

# Interactive Programming for Microcontrollers by Offloading Dynamic Incremental Compilation

Fumika Mochizuki  
fumika.maejima@csg.ci.i.u-  
tokyo.ac.jp  
The University of Tokyo  
Tokyo, Japan

Tetsuro Yamazaki  
yamazaki@csg.ci.i.u-tokyo.ac.jp  
The University of Tokyo  
Tokyo, Japan

Shigeru Chiba  
chiba@csg.ci.i.u-tokyo.ac.jp  
The University of Tokyo  
Tokyo, Japan

## Abstract

Interactive execution environments are suitable for trial-and-error basis programming for microcontrollers. However, they are mostly implemented as interpreters to meet microcontrollers' limited memory size and demands for portability. Hence, their execution performance is not sufficiently high. In this paper, we propose offloading dynamic incremental compilation and linking to a host computer connected to a microcontroller. Since the computing resources of the host computer are sufficient to execute incremental dynamic compilation, they are used to enhance the relatively poor computing resources of the microcontroller. To show the feasibility of this idea, we design a small programming language named *BlueScript* and implement its interactive execution environment. Our experiment reveals that BlueScript executes a program one to two orders of magnitude faster than MicroPython, while its interactivity is comparable to that of MicroPython despite using dynamic incremental compilation.

**CCS Concepts:** • **Software and its engineering** → *Dynamic compilers*; • **Incremental compilers**; • **Computer systems organization** → *Embedded software*.

**Keywords:** Interactive Programming, Dynamic Compilation, Embedded systems, Microcontrollers, TypeScript

## ACM Reference Format:

Fumika Mochizuki, Tetsuro Yamazaki, and Shigeru Chiba. 2024. Interactive Programming for Microcontrollers by Offloading Dynamic Incremental Compilation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24)*, September 19, 2024, Vienna, Austria. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3679007.3685062>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org). *MPLR '24, September 19, 2024, Vienna, Austria*

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1118-3/24/09

<https://doi.org/10.1145/3679007.3685062>

## 1 Introduction

An interactive execution environment of programming language, or a REPL (Read-eval-print loop), is known as being suitable for trial-and-error basis programming. For artificial intelligence and data science, Jupyter [22] is widely used as such an interactive environment in Python. For programming education, Scratch [17] is a popular interactive language and environment. A web browser such as Google Chrome provides an interactive environment in JavaScript for debugging.

In contrast, in the field of microcontroller programming, a non-interactive development environment has been mainly adopted. A program is written in the C or C++ language, statically compiled, and linked on a host machine/computer. Then, the compiled binary is written to a flash memory of a target microcontroller connected through a serial cable, and the microcontroller is rebooted to execute the program. When the behavior of the program is not satisfactory, programmers or developers repeat this linear process from the beginning. This traditional development process is still dominant, but interactive environments are becoming popular even for programming a microcontroller. Interpreter-based environments, such as MicroPython [3] and Espruino [23], have been actively developed and used. They provide a REPL, and their users can enjoy their interactivity from a host machine connected to a microcontroller.

However, such interactive environments for microcontrollers are often slow since they are interpreters. An interpreter is suitable for interactive programming. It can be implemented to run with a small amount of memory and keep portability independent of CPU architecture. This is an advantage but implies low execution performance. Dynamic incremental compilation, or dynamic compilation, would be a solution but it damages the size of an interpreter's memory footprint and portability. It would also increase response time and worsen interactivity due to the limited computing power of microcontrollers.

To address this problem, we propose to exploit the computing resources of a host machine connected to a microcontroller particularly through a wireless network such as Bluetooth. Usually, a microcontroller is connected to a host machine or computer during software development. We offload dynamic incremental compilation and linking to this

host machine. To investigate our idea in a practical interactive environment for microcontrollers, we design a small programming language named *BlueScript* and implement its interactive execution environment for Espressif Systems' ESP32 microcontroller. Although we use an off-the-shelf C compiler for ESP32 as a backend native-code generator to reduce development costs, our execution environment achieves smooth interactivity. The execution of BlueScript programs is only up to 10 times slower than C programs and one to two orders of magnitude faster than MicroPython's ones. Our contributions are twofold. One is to illustrate that offloading dynamic incremental compilation and linking still enables interactive programming for microcontrollers while significantly improving execution performance. The other is to implement a prototype of the interactive execution environment for BlueScript and design experiments to assess its interactivity.

## 2 Interactive Programming for Microcontrollers

Even when programming a microcontroller, a trial-and-error basis development is effective and useful, and it is enabled by interactive execution environments of programming languages. Suppose that we are programming a microcontroller for controlling a toy car. We first use the C language and its traditional non-interactive environment. This toy car is equipped with two motors and a front camera. Since the two motors are separately controlled and connected to two rear wheels, the car can move forward and backward and change its direction of travel. We write a program so that the car will periodically take a photo and turn to the right when it finds a red object in front of it, for example, a red plastic ball. Listing 1 is an example of this program. It is written in the C language. The main function registers an event handler `periodic_task`, which is invoked every 500 msec. The event handler takes a front photo by calling `take_picture`, and calls `find_red_object` to count the number of red pixels and determine whether a red object is found in the given photo image. If a red object is found, the car turns to the right to avoid collision. Otherwise, the car keeps going straight ahead.

The calls to `set_speed` in line 14, 18, and 19 change the motor speed. The second argument specifies the speed. Programmers have to carefully choose an appropriate value for the second argument so that the car will run smoothly. To choose it, programmers might want to take a trial-and-error approach. They might write a program with some initial values for that second argument, compile it, and run it to move the car. Then, they might observe the car moving and change the second argument to new values. They might iterate these steps until they find good values for the second argument and the car runs nicely. Iterating these steps is, however, time-consuming and tedious. For example, these steps take

```

1 bool find_red_object(uint16_t* img) {
2     int red_count = 0;
3     for (int row = 0; row < HEIGHT; row++) {
4         for (int col = 0; col < WIDTH; col++) {
5             int hue = get_hue(img[row*col+col]);
6             if (hue > 45)
7                 red_count++;
8         }
9     }
10    return red_count > RED_MINIMUM;
11 }
12
13 void go_straight() {
14     set_speed(BOTH, 80);
15 }
16
17 void go_right() {
18     set_speed(LEFT, 50);
19     set_speed(RIGHT, 10);
20 }
21
22 void periodic_task() {
23     uint16_t* img = take_picture();
24     if (find_red(img))
25         go_right();
26     else
27         go_straight();
28 }
29
30 void main() {
31     timer_t timer;
32     timer_create(&periodic_task, &timer);
33     timer_start_periodic(timer, 500);
34 }

```

**Listing 1.** Controlling a toy car

several seconds to a minute when we use ESP-IDF (Espressif IoT Development Framework) [7], which is the standard development environment provided by Espressif for the ESP32 microcontroller. It is slow to build executable binary code, write it into the flash memory of the microcontroller, and reboot the microcontroller for restarting a program.

An interactive execution environment mitigates this inefficiency. For example, MicroPython is available on the ESP32 microcontroller. It provides a REPL accessible from a host computer connected to the microcontroller through a serial cable or Wi-Fi network. Through this REPL, programmers can give code fragments after the prompt one by one to interactively define functions, redefine them, and run them. Listing 2 shows a log of programming a toy car through the MicroPython REPL. `>>>` and `...` in Listing 2 are prompts. Program texts following them are sent to the REPL. A programmer first defines four functions one by one (Line 1-12) and run them by registering `periodic_task` as a timer-event handler (Line 14-15). Then the programmer observes the move of the toy car. Since he/she thinks that the car moves

```

1 >>> def find_red_object(img):
2 ...     # omit
3 >>>
4 >>> def go_straight():
5 ...     set_speed(BOTH, 80)
6 >>>
7 >>> def go_right():
8 ...     set_speed(LEFT, 50)
9 ...     set_speed(RIGHT, 10)
10 >>>
11 >>> def periodic_task():
12 ...     # omit
13 >>>
14 >>> timer = Timer()
15 >>> timer.init(mode=Timer.PERIODIC, callback=
    periodic_task, period=500)
16 >>> # Check behavior of toy car.
17 >>>
18 >>> def go_straight(): # Redefine the function.
19 ...     set_speed(BOTH, 40)
20 >>> # Check behavior of toy car again.

```

**Listing 2.** Programming a toy car through the REPL in MicroPython

too fast, he/she redefines the `go_straight` function to pass a smaller value 40 to `set_speed` (Line 19-20). If `go_straight` did not pass a numeric literal but the value of a global variable such as `SPEED`, the programmer would not redefine a function but simply change the value of that global variable. This change is also easy through a REPL. This redefinition is immediately reflected on the car's move, and the programmer can observe its effect. If the car's move is not satisfactory, the programmer can redefine `go_straight` again. He/She can iterate these steps until the move is satisfactory. This iteration is not tedious or time-consuming.

A problem of interactive execution environments for microcontrollers is their execution speed. They often use a simple interpreter since only a limited amount of memory is equipped on a microcontroller. Their execution speed is much slower than the native machine code compiled from a program written in the C language. For example, the function `find_red_object` in the C language takes 0.45 seconds, but a MicroPython function equivalent to `find_red_object` takes 11.5 seconds on the ESP32 microcontroller, which operates at 240 MHz. The MicroPython function would be too slow to change the car's direction of travel before it collides with a red object in front of it. The travel speed of the toy car must be significantly reduced.

### Dynamic Incremental Compilation

A promising approach to accelerating the execution speed in an interactive execution environment is dynamic incremental compilation, which can trace its origin back to the 1970s [15]. Its idea is to *incrementally* compile only a new code

fragment interactively given by programmers and *dynamically* link the compiled binary to the rest of the codebase that already exists. This approach has been sophisticated [10, 25], and it is now widely adopted by a number of language virtual machines as dynamic compilation or *just-in-time* compilation, where a code fragment is dynamically selected for compilation not only when it is given by programmers but also when it is recognized as frequently executed code, so called *hotspot*, during runtime. Furthermore, modern dynamic compilers adaptively change how aggressively they optimize code to match a trade-off between compilation time and resulting speedup.

However, dynamic incremental compilation is not widely used by interactive execution environments for microcontrollers. Some execution environments like MicroPython [4] support it, but its capability is limited since the environments must contain an optimizing compiler and a linker. These components increase the memory footprint of the environment, but this increase of memory footprint is not accepted without careful consideration. A microcontroller is equipped with only a limited amount of memory. The SRAM size of the ESP32 microcontroller is 520 KB, and that of the RP2040 microcontroller for Raspberry Pi Pico is only 264 KB. These memories must be shared with application programs. Note that, from the viewpoint of product design, the total memory footprint should be reduced as much as possible to minimize product cost.

This viewpoint is also applied to the code size of execution environments. Although those microcontrollers support up to 16 MB flash memory, where the executable binary code is stored, it must be shared with applications, and hence, a smaller footprint of the execution environment is also more desirable. For example, the binary size of the V8 JavaScript engine for macOS is 1.5 times bigger than the V8 engine without dynamic compilation. Also, the Ruby virtual machine supporting the YJIT dynamic compiler is 1.3 times bigger than the original one for macOS.

Furthermore, some developers might be afraid that dynamic incremental compilation would degrade the interactivity of execution environments for microcontrollers. Since the clock speed of microcontrollers is an order of magnitude slower than high-performance processors for smartphones or server computers, it is challenging to dynamically compile code during runtime to keep a short response time.

## 3 Offloading Dynamic Incremental Compilation

To utilize incremental dynamic compilation in interactive execution environments on microcontrollers, we propose offloading it onto a host machine that is a computer used by programmers as a console to access a microcontroller. The host machine is connected to the microcontroller through a

wireless network or a serial cable, and it has larger computing resources than the microcontroller. Since these computing resources are sufficient to execute incremental dynamic compilation, they are used for enhancing relatively poor computing resources of the microcontroller.

We offload not only compilation but also linking on a host machine. This differs from a traditional approach where source code is compiled into a shared object or a dynamic library on a host machine and is dynamically loaded and linked on a target machine by `dlopen`. We offload all the steps except writing executable code on the memory of a microcontroller and reduce the memory footprint of an execution environment on a microcontroller as much as possible.

### 3.1 Overview

An interactive execution environment that offloads incremental dynamic compilation consists of two components. One component runs on a host machine and provides a REPL and a compiler. The other one runs on a microcontroller, and it is a collection of runtime routines that are needed for receiving compiled native code from the host machine and executing it. These two components are connected through a network, which is a Bluetooth wireless network in the case of our prototype mentioned later. Before rebooting the component on a microcontroller, its runtime routines are compiled, built, and written onto the flash memory of the microcontroller. The compiled runtime routines are sent from the host machine through a serial cable to the microcontroller. After rebooting, the serial cable is not necessary if the two components are connected through a wireless network.

When a programmer gives a new source-code fragment through a REPL on a host machine, a compiler on the host machine compiles it into native code for a microcontroller. We call this native code a *loading unit*. The compiler performs *incremental* compilation. If necessary, it refers to code fragments previously given by the programmer. To reduce engineering efforts, we use an existing compiler tool-chain for cross-compilation to the target microcontroller. Our prototype mentioned later uses an existing C compiler as a backend. Our compiler translates source code into a C program, and a C compiler compiles it into native code in the ELF (Executable and Linkable Format). Using an existing off-the-shelf compiler indirectly improves the portability for different microcontrollers.

Then, the compiled native code, or a loading unit, is modified on the host machine to be linked to the rest of the code. After being linked, this loading unit is sent *as it is* to the microcontroller, and it is written to memory on the microcontroller. Finally, a runtime routine calls the entry point of that loading unit to execute it. After finishing the execution, the runtime routine waits until another loading unit is sent from the host computer. The REPL also waits until the programmer gives a new code fragment.

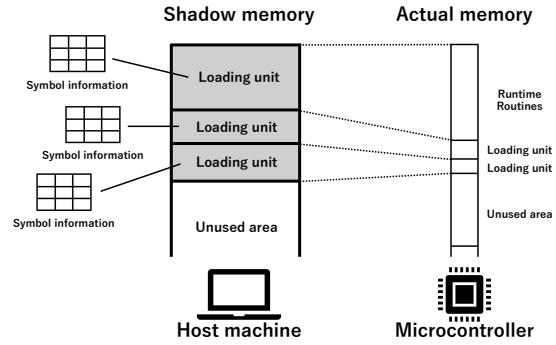


Figure 1. Shadow memory

### 3.2 Shadow Memory

To execute linking on a host machine, we maintain an abstract memory image, which we call *shadow memory*, on a host machine (Figure 1). It is an abstract copy of the memory image on a microcontroller, and it also holds symbol information needed for linking compiled native code to the rest of the native code that is already running on a microcontroller.

The shadow memory consists of several regions, which correspond to memory regions on a microcontroller. The shadow memory initially consists of regions that represent unused free areas on a microcontroller’s memory. Later, those unused areas will be gradually converted into regions containing compiled native code that we call a *loading unit*. The shadow memory also initially includes a region that corresponds to the memory region containing the runtime routines of the execution environment on a microcontroller. It may be a memory region on the flash memory of a microcontroller, where executable code is written before rebooting. This region in the shadow memory holds a symbol table that indicates the addresses of the entry points of the runtime routines.

A loading unit, which is native code obtained by compiling a source-code fragment interactively given by a programmer through a REPL, is linked on the shadow memory. First, a new region is allocated for this loading unit from an unused free area. A new memory region is allocated on a microcontroller, and a corresponding region is created in the shadow memory on a host machine. The loading unit is copied into this new region in the shadow memory on a host machine. The symbol information of the loading unit is attached to the region. Because a loading unit in our prototype system is executable binary in the ELF, a symbol table stored in the ELF section `.symtab` and a relocation table in the ELF section starting with `.rela` are attached to the region. A symbol table is a map that associates a symbol name with its address, which consists of a section name and an offset from the beginning of the section. A relocation table lists the locations of unresolved addresses in a section, for example,

```

1 let led: GPIO = new GPIO(23);
2 let isLit: boolean = false;
3
4 function toggleLED() {
5     led.set(isLit ? 0 : 1);
6     isLit = !isLit;
7 }
8
9 setInterval(toggleLED, 500);

```

**Listing 3.** A program in BlueScript for blinking an LED

the `text` section for the relocation table in the `.rela.text` section. Its entries are an offset, a symbol name, how a value is computed, and so on.

Then, the loading unit is modified in the shadow memory to be linked. All the entries in the relocation tables are resolved, and computed values are stored at their addresses in the region in the shadow memory on a host machine. During this symbol resolution, symbol information for other loading units in their regions are referred to. After this symbol resolution, the resulting machine code in the shadow memory is sent to a microcontroller, and it is written in the corresponding memory region on the microcontroller. When the machine code in other regions is updated, those new values are also sent to a microcontroller, and they overwrite previous values.

Note that the symbol information is never sent to a microcontroller. It is just kept in the shadow memory on a host machine so that it will be reused later when a new loading unit comes in to be linked or when an existing loading unit is modified for further optimization.

### 3.3 The BlueScript Language

As a prototype for exploring our idea of offloading, we have developed a new small language named *BlueScript*. We use this language for exploring our idea in a practical interactive environment for microcontrollers. The design goal of BlueScript is to balance sufficient flexibility for interactive programming and adequate execution performance on performance-poor microcontrollers. BlueScript borrows syntax from TypeScript, although it only uses a subset of TypeScript's syntax. Listing 3 is an example of a BlueScript program. It can be read mostly as a normal TypeScript program, but the semantics of BlueScript are more static than TypeScript. Most of the dynamic features available in TypeScript are not supported in BlueScript. For example, BlueScript does not support `eval`, `apply`, prototype-based inheritance, or structural typing. BlueScript adopts nominal typing and supports classes with simple single inheritance as well as function redefinition, but forbids redefining existing classes.

The language adopts simple gradual typing [24]. It supports primitive types such as 32-bit signed integer, 32-bit floating-point number, boolean, and a character string. Their

**Table 1.** Built-in classes and functions

GPIO class	Control general-purpose input/output ports.
PWM class	Access a pulse width modulation controller.
Display class	control a display (M5Stack only).
Timer functions	<code>setInterval</code> , <code>clearInterval</code> , <code>setTimeout</code> , and <code>clearTimeout</code> .
Button function	<code>buttonOnPressed</code> .

type names are `integer` (or `number`), `float`, `boolean`, and `string`, respectively. Unlike TypeScript, an integer and a floating-point number are distinguished and treated as different primitive types. The language also supports an array and an instance of a class. The current class system of BlueScript is static and provides minimal functionality. It only supports single inheritance, and an interface is not supported. A class type is nominally treated, and subtype relations are determined by considering only type names. Adding a new method or field is currently not supported.

Since simple gradual typing is adopted, the language also supports any type. A value of any can be any type of value. The type checker of BlueScript performs type inference, and it infers any type when no other types are determined because, for example, a type annotation is not given by a programmer. If necessary, the type checker may treat an expression (or a variable) of any type as being compatible with any other type of expression, and vice versa. Since this implies implicit type conversion at runtime, this may raise a runtime type error.

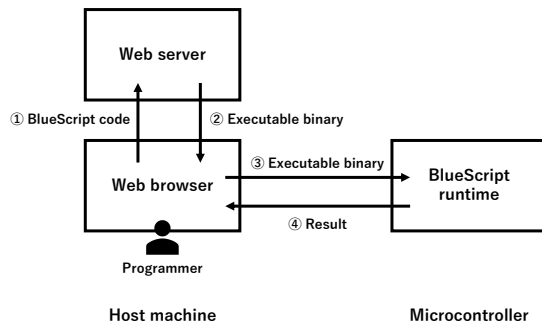
When an element type is not explicitly given, an array object is created as an array of any type; its element type is any. The index of an array element must be `integer` type. When an array element is accessed, it is checked at runtime whether a given index is inside the boundaries of the array. Hence, an array access may raise a runtime error. BlueScript also provides arrays of primitive types, such as an array of `integer` or `float`. All the elements have the same type.

A function is declared by a function declaration starting with `function` or by the arrow notation `=>` as in TypeScript. Currently, however, a function expression cannot capture a local variable. It cannot form a closure. The statements currently supported by BlueScript are expression statements, block statements, and `const`, `let`, `if`, `while`, `for`, and `return` statements. They also include non-labeled `break` and `continue`. `for...in` or `for...of` statements are not supported. `try...catch` statements or `async` functions are not supported either.

Unlike TypeScript, BlueScript currently does not provide a module system. Neither `import` nor `require` is available. BlueScript provides only a single global name scope where several built-in library functions and classes, such as `GPIO` and `PWM`, are available for accessing hardware components as listed in Table 1.



**Figure 2.** A notebook-style REPL for BlueScript



**Figure 3.** The interactive execution environment for BlueScript.

### 3.4 Implementation

We have implemented an interactive execution environment for the BlueScript language. The target microcontroller is ESP32, a 32bit microcontroller by Espressif Systems. This execution environment provides a notebook-style REPL similar to the Jupyter notebook<sup>1</sup>. As shown in Figure 2, a programmer can write a source-code fragment in a rectangular cell in a web page in a web browser on a host machine. Then, when the programmer clicks on the run button to the left of the cell, the code written in the cell is executed on an ESP32 microcontroller connected to the host machine through Bluetooth. The output of the execution is sent back to the web browser, and it is displayed on the right side of the web page. If the output is an error message, it is displayed under the cell. The wireless communication by Bluetooth between a host machine and a microcontroller may make it easy to program a microcontroller that controls the driving of a toy car, the flight of a drone, or the motion of a robot since we do not have to connect it to a host machine by a limited length of wire.

As already mentioned in Section 3.1, the execution environment consists of two components: one on a host machine and the other on a microcontroller. See Figure 3. The component on a host machine runs on Node.js and works as a web

server that serves a web page for a notebook-style REPL. The web browser is responsible for the communication between a host machine and a microcontroller. Since it uses the Web Bluetooth API, a programmer must use a web browser supporting this API such as Google Chrome. The component for a microcontroller runs on FreeRTOS with libraries provided by ESP-IDF (Espressif IoT Development Framework).

**Compilation.** The BlueScript compiler is included in the host-machine component of the execution environment. It is implemented in TypeScript and runs on Node.js. It uses the Babel parser<sup>2</sup> for parsing a BlueScript program. A non-supported statement or expression is accepted by Babel but it is treated as an error after parsing. Since a BlueScript program is incrementally given to the compiler, the compiler receives a source-code fragment one by one and translates it into a program in the C language. The compiler maintains a global name table so that functions and global variables can be accessed from other code fragments given later. Then, that C program is compiled with the `-O2` option by a cross-compiler provided by ESP-IDF, which is based on the GNU C compiler. The compiler generates executable binary code in the ELF. After being linked, it is sent to ESP32 and written in an executable RAM area of the device. The data section generated by the compiler is written in a non-executable RAM area. Note that ESP32 partly uses the Harvard memory architecture.

A function in BlueScript is translated into a function in the C language. Currently, this function in the C language is indirectly invoked so that redefinitions of a function in BlueScript can be easily implemented. Only the calls to a function bound to a const variable are compiled into direct function calls in the C language. The top-level statements included in a given source-code fragment are collected and translated into the body of one function in the C language. This function is the entry point of the loading unit generated from the given fragment. It is invoked when the loading unit is written to the memory on a microcontroller.

The integer type and the boolean type in BlueScript are translated into the `int32_t` type in the C language. The float type in BlueScript is translated into the `float` type. A string, an array, and an instance of a class are objects in BlueScript. They are translated into a heap array in the C language. The first 32 bits of this heap array are the header. The upper 30 bits in the header are used as a pointer to the type (or class) information of that object. 2 bits in the header are used as mark bits during garbage collection. The rest of the elements of the heap array are used to store a property or an array element in BlueScript. The memory layout in the heap array is statically determined for its type or class in BlueScript.

The type system of BlueScript guarantees that a statically typed expression (or variable) results in a value of (a subtype

<sup>1</sup><https://jupyter.org>

<sup>2</sup><https://github.com/babel/babel>

of) that type when that expression is evaluated at runtime if no type error is reported during compilation. Thus, the C program that BlueScript code is translated into does not include runtime type checking except an expression of the any type. It runs without runtime penalties for type checking.

**Gradual Typing.** The any type is compatible with any type. An expression statically typed as the any type can be used where an expression of any other type is expected, and vice versa. For example, it can be used where an integer expression is expected. On the other hand, an integer expression can be used where an any expression is expected. Thus, when an expression of the any type is translated where an expression of other types  $t$  is expected, a runtime type check is inserted to assure that the value is of that type  $t$ . When the type of an expression is not any and it is translated where an expression of the any type is expected, that expression is wrapped in type conversion into any. For example, this BlueScript code:

```
const a: any = 3
const b: integer = a + 4
```

is translated into C code equivalent to the following pseudo code:

```
lvars[0] = int_to_any(3)
int32_t b = any_to_int(add_any(lvars[0],
                             int_to_any(4)))
```

where `lvars` is an array holding the values of local variables that are pointers or any-type values. `int_to_any` and `any_to_int` are functions to convert an integer value into an any value and vice versa. `add_any` is a function to add two any values. Since the left operand of `+` is statically typed as any, the right operand is converted into any.

A value of the any type is implemented by a classic technique called *pointer tagging*. It is a 32bit pointer to a heap array, but the lower 2 bits are used as a tag. An integer value and a float value are packed into the upper 30 bits with a tag for identifying their types. Thus, when integer is converted into any, its value is cast from 32 bits to 30 bits. When float is converted into any, its precision is reduced from 32 bits to 30 bits. The exponent loses 2 bits and becomes 6 bits after the conversion. Furthermore, when a value is stored in an array or an instance of a class, the value is converted into an any-type value. Only when a value is stored in an integer, float, or byte array, the value is stored as it is. No runtime overhead for type conversion implies.

**Garbage Collection.** The current execution environment for BlueScript provides a mark-and-sweep garbage collector. It runs on a microcontroller. 2 bits in an object header are used as mark bits for garbage collection. The garbage collection root is implemented by a technique called *shadow stack* [13]. During the marking phase, an object is colored black, white, or gray. A gray object is an object that is reachable from the root but not yet scanned. The reachability from this

gray object is not examined. A black object is an object that is reachable from the root and scanned. The other objects are white. The collector first performs depth-first-traversal to first color objects gray and then black. If a stack overflows, the collector stops the traversal. It scans the whole heap memory from the top to the bottom, and whenever it finds a gray object, it restarts the depth-first-traversal from that gray object until no gray object is found [16].

The garbage collection for BlueScript can be interrupted. When a hardware interrupt occurs during the marking phase, the tri-color marker is interrupted, and an interrupt handler is invoked. While the interrupt handler is running, when a pointer to a white object is stored in a black object, that white object is changed to gray and pushed into the marker's stack. This is performed by a write barrier inserted before every store operation.

The garbage collector currently reclaims only the memory occupied by an object. It does not reclaim the memory occupied by executable native code. Extending the garbage collector to reclaim the memory occupied by executable native code is our future work.

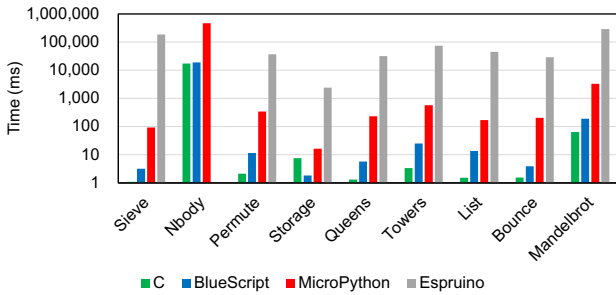
**Libraries.** BlueScript provides a built-in library. To control hardware, programmers can use classes and functions listed in Table 1. The `setInterval` function repeatedly calls a given function at specified intervals. It adds the given function to a table, and the functions in the table are periodically called by a separate thread of FreeRTOS. Since this thread is bound to the same core as the main application thread, when that thread is executing the function passed to `setInterval`, the main application thread is suspended. The functions passed to `setInterval` and the main program are mutually exclusive in BlueScript.

## 4 Experiments

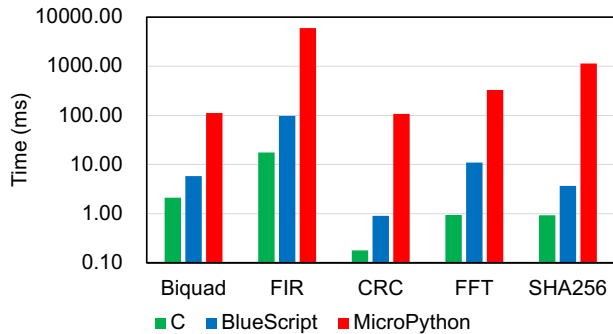
We use our prototype implementation of BlueScript language to conduct experiments. We compare the execution speed, the response time, and the code size of the language's execution environment on a microcontroller, between BlueScript, MicroPython, Espruino, and the C language. Espruino is a JavaScript virtual machine for microcontrollers. Our research question is whether or not our idea of offloading incremental dynamic compilation achieves both smooth interaction and fast execution speed on modern personal computers and microcontrollers. We use MicroPython V1.19.1 and Espruino 2V20 for experiments. The C compiler is `xtensa-esp32-elf-gcc (crosstool-NG esp-2022r1) 11.2.0`. As a target microcontroller, we use M5Stack Fire. It is an IoT development kit based on the ESP32-D0WDQ6 microcontroller with 520 KB of RAM, 16 MB of Flash memory, and 8 MB of PSRAM. As a host machine, we use MacBook Pro, which is equipped with Apple M1 Pro, 16 GB of memory, and 512 GB of storage.

**Table 2.** Benchmarks programs for execution times (the upper programs are from “Are we fast yet?” and the lower ones are from ProgLangComp)

Name	LOC	Description
Bounce	93	Simulates a ball bouncing within a box.
List	79	Recursively creates and traverses lists.
Mandelbrot	83	Calculates the classic fractal.
NBody	185	Simulates the movement of planets in the solar system.
Permute	42	Generates permutations of an array.
Queens	61	Solves the eight queens problem.
Sieve	38	Finds prime numbers based on the sieve of Eratosthenes.
Storage	57	Creates and verifies a tree of arrays to stress the garbage collector.
Towers	83	Solves the Towers of Hanoi game.
Biquad	43	Converts waveform (floating point) by a digital biquad filter.
FIR	55	Converts waveform (floating point) by an FIR filter.
CRC	113	Computes CRC32-IEEE checksums using a precomputed table.
FFT	97	Computes FFT (fixed point, complex pair of int16).
SHA256	175	Computes SHA256 hashes.



**Figure 4.** Execution time of the “Are we fast yet?” benchmark suite



**Figure 5.** Execution time of the ProgLangComp benchmark suite

### 4.1 Execution Time

A BlueScript function equivalent to `find_red_object` in Listing 1 runs in 0.57 seconds, while the original function in the C language runs in 0.45 seconds, and the MicroPython equivalent runs in 11.5 seconds. We also measure the execution time of programs taken from two benchmark

suites listed in Table 2, and compare the times among BlueScript, MicroPython, Espruino, and C. One benchmark suite is the “Are we fast yet?” benchmark suite [19]<sup>3</sup>. This benchmark suite was developed to compare execution performance among various languages. We use its micro benchmark programs in Python and JavaScript and write equivalent BlueScript and C versions. The other is the ProgLangComp benchmark suite [21]<sup>4</sup>. It is a collection of programs written for evaluating the performance of the ESP32 microcontroller. They perform signal processing and hash functions, which are regarded as typical computation by microcontrollers. We use MicroPython programs from this suite and write equivalent C and BlueScript programs. We fully type-annotate all the BlueScript programs used for the experiment.

For BlueScript, we compile and link a whole BlueScript program and runtime routines all at once on a host machine. The resulting executable binary is written on flash memory before a microcontroller is booted to run. For MicroPython and Espruino, we run benchmark programs according to the documents on their official pages. We download their virtual machines from the official page and install them on the microcontroller before rebooting. Then, we write each benchmark program on the host machine and send and execute it to the microcontroller. For the C language, we use the ESP-IDF build tool to compile and execute benchmark programs. We give the `-O2` option to the compiler.

Figure 4 shows the execution times of the “Are we fast yet?” benchmark suite. We compare C, BlueScript, MicroPython, and Espruino. The execution of Nbody by Espruino is timeout. Figure 5 shows the execution times of the ProgLangComp benchmark suite. For this, we compare C, BlueScript, and MicroPython. All the execution times are the means of five runs. Note that the Y axes are log scales.

<sup>3</sup><https://github.com/smarr/are-we-fast-yet>

<sup>4</sup>[https://github.com/ignasp/ProgLangComp\\_onESP32](https://github.com/ignasp/ProgLangComp_onESP32)

```

1 // Code fragment #1
2 let dspl = new Display();
3 let colorWhite = dspl.color(255, 255, 255);
4 let colorRed = dspl.color(255, 0, 0);
5 dspl.showIcon(dspl.ICON_HEART, colorRed,colorWhite
  );
6 print("$$");
7
8 // Code fragment #2
9 dspl.showIcon(dspl.ICON_HEART, colorRed,colorWhite
  );
10 dspl.fill(colorWhite);
11 print("$$");
12
13 // Code fragment #3
14 let isSmall = false;
15 setInterval(() => {
16   if (isSmall) {
17     dspl.showIcon(dspl.ICON_SMALL_HEART, colorRed,
18       colorWhite);
19   } else {
20     dspl.showIcon(dspl.ICON_HEART, colorRed,
21       colorWhite);
22   }
23   isSmall = !isSmall;
24 }, 500);
25 print("$$");

```

**Listing 4.** The Flashing Heart program in BlueScript

The figures show that BlueScript is one to two orders of magnitude faster than MicroPython and three to four magnitudes faster than Espruino. The performance difference is much larger when a program involves a large number of arithmetic operations on primitive types. Such programs are Bounce, FIR, CRC, and SHA256. This result is because the BlueScript programs are type-annotated and thus they less frequently perform runtime type checks and boxing/unboxing. Compared to the C language, BlueScript is up to 10 times slower. The slowdown is significant when a program involves a large number of function calls since they are indirectly invoked in BlueScript. Such programs are Towers and List.

## 4.2 Interactivity

We evaluate the interactivity of BlueScript by comparing it to MicroPython and the C language. We incrementally write a program step by step according to a given scenario, and measure the response time for every step.

**Benchmark Programs.** We develop a new benchmark suite for our experiment. We take five scenarios from a tutorial course on the website of MakeCode<sup>5</sup>, which provides programming lessons using the `micro:bit` [8] microcontroller for beginners. We develop three versions of programs for

<sup>5</sup>See <https://makecode.com>. It is open source under MIT license, see <https://github.com/microsoft/pxt>

each scenario in BlueScript, MicroPython, and the C language. Table 3 lists the benchmark programs we develop. In each scenario, a programmer writes a code fragment, and runs it to test its behavior step by step. Each scenario consists of three or four code fragments. The goal of each scenario is to display an icon and text on the screen. The target microcontroller is M5Stack Fire for all the scenarios. We implement a library in the C language to manipulate the screen and the buttons of M5Stack Fire. We make this library accessible from BlueScript and MicroPython through a similar interface as well as the C language. The microcontroller is connected to a host machine through Bluetooth for BlueScript, WiFi for MicroPython, and a serial cable for the C language.

Listing 4 shows code fragments in BlueScript for the scenario Flashing Heart. This scenario consists of three code fragments. A programmer types and runs each code fragment step by step to finally build a small application program that shows a heart icon on the screen. The code fragment #1 initializes the display and shows a heart icon on the screen. The code fragment #2 shows a small heart icon and erase it. The code fragment #3 calls `setInterval` so that the given function will be repeatedly called and the size of the icon will change every 500 msec. Every code finally prints a text to notify the end of the execution to a host machine.

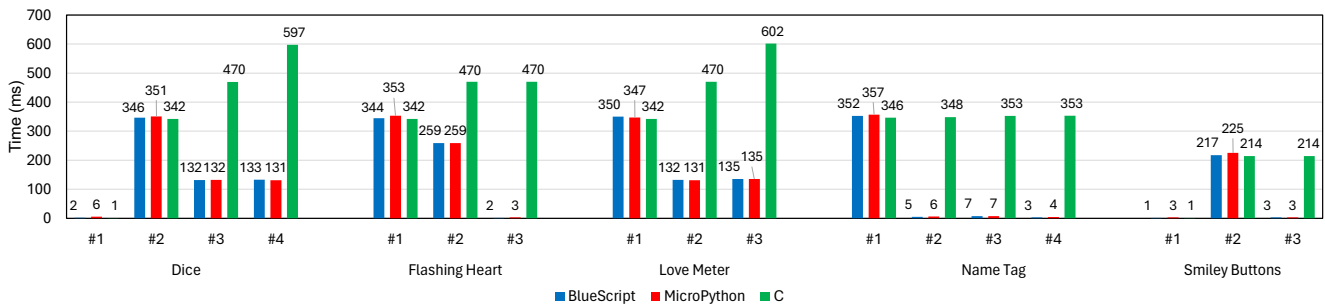
For MicroPython, the code in each code fragment is transformed into a single line since the REPL of MicroPython receives only a single line at once. We combine all the statements in the code fragment into a single line where statements are separated by a semicolon. Since the execution environment ESP-IDF for the C language, does not directly support interactive programming, we execute the previous code fragment again whenever we execute the new code fragment. For example, when we execute the code in the third fragment, we combine the first, second and third fragments into a single program, compile it, and execute it. This is because we must reboot M5Stack Fire to start a new program, and hence, all the devices must be initialized again after rebooting.

**Results.** We execute the code fragments of each scenario three times and measure the response time. It is the elapsed time since we press the button to start executing a code fragment till a string `$$` is printed as an execution result on the screen. Every code fragment prints `$$` at the end as shown in Listing 4. We also measure the execution time, excluding compilation time and communication time from the response time. For BlueScript and the C language, the times shown in figures are the means of the three runs of each scenario. For MicroPython, those are the shortest ones because of their instability and large distribution.

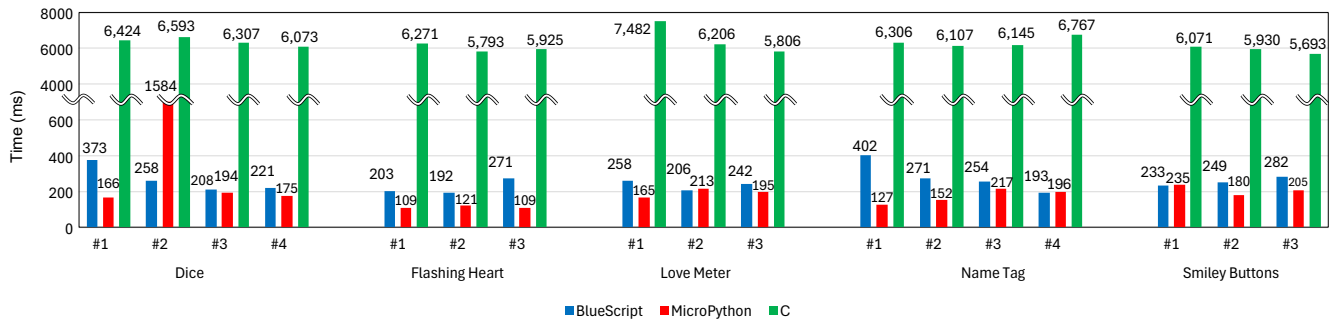
Figure 6 shows the execution time of every code fragment. Since every code fragment is very short and it just calls a few library functions only, the majority of the execution time is the time for executing library functions. Thus, we

**Table 3.** Benchmark programs for interactivity

Scenario	Fragment	LOC	Description
Dice	1	3	Bind an empty function to button B by using buttonOnPressed built-in function.
	2	8	Initializes the display module and bind a function for displaying 0 to button B.
	3	5	Bind a function for displaying a random integer from 0 to 10 to button B.
	4	5	Bind a function for displaying a random integer from 0 to 6 to button B.
Flashing Heart	1	5	Initialize the display module, set global variables to color code, and then display a heart icon.
	2	3	Display the same heart icon again, and then erase it. Reuse the display module and color code.
	3	10	Use setInterval function to display a large heart and a small heart alternately.
Love Meter	1	9	Initialize the display module and bind a function for displaying a random integer 0-10 to button B.
	2	5	Bind a function for displaying a random integer from 0 to 100 to button B.
	3	6	Display "Love meter" and bind a function for displaying a random integer from 0 to 100.
Name Tag	1	6	Initialize the display module, set global variables to color code, and then display "My name is:".
	2	2	Display "Sara!."
	3	2	Display "My age is:".
	4	2	Display an integer 9.
Smiley Button	1	3	Bind an empty function to button B by using buttonOnPressed built-in function.
	2	7	Initialize the display module and bind a function for displaying a happy face icon to button B.
	3	5	Bind a function for displaying a sad face icon to button C.



**Figure 6.** Execution times of every code fragment



**Figure 7.** The time of compilation and others for every code fragment in BlueScript, MicroPython and C

do not observe notable differences between the three languages. The execution times of fragment #3 and #4 in the C language are significantly slow. This is because we execute the fragment #1 and #2 (and #3) again when we execute the fragment #3 (or #4). The readers might think that this is odd and unfair, but this is typical software development by using a non-interactive environment for the C language.

Whenever we change a program and test it, we must reboot a microcontroller and run the program again from the beginning.

Figure 7 shows the times for compilation and others excluding execution. The times include communication between a host machine and a microcontroller. They are calculated by subtracting the net execution time from the total

**Table 4.** The size of runtime routines on a microcontroller

Environment	Bluetooth	WiFi	Size (MB)
BlueScript	✓	-	0.86
	-	-	0.27
MicroPython	✓	✓	1.64
	-	✓	1.40
	✓	-	1.26
	-	-	0.97

response time. The time of fragment #2 for the scenario Flashing Heart in MicroPython is extremely long for an unknown reason, which we are still investigating. The time is 1584 ms, which is significantly longer than the time for BlueScript. In other cases, BlueScript is up to 3.2 times (the mean is 1.6) slower than MicroPython since BlueScript performs incremental dynamic compilation. However, the times in BlueScript are approximately less than 300 msec. despite the use of a relatively slow off-the-shelf C compiler as a backend native-code generator. It would be slower than a dedicated highly-tuned dynamic compiler. Those times are still acceptable for interactive programming. According to the literature [20], the users feel that a response is immediate when it takes 0.5 to 1 seconds. The times for compilation and others in the C language are an order of magnitude longer than the times in the other two languages. Those times are approximately 6 seconds. They include compilation, linkage of all object code including libraries and an operating system, and writing the resulting executable binary code to flash memory of a target microcontroller. The C language is not acceptable for interactive programming.

### 4.3 Runtime Size

Table 4 lists the memory size of runtime routines on a microcontroller. For BlueScript, it is 0.86 MB but is 0.27 MB if the Bluetooth library is excluded. For MicroPython, the size of its virtual machine is 1.64 MB but is 1.40 MB if the Bluetooth library is excluded. It is 1.26 MB if the Bluetooth library is included but the WiFi library is excluded. The size of the virtual machine without Bluetooth or Wifi libraries is 0.97 MB. However, that virtual machine still includes a number of libraries for accessing hardware. Note that these runtime routines and the code of the virtual machine are stored in flash memory but not in a limited size of SRAM.

## 5 Related Work

Espruino's "compiled" tag [11] is most relevant to our work. The source code of a function with this tag is sent to a remote web service. It is compiled there into native code and sent back to a target microcontroller. Since this dynamic incremental compilation is supported only for official Espruino boards, it is not used for the experiment in Section 4.1. A difference from our work is that Espruino's compilation

has only limited capability to link compiled native code. A global variable is not linked, and hence Espruino searches the symbol table when it accesses a global variable.

MicroPython provides two dynamic incremental compilers: the Native code emitter and the Viper code emitter [4]. They run on microcontrollers. If a MicroPython function is decorated with `native` or `viper`, it is compiled into native code when its declaration is evaluated. Since those compilers run on microcontrollers, their compilation capability is limited. They support only a subset of the language, and the compiled native code must be compatible with the bytecode interpreter for passing and returning a value beyond function boundaries. Thus, the performance improvement is limited. According to our experiment using the benchmark programs in Section 4.1, the Viper code emitter improves the execution performance of the benchmark programs only by a factor of up to 4.6. Recall that BlueScript runs one or two orders of magnitude faster than MicroPython.

LuaRTOS [14] provides an interactive shell and dynamic incremental compilation similar to the Native code emitter of MicroPython. uLisp [5] also provides an interactive shell for microcontrollers. Although it does not provide dynamic incremental compilation, it supports an inline assembler.

There have been several research activities [9, 18, 26], where the techniques for dynamic compilers are applied to virtual machines running on microcontrollers, but the design and implementation of such compilers is still a challenging topic. Our work proposes a different design of dynamic compiler targeting interactive environments for microcontrollers, rather than simply porting a dynamic compiler designed for high-performance processors to microcontrollers.

Warduino [12] is a WebAssembly (Wasm) virtual machine for microcontrollers. Like our work, Warduino utilizes the computing resources of a host machine to compensate the poor resources of microcontrollers. It offloads debugging capability to a host machine although ours offloads dynamic incremental compilation.

StaticTypeScript [2] is a subset of TypeScript, and its design is similar to our BlueScript. To achieve good execution performance on a micro:bit microcontroller, StaticTypeScript provides an offline compiler running in a web browser on a host machine. Unlike BlueScript, StaticTypeScript does not provide an interactive shell.

Contiki [6] uses a dynamic linker for reprogramming a wireless sensor node, which is run by a low-power microcontroller for energy efficiency. Contiki allows a memory-constrained sensor node to dynamically load and link native code modules. Unlike Contiki, BlueScript allows dynamic loading without dynamic linking on a microcontroller since linking is offloaded.

JTAG [1] and SWD<sup>6</sup> are debugging protocols for microcontrollers. They enable breakpoints, step execution, and reading/writing microcontrollers' memory. Compared to such debugging protocols, our shadow memory can be regarded as a software implementation of debugging protocol although it currently allows only reading and writing microcontroller's memory. Our shadow memory is, however, not only for debugging but also for interactive programming. Conversely, it would also be possible to implement shadow memory using JTAG.

## 6 Conclusion

We presented an interactive execution environment for our programming language *BlueScript*. To run on a microcontroller with a small amount of SRAM, this execution environment offloads dynamic incremental compilation and linking to a host machine that is a computer connected to that microcontroller and used by programmers to access it for programming. We illustrated that *BlueScript* programs run only up to 10 times slower than C programs and one to two orders of magnitude faster than MicroPython's ones. Our experiments showed that our execution environment responded within approximately less than 300 msec. excluding the execution time of a given *BlueScript* program, although it reuses an off-the-shelf C compiler to reduce its development costs and improve its portability. The source code of *BlueScript* is available on our website <https://github.com/csg-tokyo/bluescript>.

## Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers JP20H00578 and JP24H00688.

## References

- [1] 2001. IEEE Standard Test Access Port and Boundary Scan Architecture. *IEEE Std 1149.1-2001* (2001), 1–212. <https://doi.org/10.1109/IEEESTD.2001.92950>
- [2] Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: An Implementation of a Static Compiler for the TypeScript Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) (*MPLR 2019*). Association for Computing Machinery, 105–116. <https://doi.org/10.1145/3357390.3361032>
- [3] Damien P. George. 2014. MicroPython. <https://micropython.org>.
- [4] Damien P. George and Paul Sokolovsky and contributors. 2014. Maximising MicroPython Speed. [https://docs.micropython.org/en/v1.9.3/pyboard/reference/speed\\_python.html](https://docs.micropython.org/en/v1.9.3/pyboard/reference/speed_python.html).
- [5] David Johnson-Davies. 2016. uLisp. <http://www.ulisp.com/>.
- [6] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems* (Boulder, Colorado, USA) (*SenSys '06*). Association for Computing Machinery, New York, NY, USA, 15–28. <https://doi.org/10.1145/1182807.1182810>
- [7] Espressif Systems (Shanghai) Co., Ltd. 2016. ESP-IDF Programming Guide. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html>.
- [8] Micro:bit Educational Foundation. 2017. micro:bit. <http://microbit.org/teach/>.
- [9] Giuseppe Di Giore, Antonella Di Stefano, Giovanni Morana, and Corrado Santoro. 2006. JIT compiler optimizations for stack-based processors in embedded platforms. In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems* (Paris, France) (*JTRES '06*). Association for Computing Machinery, New York, NY, USA, 212–217. <https://doi.org/10.1145/1167999.1168034>
- [10] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [11] Gordon Williams. 2015. JavaScript Compilation. <https://www.espruino.com/Compilation>.
- [12] 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* (Athens, Greece) (*MPLR 2019*). Association for Computing Machinery, 27–36. <https://doi.org/10.1145/3357390.3361029>
- [13] Fergus Henderson. 2002. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany) (*ISMM '02*). Association for Computing Machinery, New York, NY, USA, 150–156. <https://doi.org/10.1145/512429.512449>
- [14] Jaume Olivé Petrus. 2015. Lua RTOS. <https://github.com/whitecatboard/Lua-RTOS-ESP32>.
- [15] Ronald L. Johnston. 1979. The Dynamic Incremental Compiler of APL 3000. In *Proceedings of the International Conference on APL: Part 1* (New York, New York, USA) (*APL '79*). Association for Computing Machinery, New York, NY, USA, 82–87. <https://doi.org/10.1145/800136.804442>
- [16] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st ed.). Chapman & Hall/CRC.
- [17] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (nov 2010), 15 pages. <https://doi.org/10.1145/1868358.1868363>
- [18] Geetha Manjunath and Venkatesh Krishnan. 2000. A small hybrid JIT for embedded systems. *SIGPLAN Not.* 35, 4 (apr 2000), 44–50. <https://doi.org/10.1145/346443.346451>
- [19] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking: Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) (*DLS 2016*). Association for Computing Machinery, 120–131. <https://doi.org/10.1145/2989225.2989232>
- [20] Steven Seow Ph.D. 2008. *Designing and Engineering Time: The Psychology of Time Perception in Software* (1st ed.). Addison-Wesley Professional.
- [21] Ignas Plauska, Agnius Liutkevičius, and Audronė Janavičiūtė. 2023. Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics* 12, 1 (2023). <https://doi.org/10.3390/electronics12010143>
- [22] Project Jupyter. 2014. Jupyter. <https://jupyter.org/>.
- [23] Pur3 Ltd. 2013. Espruino. <https://www.espruino.com>.
- [24] Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming*.
- [25] David Ungar and Randall B. Smith. 2007. Self. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*

<sup>6</sup><https://developer.arm.com/documentation/ih0031/a/The-Serial-Wire-Debug-Port--SW-DP-/Introduction-to-the-ARM-Serial-Wire-Debug--SWD--protocol>

- (San Diego, California) (*HOPL III*). Association for Computing Machinery, New York, NY, USA, 9–1–9–50. <https://doi.org/10.1145/1238844.1238853>
- [26] Yuan Zhang, Min Yang, Bo Zhou, Zheming Yang, Weihua Zhang, and Binyu Zang. 2012. Swift: a register-based JIT compiler for embedded JVMs. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (London, England, UK) (*VEE '12*). Association for Computing Machinery, New York, NY, USA, 63–74. <https://doi.org/10.1145/2151024.2151035>