WARDuino

A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers

Robbert Gurdeep Singh
Ghent University
Ghent, Belgium
Robbert.GurdeepSingh@ugent.be

Christophe Scholliers
Ghent University
Ghent, Belgium
Christophe.Scholliers@ugent.be

Abstract

It is extremely hard and time-consuming to make correct and efficient programs for microcontrollers. Usually microcontrollers are programmed in a low level programming language such as C which makes them hard to debug and maintain. To raise the abstraction level, many high level programming languages have provided support for programming microcontrollers. Examples include Python, Lua, C# and JavaScript. Using these languages has the downside that they are orders of magnitude slower than the low-level languages. Moreover, they often provide no remote debugging support.

In this paper we investigate the feasibility of using WebAssembly to program Arduino compatible microcontrollers. Our experiments lead to extending the standard WebAssembly VM with: 1) safe live code updates for functions and data 2) remote debugging support at the VM level 3) programmer configurable (Arduino) modules in order to keep the virtual machine's footprint as small as possible. The resulting WARDuino VM enables the programmer to have better performance than an interpreted approach while simultaneously increasing the ease of development.

To evaluate our approach, we implemented a simple breakout game and conducted micro benchmarks which show that the VM runs approximately 5 times faster than Espruino, a popular JavaScript interpreter for the ESP32 microcontroller.

CCS Concepts • **Software and its engineering → Virtual machines**; **Software testing and debugging**.

Keywords WebAssembly, Virtual Machine, Arduino, Live Code Updates

MPLR '19, October 21–22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6977-0/19/10…$15.00
https://doi.org/10.1145/3357390.3361029

ACM Reference Format:
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 '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357390.3361029

1 Introduction

Microcontrollers can be programmed with *low-level* languages such as C [9] and C++ [16] but there are also various *high-level* programming languages available. Low-level programming languages have the advantages that they can provide the best speed for the devices. The disadvantages of low-level programming languages are: 1) Finding and debugging memory bugs is tedious without a (expensive) hardware debugger. 2) Due to the tight coupling of the programming model and microcontroller porting a program form one microcontroller to another is not straightforward. 3) Uploading a program to the chip (flashing) is slow and tedious, a small change in the program requires the whole program to be re-uploaded which can take up to a minute.

Programming microcontrollers with a high-level language has a lot of benefits because the available abstractions hide the complexity of programming microcontrollers. The downsides of these languages are that: 1) they tend to be a lot slower, 2) many of the them do not provide good remote debugging support, 3) for performance reasons access to the peripheral devices is often baked into the programming language which means that the language has limited use if your peripheral device is not supported by the language.

In this paper we investigate the use of a *memory-safe* virtual machine employing WebAssembly (WA) as a middle ground between high-level and low-level programming languages. The advantages of the virtual machine is that it provides safety and exposes the underlying hardware uniformly so that switching microcontroller does not require a rewrite of the programs.

In this work, we extend WebAssembly with safe live code updates and remote debugging. Moreover, we use the existing functionality provided by WA to define modules which exposes the underlying functionality provided by the ESP32 chip. Micro benchmarks show that the use of the VM provides good speedups compared to a interpreter based approach.

2 The Need for Something New

In order to demonstrate the need to use of WebAssembly on microcontrollers we note that the following hurdles exist when programming these devices with a low-level language:

1. The development cycle is very long, uploading a new version of the program takes the same (long) time no matter how small the change.
2. Debugging is done by using `print` statements.
3. The microcontroller can crash leaving no clear information about where the program went wrong.

Some of these hurdles can be mitigated by using one of the high-level programming languages available for programming microcontrollers. This is not a full solution, as there is now a new set of problems:

1. The debugging support is often still done by using `print` statements.
2. The runtime of the language takes a lot of space on the chip no matter how few functionalities are used.
3. A high-level language is significantly slower than a low level programming language, therefore writing device drivers in the high level programming language is futile.

In this paper we investigate whether a WebAssembly virtual machine can offer the programmer the following benefits:

1. Support a wide range of programming languages.
2. Allow fast and safe live code updates.
3. Provide remotely debugging facilities.
4. Enable configurable runtime where the programmer only pays for the used functionality.
5. Good performance.

3 WARDuino Overview

In this section we give an overview of our virtual machine based approach towards programming microcontrollers. Our virtual machine called WARDuino builds upon two major pillars Arduino and WebAssembly. We first give a short overview of both. After that, we briefly describe our extensions to WebAssembly that aim to increase its usability as a target platform for programming microcontrollers.

3.1 WebAssembly

WebAssembly (WA) is a recently proposed low-level code platform initially designed for the web [5]. Its main goals are to provide safety, portability, compact code representation and fast code execution. The difficulty in its design was to offer all these properties at once. While the primary motivation of WA is to provide a unifying platform for the Web, the specification hints that WA can be used as a virtual machine for wide range of platforms. Moreover, the specification has made a point out of not depending in any way on the JavaScript environment found on the web. Interestingly the design decisions made for WA are also very desirable when programming microcontrollers.

While the design decisions for WA make a very good starting point for programming microcontrollers there are still a number of desirable properties which are not directly addressed. First, the standard does not specify anything about how WA may be debugged. Second, there is no specification of how WA programs can be updated safely. Finally, the standard does not provide primitives to access the underlying hardware.

3.2 Arduino

Arduino [1] is an open-source electronics platform for a wide range of microcontrollers. A thin layer on top of C and a vivid community makes starting to program microcontrollers a lot less painful. The Arduino platform does an excellent job in defining uniform libraries. For example, getting the iconic blinking LED example working on your microcontroller is the same for all supported microcontrollers. The reason is that all microcontroller boards implement a core set of libraries to access the input-output pins, and they all define a number of constants for example `LED_PIN`.

We build on the success of the Arduino libraries by exposing them as modules in WA. By building upon the Arduino libraries we obtain the same kind of reusability for porting programs written in WA to different kinds of microcontrollers.

3.3 Architectural Overview

While the WebAssembly layer allows programmers to execute programs that compile to WA, it does not allow the programmer to easily update the program or access the underlying hardware of the microcontroller. Our extensions to WA aim to resolve this. We have two extensions one for updating live functions and data and a second to allow the virtual machine to be debugged remotely. Finally, we also defined a set of modules: `GPIO`, `SPI`, `AD/DC`, and `PWM` built on top of Arduino. Each of these modules provides the programmer with support for accessing the low level hardware modules of the microcontroller. These modules live in the VM, between the WA and chip layer.

To allow debugging, we allow receiving debug messages trough a wide variety of channels[1]. With these messages, the programmer can alter the running state of the VM. We allow "*pause*", "*play*", "*step*", "*dump*", "$break^{+}\ id$" and "$break^{-}\ id$" messages. The first 3 messages respectively pause the execution of the VM, continue it and allow stepping to the next WA instruction. The *dump* message dumps the state of the VM. This includes instruction pointer, the callstack, the locals and so on. Finally, the "$break^{+}\ id$" ("$break^{-}\ id$") message adds (removes) a breakpoint by adding (removing)

[1] Any channel that we can place interrupts on: Serial bus, Wi-Fi, ...

it to set of pending breakpoints. If a breakpoint is hit, WARDuino pauses. For live code updates developers can send an "*upload*", "$update_f$" or "$update_l$" message with new data as payload. The provided data will be used to respectively update the entire WebAssembly state (restart with new code), do a live update of a function or update a local. The full formal semantics can be found in appendix A.

4 WARDuino Modules Extensions

The WebAssembly standard allows the definition of custom modules provided by the runtime. In web browsers these modules provide interoperability with JavaScript. Here we make use of the same mechanism to provide access to the hardware functionalities of the microcontroller.

4.1 Digital Input-Output

A first module exposes the hardware pins of the microcontroller. Each pin of the microcontroller has multiple functionalities or modes, it is therefore important to make sure that the mode of the pin is set correctly. Once the pin mode has been set, it can be used for reading or writing.

In a microcontroller each pin is connected through a certain port, as popularized by Arduino the division in ports and pins is abstracted away through a simple API. This API allows to set the pin mode, read the pin or write a (digital) value to the pin. In WARDuino we defined a naive implementation of these functions in a module. The signatures of the functions in the WARDuino IO module are:

$$\begin{array}{ll} pinMode(pin, mode) & int \times int \rightarrow () \\ digitalWrite(pin, value) & int \times int \rightarrow () \\ digitalRead(pin) & int \rightarrow int \end{array}$$

An example WARDuino program which blinks a LED is shown in figure 1. This program is defined as a module which specifies a number of function declarations, defines which functions are imported and finally defines a function `$blink--arduino`. This program defines two function types: the first called `$void->void` and a second called `$int->int->void`. Both are function types, the first one specifies that it takes an empty parameter list and returns an empty result list. The second one takes two parameters both of type *i*32 and has an empty return list. Subsequently, two functions are imported into the module, `pin_mode` and `digital_write`, both functions are imported form the `IO` module. This module is implemented inside of the WARDuino VM.

This piece of code is verified in two ways by the WebAssembly tools. First, the module is type checked and any misuse of the imported functions will be signaled by the typechecker. Second, once this module is loaded onto the microcontroller the WARDuino virtual machine verifies whether the type declaration of the imported functions corresponds with the known types. Those two checks protect the programmer from executing ill-formed programs.

```
(module
 (; Type declarations ;)
 (type $void->void (func (param) (result)))
 (type $int->int->void
    (func (param i32) (param i32) (result)))
 (; Imports ;)
 (import "IO" "pin_mode"
     (func $pin_mode  (type $int->int->void)))
 (import "IO" "digital_write"
     (func $dig_write (type $int->int->void)))
 (; blink function ;)
 (func $blink_arduino (type $void->void)
  (call $pin_mode (i32.const 16) (i32.const 1))
  (loop
   (call $dig_write (i32.const 16) (i32.const 0))
   (call $wait)
   (call $dig_write (i32.const 16) (i32.const 1))
   (br 0))))
```

Figure 1. WARDuino Blink Example

4.2 Pulse Width Modulation

The pulse width modulation (PWM) module of a microcontroller allows the programmer to send out a square wave to one of the output pins without having to write a busy loop. The PWM module is often used to simulate analog output trough digital means. The prototypical example is to dim a LED light. Other uses of the PWM module include generating sounds.

To control the PWM module the programmer has one API function, `setPinFrequency`, shown below this paragraph. This function changes the default frequency given a certain pin. For example when the default frequency on a pin `D1` is 31250 Hz a call to `(setPinFrequency D1 8)` will change the frequency on the pin to 31250/8 Hz.

$$\texttt{setPinFrequency}(\texttt{pin}, \texttt{divider}) \quad int \times int \rightarrow ()$$

4.3 Serial Peripheral Interface

The serial peripheral interface is a bus protocol which is commonly used to communicate between a microcontroller and small peripheral devices such as sensors, SD-cards, displays, and shift registers. The bus consists of three data lines a data line to indicate the clock SCK, a data line for master input slave out communication MISO, and a data line for master out slave in communication MOSI. Next to the data communication lines there is also an optional chip select line (per peripheral device). A single SPI bus can communicate with multiple peripheral devices at once. In order to indicate to which device the master wants to communicate it first needs to make one of the chip selection lines high.

SPI communication can be implemented in hardware or in software. When making use of the hardware implementation the programmer must use specific pins of the microcontroller. In software the programmer is free to use any of the available

input-output pins. Software implementations are however, significantly slower than making use of the hardware implementation.

The functions governing access to the hardware SPI bus are shown in figure 2. The functions `spiClockDivider`, `spiBitOrder`, `spiDataMode` are configuration functions to specify how data will be transferred. Before actually using the SPI bus the programmer first needs to call the `spiBegin` which initializes the SPI module. Once opened, the programmer can start transferring data from the chip to the peripheral device by using one of the transfer functions. We included two kinds of transfer functions one for 8bit transfers and one for 16bit transfers. For both variants we also included a bulk mode which sends the same data a specific kind of times. The inclusion of the bulk operations improves the performance of a display driver greatly.

$spiBitOrder(bitorder)$	$int \rightarrow ()$
$spiClockDivider(divider)$	$int \rightarrow ()$
$spiDataMode(mode)$	$int \rightarrow ()$
$spiBegin()$	$() \rightarrow ()$
$spiTransfer8(data)$	$int \rightarrow ()$
$spiTransfer16(data)$	$int \rightarrow ()$
$spiBulkTransfer8(count, data)$	$int \times int \rightarrow ()$
$spiBulkTransfer16(count, data)$	$int \times int \rightarrow ()$
$spiEnd()$	$() \rightarrow ()$

Figure 2. Signature of the WARDuino SPI module

5 WARDuino Implementation

The WARDuino VM is written in C++ as an extension of an open source project by Joel Martin [2]. It is a stack based virtual machine which implements most of the WebAssembly standard. The major changes we had to make to the VM is to: 1) introduce an interrupt driven messaging system for listening to debugging messages 2) implement native modules for accessing the hardware modules 3) rewrite the memory layout to allow for live code updates.

5.1 Implementation

The WARDuino VM is a stack based virtual machine. It processes the code instruction by instruction while keeping track of a program counter. The VM also has run state, "PLAY", "PAUSE" or "STEP", which is checked before processing each instruction. An interrupt driven queuing system handles incoming messages from the remote debugger.

5.1.1 Debugging Queue

Debug messages can be sent to WARDuino by various means. Such a message always starts with a byte identifying the type of debugging message, and it ends with another specific byte sequence.

When a debug message is sent to the microcontroller, it is caught by an interrupt handler. This handler reads the available data and passes it on to the VM. The VM in turn waits for a full debugging package to arrive. Once a package is complete, it is placed in a queue for final processing.

The debugging queue is checked before each instruction executed by the virtual machine. If a message is present in the queue, appropriate action is taken. The *play*, *pause* and *step* messages respectively run, pause or step the currently executing program by setting the run state appropriately.

When a *dump* message is received, the run state is set to PAUSE and a JSON representation of the current state of the VM is sent back to the user. The JSON object contains the callstack, a list of functions, and the current instruction pointer. An example output is shown in figure 9 of Appendix B. In our implementation we also allow querying only specific elements of the state, such as the local variables.

5.2 Breakpoints

The remote debugging messages $break^+$ and $break^-$ carry a pointer to the code a user wishes to PAUSE execution at. These breakpoints are stored in a set. The set is checked before each instruction. When a breakpoint is hit, the run state is set to PAUSE, and an acknowledgment message is sent to the remote debugger.

5.2.1 Native Modules

WA is a module based system. To transparently provide access to the hardware we provide modules that can be included in the WA program. The code that handles imports can now handle certain "built-in" modules that provide a set of functions to the programmer. These functions can be implemented straightforwardly in C. To make the VM more lightweight, these built-in modules can be turned on or off. This way the developer only pays for the built-ins they use.

5.2.2 Live Code Updates

The remote debugging messages $update_f$ and $update_l$ contain two things: the id of the function or local to update and the new value. According to the semantics, the VM should be in the PAUSE state to process such as change. If the virtual machine is not yet in the PAUSE state, it is set to it and the change is processed.

Updating a local simply updates the appropriate value on the stack. Updating a function on the other hand is slightly more elaborate. First, the bytecode of the function is parsed and the appropriate structures are built. If the new function has an identical type, the pointer in WARDuino's function table is replaced with a reference to the new code. Any running call of the existing function will continue to work. A new call to a replaced function will use the updated code.

[2] https://github.com/kanaka/wac

5.3 Benchmarks

In order to get a preliminary idea of the performance of the virtual machine we compare WARDuino with a JavaScript based VM called Espruino[3] [19]. We compare both the execution speed and the size of the programs for both WARDuino and Espruino. In our measurements we do not take into account the uploading and initialization time of the virtual machines. The measurements are performed on an ESP DEVKITV1. This board features a ESP-WROOM-32 chip that operates at 240 MHZ, with 520KiB SRAM, integrated Wi-Fi, dual-mode Bluetooth, UART, ETH, PWM, IR, Touch Sensors, and a SPI interface.

5.3.1 Execution Speed

We start by giving an overview of the execution speed measurements. Our benchmark consists of six computationally intensive programs implemented in both JavaScript (for Espruino) and WebAssembly (for WARDuino).

For each program we report the time that elapses between starting and ending the execution of the program ten times. The WebAssembly code was generated from C code with Emscripten. To ensure an honest comparison this C code is identical in structure to the JavaScript code except for the addition of types. Additionally, we prohibited loop unrolling and inlining of the benchmark functions.

The benchmarks results are shown in the top graph of figure 3. The blue bars indicate the execution time of Espruino, the red bars shows the execution time for WARDuino, both on a log scale. We see that WARDuino consistently outperforms Espruino by a factor of 5. Note that the difference is even larger for the `tak` benchmark. This may be attributed to the extreme amount of recursion the `tak` function exhibits. Our suspicion seems to be confirmed by the (iterative) `fib` benchmark which calculates Fibonacci numbers without recusing. In this benchmark the performance difference is indeed less pronounced as in the `tak` benchmark.

We also compared our results to a native C implementation of the same code. Detailed results can be found in Appendix C. Because the native implementation in C is not based on a stack machine, it is much faster. To add two numbers for example, no stack access is needed in native C. Since WA is a stack machine, and our implementation does not yet feature a JIT compiler, memory access is required to perform all basic operations. WARDuino is about 465 times slower than a native C equivalent, Espruino is 4663 times slower[4].

5.3.2 Code Size

Another interesting metric is the size of the code to be uploaded to an instantiated VM. The second graph of figure 3 shows the size of the code for each of the benchmarks. For WARDuino, in red, the size of the WebAssemby binary (`wasm`) is shown. For Espruino the size of the JavaScript file is shown. The `wasm` files are smaller because they are in a binary format. The size of the JavaScript files could be decreased trough a process called minimization, which makes the JavaScript smaller and no longer readable for humans. We do not take this into account as the Espruino IDE does not minimize programs automatically. An additional downside of JavaScript is that it is harder to parse compared to the binary `wasm` files.

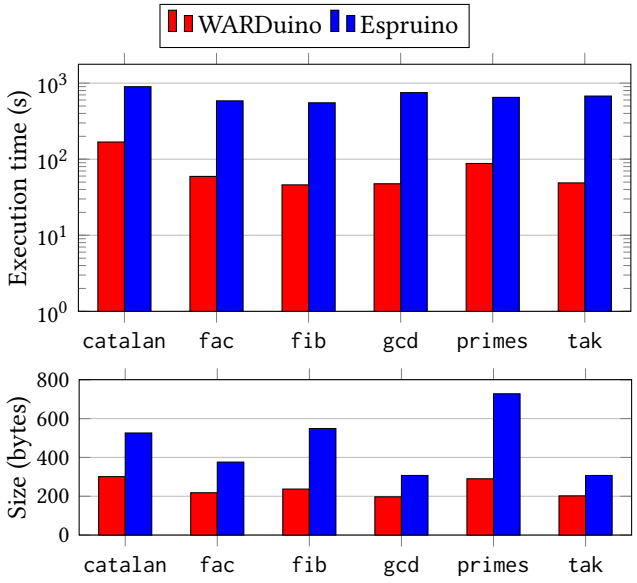

Figure 3. Execution times as multiples of the native C execution time (top) and bytecode sizes (bottom) of Espruino programs (blue) compared to the same metric for the same program implemented in WARDuino with Emscripten (red).

From these benchmarks we can conclude that WARDuino has reasonable performance compared to Espruino in both execution time and code size.

5.4 A Breakout Game

We also investigated how well the WARDuino VM performs for programming microcontrollers in practice by implementing a simple breakout game rendered on a 128x160 display. To stress test the WARDuino VM we implemented the display driver *entirely* in WA. This minimal driver consists of three functions, `initialise-display`, `clear-screen` and `fillrectangle`. Our first implementation had reasonable performance but calling the clear-screen function was noticeably slower than a pure C implementation. The reason is that every pixel consists of 16 bytes which required two calls to the SPI driver. Clearing the screen requires 20480 pixels to be transferred and thus 40960 invocations. To speed up the display driver we implemented a 16 bytes SPI burst transfer primitive. Performance of the display with the 16 byte primitive is still slower than the C version but fast enough to render the game.

[3]Version 2v03: `espruino_2v03_esp32.bin`

[4]Geometric mean execution of time normalized to the native time.

6 Related Work

In this paper we have investigated the use of WebAssembly for programming microcontrollers. Our related work spreads over many areas going from programming languages for microcontrollers to techniques for updating and debugging virtual machines. In the following sections we give a short overview of the most related techniques and how they differ from our experiment.

6.1 Languages for Programming Microcontrollers

The world of microcontrollers and programming languages for programming them has a very rich culture. A wide range of programming languages have been ported to various hardware platforms: Forth [14], BASIC [8], Java [4], Python[15], Lua [7] to name a few. Here we restrict ourselves to compare popular approaches for programming the ESP family of microcontrollers, the microcontroller platform on which we tested WARDuino.

Still, the predominant programming language for programming the ESP processor is the C language [9]. The advantage of using C is that the programs can execute fast and that debugging can be done with a so called JTAG hardware debugger. This (costly) hardware debugger is usually not available for the high level programming languages running on these devices. Downside of the C language is that once a potential bug is found the programmer needs to re-flash the hardware and restart the device completely. Flashing the chip can take long, making the development of microcontroller software a rather slow process.

The Zerynth Virtual Machine [10] is a virtual machine for Python programs running on microcontrollers. They allow accessing all the hardware primitives from within Python and even allow debugging of the C extensions of the Python environment. Unfortunately, as far as we can tell the VM does not allow breakpoints on the Python code itself.

Espruino [19] allows programmers to use a dialect of JavaScript by running a JavaScript interpreter on the chip. Espruino allows the programmer to interactively load code into the interpreter. From version 1.8 basic debugging support is provided for Espruino. The VM is unfortunately too slow to program the device drivers in JavaScript. Therefore, most support for displays and sensors is hardcoded.

C# [6] can be executed on the ESP32 microcontroller on the nanoframework virtual machine. Videos show that the nanoframework supports debugging. From the documentation it only seems possible to make use of the C# language. Finally, the footprint of the virtual machine is quite big. Installing the nanoframework VM seems only possible on a Windows machine. Moreover, the precompiled binaries were no longer available on the github page at the time of writing.

The chip can also be programmed in Lua which is interpreted and uploaded to the chip. In order to speed up the execution of the Lua programs developers can implement components in C and call them from within Lua. The VM is sculpted to the execution of Lua programs.

MicroPython[3] is a highly optimized version of a subset of the Python programming language. It provides on the chip compilation of Python programs. Micro benchmarks showed that MicroPython is much faster than our current implementation of WARDuino. We plan to improve the performance of WARDuino by implementing a JIT compiler. Finally, MicroPython unfortunately does not provide any means for remote debugging.

6.2 Dynamic Software Updates

In literature there are many techniques for dynamic software proposed. Tesone et al propose *gDSU* [17]. A dynamic software updating mechanism for both live programming and production environments. They provide safe update point detection using call stack manipulation. They show that the update mechanism does not incur any overhead of the global performance of the application outside the update window.

Cazolla et al. have made use of static analysis over Java programs to only perform updates when they are considered safe [2]. Compared to other work the disadvantage of this technique is that the statical analysis needs to be performed over the whole code base. This means that the technique does not scale for third party libraries.

Wernli et al. propose a solution by making use of proxies to incrementally replace the updated objects [18]. Unfortunately this creates some problems with respect to object identity during the update window. Many other techniques provide a form of manual update mechanism [11, 13].

It is clear that there are many techniques that could be applied in the context of WARDuino to safely update functions and data even when they are not of the same type. Investigating such techniques is part of future work.

7 Conclusion

In this paper we have presented WARDuino an extension to the WebAssembly standard specifically for programming microcontrollers. These extensions provide WebAssembly with live code updates, remote debugging, and give access to the hardware modules of the microcontroller. Live code updates make sure that only type correct functions can be replaced. Micro benchmarks show that the VM has good performance compared to Espruino, a popular JavaScript VM for programming the ESP32 microcontroller. While performance compared to Espruino is good, there is still a quite large gap with the execution speeds one can expect from C. In future work we aim to reduce this gap by implementing a JIT compiler.

(*Store*)	s	::=	{inst $inst^*$,tab $tabinst^*$,mem $meminst^*$ }
(*Intances*)	$inst$	::=	{func cl^* ,glob v^* ,tab $i^?$,mem $i^?$ }
(*Closure*)	cl	::=	{inst i,code f}
(*Values*)	v	::=	t.**const** c
(*Admin. oper.*)	e	::=	.. | **call** cl | **label**$_n${e^*} e^* **end** | **local**$_n${$i; v^*$} e^* **end**
(*Local contexts*)	L^0	::=	v^* [_] e^*
	L^{k+1}	::=	v^* **label**$_n${e^*} L^k **end** e^*

Figure 4. WebAssembly syntax

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{s; v^*; L^k[e^*] \hookrightarrow_i s'; v'^*; L^k[e'^*]} \text{(STEP-I)}$$

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{s; v_0^*; \mathbf{local}_n\ \{i; v^*\}\ e^*\ \mathbf{end} \hookrightarrow_{d,i} s'; v_0^*; \mathbf{local}_n i; v' * e'^*\ \mathbf{end}} \text{(STEP-LOCAL)}$$

Figure 5. WebAssembly Meta-Rules

(*DBState*)	dbg	::=	{ rs,msg_i,msg_o, s, bp}
(*RunningState*)	rs	::=	PLAY | PAUSE
(*Msg*)	msg	::=	$\emptyset$ | *pause* | *play* | *step* | *dump* | $break^+$ *id* | $break^-$ *id*

Figure 6. WARDuino Debugger Extension

A Formal Specification of WARDuino

In this appendix we give the formal semantics of WARDuino. We specify the formal semantics as an extension of WebAssembly as specified by Haas et al. [5]. Details about the primitives and the type system are omitted here to focus on those parts of the operational semantics which we extended.

In figure 4, we give an overview of the runtime syntax of the stack based WebAssembly virtual machine. The store s consists of a set of module instances, table instances and memory instances. A module instance consists of closures, global variables, tables and memories. A closure is represented by a tuple of the module instance and a code block. Values consist of constants. To elegantly represent the semantics a number of administrative operators are introduces. The most important ones are **local** to indicate a call frame for function invocation (possibly over module boundaries) and **label** witch marks the extend of control construct.

Reductions operate over a configuration $s; v^*; e^*$ which consists of a global store, the local values v^* and the active instruction sequence e^* being executed.

The two reduction rules which govern the order of evaluation are shown in figure 5, The STEP-I rule splits a configuration into its context and its focus and takes one step of the $\hookrightarrow_i$ relation. The second rule STEP-LOCAL explains how to evaluate a function which might reside in a different module. Note that it changes the currently executing module and the local variables.

A.1 Remote Debugging Extensions

To facilitate debugging of WebAssembly programs we extend the semantics with remote debugging constructs. We follow the style for defining a debugger semantics as outlined by Torres et al [12]. The goal of these constructs is to provide lightweight extensions to the operational semantics of WebAssembly which is strong enough to provide the most common remote debugging facilities. We first give an overview of the extensions and then show how these extensions serve as the basis to build more elaborate debugging operations.

In figure 6 we give an overview of our syntactic extensions to the operational semantics of WebAssembly to provide basic remote debugging operations. In the semantics we make abstraction of the underlying communication primitives. A concrete implementation may allow communication over the serial port, an HTTP connection or the SPI bus. For ease of exposition all these possible communication possibilities are modeled through (incoming and outgoing) messages.

The main state of the debugger *dbg* is represented as a 5-tuple which encapsulates: the running state rs, the last incoming message msg_i the last outgoing message msg_o, the WebAssembly store s and a set of breakpoints bp. The running state indicates whether the virtual machine is in the paused state (PAUSE) or is running (PLAY).

The semantics of the debugger consists of a transitioning system where each state consists of a debugger state *db*, zero or more local values v^* and a focused operation e^*, operating in a module instance i.

The reduction rules for remote debugging are shown in figure 7:

vm-run When in the PLAY state with no incoming or outgoing messages and no applicable breakpoints, the debugger takes one small step of the small step operational semantics $\hookrightarrow_i$.

db-pause When the debugger receives a *pause* message, the debugger transitions to the PAUSE state. Note that it is allowed to transition from any previous state to the paused state. After transitioning to the paused state, the rule VM-RUN is no longer applicable.

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^* \qquad id(e^*) \notin bp}{\{\text{PLAY}, \emptyset, \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{\text{PLAY}, \emptyset, \emptyset, s', bp\}; v'^*; e'^*} \text{(VM-RUN)}$$

$$\frac{}{\{\text{RS}, pause, \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{\text{PAUSE}, \emptyset, \emptyset, s, bp\}; v^*; e^*} \text{(DB-PAUSE)}$$

$$\frac{msg = json(\text{bp}, s, v^*, e^*)}{\{\text{PAUSE}, dump, \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{\text{PAUSE}, \emptyset, msg, s, bp\}; v^*; e^*} \text{(DB-DUMP)}$$

$$\frac{}{\{\text{PAUSE}, run, \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{\text{PLAY}, \emptyset, \emptyset, s, bp\}; v^*; e^*} \text{(DB-RUN)}$$

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{\{\text{PAUSE}, step, \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{\text{PAUSE}, \emptyset, \emptyset, s', bp\}; v'^*; e'^*} \text{(DB-STEP)}$$

$$\frac{}{\{rs, (break^+ id), \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{rs, \emptyset, \emptyset, s, (bp\ U\ id)\}; v^*; e^*} \text{(DB-BP-ADD)}$$

$$\frac{}{\{rs, (break^- id), \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{rs, \emptyset, \emptyset, s, (bp \backslash id)\}; v^*; e^*} \text{(DB-BP-REM)}$$

$$\frac{id(e^*) \in bp}{\{\text{PLAY}, \emptyset, \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{\text{PAUSE}, \emptyset, \emptyset, s, bp\}; v^*; e^*} \text{(DB-BREAK)}$$

Figure 7. WARDuino reduction rules for remote debugging

db-dump In the paused state the debugger can request a dump of the state of the virtual machine. This dump is communicated to the debugging host by means of an outgoing message which next to all the WebAssembly state also contains the breakpoints of the debugger.

db-run When the debugger is in the PAUSE state, the programmer can restart execution by sending a *run* message.

db-step When the debugger receives the *step* message in the PAUSE state, it takes one step ($\hookrightarrow_i$). The debugger remains in the PAUSE state.

db-bp-* In the pause state breakpoints can be added and removed.

db-break When the debugger is in the PLAY state and the *id* of currently executing expression is in the list of breakpoints the debugger transitions to the PAUSE state.

While we model a basic set of stepping primitives, there are a number of traditional breakpoints that can be encoded with the given primitive debugging operations.

step-into This stepping command is offered only when the current instruction is a function call. In order for the debugger client to verify whether this command should be active it can request a dump of the current execution and enable the step-into command in the GUI. Execution of the STEP-INTO command is exactly the same as DB-NEXT.

step-out When the programmer is debugging inside of a function, they might want to step out of the function call. Because the end of a function is an actual instruction in WebAssembly the debugger can inspect the body of the function and add breakpoints for all the exit points of the function. Important here is that the debugger needs to take note of the callstack at the moment a STEP-OUT is requested. To handle recursive calls correctly, the program should only be paused if one of the breakpoints is hit while the callstack has the same height. If the breakpoint is hit on a larger callstack, the program should be resumed (by sending *run*).

step-over Similar to step-into, step-over should only be activated when the next instruction is a call instruction. Instead of following the call the step-over stepping command stops the debugger when the call is finished. The instruction sequence to express step-over with our basic debugging constructs are: take one step to go into the function (DB-STEP), execute the STEP-OUT stepping command.

A.2 Safe Dynamic Code Updates

Next to remote debugging the program we allow the programmer to upload new programs and to update functions, global variables, tables and memory. Figure 8 provides an overview of the reduction rules to dynamically update the WebAssembly program.

upload-s When updating the whole WebAssembly store, the pending breakpoints are reset and a call to the main entry of the program is pushed on the stack. Note that after uploading the new store the debugger is still in the PAUSE state allowing the programmer to add breakpoints before starting the execution.

update-f Partial updates of the store are captured in the UPDATE-F rule. We only show the rule for updating functions, updating globals, tables and memory of a store instance follow the same form. Important is that updates are only allowed in case the type of the values being overwritten corresponds with the updated value.

$$
\begin{array}{llll}
(DBState) & dbg & ::= & \{rs, msg_i, msg_o, s, bp\} \\
(RunningState) & rs & ::= & \textsc{play}|\textsc{pause} \\
(Msg) & msg & ::= & \emptyset|pause|play|step \\
 & & & |dump|break^{+}\ id \\
 & & & |break^{-}\ id \\
 & & & |upload\ s \\
 & & & |update_f\ id_i\ id_f\ code\ f \\
 & & & |update_l\ j\ v
\end{array}
$$

(UPLOAD-S)

$$\frac{id_{main} = main(s')}{\{\textsc{pause}, upload\ s', \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{\textsc{pause}, \emptyset, \emptyset, s', \emptyset\}; (\textbf{call}\ id_{main})}$$

(UPDATE-F)

$$\frac{s' = update_f(s, id_i, id_f, codef)}{\{\textsc{pause}, update_f\ id_i\ id_f\ code\ f, \emptyset, s, bp\}; v^*; e^* \hookrightarrow_{d,i} \{\textsc{pause}, \emptyset, \emptyset, s', bp\}; v^*; e^*}$$

(UPDATE-LOCAL)

$$\frac{}{\{\textsc{pause}, update_l\ j\ v', \emptyset, s, bp\}; v_1^j v v_2^k; e^* \hookrightarrow_{d,i} \{\textsc{pause}, \emptyset, \emptyset, s, bp\}; v_1^j v' v_2^k; e^*}$$

Figure 8. Reduction rules for code updates

update-local Next to updating the store there are also local variables which might need to be updated. Similar to partial updates to the store, local variables can only be updated with values of the same type.

Our update strategy for functions and data is quite simple. We only allow code updates if the underlying types remain the same. While this provides safety, it can still have undesirable effects. For example when updating, in the middle of a recursive function the new base conditions might have already been exceeded. The WARDuino VM does not tackle these kinds of problems. In future work we hope to improve on this by incorporating techniques from work on dynamic software updates [17].

B Debug Dump

An example of the debug dump of a running program discussed in section 5.1.1 can be found in the listing of figure 9.

C Full Benchmark Results

Figure 10 and table 1 show the results of the micro benchmarks discussed in Section 5.3 on page 5. They contain the same results as Figure 3. We see that WARDuino outperforms Espruino on all micro benchmarks by at least a factor 5.

```
{ "pc": "0x559ec7578271",
  "breakpoints": ["0x560c56a0e26b"],
  "mod" : [{
      "functions": [
          {"fidx": "0x4",
           "from": "0x559ec7578251",
           "to": "0x559ec7578256"},
          {"fidx": "0x5",
           "from": "0x559ec7578259",
           "to": "0x559ec7578278"}
      ],
      "glob":[],
      "tab":[], "mem": []
  }],
  "tables": []
  "mem": []
  "callstack": [
      {"type": 0,"fidx": "0x5", "sp": -1,
       "fp": -1,"ra": null},
      ...,
      {"type": 3, "fidx": "0x0", "sp": -1,
       "fp": 0, "ra": "0x559ec7578261"}
  ]
}
```

Figure 9. Debug dump of a running program. `pc` stands for program counter, `fidx` is the function id, `from` and `to` indicate the memory location of a function. The `fp`, `sp` and `ra` are the frame pointer, stack pointer and return address respectively.

An implementation of the same program in native C is be much faster than WARduino. A native C implementation uses registers. WebAssembly is a stack machine and therefore requires a lot of (slow) memory access to get arguments to operations. Using JIT compilation is one of the ways WARDuino could reduce the amount of memory operations and speed up. Implementing JIT compilation for WARDuino is part of our future work.

Acknowledgments

This research is supported by a doctoral fellowship from the Special Research Fund (BOF) of UGent (BOF18/DOC/327).

References

[1] Massimo Banzi. 2008. *Getting Started with Arduino* (ill ed.). Make Books - Imprint of: O'Reilly Media, Sebastopol, CA.

[2] W. Cazzola and M. Jalili. 2016. Dodging Unsafe Update Points in Java Dynamic Software Updating Systems. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 332–341.

[3] Damien P. George. 2014. MicroPython. https://github.com/micropython/micropython.

[4] James Gosling, Bill Joy, and Guy L. Steele. 1996. *The Java Language Specification* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Table 1. Left: Absolute execution times in second for all tests. Right: Execution time of tests normalized to the native C implementation. The geometric mean of Espruinos overhead compared to native is 4,662.8. The geometric mean of WARduinos overhead compared to native is 464.82.

name	Espruino (s)	WARDuino (s)	C (s)	$\frac{\text{Espruino}}{\text{C}}$	$\frac{\text{WARduino}}{\text{C}}$
catalan	897.26	168.29	0.221	4,060.61	761.60
fac	584.85	59.39	0.117	5,001.22	507.84
fib	551.45	45.98	0.125	4,420.04	368.51
gcd	750.53	47.53	0.216	3,477.83	220.26
primes	649.76	87.85	0.127	5,112.61	691.28
tak	676.47	48.81	0.105	6,445.74	465.10

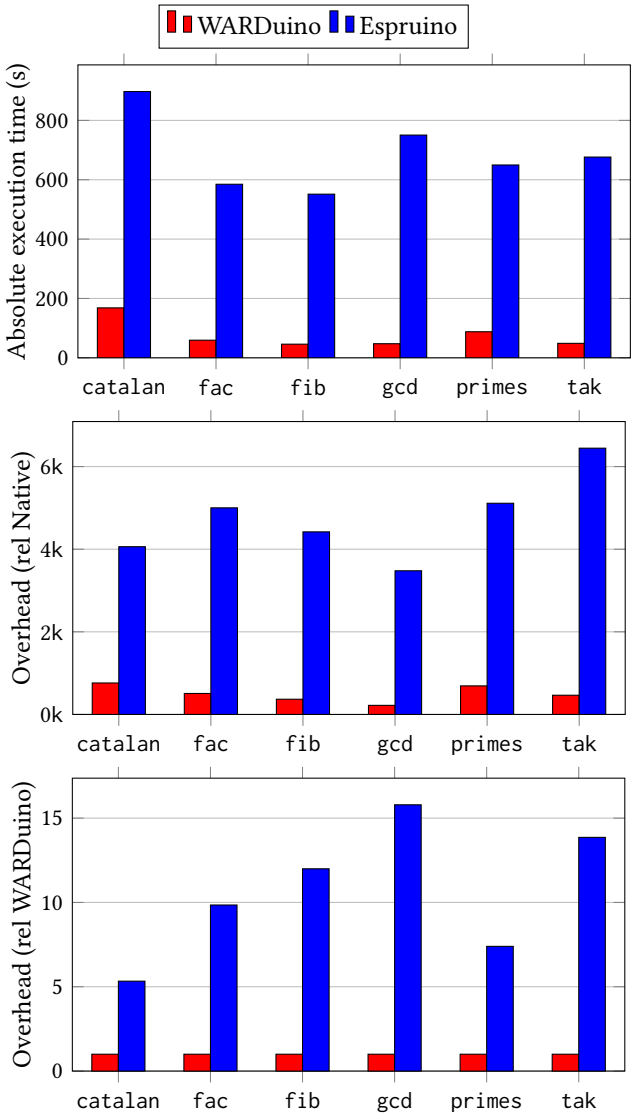

Figure 10. The execution times of WARDuino and Espruino. From top to bottom: absolute results in seconds, time normalized to native C execution time, time normalized to the WARDuino execution time.

[5] 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. *SIGPLAN Not.* 52, 6 (June 2017), 185–200.

[6] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. 2010. *C# Programming Language* (4th ed.). Addison-Wesley Professional.

[7] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 1996. Lua&Mdash;an Extensible Extension Language. *Softw. Pract. Exper.* 26, 6 (June 1996), 635–652. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P

[8] John G Kemeny, Thomas E Kurtz, and David S Cochran. 1968. *Basic: a manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System.* Dartmouth Publications.

[9] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language.* Prentice Hall Press, Upper Saddle River, NJ, USA.

[10] Kinzica Ventures LLC. [n.d.]. Zerynth Virtual Machine. https://www.zerynth.com/zerynth-virtual-machine/. Accessed: 2019-09-04.

[11] J. Kramer and J. Magee. 1985. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering* SE-11, 4 (April 1985), 424–436.

[12] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. 2019. Multiverse Debugging: Non-deterministic Debugging for Non-deterministic Programs. In *ECOOP 2019 (Leibniz International Proceedings in Informatics (LIPIcs)).* Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[13] P. Oreizy, N. Medvidovic, and R. N. Taylor. 1998. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering.* 177–186.

[14] Elizabeth D. Rather and Charles H. Moore. 1976. The FORTH Approach to Operating Systems. In *Proceedings of the 1976 Annual Conference (ACM '76).* ACM, New York, NY, USA, 233–240. https://doi.org/10.1145/800191.805586

[15] Guido Rossum. 1995. *Python Reference Manual.* Technical Report. Amsterdam, The Netherlands, The Netherlands.

[16] Bjarne Stroustrup. 2013. *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.

[17] Pablo Tesone, Guillermo Polito, Noury Bouraqadi, Stephane Ducasse, and Luc Fabresse. 2018. Dynamic Software Update from Development to Production. *Journal of Object Technology* 17, 1 (Nov. 2018), 1:1–36. https://doi.org/10.5381/jot.2018.17.1.a2

[18] Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz. 2012. Incremental Dynamic Updates with First-Class Contexts. In *Objects, Models, Components, Patterns*, Carlo A Furia and Sebastian Nanz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–319.

[19] Gordon Williams. 2014. Espruino. https://www.espruino.com/.