of WAIT could reduce the compile-time energy consumption for writing the Flash to a large extent, which finally turns out to be larger benefits when the module is more frequently updated.

Table 3: Effect 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]

[1] N/E: Not executable due to its RAM requirement exceeds the available RAM.

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

The IoT applications are executed intermittently with duty cycles. Hence, to further characterize the energy gain brought by the two energy optimizations of WAIT, we illustrate the device lifetime variation with respect to each optimization against sleep interval between two executions in Figure 13. We use two benchmarks as representatives: the *Outlier* is an I/O-intensive benchmark that reads the peripherals massively while *FFT* is a computation-intensive task. We can see from the figure that the I/O direct accessing optimization achieves better performance when the sleep interval is small. Moreover, this optimization is more obvious for the I/O intensive task, which is also reasonable because the I/O intensive tasks trigger more times of our optimization than the computation-intensive ones.

6.5 Overhead of WAIT

The run-time overhead of WAIT mainly comes with three aspects: the RAM usage of WAIT runtime (discussed in Section 6.2), the extra cycles brought by sandbox checks (discussed in Section 6.3), and the time used for accessing remapped constants. Hence, we evaluate the overhead of constant remapping and illustrate the results in Table 3.

The most noticeable benefit of constant remapping is that the *HeatC* is executable only with this optimization. With remapping, WAIT moves constants of *HeatC* (16,798 bytes) to Flash during AOT compilation to facilitate its execution. This technique could also be enabled for executable applications (e.g., *FFT* and *LEC* benchmark in Table 3). 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*).

7 DISCUSSION

In this section, we discuss several open issues in the design of WAIT.

7.1 Usage scenarios of WAIT

WAIT facilitates a “seamless” device-cloud integrated IoT application with the aid of WebAssembly, which means the computation module could be transmitted among the devices without any modification. To be more specific, WAIT is especially beneficial to the following two types of device-cloud integrated applications.

(1) The application involves devices of heterogeneous ISA. This situation is common both in cloud-device integrated applications (the server uses x86 and IoT uses AVR or ARM) such as smart healthcare applications (e.g., multi-device collaborated seizure onset detection [41]), and in the collaborative applications only with IoT devices (IoT uses ARM32, ARM64, Xtensa, etc.) such as federated learning with IoT devices [42].

For example, WAIT is beneficial to the city-scale surveillance applications [24], which need to deploy different computation stages on a hierarchy of computing devices – cameras, private clusters and public clouds and the application to be deployed changes over time.

(2) Developers of the application are from diverse programming language backgrounds. For example, the prevalent Function-as-a-Service (FaaS) computing schema decomposes an application into many functions and each function could be implemented using various languages. Existing works use Docker as the underlying runtime to execute the functions, which is not portable among heterogeneous devices and too heavyweight for IoT devices. By virtue of WebAssembly's multi-language support, WAIT could facilitate FaaS on a vast range of devices, even on resource-constrained IoT devices, by deploying functions as WebAssembly modules.

7.2 Design alternatives of WAIT

We now discuss the alternatives we considered during the design and implementation of WAIT, and why we adopted the current design.

LLVM or build from scratch? Existing AOT runtimes of WebAssembly [9, 51] commonly use LLVM [38] to perform the translation from WebAssembly to native assembly. By virtue of LLVM's modularity and reusability, these runtimes could be ported to different hardware platforms with relatively low effort. Nevertheless, the usage of LLVM incurs a significant footprint, especially for the AOT stage before execution, as shown in Section 6.2. Hence, WAIT chooses to build a resource-friendly WebAssembly AOT runtime from scratch. LLVM is a good choice if the computation resources are abundant, while WAIT is surely a better choice for IoT devices which are usually resource-constrained. As we described in Section 5, we consider the portability of WAIT all through the design. We also consider the automatic approach that porting WAIT to new platforms as our future work (e.g., similar to [23]) to remove the human-in-the-loop and increase the portability of WAIT.

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 WAIT's on-device AOT compilation. Performing AOT compilation on the cloud could reduce the run-time 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 WAIT'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 [3, 5], mobile agent computing [10, 40], and personal area network (PAN) [17, 46]. We use the 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 WAIT'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. (2) Applications with bi-directional security requirements, i.e., the application does not trust the computing device and vice versa. The smart contract application [22, 27, 52] is the most suitable case. The sandboxed execution facilitated by WebAssembly and WAIT is necessary for these applications, and compiling to native code on the cloud enlarges the attack surface.

7.3 Tradeoff of the lightweight design of WAIT

As we describe in Section 4, WAIT adopts several lightweight approaches to run on resource-constrained devices. Some of the high-end features of WebAssembly are not included in WAIT for this lightweight design.

We categorize the unsupported features by their distance to the core WebAssembly specification. (1) *The minimal viable product (MVP) features.* The MVP features form the initial release of WebAssembly, including the binary format, linear memory, etc. WAIT implements all the MVP features of WebAssembly and adds the IoT-related APIs for developers. (2) *The post-MVP features.* Based on MVP, WebAssembly continues its evolution and the newly added features are post-MVP features. Among the post-MVP features, WAIT currently does not support the threading, 128-bit SIMD (single-instruction multiple-data), and 64-bit addressing. We argue that these high-end features are barely used in IoT applications or are even not supported by IoT devices. (3) *Extra features that are not included in the specification or those are still in the proposal stage.* These features are not yet standardized, and may vary in the future, e.g., GPU-accelerator support. Hence, we also omit these high-end features in WAIT for lightweight.

It is worth noting that these features are not implemented by choice, and WAIT can be extended to support the high-end features. For example, to implement the 128-bit SIMD post-MVP feature, we could add a new entry for the SIMD instruction of WebAssembly (e.g., the `I8x16.MUL` instruction for simultaneously multiplying 16 8-bit integers) similar to Figure 6, and implement the translation with native SIMD assemblies. In the short term, our lightweight design could enable the first step in bringing WebAssembly to existing IoT devices and accelerate the adoption of WebAssembly in the IoT domain. Furthermore, in the long-term future, WAIT could evolve with advances in IoT hardware, help developers to build hardware-independent applications, and facilitate the Web of Things vision [49].

8 RELATED WORK

In this section, we summarize the efforts towards device-cloud integration, the attempts to bring WebAssembly outside the browser, and the safety guarantees for IoT devices.

Systems for device-cloud integrated applications. Many systems have been proposed for supporting the device-cloud integrated applications. These approaches could be classified to three categories: VM-based [44, 48], native-based [20, 31], and CLR-based [11, 26, 29, 33] solutions by the underlying technologies they use.

The VM-based approaches [44, 48] share the application logic between different platforms by migrating the whole virtual machine image. These approaches require too many resources which are not available on IoT devices.

The most recent native-based methods are EdgeProg [31] and TinyLink 2.0 [20]. These works propose domain specific languages (DSL) to enable users to express the application logic across multiple hardware devices in one piece of code. Taking the applications written in DSLs as input, EdgeProg and TinyLink 2.0 separate the code and generate the executables for each device. The native-based method is hard to support fine-grained program state migration due to the different memory layouts between architectures and each partition update at runtime needs binary re-generation.

Considering the drawbacks of the above two categories, WAIT chooses the CLR-based solution to achieve device-cloud integration on resource-constrained IoT devices. The existing approaches leverage common language runtimes to achieve seamless migration. For example, MAUI [11] uses C#, ThinkAir [29] and COMET [18] uses Java, and EveryLite [36] uses JavaScript. Among the existing CLR-based approaches, the most related one is WiProg [33], which uses WebAssembly as the underlying migration technology. WiProg focuses on how to provide offloading annotations to label the placement of each method and how to efficiently migrate the WebAssembly module among devices. Orthogonal to WiProg, WAIT focuses on how to embrace the resource-constrained devices to the WebAssembly-based device-cloud integration.

Approaches towards safe execution on IoT devices. Ensuring the safe execution of applications is one of the vital characteristics of an IoT system.

Some efforts to protect the safe execution on IoT devices are through software-based approaches. For example, Gu et al. propose t-kernel [19], which is a safe kernel for IoT devices. The t-kernel advocates a software-based paging mechanism for IoT devices to keep memory safety. Harbor [30] employs a memory map to record the ownership and layout information of memory regions and implements a safe stack that stores return addresses in a protected memory region to ensure the control-flow integrity. Moreover, CapeVM [43] leverages the metadata in Java bytecode to perform compile-time and run-time sandbox checks to ensure the safe execution of Java bytecode on IoT devices. Compared with these existing approaches, WAIT moves most of the checks to the AOT compiling process, which could reduce the overhead of safety checks.

9 CONCLUSION

In this paper, we propose WAIT, a WebAssembly AOT runtime on resource-constrained devices for device-cloud integrated applications. WAIT is the first work to facilitate the AOT compilation of WebAssembly on constrained devices by leveraging several lightweight methods. Moreover, WAIT achieves efficient sandbox execution by moving most of the run-time checks to compile-time. Furthermore, WAIT advocates a set of IoT-related APIs to enable the full IoT programming for WebAssembly and optimizes the runtime power consumption by bulk instruction writing and I/O direct accessing. Results show that WAIT reduces the RAM usage by 85× and the energy consumption by 1.2× to 4.9×.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. We sincerely appreciate the conscientious effort and insightful suggestions of our shepherd, Jeremy Andrus. This work is supported by NSFC under Grant No. 62072396, Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under Grant No. LR19F020001, the Fundamental Research Funds for the Central Universities (No. 226-2022-00087).

REFERENCES

[1] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. 2018. The signpost platform for city-scale sensing. In *Proc. of ACM/IEEE IPSN*.

[2] Arduino. 2022. *Forum*. Retrieved March, 23, 2022 from https://www.arduino.cc/reference/en/

[3] Hamidreza Bagheri, Md Noor-A-Rahim, Zilong Liu, Haeyoung Lee, Dirk Pesch, Klaus Moessner, and Pei Xiao. 2021. 5G NR-V2X: Toward connected and cooperative autonomous driving. *IEEE Communications Standards Magazine* 5, 1 (2021), 48–54.

[4] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, Martin Vetterli, Olivier Couach, and Marc Parlange. 2008. Sensorscope: Out-of-the-box environmental monitoring. In *Proc. of IEEE IPSN*.

[5] Lucas Bréhon-Grataloup, Rahim Kacimi, and André-Luc Beylot. 2022. Mobile edge computing for V2X architectures and applications: A survey. *Computer Networks* 206 (2022), 108797.

[6] Bytecode Alliance. 2022. *Cranelift Code Generator*. Retrieved March, 23, 2022 from https://github.com/bytecodealliance/wasmtime/tree/main/cranelift

[7] Bytecode Alliance. 2022. *Lucet*. Retrieved March, 23, 2022 from https://github.com/bytecodealliance/lucet

[8] Bytecode Alliance. 2022. *Wasmtime*. Retrieved March, 23, 2022 from https://github.com/bytecodealliance/wasmtime

[9] Bytecode Alliance. 2022. *WebAssembly Micro Runtime*. Retrieved March, 23, 2022 from https://github.com/bytecodealliance/wasm-micro-runtime

[10] Yu-Cheng Chou, David Ko, and Harry H Cheng. 2009. Mobile agent-based computational steering for distributed applications. *Concurrency and Computation: Practice and Experience* 21, 18 (2009), 2377–2399.

[11] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. In *Proc. of ACM MobiSys*.

[12] DFRobot. 2022. *Forum*. Retrieved March, 23, 2022 from https://www.dfrobot.com/forum

[13] Edsger W Dijkstra. 1968. Letters to the editor: go to statement considered harmful. *Commun. ACM* 11, 3 (1968), 147–148.

[14] Wei Dong, Chun Chen, Jiajun Bu, and Wen Liu. 2015. Optimizing relocatable code for efficient software update in networked embedded systems. *ACM Transactions on Sensor Networks (TOSN)* 11, 2 (2015), 22.

[15] Prabal Dutta, Mike Grimmer, Anish Arora, Steven Bibyk, and David Culler. 2005. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proc. of IEEE IPSN*.

[16] Yi Gao, Wei Dong, Kai Guo, Xue Liu, Yuan Chen, Xiaojin Liu, Jiajun Bu, and Chun Chen. 2016. Mosaic: A low-cost mobile sensing system for urban air quality monitoring. In *Proc. of IEEE INFOCOM*.

[17] Yi Gao, Siyu Zeng, Ji Zhao, Wenxin Liu, and Wei Dong. 2022. AirText: One-Handed Text Entry in the Air for COTS Smartwatches. *IEEE Transactions on Mobile Computing* (2022).

[18] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. 2012. {COMET}: Code Offload by Migrating Execution Transparently. In *Proc. of USENIX OSDI*.

[19] Lin Gu and John A Stankovic. 2006. t-kernel: Providing reliable OS support to wireless sensor networks. In *Proc. of ACM SenSys*.

[20] Gaoyang Guan, Borui Li, Yi Gao, Yuxuan Zhang, Jiajun Bu, and Wei Dong. 2020. TinyLink 2.0: integrating device, cloud, and client development for IoT applications. In *Proc. of ACM MobiCom*.

[21] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proc. of ACM PLDI*.

[22] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. {EOSAFE}: Security Analysis of {EOSIO} Smart Contracts. In *Proc. of USENIX Security*. 1271–1288.

[23] Luke Hsiao, Sen Wu, Nicholas Chiang, Christopher Ré, and Philip Levis. 2020. Creating hardware component knowledge bases with training data generation and multi-task learning. *ACM TECS* (2020).

[24] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. 2018. Videoedge: Processing camera streams using hierarchical clusters. In *Proc. of ACM/IEEE SEC*.

[25] IETF. 2022. *RFC7228: Terminology for Constrained-Node Networks*. Retrieved March, 23, 2022 from https://www.rfc-editor.org/rfc/rfc7228.html

[26] Hyuk-Jin Jeong, Chang Hyun Shin, Kwang Yong Shin, Hyeon-Jae Lee, and Soo-Mook Moon. 2019. Seamless Offloading of Web App Computations From Mobile Device to Edge Clouds via HTML5 Web Worker Migration. In *Proc. of ACM SoCC*.

[27] Bo Jiang, Yifei Chen, Dong Wang, Imran Ashraf, and WK Chan. 2021. WANA: Symbolic Execution of Wasm Bytecode for Extensible Smart Contract Vulnerability Detection. In *Proc. of IEEE QRS*. IEEE, 926–937.

[28] Kanaka. 2022. *WebAssembly interpreter in C*. Retrieved March, 23, 2022 from https://github.com/kanaka/wac

[29] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proc. of IEEE INFOCOM*.

[30] Ram Kumar, Eddie Kohler, and Mani Srivastava. 2007. Harbor: software-based memory protection for sensor nodes. In *Proc. of ACM/IEEE IPSN*.

[31] Borui Li and Wei Dong. 2020. EdgeProg: Edge-centric Programming for IoT Applications. In *Proc. of IEEE ICDCS*.

[32] Borui Li, Wei Dong, Gaoyang Guan, Jiadong Zhang, Tao Gu, Jiajun Bu, and Yi Gao. 2021. Queec: QoE-aware Edge Computing for IoT Devices under Dynamic Workloads. In *ACM Transactions on Sensor Networks*.

[33] Borui Li, Wei Dong, and Gao Yi. 2021. WiProg: A WebAssembly-based Approach to Integrated IoT Programming. In *Proc. of IEEE INFOCOM*.

[34] Borui Li, Chenghao Tong, Yi Gao, and Wei Dong. 2021. S2: a Small Delta and Small Memory Differencing Algorithm for Reprogramming Resource-constrained IoT Devices. In *Proc. of IEEE INFOCOM Demos and Posters*.

[35] Huan Li, Dong Liang, Lihui Xie, Gong Zhang, and Krithi Ramamritham. 2014. Flash-optimized temporal indexing for time-series data storage on sensor platforms. *ACM Transactions on Sensor Networks (TOSN)* 10, 4 (2014), 1–30.

[36] Zhenying Li, Xiaohui Peng, Lu Chao, and Zhiwei Xu. 2018. EveryLite: A lightweight scripting language for micro tasks in IoT systems. In *Proc. of IEEE/ACM SEC*.

[37] Yunhao Liu, Yuan He, Mo Li, Jiliang Wang, Kebin Liu, Lufeng Mo, Wei Dong, Zheng Yang, Min Xi, Jizhong Zhao, et al. 2011. Does wireless sensor network scale? A measurement study on GreenOrbs. In *Proc. of IEEE INFOCOM*.

[38] llvm-admin team. 2022. *The LLVM Compiler Infrastructure*. Retrieved March, 23, 2022 from https://llvm.org/

[39] Francesco Marcelloni and Massimo Vecchio. 2009. An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks. *the computer journal* 52, 8 (2009), 969–987.

[40] Stephen S Nestinger, Bo Chen, and Harry H Cheng. 2009. A mobile agent-based framework for flexible automation systems. *IEEE/Asme Transactions on Mechatronics* 15, 6 (2009), 942–951.

[41] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. 2009. Wishbone: Profile-based Partitioning for Sensornet Applications.. In *Proc. of USENIX NSDI*.

[42] Dinh C Nguyen, Ming Ding, Pubudu N Pathirana, Aruna Seneviratne, Jun Li, and H Vincent Poor. 2021. Federated learning for internet of things: A comprehensive survey. *IEEE Communications Surveys & Tutorials* (2021).

[43] Niels Reijers and Chi-Sheng Shih. 2018. CapeVM: A Safe and Fast Virtual Machine for Resource-Constrained Internet-of-Things Devices. In *Proc. of ACM SenSys*.

[44] Bin Shi and Haiying Shen. 2019. Memory/disk operation aware lightweight vm live migration across data-centers with low performance impact. In *Proc. of IEEE INFOCOM*.

[45] Xingzhe Song, Boyuan Yang, Ge Yang, Ruirong Chen, Erick Forno, Wei Chen, and Wei Gao. 2020. SpiroSonic: monitoring human lung function via acoustic sensing on commodity smartphones. In *Proc. of ACM MobiCom*.

[46] Milan Stute, David Kreitschmann, and Matthias Hollick. 2018. One billion apples' secret sauce: Recipe for the apple wireless direct link Ad hoc protocol. In *Proc. of ACM MobiCom*. 529–543.

[47] MSPGCC team. 2022. *MSP Instruction Set*. Retrieved May, 4, 2022 from http://mspgcc.sourceforge.net/manual/x223.html

[48] Konstantinos Tsakalozos, Vasilis Verroios, Mema Roussopoulos, and Alex Delis. 2017. Live VM migration under time-constraints in share-nothing IaaS-clouds. *IEEE Transactions on Parallel and Distributed Systems* 28, 8 (2017), 2285–2298.

[49] W3C. 2022. *Web of Things*. Retrieved May, 4, 2022 from https://www.w3.org/WoT/

[50] Wasm3 Labs. 2022. *Wasm3*. Retrieved March, 23, 2022 from https://github.com/wasm3/wasm3

[51] Wasmer.io. 2022. *Wasmer*. Retrieved March, 23, 2022 from https://github.com/wasmerio/wasmer

[52] Zhiqiang Yang, Han Liu, Yue Li, Huixuan Zheng, Lei Wang, and Bangdao Chen. 2020. Seraph: enabling cross-platform security analysis for evm and wasm smart contracts. In *Proc. of IEEE/ACM ICSE Companion*. IEEE, 21–24.

[53] Hanbin Zhang, Gabriel Guo, Chen Song, Chenhan Xu, Kevin Cheung, Jasleen Alexis, Huining Li, Dongmei Li, Kun Wang, and Wenyao Xu. 2020. Pdlens: smartphone knows drug effectiveness among parkinson's via daily-life activity fusion. In *Proc. of ACM MobiCom*.