Event-Based Out-of-Place Debugging

Tom Lauwaerts
Universiteit Gent
Gent, Belgium
Tom.Lauwaerts@UGent.be

Carlos Rojas Castillo
Vrije Universiteit Brussel
Brussels, Belgium
crojcas@vub.be

Robbert Gurdeep Singh
Universiteit Gent
Gent, Belgium
Robbert.GurdeepSingh@UGent.be

Matteo Marra
Vrije Universiteit Brussel
Brussels, Belgium
mmarra@vub.be

Christophe Scholliers
Universiteit Gent
Gent, Belgium
Christophe.Scholliers@UGent.be

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussels, Belgium
egonzale@vub.be

ABSTRACT

Debugging IoT applications is challenging due to the hardware constraints of IoT devices, making advanced techniques like record-replay debugging impractical. As a result, programmers often rely on manual resets or inefficient and time-consuming debugging techniques such as printf. Although simulators can help in that regard, their applicability is limited because they fall short of accurately simulating and reproducing the runtime conditions where bugs appear. In this work, we explore a novel debugging approach called *event-based out-of-place debugging* in which developers can capture a remotely running program and debug it locally on a (more powerful) machine. Our approach thus provides rich debugging features (e.g., step-back) that normally would not run on the hardware restricted devices. Two different strategies are offered to deal with resources which cannot be easily transferred (e.g., sensors): pull-based (akin to remote debugging), or push-based (where data updates are pushed to developer's machine during the debug session). We present EDWARD, an event-based out-of-place debugger prototype, implemented by extending the WARDuino WebAssembly microcontroller Virtual Machine, that has been integrated into Visual Studio Code. To validate our approach, we show how our debugger helps uncover IoT bugs representative of real-world applications through several use-case applications. Initial benchmarks show that event-based out-of-place debugging can drastically reduce debugging latency.

CCS CONCEPTS

• **Computer systems organization** → *Embedded software*; • **Software and its engineering** → **Software testing and debugging**; *Integrated and visual development environments.*

KEYWORDS

Out-of-place debugging, Debugger, Internet-of-Things, WebAssembly, WARDuino, Virtual Machine

MPLR '22, September 14–15, 2022, Brussels, Belgium

ACM ISBN 978-1-4503-9696-7/22/09…$15.00
https://doi.org/10.1145/3546918.3546920

ACM Reference Format:
Tom Lauwaerts, Carlos Rojas Castillo, Robbert Gurdeep Singh, Matteo Marra, Christophe Scholliers, and Elisa Gonzalez Boix. 2022. Event-Based Out-of-Place Debugging. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22), September 14–15, 2022, Brussels, Belgium.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3546918.3546920

1 INTRODUCTION

Despite hardware advances enabling computation and analytics on IoT devices, debugging IoT applications is difficult due to the limited memory and computational power of these devices. A 2021 survey by Makhshari and Mesbah [17] of 194 IoT developers reported that *63%* attributed the challenges of developing IoT systems to the resource constraints of the devices. The same study collected 5,565 bugs reports from 91 IoT projects and reported that the most frequent types of bugs are related to software development (48%), device issues (29%), and communication issues (19%).

Debugging IoT systems during development is often aided by simulators. The differences between the simulator and the actual hardware, however, makes finding device issues extremely difficult. Moreover, due to the heterogeneity of IoT devices, it is impossible to simulate all types of devices [17].

Another common debugging approach is to log the execution on the real device. As the hardware is not simulated, this technique can capture device-related issues [17]. Debugging tools help with this task by enabling programmers to run and monitor the program's execution [21]. For example, record and replay debugging *automatically* records events of a program's execution in a log or trace, which can later be used to replay the program. This facilitates reproducing bugs since a recorded execution that activated a bug can be deterministically replayed. On the other hand, back-in-time debugging (aka reverse or omniscient debugging) records the application state in a snapshot, enabling the backward inspection of the program state by restoring a previously recorded snapshot [8].

These *offline debugging* techniques typically have a high memory and computational overhead [23], making them impractical for use on IoT devices. As a result, developers turn to *manual* logging solutions such as serial printf [17]. Despite its popularity, this solution relies on developers wisely choosing what to log: capturing too little may not provide enough contextual information to find the bug, while capturing too much adds unhelpful noise to the analysis [24].

Alternatively, *online debuggers* let the developers control program execution, inspect and manipulate execution state without resorting to logs [21]. Modern runtime environments for IoT devices provide some limited remote online debugging support [9, 11]. Crucially, remote debuggers suffer from debugging latency. Each debugging operation causes communication with the device, potentially consuming essential resources [1]. To alleviate the debugging latency problem, we proposed out-of-place debugging in previous work [19]. With an out-of-place debugger, the application state is captured when an exception or breakpoint is hit. The state is subsequently transferred to the developer's machine, and restored into a new debug session. Developers can now debug the application locally, reducing the debugging latency and communication load.

In this work, we introduce a variant of out-of-place debugging for IoT systems, where resource constrained devices run applications driven by events (from sensors, button presses, network messages, etc.). To this end, we propose *event-based out-of-place debugging* in which developers can debug IoT applications externally. They may, for example, use a more powerful desktop machine while still using the actual hardware for obtaining sensor values and communicating with other IoT devices.

We introduce three main innovations with respect to the original out-of-place debugging specially designed for the IoT environment. First, event-based out-of-place debugging offers developers fine-grained access to non-transferable resources, i.e., resources that cannot be easily transferred (e.g., sensors). During the local debug session values of resources can either (1) be queried when needed, leading to a *pull-based* access strategy akin to classic remote debugging, or (2) be pushed to developer's machine from the remote device when there is an update, leading to what we call a *push-based* access strategy. Second, event-based out-of-place debugging permits advanced debugging features to be brought to IoT, such as stepping back commands. Finally, event-based out-of-place debugging allows a debug session to be started on-demand, where previous implementations initiated a session only when an error occurs.

We implement an event-based out-of-place prototype debugger called EDWARD. The debugger backend has been implemented by modifying an existing Virtual Machine (VM) meant for resource-constrained devices. More concretely, we extend WARDuino [13], a microcontroller VM for WebAssembly [14], with event-based out-of-place debugging support. EDWARD's frontend is integrated into WARDuino's Visual Studio Code (VSC) plugin [27].

To validate our approach, we perform a qualitative evaluation that assesses the ability of our debugger to help solve the most frequent types of bugs in IoT systems: device-related bugs and software development issues [17]. We conduct a preliminary quantitative analysis of network latency, which shows that out-of-place debugging can improve performance over remote debugging by reducing network access.

The remainder of this paper is organized as follows: Section 2 introduces the challenges in debugging IoT applications. Section 3 details the main ingredients of event-based out-of-place debugging. Then, Section 4 presents EDWARD, including both the changes to WARDuino and the debugger frontend in VSC. In Section 5, we qualitatively and quantitatively evaluate our debugging technique

```
1  import * as wd from warduino;
2
3  const SENSOR: u32 = 36;
4
5  function readTemperature(): f32 {
6    const adc_value: u32 = wd.analogRead(SENSOR);
7    const voltage: f32 = adc_value * (3.3 / 1024.0);
8    return voltage / 10;}
9
10 function readAndPrintTemperature() : void  {
11   wd.println(`Temperature: ${readTemperature()} C.`);}
12
13 export function main() : void {
14   while(true) {
15     readAndPrintTemperature();
16     wd.delay(250);}}
```

Figure 1: LM35 temperature sensor example written in AssemblyScript.

followed by a discussion of related work in Section 6. Section 7 concludes the paper.

2 DEBUGGING IOT APPLICATIONS

In this section, we start by giving a motivating example to frame the kind of applications we are targeting and motivate why it is difficult to debug such applications using current state-of-the-art debuggers. Subsequently, we identify problems with the current debugging tool support, which forms the motivation for this work.

2.1 Motivating Example

Consider a simple smart monitoring system using an IoT device that reads a temperature sensor. The device reads out a voltage from the temperature sensor, which is then converted into a numeric value shown to the user.

We assume that IoT applications are written in a high-level language that compiles to WebAssembly (e.g., Swift, Rust, AssemblyScript, etc.). Figure 1 shows an implementation of this program written in AssemblyScript. Line 1 imports the `warduino` library (bound to `wd`) to be able to access the peripherals of the IoT device. Line 3 defines a constant `SENSOR` to indicate from which pin the voltage should be read. Subsequently, the program consists of three functions: `main`, `readTemperature`, and `readAnd-PrintTemperature`. The `main` function serves as the entry point of the application, and starts an infinite loop that repeatedly reads and prints the temperature approximately every 250 milliseconds. To this end, it calls the `readAndPrintTemperature` helper function that reads the temperature and prints it to the console. Finally, the `readTemperature` function uses the `wd` module to read the voltage level of the pin and subsequently converts the voltage into a temperature value.

2.2 Debugging the Motivating Example

During the development of this simple application, developers can be confronted with bugs of the two most frequent categories [17]: device-related issues (e.g., a pin not working correctly) or software development-related issues (e.g., the wrong pin is read, the calculation to convert the voltage to a temperature could be wrongly implemented, etc.). Consider a running IoT application displaying

the values read from a temperature sensor. Sometimes, the displayed values are wrong, e.g., not-a-number is displayed. We now describe the current state-of-the-art techniques for debugging such an issue, and we will identify their shortcomings.

Testing and Simulation. IoT developers could use testing frameworks and simulation solutions to identify bugs [6, 15]. These tools are very important to reduce the cost of developing the application because testing on hardware is typically orders of magnitude slower than testing on the local machine. It is only after the developer is sufficiently confident that their code is correct, that they will test the application on the actual hardware.

In our example scenario, consider that there exists a simulator tool with a hardware profile for the IoT device where we deploy the application, i.e., an ESP32 microcontroller. Even if the unit-testing and simulations of the code show that the code is working as expected, the temperature values could still be wrongly displayed when running the application on the actual device. To investigate that mismatch, developers would then turn to a debugger.

Debugging. Modern runtime environments for IoT devices, such as WARDuino [13], Espruino [9], provide online debugging support allowing developers to execute the program in debug mode on their machine. This boils down to some form of *remote debugging*, which allows developers to perform step-wise execution of the program running on the IoT device from their machine. Every debugging operation results in network communication to the IoT device (e.g., for stepping commands, state inspection, etc).

Whenever the program needs to access a physical resource (e.g., a temperature sensor), the debugger requests these values from the device. In our example, whenever the `analogRead` function is called the debugger queries the microcontroller to read out the latest sensor value. When debugging the program with the remote debugger, it becomes clear that the analog pin values fluctuate heavily as if the pin is not connected. Inspection of the circuit shows that the SENSOR pin value of the board is different from the one used for simulation. This is a simple example of a *device-related issue that can only be found by debugging live on the device.*

Limitations. In this simple application, it was enough to query the device through the debugger whenever a certain sensor value is required. However, remote debugging may not be suitable when network usage must remain limited. Moreover, such a *pull-based* strategy to access sensor data does not capture all use cases for debugging IoT devices. Many of the peripheral devices attached to a microcontroller use an interrupt-driven interface instead. Such interrupts are generated when certain external events happen, for example when an input-pin changes from low to high.

As a concrete example, consider we modify code in fig. 1 to use an interrupt to measure the temperature whenever a button is pressed. Figure 2 shows such an event-driven implementation. In this case, the developer needs a debugger that is able to intercept the interactions with the IoT device when a new event is generated (in this case a new temperature value is produced by pushing the button), not by querying the sensor value. Unfortunately, such an *push-based* strategy to access sensor data is not provided by debuggers for IoT devices. With current remote debugging techniques,

```
1 export function main() : void {
2   wd.interruptOn(BUTTON, wd.FALLING,
3                  readAndPrintTemperature);
4 }
```

Figure 2: Event-driven implementation for reading temperature sensor in AssemblyScript.

developers can not see an event being queued in the runtime or control the timing of those events.

Alternatively, developers could use offline debugging techniques like time-travelling and record and reply debugging [2, 4, 5, 8] to investigate the events that lead to the occurrence of a bug. Time-travelling debugging allows restoring a program execution to an earlier point in time. Record and replay, on the other hand, helps deal with the non-reproducibility of bugs due to nondeterministic interrupts and signals. This technique records relevant events that happen during a program's execution in a trace to replay the execution later. Once a program execution manifesting the bug is recorded, it can be reliably reproduced until the root cause is identified. While offline debugging techniques such as record and replay have become mature for modern commodity hardware [5], they have only been explored limitedly in IoT (e.g., [12]) because of the resource constraints of these devices. Improved support is thus needed to enable such advanced debugging techniques which would help developers deal with the nondeterminism inherent of IoT applications.

2.3 Problem Statement

From the analysis of the motivating example, we want to highlight the following problems which motivate the proposition of our novel debugging approach:

(1) Unit testing and simulation frameworks are not always sufficient to debug IoT applications
(2) Traditional remote debugging is often slow due to network latency and only offers a pull-based strategy to access device resources (e.g., sensors)
(3) No remote debugging approach provides mechanisms to control the timing and order of events (i.e., sensor updates, incoming MQTT [3] network messages, etc.) so characteristic of IoT applications
(4) Offline debugging techniques are too heavy-weighted for IoT devices.

3 EVENT-BASED OUT-OF-PLACE DEBUGGING

In this section we introduce our novel debugging technique for IoT applications inspired by the ideas of out-of-place debugging proposed in prior work [18, 19]. Out-of-place debugging is an online remote debugging approach that aims to mitigate network latency issues by making debugging a local activity. When a breakpoint is hit, or an exception occurs in an application, the execution context of the program is extracted, and transferred to a different machine where a debug session is created with a copy of the application. Debugging then becomes an in-place activity with reduced latency since any debug action is performed on the local application and thus no longer requires network communication. When accessing

resources that cannot be transferred (e.g. files, sensors, etc.), an out-of-place debugger behaves as a classic remote debugger querying the value of the non-transferable resource over the network.

In this work we propose *event-based out-of-place debugging*, which rethinks the concepts of out-of-place debugging to deal with the unique characteristics of IoT systems. Our technique differs from existing out-of-place debuggers [18, 19, 26] in three important aspects. First, creating a local debug session is now initiated by a user request rather than being triggered by an exception on the remote device. Such an on-demand debugging strategy is crucial to decrease the load on the remote devices. Secondly, and most importantly, the debugger offers both a pull-based and push-based strategy for accessing non-transferable resources. As argued in the previous section, both strategies are needed due to the event-driven nature of IoT applications. Finally, since the debug session happens locally, on a more powerful machine, advanced debugging features from online debugging that are too heavy-weighted for the IoT devices can now be offered. Developers can debug programs stepwise backwards by restoring previous states of the program (stored at each debug step).

While event-driven out-of-place debugging is a general technique that can be implemented for compiled languages, in this work, we implement it by extending a VM targeting microcontrollers. In what follows we give an overview of the VM requirements and the debugging architecture. We will then discuss each step in the debugging cycle chronologically, and present how the VM supports each step.

3.1 Virtual Machine Requirements

In order to implement event-based out-of-place debugging, any candidate VM must support the following:

(1) Halt and step through the execution of a program.
(2) Capture and serialize the state of the running program.
(3) Recreate and execute a captured state in another VM instance where the program can be debugged.
(4) Query the state of peripherals.
(5) Capture asynchronous events.
(6) Receive and process simulated events.

Halting and stepping through a program are the most elementary operations offered by classic online debuggers. Likewise, these two operations should be supported for the most minimal out-of-place debugger. To initiate a debug session that runs the target program on a different machine, a VM should support to capturing and serializing a running program's execution. Based on the serialized representation, the VM must be able to recreate the captured state and continue executing the program. In the IoT context, the VM must be able to query the state of the peripherals on a remote device. More generally, the debugger must be able to query non-transferable resources, which are only available on the other instance of the VM. The VM must be able to capture asynchronous events. In the context of IoT these will often be hardware interrupts, but other kinds of asynchronous events such as receiving a message over MQTT must be captured too. Finally, the VM must be able to receive notifications of such events and process simulated versions of them.

3.2 Debugging Architecture

We explain the main architecture of event-based out-of-place debugging through a high-level overview of the steps involved in a debugging cycle. Debugging with our approach consists of four major stages: *deployment*, *remote debugging*, *capturing an out-of-place debug session*, and *local debugging*. Within these stages, all steps will be performed by three major components, the *debugger frontend*, the *proxy debugger* and the *debugger backend* (detailed below).

Deployment. To debug an IoT application, the debugger frontend first deploys the application onto the target device. The debugger frontend is a process running on the developer's machine. It offers dedicated views to developers to debug an application. Additionally, it hosts a debugger manager, which interacts with the VM instance running on the IoT device. After deployment, the device acts as the *debugger backend*. More concretely, the debugger frontend establishes a communication channel (e.g., serial, Wi-Fi, etc.) with the debug monitor that is used for any interaction between frontend, and the VM instance at the remote device.

Remote Debugging. Once deployed, the developer can use the debugger as a traditional remote debugger where debugging operations (e.g., step, add breakpoint, etc.) are sent to the debugger backend running on the device. To reduce the network latency and benefit from advanced debugging features (e.g., control over events, stepping backwards, etc.), the developer can switch to out-of-place debugging by requesting the capture of a debug session via a dedicated view. These first two stages are discussed further in section 3.3.

Capturing an Out-of-Place Debug Session. When the debugger backend handles a request to capture an out-of-place debug session, it will pause the target program's execution and create a session by snapshotting the application. More concretely, the debug session contains all the state (as detailed in section 3.4) that is needed to make local debugging possible at another device. The debug session is then transferred to the developer's machine, and the debugger frontend starts a new debugger backend locally by restoring the captured debug session. Subsequently, the debugger frontend switches communication to the newly created backend, and the remote device will relinquish its role as backend and instead become a *proxy debugger*.

Local Debugging. After capturing, all debugging operations (e.g., step, add breakpoint, etc.) are sent to the local backend, but the VM on the remote device is kept active to provide access to non-transferable resources (e.g., sensors, network messages from neighboring devices, etc.). When the debugger backend executes code of the target application which accesses non-transferable resources, developers can choose between two strategies: (1) a pull-based access strategy will request the proxy debugger to send the latest state of the resource (e.g., temperature sensor), and (2) a push-based access strategy will request the proxy debugger to store all events produced by the resource (e.g., new temperature value) in an event queue and also notify the frontend of each received event. Details on this stage and these two strategies are discussed in section 3.5.

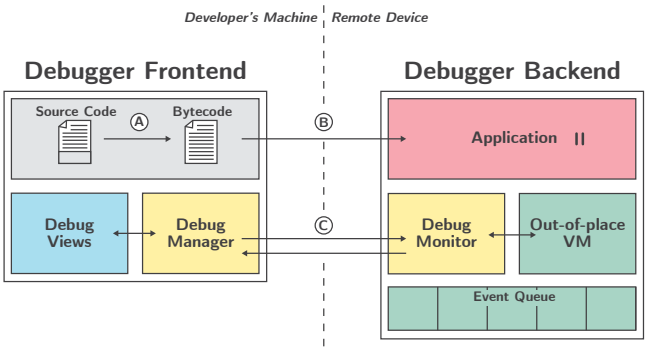

Figure 3: Stage 1 and 2: Deployment and Remote Debugging. The debugger frontend flashes a program (A) on the VM instance on the device (B) to enable remote debugging (C).

The latency of local debugging operations is much lower compared to remote debugging operations. Furthermore, local debugging enables the implementation of advanced debugging features since the local device has far less hardware restrictions than IoT devices. In this work, we provide developers with *step back* operations during an out-of-place debug session. More details are provided in section 3.6.

In what follows we detail the main features of event-based out-of-place debugging according to each stage.

3.3 Deployment and Remote Debugging

Figure 3 shows the interactions between the debugger frontend and backend in the first two stages, i.e., deployment and remote debugging. The diagram shows the developer's machine, and the remote device at the left and right-hand side, respectively. During the first two stages, the developer's machine hosts only the debugger frontend representing the integrated development environment (IDE). This is where the source code of the application is compiled to bytecode (shown as Ⓐ in fig. 3). Once the byte code is generated, the application can be uploaded to a remote device running the event-based out-of-place VM Ⓑ. The remote VM executes the application, but it also contains a debug monitor, which listens for instructions from the frontend's debug manager. At the developer's device, the debug manager is a piece of software, typically embedded in an IDE, which handles all the debugging operations requested by developers. Typically, IDEs offer developers several debug views to debug applications. When a developer requests the program to be paused, the debug manager sends the correct interrupt message to the remote debug monitor Ⓒ. The debugger will then change the state of the VM accordingly, and also sends back information about the current state of the program. The debug manager can then update the debug views to display relevant information, such as the current line, breakpoints, local variables, etc.

3.4 Capturing a Debug Session

An essential aspect of our approach is capturing an out-of-place debug session of a running application at the remote device. A debug session contains all the state (i.e., application and execution

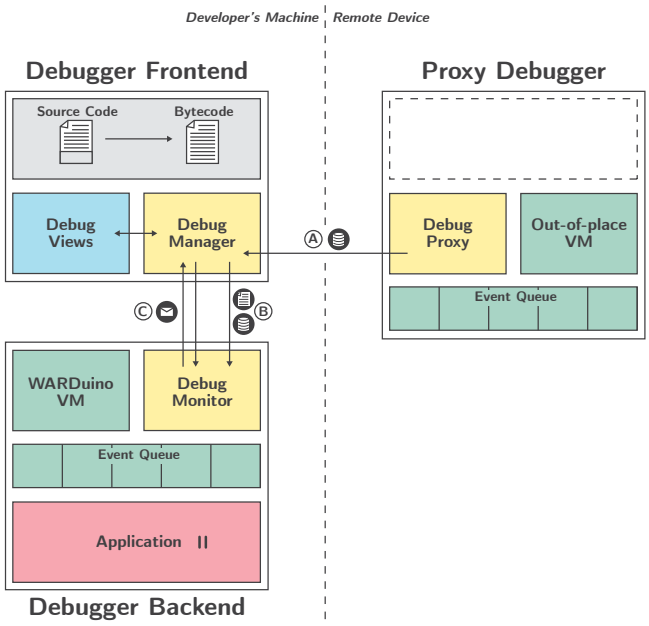

Figure 4: Stage 3: Capturing an out-of-place debug session. A snapshot of the currently executing program is created at the device and recreated at the developer's machine.

state) of a target application, which makes local debugging possible. The idea is to transfer the captured remote state to a local debugger backend, on the developer's machine, to run the target application locally.

State of the Session. What information constitutes an out-of-place debug session varies per VM and programming model [19]. As an example, the following state is captured for a WebAssembly-based VM we used to implement our approach (section 4.1):

- *Program counter*
- *Call stack* i.e., the trace of functions called to the point where the session got captured.
- *Breakpoints* that were set during remote debugging.
- the *stack of values* populated during program execution (e.g., function argument, function call results, etc.).
- *Global* and *local* variables
- State regarding WebAssembly *tables*: the initial size, maximum size, and table entries.
- *Branching table* needed by WebAssembly to branch to different blocks.
- State regarding *memory pages*: maximum pages, initial pages, and the pages content.
- The *events* that were not yet processed by the remote device when capturing the session.
- The *callback mapping*: a mapping defining which functions should process what events.

Recreating the Session. Figure 4 shows the main steps involved in the *capture* stage. When the debugger backend shown in fig. 3 receives the request for capturing a debug session, it pauses the

target program execution and serializes a debug session. From that moment on, the debugger backend becomes a *proxy debugger* as shown in fig. 4. This means that it will not provide step-wise control of the execution of the target application anymore (it is paused), and it transfers the serialized session to the debugger frontend (depicted in the figure as Ⓐ). The debugger frontend receiving this serialized state, will start a new debugger backend on the developer's machine. Once the backend is ready to communicate, the frontend sends the received captured session to it along with the byte code of the application Ⓑ. The local debugger backend can then load the application and restore it to the captured state. As such, developers are shown a debug view in which the application is paused at the point where the debug session was captured. From that moment on, developers can begin debugging locally which results in exchanges of messages between the debugger frontend and the new debugger backend Ⓒ.

3.5 Accessing Non-transferable Resources

Once a developer is presented with an out-of-place debug session at their machine, most debugging operations (e.g., stepping) will be performed locally. While state inspection is typically also a local feature in online debuggers, an event-based out-of-place debugger distinguishes between access to non-transferable resources and location-independent accesses (which can be performed locally). When the program accesses state corresponding to non-transferable resources our event-based out-of-place debugging approach provides a mechanism to fetch them from the remote device based on two strategies: (1) pull-based access (that directly requests the value to the remote device) and (2) event-based access (that requests the VM to be notified on value changes when they happen at the remote device). In what follows, we employ fig. 5 to detail the key ideas of each strategy. The figure depicts the interactions between the different debugger's components for accessing non-transferable resources during the local debugging stage.

3.5.1 Pull-based Non-transferable Resources. To support pull-based access to non-transferable resources, we use *proxy calls*, i.e., function calls on a foreign VM. Recall that operations performed by the developer during local debugging result in calls to the debugger manager from the debug views. Those calls in turn result in debug messages sent to the debugger backend which controls the target application Ⓐ. Whenever the debugger backend encounters the call of a function marked as a non-transferable resource, the local VM will resolve the call through a proxy call. A proxy call is requested by sending a debug message to the proxy debugger at the device with the arguments and the identity of the function to call Ⓑ. When the debug proxy receives the request, the out-of-place VM will execute the function using the received arguments. After the function is performed, the debug proxy sends back the return values of the function (if any) or an exception (if the function fails) to the debug monitor.

When the local VM receives the return values of the proxy device, it will place the values on the stack. This way the remote invocation behaves exactly like a normal primitive call to the rest of the VM. This also means that the debugger frontend will not be aware that a proxy call was executed, except for the higher latency of this access compared to a location-independent access.

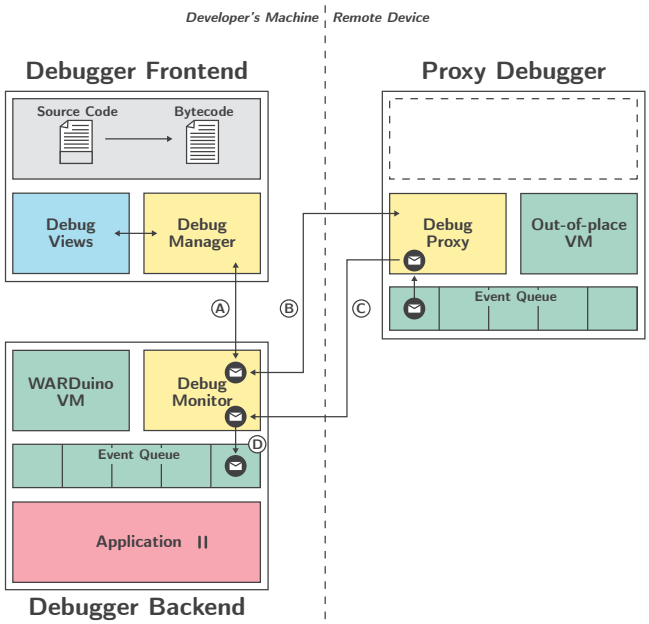

Figure 5: Stage 4: Local debugging. The debugger instructs the locally executing debugger backend while receiving and querying the proxy debugger. The proxy debugger sends any asynchronous events from its event queue to the backend.

3.5.2 Push-based Non-transferable Resources. In order to support push-based access to non-transferable resources, we extended the VM with the concept of an *event queue*. This queue stores asynchronous events received on the device, which model new sensor updates, MQTT messages, etc. When an event is added into the event queue, the debug proxy forwards it to the local debugger backend (depicted in fig. 5 Ⓒ) at the developer's machine. The backend stores the forwarded events in a local event queue, mirroring the event queue at the proxy debugger Ⓓ. Events received by the debugger backend will not be automatically handled. Instead, they are forwarded to the frontend, to be shown in a dedicated view. Events will be manually resolved upon the developer's request. When the VM receives such a request from the frontend, it will only process the event at the head of the queue. This way, the developer can choose at what point in the code an event should be handled, enabling the reproduction of multiple situations.

3.6 Local Debugging Support

By moving the debug session to the developer's machine, event-based out-of-place debugging can exploit the hardware of the programmer's machine to provide more advance debugging techniques that are too heavy-weighted for IoT devices. In this work, we do a first exploration of applying offline techniques to the local debug session by providing support for time-traveling debugging. This allows developers to debug the target application backwards, reverting the execution to an earlier state. To this end, the debug monitor keeps track of the generated state at each execution step within a local debug session. When an application is paused (due

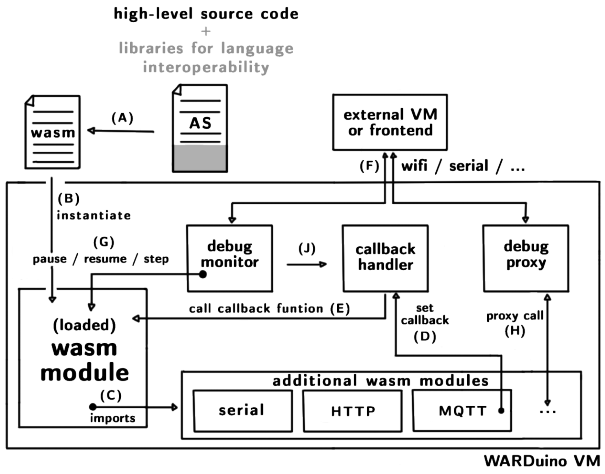

Figure 6: Overview of the extended WARDuino architecture.

to a breakpoint, or a pause debug command), developers can restore each previously stored state by issuing step-back commands. This can potentially shorten the debugging time since it allows developers to identify the root cause of the bug by going back to previous states, instead of having to rerun the application from the beginning to reproduce a bug.

4 EDWARD: AN EVENT-BASED OUT-OF-PLACE DEBUGGER FOR WARDUINO

We implemented a prototype event-based out-of-place debugger on top of a microcontroller VM for WebAssembly (Wasm) called WARDuino [13]. The prototype provides the features described in the previous section for IoT applications that compile to Wasm. The frontend of EDWARD is implemented as a Visual Studio Code (VSC) [22] plugin. In this section, we will first discuss the extensions to the WARDuino VM for supporting event-based out-of-place debugging, and subsequently present EDWARD's frontend.

4.1 Virtual Machine Extensions

In this section, we describe the extensions to WARDuino which turn it into an *event-based out-of-place* VM. Figure 6 gives a high-level overview of the extended WARDuino virtual machine. IoT applications are written in a high-level language that compiles to WebAssembly (e.g., AssemblyScript) and to use language-specific libraries to interoperate with the low-level interfaces of our WebAssembly primitives.

To run an application on WARDuino, it must first be compiled to WebAssembly Ⓐ. The resulting WebAssembly program is then instantiated in the VM Ⓑ. The program can import WARDuino primitives to access the hardware peripherals of the device Ⓒ. These primitives are are black-boxes written in C, which are exposed to WebAssembly programs through the *env* module. This way, all access to external hardware is encapsulated and accessed uniformly. This facilitates the implementation of pull-based access to non-transferable resources as we detail later.

The original WARDuino VM allows programs to be updated and debugged by developers remotely. The VM features a debug monitor that can receive updates and debug messages over different channels Ⓕ, such as Wi-Fi or the serial port. Debug messages can instruct the VM to pause, step or resume execution Ⓖ. We extended the existing debug monitor to handle the new debug messages for event-based out-of-place debugging (e.g., message for capturing out-of-place debug sessions). Moreover, we introduced a novel callback handling system to handle asynchronous events. It uses a FIFO queue to aggregate all asynchronous events generated by the environment. The handler will dispatch entries from this event queue to registered callback functions. Callbacks and events are associated through a topic string. The MQTT module, for instance, can register a Wasm function from the instantiated program as a callback for a specific topic Ⓓ. Whenever events for that topic occur, our callback handler invokes the registered function and passes the event to it Ⓔ.

We now highlight the most important implementation details for the different stages in our approach (cf. section 3).

Capturing a Debug Session. To capture and reconstruct a debug session, we extended WARDuino with three debug messages: *captureSession*, *recvSession*, and *monitorProxies*. The *captureSession* message is issued by EDWARD's frontend to fetch the current state of a program run on the remote device. When the debug monitor receives the *captureSession* message, it stops the program and creates a debug session including the information described in section 3.4. The *recvSession* message is used to send the captured debug session back to EDWARD's frontend. EDWARD's frontend will then create a new VM instance locally and forward the *recvSession* message to it. The newly created VM instance uses the captured state as its own new state. Finally, EDWARD's frontend sends the *monitorProxies* message to the local debug monitor, containing the IP address of the proxy debugger on the remote device such that EDWARD's backend can query or receive updates on non-transferable resources.

Accessing Pull-based Resources. We now detail how we implemented *proxy calls* to enable pull-based access to non-transferable resources. Identifying what function calls need to be invoked remotely is often a difficult task. In WARDuino, however, this is easier as all pull-based non-transferable resources are accessed through our primitives. We thus only need to remotely invoke those primitives from the *env* module. To enable this, we extended WARDuino with the *proxyCall* debug message, containing a function index and a list of function arguments. Whenever EDWARD's backend encounters a primitive call, it sends the *proxyCall* message to the proxy debugger with the correct index of the primitive and its arguments. When a *proxyCall* message is received, the debug proxy on the remote device executes the requested primitive (Ⓗ in fig. 6). The result (or exception) is then sent to the debug monitor at EDWARD's backend.

Accessing Push-based Resources. Asynchronous events are push-based non-transferable resources and cannot be proxied in the way that primitives are. As mentioned before we introduced a callback handling system in WARDuino. This means that the VM acts as the event dispatching authority. In the callback handling system,

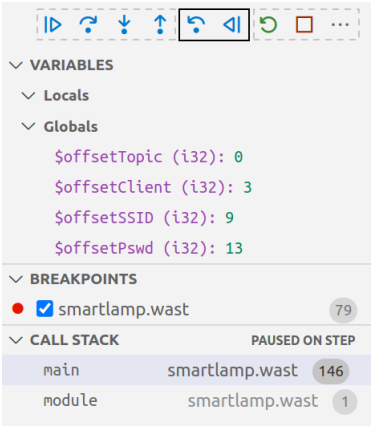

Figure 7: The standard components of the debugger sidebar in VSC for online debugging.

executing callbacks can be done locally aside from primitive calls, which are proxied. In other words, the VM captures all asynchronous events in a *single* event queue. This allows the VM on the remote device to redirect every generated event from its event queue to EDWARD's backend. To this end, we added a new debug message called *pushEvent*. When EDWARD's backend receives this message along with a serialized event, it adds the new event to its own event queue (Ⓙ in fig. 6). We also added two debug messages that are issued from EDWARD's frontend when developers want to view and control the event queue: *dumpEvents* gets all events, and *popEvent* processes the event at head of the queue.

4.2 EDWARD's Frontend in Visual Studio Code

EDWARD's frontend has been implemented in the Visual Studio Code IDE (VSC) [22]. More concretely, we built a plugin on top of VSC's own debugger extension API. Thanks to this API, we may leverage the standard VSC debugger user interface to offer views of both local and global variables, the callstack, and breakpoints.

Since WARDuino is a WebAssembly runtime, the VM and the debugger execute WebAssembly bytecode. By using WABT [29], we also provide support for debugging a textual representation of WebAssembly. While tool support for high-level languages compiled to WebAssembly varies, most provide some form of source code mapping such as DWARF. In future work, we will use these source-code mappings to support textual debugging of high-level code in the plugin.

Standard Online Debugging Views. Figure 7 shows the standard online debugging views containing local and global variables, breakpoints and the callstack. In fig. 7, EDWARD is paused due to a breakpoint, the *variables* view shows the state of the variables in scope. Next to inspecting the current value of the variables, the user can also change their value. Changing the value of a variable also changes the value in EDWARD's backend. The global variables are always displayed in the plugin, even when the program is not paused. Local variables, on the other hand, are only shown when the program is paused.

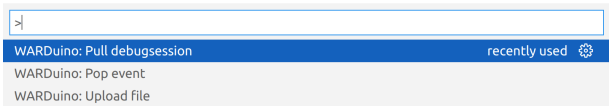

Figure 8: Pulling a debug session via the command palette.

Below the variables view, the plugin displays a list of all the current breakpoints (which can be enabled and disabled) and their line number in the source code. Finally, the *callstack* view shows a list of frames representing function activations, with the currently executing function at the top and the program entry at the bottom. Each frame also contains the line in source code as well as the filename. For the top frame, this line corresponds with the current position.

Figure 7 also shows, marked with a dashed gray line, the standard operations that are offered during remote debugging mode including pause, step, stop and resume execution. In what follows, we describe the special debug views and commands for event-based out-of-place debugging in EDWARD's frontend.

Capturing an Out-of-place Debug Session. To request the creation of an out-of-place debug session, developers can use a dedicated command available at the VSC command palette. More concretely, the *Pull debug session* command was added to the palette as shown in fig. 8. When executed during remote debugging, this command will capture the current session and switch to event-based out-of-place debugging by executing all the operations detailed in section 3.4. This operation happens transparently to the user, who is notified in the status bar of the operation's progress. As soon as a debug session is successfully recreated at the developer's machine, EDWARD enables two additional views specific to event-based out-of-place debugging presented in fig. 9 and further detailed in what follows.

Accessing Remote Resources. The *proxies* view, in the bottom part of fig. 9, shows an overview of the functions that are currently being proxied to the remote device. In the current implementation, when developers disable one of the proxies a random value is returned (as a mocked value) instead of performing the proxy call. In future work, we plan to explore others strategies to handle push-based access to non-transferable resources that are not being proxied.

Event-based Debugging. The *events* view, displayed above the proxies view in fig. 9, shows the queue of events captured by the debugger. Events consist of a topic and payload. Events are displayed by their topic, e.g., MQTT topic or the pin number for a hardware interrupts. An event can be expanded to show its content if a payload is available. This is the case for an MQTT event but for many events such as hardware interrupts this payload will be empty.

During event-based out-of-place debugging the timing of events can be controlled by the user. To enable this in EDWARD, developers can use the arrow in the top right corner of the events view to debug the execution of the first event of the queue. In future work, we plan to further explore debugging operations for events, e.g., let developers reorder the events, choose which one to debug, etc.

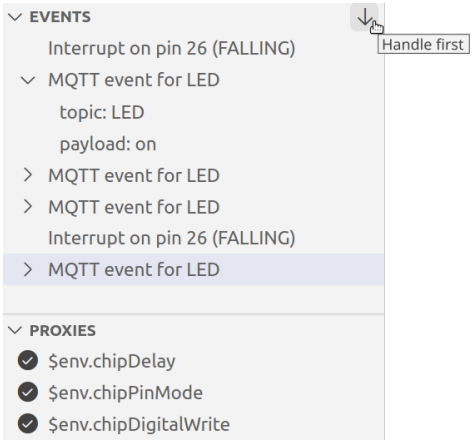

Figure 9: The event-based out-of-place debugging specific components of the VSC sidebar; contains the event queue and the proxied calls for accessing non-transferable resources.

Back-in-time Debugging. Figure 7 also shows that EDWARD offers developers stepping operations to debug programs backwards (marked with a black line). These two operations are only available during event-based out-of-place debugging, i.e., after capturing an out-of-place debug session. EDWARD currently supports stepping backwards via the button on the left. When that button is pressed, the debugger updates the variables and calls stack views to show the state at the previous execution step and updates the current line in the source code window to reflect the step back.

5 EVALUATION

In this section, we evaluate our event-based out-of-place debugging technique in two ways. First, we use EDWARD to detect and fix common bugs in IoT applications. Second, we conduct a performance assessment of event-based out-of-place debugging with respect to remote debugging.

5.1 Debugging Common Bug Issues

As mentioned in the introduction, a 2021 study on 5,565 bugs in 91 IoT projects showed that the most frequent types of bugs are related to software development and device issues [17]. In this section, we show an example program illustrating how event-based out-of-place debugging better accommodates finding and solving device issues than regular remote debugging. Appendix A provides a similar comparison but for a software development issue due to concurrency.

The Bug. Many device issues are related to handling interrupts [17]. Figure 10 shows a simple AssemblyScript application that toggles an LED when a button is pressed. The code listens for a hardware interrupts triggered on the falling edge of the button pin (line 11). Upon receiving an interrupt, the `buttonPressed` function is called, which toggles the LED (line 7). While the code may not contain errors, the hardware can cause bugs in it. Consider the following bug scenario: when testing the application with a real button, the LED sometimes does not change despite the button being pressed.

```
1  import * as wd from warduino;
2
3  const LED: u32 = 25;
4  const BUTTON: u32 = 26;
5
6  function buttonPressed(): void {
7    wd.digitalWrite(LED, !wd.digitalRead(LED));
8  }
9
10 export function main() : void {
11   wd.interruptOn(BUTTON, wd.FALLING, buttonPressed);
12   while(true);
13 }
```

Figure 10: A simple AssemblyScript program that toggles an LED when a button is pressed.

Bugfixing with a Remote Debugger. With a regular remote debugger, developers could start their diagnosis by adding a breakpoint in the `buttonPressed` callback function triggered when pressing the button. Note that in this simple example, there is only one single callback function, but in more complex IoT applications developers may need to place breakpoints in many callback functions as it is difficult to rule out which ones are not causing to the faulty behavior.

Stepping through code with asynchronous callbacks is generally not easy with current state of the art remote debuggers. Keeping track of all the asynchronous callbacks increases the number of times a developer needs to manually step through the application before discovering the error, complicating debugging. Moreover, stepping through the code is relatively slow, as the network latency between the developer's machine and the remote device slows down the debug session. Finally, most applications will not feature a busy loop as in our example, but the main thread runs concurrently with the asynchronous invocations, making it harder to notice errors.

Once the developer has stepped through all the asynchronous code letting the callbacks execute, the de developer might notice that the `buttonPressed` callback is strangely invoked multiple times. The reason is that a single button press can trigger multiple hardware interrupts due to a common problem of physical buttons called *contact bouncing* [20]. Contact bouncing happens when the voltage of a mechanical switch pulses rapidly, instead of performing a clean transition from high to low. In that case, the pin can register a falling edge multiple times in a row. Subsequently, the `buttonPressed` function is triggered multiple times for a single press. If contact bouncing causes the function to be triggered an even number of times, the state of the LED seems to remain the same, making the developer believe the code does nothing. It is not trivial to deduce the underlying contact bouncing problem by only stepping through the program.

Bugfixing with EDWARD. Let us now revisit the scenario using event-based out-of-place debugging. With EDWARD, developers can pull an out-of-place debug session from the remote device, and begin debugging locally at their machine. EDWARD provides the developer with a dedicated view on the event queue with all asynchronous events that happen at the remote device, and the ability to choose when the next event happens. When the developer pushes

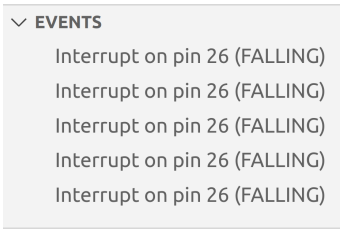

Figure 11: The debugger frontend shows a list of identical interrupts after a single button press.

	#Instructions	
Location independent	2092	99.15%
Non-transferable	18	0.85%
Total	2110	100%

Figure 12: Labeling of Wasm instructions for a smart lamp application (fig. 14 in appendix A).

the physical button once during debugging, they will immediately notice that EDWARD's events view suddenly contains multiple identical events for a single button press, as shown in fig. 11. This information enables the developer to more easily detect the contact bouncing issue.

If the developer has not yet deduced the root cause of the bug, they could use stepping through the code in a similar way than when using the remote debugger. However, this time, stepping through the code is fast as debugging happens locally without incurring in network communication. Moreover, EDWARD allows debugging the program backwards. This means that during debugging when the LED does not turn on, the developer can step back to the previous step to diagnose what exactly went wrong during the execution. There is no need to restart the program and try to guess what the right conditions for the bug were.

Conclusion. This example illustrates that using event-based out-of-place debugging makes a difference when debugging device issues compared to a remote debugger. Since EDWARD captures all non-transferable resources and provides a view on the event queue with all asynchronous events that happened at the remote device, developers can more easily diagnose device issues. For those cases where stepping is still needed, this happens with low latency. EDWARD also allows developers to step backwards, potentially reducing the debugging time as applications may not need to be restarted to reproduce the conditions for the bug to appear.

5.2 Quantitative Evaluation

We now present some preliminary quantitative evaluation of EDWARD, to underscore the potential of our approach to reduce performance impact while debugging IoT devices.

Code Analysis. To analyze the potential communication needed between debugger and remote device, consider the smart lamp application written in AssemblyScript allowing users to control the brightness of an LED (cf. fig. 14 in appendix A).

While remote debugging requires network communication between debugger and the remote device for *all* debugging operations and all types of instructions, event-based out-of-place debugging only requires network communication for those instructions that access non-transferable resources. In order to get an estimate of the amount of instructions which are location dependent compared to the non-transferable instruction we labeled each of the Wasm instructions of the smart lamp application's code as a location-independent instruction, or an instruction that accesses a non-transferable resource. The results shown in fig. 12 confirm our suspicion that location-independent instructions outweigh instructions accessing non-transferable resources.

Network Overhead. In order to get an estimate of the difference in network overhead between remote debugging and event-based out-of-place debugging we benchmarked the (debugging) network overhead of the smart lamp application. Our benchmarks were performed on a M5StickC [16] connected to a MacBook Pro with an Apple M1 Pro chip operating at 3.2 GHz CPU and 32GiB of RAM, through a local network.

Figure 13 plots the network overhead of stepping through the application with a remote debugging session. As we can see, there are small step-wise fluctuations caused by the changing amount of local variables in the program. The network overhead for each debugging step is approximately 2.2 kB.

For event-based out-of-place debugging, we benchmarked the network overhead of taking a full snapshot at each remote stepping operation, i.e. the network overhead involved with starting an out-of-place debugging session. Note that in practice the developer needs to perform this operation only once. Figure 15 shows the results of taking a snapshot at each stepping operation of the smart-lamp application. As expected, the network overhead involved with taking a full snapshot is much higher than a single debugging step, each full snapshot takes approximately 130 kB.

The significant difference in network overhead between a remote debugging step and a full snapshot is expected and is mostly because the snapshot captures the stack and a full memory dump of the running application.

Luckily, once a snapshot has been taken the debugging session can be executed locally and the subsequent debugging session will be much faster. Avoiding access to the remote device reduces network overhead and lowers debugging latency. The network overhead for proxied calls is much smaller than a normal debugging step and takes at most 10 bytes per remote call with an additional 4 bytes per argument. More importantly the network overhead for stepping through each of the location independent instructions is zero.

Latency. Finally, we also benchmarked the difference in latency between local debugging steps and remote debugging steps. When stepping through the smart-lamp application with event-based out-of-place debugging, we find that local steps take on average approximately 5ms while a remote *proxy call* takes approximately 500ms.

In practice this means that the developer using event-based out-of-place debugging will perceive almost instantaneously local debugging steps interleaved with remote calls which are perceived slower. As these non-transferable instructions make up less than

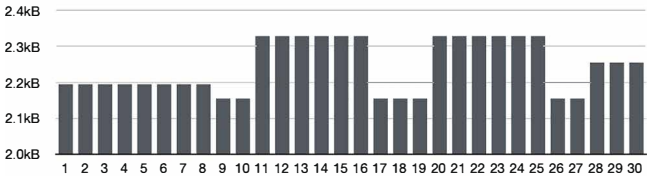

Figure 13: Network communication overhead of 30 step operations using remote debugging. Note that the overhead axis starts at 2kB.

1% of the code, most debugging steps will be able to be executed fast.

6 RELATED WORK

IoT systems are often built using low-level programming languages (e.g., C/C++) which are known to cause memory errors that are often hard to debug. As a result, several VMs specially tailored for microcontrollers have been proposed to hide the complexities of low-level programming. For instance, Espruino for Javascript [9], MicroPython [11] for Python, and WARDuino & Wasm3 for any language that compiles to WebAssembly [13, 28].

Those VMs incorporate some debugging support. In Espruino [9], developers can remotely debug JavaScript programs and through source-code modification log runtime information (e.g., stack traces, etc.) that is either forwarded to the debugger client or saved on the MCU while the debugger client is disconnected. In contrast, MicroPython [11] does not provide a remote debugger, so programmers need to use printf statements. Wasm3 proposes a GDB source code level remote debugger [7], but this work is at an early stage and has been inactive for two years. Compared to those VMs, our work not only provides a remote debugging experience, it also features online debugging with minimal latency on demand. Furthermore, to the best of our knowledge, no other approach allows developers to access the generated events of the remote device and control its processing.

With respect to prior work on out-of-place debugging [18, 19, 26], our approach provides sessions on-demand, a push-based strategy for accessing remote resources and reintroduces offline features like step-backwards into the debug session. Rojas Castillo et al. [26] did the first application of out-of-place debugging for IoT devices as an extension to WARDuino instead of relying on a reflective runtime. However, WOOD only offers pull-based accesses to non-transferable resources, it does not feature time-travelling debugging, nor a debugger frontend integrated into an IDE.

IoT debugging solutions based on logging events have also been explored. In IoTReplay [10] a network gateway records external events that are broadcasted on the network and replays them on emulators or copy hardware. In contrast, our work captures all the events (external & internal) directly on the VM at the IoT device, giving control to the developer when to process them. Resense [12] logs internal events such as sensor values, for later replaying the logged values through sensor emulation. In contrast, our work allows developers to access and debug both internal and external events through a debugger. Moreover, Resense is applied to Raspberry Pi devices which are much more powerful than the ones we aim to debug (e.g., ESP32 devices with a few hundreds of kBs of memory).

7 CONCLUSION

In this paper, we introduced *event-based out-of-place debugging*, a novel debugging approach in which program execution is captured on a remote IoT device and recreated in a debug session at a more powerful machine. During debugging, the classic online debugging operations such as stepping are complemented with features specially targeted to the IoT environment. First, non-transferable remote resources can be accessed in a *pull-based* way by proxy calls, or in a *push-based* way by giving control over all asynchronous events such as interrupts from buttons, sensor updates, network communication, etc. Second, event-based out-of-place debugging permits reintroducing support for more advanced debugging features, like time-traveling debugging, not available for resource-constrained IoT devices.

We presented *EDWARD*, a prototype implementation of our approach for the WARDuino VM. EDWARD's frontend has been implemented as a VSC plugin, in which developers can perform classical remote debugging but also pull debug sessions to debug locally, manage the event queue, inspect proxied function calls, and step backwards. We validate our solution through two application scenarios by showing applicability in detecting and solving the most frequent types of bug in IoT devices, i.e., device issues like contact bouncing and software development issues such as concurrency problems. These examples showed how the debugging facilities enabled by EDWARD ease the debugging process in comparison to a remote debugger. Finally, initial benchmarks show that the latency for local stepping is approximately 100 times less than when using remote debugging.

ACKNOWLEDGMENTS

Tom Lauwaerts is funded by the Research Foundation Flanders (FWO) with file number G030320N. Robbert Gurdeep Singh is funded by a doctoral fellowship from the Special Research Fund (BOF) of UGent (BOF18/DOC/327). We would also like to thank the anonymous reviewers for their feedback.

A DEBUGGING CONCURRENCY ISSUES

Apart from the device related bugs discussed in section 5.1, software development issues are common in IoT applications [17]. Within this type, a common root cause is concurrency faults [17]. Figure 14 shows the implementation of a smart lamp application in AssemblyScript. The code allows users to control the brightness of an LED with MQTT messages or two hardware buttons. The application listens for messages on topics, *increase* and *decrease* (lines 58 - 61). For each message, the code increases or decreases the brightness of the LED by five percent, respectively. Instead of changing the brightness abruptly, the code gradually changes the brightness. For this reason, the callback function does not directly change the LED brightness, but it changes the variable `delta` to record the requested change. The function `updateBrightness` called in the main application loop changes the actual brightness gradually. Every time it is called,

```
1  import * as wd from "warduino";
2
3  const LED: i32 = 10;
4  const MAX_BRIGHTNESS: i32 = 255;
5  const UP_BUTTON: i32 = 37;
6  const DOWN_BUTTON: i32 = 39;
7  const CHANNEL: i32 = 0;
8  const SSID = "local-network";
9  const PASSWORD = "network-password";
10 const CLIENT_ID = "random-mqtt-client-id";
11
12 let brightness: i32 = 0;
13 let delta: i32 = 0;
14
15 function until_connected(connect: () => void,
16                          connected: () => boolean): void {
17     while (!connected()) {
18         wd.delay(1000);
19         connect();}}
20
21 function check_connection(): void {
22     until_connected(
23         () => { wd.mqtt_connect(CLIENT_ID);
24                 wd.mqtt_loop(); },
25         () => { return wd.mqtt_connected(); });}
26
27 function init(): void {
28     wd.analogSetup(CHANNEL, 5000, 12);
29     wd.analogAttach(LED, CHANNEL);
30
31     // Connect to Wi-Fi
32     until_connected(
33       () => { wd.wifi_connect(SSID, PASSWORD); },
34       ()=>{return wd.wifi_status() == wd.WL_CONNECTED;});
35     let message = "Connected to wifi network with ip: ";
36     wd.print(message.concat(wd.wifi_localip()));
37
38     // Connect to MQTT broker
39     wd.mqtt_init("192.168.0.24", 1883);
40     check_connection();}
41
42 function updateBrightness(): void {
43     brightness += delta;
44     if (brightness < 0) {
45         brightness = 0;
46     }
47     if (brightness > MAX_BRIGHTNESS) {
48         brightness = MAX_BRIGHTNESS;
49     }
50     wd.analogWrite(CHANNEL, brightness, MAX_BRIGHTNESS);
51     delta = 0;}
52
53
54 export function main(): void {
55     init();
56
57     // Subscribe to MQTT topics and turn on LED
58     wd.mqtt_subscribe("increase",
59         (topic: string, payload: string) => {delta = 5});
60     wd.mqtt_subscribe("decrease",
61         (topic: string, payload: string) => {delta = -5});
62     while (true) {
63         check_connection();
64         if (delta !== 0) updateBrightness();}}
```

Figure 14: The full code of the example application illustrating a concurrency problem in IoT

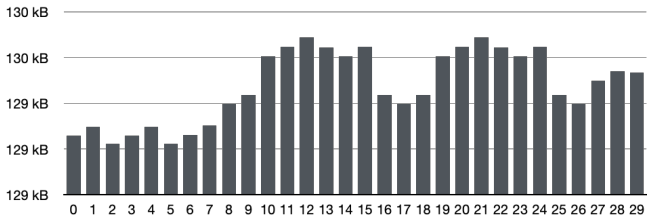

Figure 15: Network communication overhead for full snapshotting. Note that the y-axis starts at 129kB.

it changes the brightness by one percent in the direction dictated by the sign of `delta`. After doing this, the absolute value of `delta` is lowered by one. In this way, the application only needs to check the value of the delta variable and change the brightness whenever it does not equal zero (line 64).

The Bug. When testing this program with a real hardware setup, the developer notices that the brightness changes irregularly. When sending two messages to the *increase* topic, the LED increases its intensity by only 5% instead of 10%. The reason is that the second message overwrites the value of the `delta` variable before the `updateBrightness` function updates the LED. Such concurrency bugs are often time sensitive, and do not always manifest [21]. In our example the bug only manifests when sending two MQTT messages rapidly. This made finding the exact conditions for the bug very difficult. Moreover, the effects of the bug can happen long after the root cause [25], i.e. when the main loop updates the brightness.

Bugfixing with a Remote Debugger. As mentioned before, stepping through code with asynchronous callbacks using current state of the art remote debuggers is very difficult. The developer has no control over when the asynchronous callbacks are called. This makes it difficult to reproduce the exact conditions in which the bug manifest. In turn, this increases the times a developer needs to manually step through the application before reproducing the error, drastically complicating debugging. Moreover, the developer needs to keep track of all the steps taken and remember these steps carefully for when the bug manifest later, and a new debugging session is needed. Finally, stepping through the code is relatively slow due to network latency between the developer's machine and the remote device.

Bugfixing with EDWARD. EDWARD can help debugging these concurrency bugs thanks to its event scheduling and time-traveling debugging features. By using EDWARD, developers can inspect the generated events and schedule their execution one after another (as the developer suspects that this is when the bug manifests). When they step through the code, they can visually inspect the brightness of the LED and observe that the bug has indeed manifested. If the root cause was not discovered during the initial debugging session the developer can easily step back in time and go through the code as many times as needed. During this time-traveling debugging session they can then notice that when receiving two messages in a row, the second may overwrite the `delta` parameter set by the first message before it was processed by the main loop, revealing the

cause of the bug. Lines 59 and 61 should increase (and decrease) the value of `dela` instead of overwriting it.

REFERENCES

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A Survey. *Computer Networks* 54, 15 (Oct. 2010), 2787–2805. https://doi.org/10.1016/j.comnet.2010.05.010

[2] Dominik Aumayr, Stefan Marr, Clément Béra, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. Efficient and deterministic record & replay for actor languages. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes - ManLang '18*. ACM Press. https://doi.org/10.1145/3237009.3237015

[3] Andrew Banks and Rahul Gupta. 2014. MQTT Version 3.1. 1. *OASIS standard* 29 (2014).

[4] Earl T. Barr and Mark Marron. 2014. Tardis: Affordable Time-Travel Debugging in Managed Runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/2660193.2660209

[5] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. 2015. Deterministic Replay: A Survey. *ACM Comput. Surv.* 48, 2, Article 17 (Sept. 2015), 47 pages. https://doi.org/10.1145/2790077

[6] CodeMagic LTD. 2022. Welcome to Wokwi! https://docs.wokwi.com/?utm_source=wokwi. Accessed: May 12, 2022.

[7] Wasm3 Debug. 2020. Espruino. https://github.com/wasm3/wasm-debug. Accessed: May 12, 2022.

[8] Jakob Engblom. 2012. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. 1–6.

[9] Espruino. 2021. Espruino. https://www.espruino.com/. Accessed: May 12, 2022.

[10] Kaiming Fang and Guanhua Yan. 2020. IoTReplay: Troubleshooting COTS IoT Devices with Record and Replay. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. 193–205. https://doi.org/10.1109/SEC50012.2020.00033

[11] Damien George. 2021. MicroPython. https://micropython.org/. Accessed: May 12, 2022.

[12] Dimitrios Giouroukis, Julius Hülsmann, Janis von Bleichert, Morgan Geldenhuys, Tim Stullich, Felipe Oliveira Gutierrez, Jonas Traub, Kaustubh Beedkar, and Volker Markl. 2019. Resense: Transparent Record and Replay of Sensor Data in the Internet of Things. In *22nd International Conference on Extending Database Technology (EDBT)*.

[13] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes - MPLR 2019*. ACM Press, 27–36. https://doi.org/10.1145/3357390.3361029

[14] 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 *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[15] Gabor Kecskemeti, Giuliano Casale, Devki Nandan Jha, Justin Lyon, and Rajiv Ranjan. 2017. Modelling and Simulation Challenges in Internet of Things. *IEEE Cloud Computing* 4, 1 (2017), 62–69. https://doi.org/10.1109/MCC.2017.18

[16] M5STACK. 2021. M5StickC. https://docs.m5stack.com/en/core/m5stickc/. Accessed: May 12, 2022.

[17] Amir Makhshari and Ali Mesbah. 2021. IoT Bugs and Development Challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 460–472. https://doi.org/10.1109/ICSE43902.2021.00051

[18] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2021. Practical Online Debugging of Spark-like Applications. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 620–631. https://doi.org/10.1109/QRS54544.2021.00072

[19] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2018. Out-Of-Place debugging: a debugging architecture to reduce debugging interference. *The Art, Science, and Engineering of Programming* 3, 2 (nov 2018). https://doi.org/10.22152/programming-journal.org/2019/3/3

[20] J.W. McBride. 1989. Electrical contact bounce in medium-duty contacts. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology* 12, 1 (March 1989), 82–90. https://doi.org/10.1109/33.19016

[21] Charles E. McDowell and David P. Helmbold. 1989. Debugging Concurrent Programs. *Comput. Surveys* 21, 4 (Dec. 1989), 593–622. https://doi.org/10.1145/76894.76897

[22] Microsoft. 2022. Visual Studio Code: Extension API. https://code.visualstudio.com/api. Accessed: May 12, 2022.

[23] Armando Miraglia, Dirk Vogt, Herbert Bos, Andy Tanenbaum, and Cristiano Giuffrida. 2016. Peeking into the Past: Efficient Checkpoint-Assisted Time-Traveling Debugging. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 455–466. https://doi.org/10.1109/ISSRE.2016.9

[24] David Pacheco. 2011. Postmortem Debugging in Dynamic Environments. *Commun. ACM* 54, 12 (Dec. 2011), 44–51. https://doi.org/10.1145/2043174.2043189

[25] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2016. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2016), 83–110. http://dblp.uni-trier.de/db/journals/sqj/sqj25.html#PerscheidSTH17

[26] Carlos Rojas Castillo, Matteo Marra, Jim Bauwens, and Elisa Gonzalez Boix. 2021. WOOD: Extending a WebAssembly VM with Out-of-Place Debugging for IoT applications. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Chicago IL, USA) (*VMIL 2021*). pre-print http://soft.vub.ac.be/Publications/2021/vub-tr-soft-21-11.pdf.

[27] TOPLLab. 2021. WARDuino-VSCode. https://github.com/TOPLLab/WARDuino-VSCode. Accessed: May 12, 2022.

[28] Wasm3. 2020. Wasm3. https://github.com/wasm3/wasm3. Accessed: May 12, 2022.

[29] WebAssembly. 2022. WABT: The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt. Accessed: May 12, 2022.