mruby on Resource-Constrained Low-Power Coprocessors of Embedded Devices

Go Suzuki
Tokyo Institute of Technology
Tokyo, Japan
gosuzuki@psg.c.titech.ac.jp

Takuo Watanabe
Tokyo Institute of Technology
Tokyo, Japan
takuo@psg.c.titech.ac.jp

Sosuke Moriguchi
Tokyo Institute of Technology
Tokyo, Japan
chiguri@psg.c.titech.ac.jp

Abstract

As IoT devices advance, their microcontroller systems-on-a-chip (SoCs) demand higher speeds, more memory, and advanced peripherals, leading to increased power consumption. Integrating low-power (LP) coprocessors in SoCs can reduce power usage while maintaining responsiveness. However, switching application execution to and from the coprocessors generally involves complex and platform-specific procedures. We propose a JIT compilation method for managed programming languages to streamline LP coprocessor use. Our prototype for the programming language mruby includes a JIT compiler and a seamless processor-switching mechanism, enabling rapid development of IoT applications leveraging LP coprocessors. This work-in-progress paper describes the design and implementation of the extended mruby interpreter and presents preliminary evaluations of its power consumption and latency on ESP32-S3 and ESP32-C6.

CCS Concepts: • **Software and its engineering** → **Interpreters**; **Just-in-time compilers**; • **Computer systems organization** → *Embedded software*.

Keywords: managed languages, embedded systems, microcontrollers, low-power coprocessors, power consumption

ACM Reference Format:
Go Suzuki, Takuo Watanabe, and Sosuke Moriguchi. 2024. mruby on Resource-Constrained Low-Power Coprocessors of Embedded Devices. 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, 7 pages. https://doi.org/10.1145/3679007.3685064

1 Introduction

Recent microcontroller systems-on-a-chip (SoCs) accommodate rich peripherals and large enough memory resources to meet the requirements for various Internet-of-Things (IoT) applications. For example, an ESP32-S3 SoC module incorporates WiFi, 512 KiB Static RAM, and 4 MiB or more Flash memory [7]. Despite the growing demands of complex tasks (e.g., communication via MQTT, HTTPS with JSON) performed in such devices, C, C++, and assembly language are still used as the primary implementation languages. So, memory errors and the difficulty of debugging optimized compiled binaries have plagued developers.

To help the rapid development of IoT applications, high-level programming languages with rich programming environments have been proposed for embedded devices [14, 16–18, 23, 25, 28]. Language features, such as dynamic typing and garbage collection (GC), can facilitate the development of complicated but memory-safe IoT devices. For the languages listed above, bytecode VMs running on the target devices are often used instead of native code compilers running on the development host machines. These provide rapid, interactive development. In addition, they make live code updates, such as Over-The-Air updates, much easier than with C/C++. While these are not suitable for real-time systems due to a (naïve) GC, there are many applications suitable for these languages, such as agriculture, meteorological observation, etc.

However, while low power consumption and responsiveness are essential for IoT devices, execution by a VM consumes more power than native code. Putting the processor in sleep mode reduces power consumption but at the expense of responsiveness. To solve this problem, SoCs with *low-power coprocessors* (LP coprocessors) that operate while the main processor is sleeping are gaining popularity. For this purpose, LP coprocessors operate with limited resources. The amount of available memory and accessible peripherals should be limited. Moreover, the address space and processor architecture may also differ from those of the main processors. Thus, developing applications that take advantage of LP coprocessors usually requires writing complex procedures in C/C++ that directly access the hardware.

This work aims to facilitate the development of applications that utilize LP coprocessors using a dynamically typed

Table 1. The Specifications of the Targets

	ESP32-S3	ESP32-C6
Main Processor ISA	Xtensa LX7	RV32IMAC
LP Coprocessor ISA	RV32IMC	RV32IMAC
Main SRAM [KiB]	512	512
RTC Slow Memory [KiB]	8	16

managed language. Toward this goal, we introduce a JIT compiler and an execution migration mechanism into mruby/c[1] to execute code on LP coprocessors. These mechanisms allow, within an mruby script, seamless switching between the main processor and coprocessor execution in a SoC. Our current contributions are implementations and preliminary evaluation of the mechanisms.

We target two microcontroller SoCs, ESP32-S3 [6] and ESP32-C6 [8]. The main processor and the LP coprocessor are connected via the interconnect. They interact via the RTC Slow Memory, a memory for the LP coprocessor. During the light-sleep state, the LP coprocessor can access only the RTC Slow Memory. This memory can be accessed via the interconnect and is memory-mapped. In ARM-based microcontrollers, a coprocessor, processors, and memories are connected via the AXI interconnect bus [12, 21, 22]. We guess that ESP SoCs are implemented like ARM-based microcontrollers. Table 1 shows the specifications of these targets. The LP coprocessor has limited accessible memory space, accessible peripherals, and processor performance, compared to the main processor. Thanks to these limitations, the LP coprocessor of ESP32-S3 consumes 200 µA, while the main processor consumes 13.2 mA at the lowest frequency (40 MHz) [6].

The rest of this work-in-progress paper is organized as follows. The next section describes related works. Section 3 provides two simple examples to illustrate how to switch execution between the main processor and the LP co-processor. Section 4 describes our proposed method for JIT compilation. Then, Section 5 presents the preliminary evaluation results. Finally, Section 6 discusses the future work and Section 7 concludes the paper.

2 Related Work

2.1 Ahead-of-Time Compilation

Static TypeScript [1] is a subset of TypeScript with an Ahead-of-Time (AoT) compiler. It targets small-scale embedded devices, which have only 256–512 KiB ROM and 16–256 KiB RAM. Although AoT compilation gives better performance, it can result in larger compiled binaries occasionally. Our target can communicate over WiFi and potentially manipulate JSON data (an example of the highly dynamic data structures). Thus, conservative type checking may generate larger codes. However, AoT compilation can generate better codes for well-typed programs and is suitable for real-time systems. Depending on the application, it is important to employ AoT or JIT compilation properly.

2.2 Tiny Interpreters

The Ribbit system [30] is a compact Scheme interpreter with a footprint of 4 KiB, and the following work [24] implements the R^{4}RS standard in 7 KiB by using LZ compression for bytecodes. Without compression, it exceeds 8 KiB. This suggests that compiling or transferring functions on demand is necessary. Even though bytecodes tend to be smaller than machine code [3], the small memory of the LP coprocessor cannot accommodate all of the standard libraries. We must declare functions used in LP coprocessors, or functions must be compiled/transferred on demand. Moreover, while Scheme is simple, popular high-level programming languages in embedded systems such as Ruby and Python are more complicated. Hence, we think that it is difficult to implement an interpreter for such languages within 8 KiB.

2.3 Just-in-Time Compilation

2.3.1 On Embedded Devices. Some works [10, 19, 27] developed just-in-time (JIT) compilers for resource-constrained devices. These works imply that ESP32 can do JIT compilation. With the fact that programs on LP coprocessors are usually small, it is possible to run a JIT compiler on main processors to generate programs for LP coprocessors. The conventional main purpose of JIT compilers is the code speed, but our purpose is a small memory resource. An efficient code is not always minimum, for example, excessive code duplication (code specialization) should be avoided.

2.3.2 For Dynamic Languages. *Lazy basic block versioning* [4] is a JIT compilation technique suitable for dynamic programming languages and is used in the Ruby compiler YJIT [5]. The compiler incrementally compiles one basic block at a time. Compiled basic blocks have branch stubs for branches to uncompiled basic blocks. When execution reaches such stubs, the compiler resumes the code generation. The header of each basic block has a typing context for local variables, and basic blocks are specialized according to the typing context. The destination of compiled code of branches is determined not only by the program location but also by the typing context. This technique avoids heavy and complex implementations such as type analysis. However, it still requires its own partial evaluator for compiler optimizations such as constant folding and devirtualization.

Trace-based JIT [10] is also a JIT compilation technique to detect and compile frequently executed program paths. It gathers constant-foldable values and types of local variables

[1] mruby [20, 29] is a lightweight implementation of Ruby, and mruby/c [23] is an implementation of the mruby runtime that runs on resource-constrained devices. We chose mruby for this study because information on RiteVM (the mruby VM) is relatively easy for us to obtain compared to MicroPython [25] or other languages.

by executing programs by the interpreter. It reduces the code size of the compiler, which is suitable for resource-constrained systems. However, generated codes by trace-based JIT may occur too many code duplications. As a result, the generated code may exceed the memory resource of the LP coprocessor.

2.4 Execution Migration

Execution migration is to migrate a running program across heterogeneous-ISA (Instruction Set Architecture). One of the purposes of execution migration is to run programs on suitable processors for performance or power consumption. Some works [2, 11] compile statically and generate bare binaries, unlike our approach. In embedded systems, a big issue of execution migration is code size overheads for state transformation (transforming stack frames and the heap) at migration points. It is possible that the compiled binary is twice as big. UNIFICO [15] solves this problem by adjusting the stack and limiting the number of registers. These works allow execution migrations at any call site. We believe that a carefully designed migration granularity can reduce such overheads in applications of LP coprocessors.

Emerald [13] is a managed programming language designed for distributed computing. Objects in Emerald can freely move within the distributed system, and developers can explicitly move objects. Unlike Emerald, in our work, the LP coprocessor and the main processor work exclusively. Thus, developers do not need to think about the concurrency so we will not provide primitives such as monitor. Moreover, we try to move objects automatically on demand without modifying the object system in Ruby.

3 Motivating Example

Mainly, (general purpose) coprocessors of microcontrollers have two purposes: for lower power consumption and for real-time tasks. Our approach is not suitable for real-time systems because we employ JIT compilation that results in a long pause time when execution reaches uncompiled program locations. Therefore, we focus on the lower power consumption purpose. In this section, we show two motivating examples. The former is used in the preliminary evaluation.

3.1 LED Blinking

LED blinking is a popular test case of embedded systems. Listing 1 is an example using the LP coprocessor, written in Ruby. In this example, the GPIO4 and GPIO5 pins are connected to an LED and a tactile switch, respectively. When the `Copro#run` method is called, the JIT compiler starts to compile the given block. Eventually, the LP coprocessor executes the compiled code and the main processor sleeps (explained in Section 4.1). When the given block is finished (*i.e.*, after the tactile switch is pressed), the main processor wakes up and executes the following program.

Listing 1. LED blinking in Ruby

```ruby
Copro.run do
  prevPress = false
  press = false
  # While the tactile switch is not pushed.
  while (!prevPress ||
    press) do # Negative Edge
    Copro.gpio(4, true)  # Turn on the LED.
    Copro.delayMs(30)    # Sleep for 30 ms.
    Copro.gpio(4, false) # Turn off the LED.
    Copro.delayMs(30)
    prevPress = press
    press = Copro.gpio?(5) # Check the switch.
  end
end
# LP coprocessor never executes here.
```

Listing 2. IoT Sensor in Ruby

```ruby
sensor = SHT3xSensor.new(I2C.new(5,4))
   # May be set by the JSON config.
buffer = Array.new(60)
Copro.run do
  (0...60).each do |i|
    Copro.delayMs(1000*60)    # Sleep for 1 min.
    buffer[i] = sensor.read() # Read from sensor.
  end
end
Network.send(buffer) # Send buffered data.
```

In this way, we can save the number of migration points discussed in Section 2.4, if we use a static compilation approach. Migration points are only before/after the call sites of `Copro#run`. Stack frame transformations are unnecessary; it only transfers the closure (`Proc` object in Ruby) to the LP coprocessor because the LP coprocessor never executes outside the `Copro#run`.

3.2 IoT Sensor

IoT sensors are popular applications to sense humidity, motion, pressure, etc. They gather environmental information from sensors and send gathered information over the network to a central server. Some sensors consume lower current than the main processor, *e.g.*, a humidity and temperature sensor consumes 600 µA typically [26]. The LP coprocessors of our targets can interact with sensors connected over communication methods such as I^{2}C, 1-Wire, and analog-to-digital converters. If the IoT sensor sends to the server infrequently, waking the main processor for measurements affects the battery life.

Listing 2 is an example of an IoT sensor in Ruby. Ruby's subtyping mechanism allows the creation of interfaces that are independent of specific sensors or communication methods. In our implementation, this mechanism can also be used on the LP coprocessors. Thus, if the production of components, such as sensors, becomes discontinued, we can prepare a new code for the replacement components with small changes.

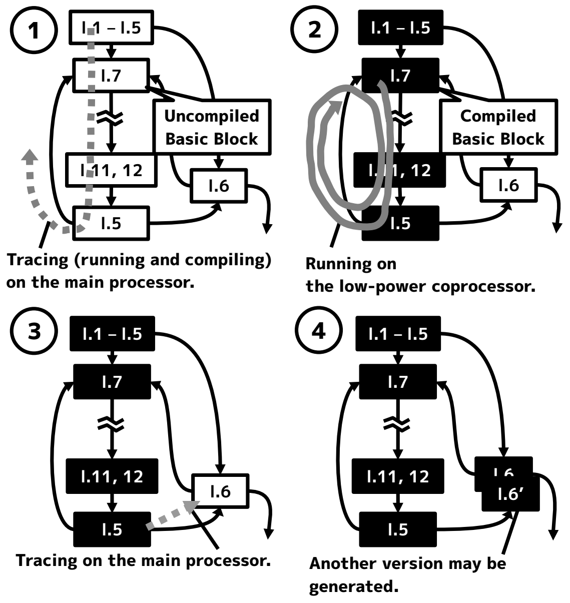
1
l.1 – l.5
l.7
l.11, 12
l.5
l.6
Uncompiled Basic Block
Tracing (running and compiling) on the main processor.
2
l.1 – l.5
l.7
l.11, 12
l.5
l.6
Compiled Basic Block
Running on the low-power coprocessor.
3
l.1 – l.5
l.7
l.11, 12
l.5
l.6
Tracing on the main processor.
4
l.1 – l.5
l.7
l.11, 12
l.5
l.6
l.6'
Another version may be generated.

Figure 1. An Overview of Just-in-Time Compilation

4 Summary of Proposed Method

However, the LP coprocessors of our targets do not have enough memory to run the VM, so we introduce a JIT compiler running on the main processor to generate code for the coprocessor dynamically. This section briefly describes our JIT compiler and discusses object management between the main processor and coprocessor.

4.1 Just-in-Time Compilation

Our method is based on lazy basic block versioning [4]. It avoids traditional type analyses. Type analyses can be complex and heavy to support richer type representations. The generated programs for the LP coprocessor also may become inefficient and large due to the conservative type checking. We consider that gathering types by running the program is a cheap way for the footprint and the computation complexity.

However, we use the interpreter for the first execution like trace-based JIT [10] to reduce the processor wake-ups and the execution migrations. Unlike trace-based JIT, the generated codes are divided into basic block versions. This allows compiled basic blocks (including functions) to be reused in the different code paths. Similar to trace-based JIT, our method reuses the original mruby/c interpreter. It reduces the runtime footprint[2]. In addition, if an executing basic block is not appropriate to compile and execute on the LP coprocessor (*e.g.*, low frequently executed, or using not supported features (*e.g.*, too dynamic features) on the LP coprocessor), it allows to run on the main processor seamlessly.

Figure 1 shows the control flow graph corresponding to Listing 1. Each node is a basic block split by method callings and branches, whose label represents the line number on Listing 1. First, uncompiled basic blocks are executed and traced on the main processor (1). The traced basic blocks are compiled while applying optimizations such as type specialization. Then, when it reaches a compiled basic block, the LP coprocessor executes the compiled basic blocks (1 → 2). During execution on the LP coprocessor, the main processor sleeps. After that, when the LP coprocessor reaches the uncompiled basic block, the main processor wakes up and executes and compiles the uncompiled basic block (3). In the LED blinking example, this is happened when the tactile switch is pushed. Like lazy basic block versioning, new basic block versions may be generated (4). If the `Copro#gpio?` (*l*.12) does not return a boolean value at the second time, a new basic block version for *l*.6 is created.

4.2 Objects

During execution on the LP coprocessor, shapes of objects are fixed. Unlike Python, Ruby does not allow to access instance variables outside instance methods. We assume that programs on the LP coprocessor do not use too dynamic features. On the LP coprocessor, too dynamic features such as `Object#extend` and class definitions are disallowed. As a result, objects can be realized without hash tables. We note that dynamic features still can be used on the main processor (outside `Copro#run`).

A `Method` object on the main processor can contain a pointer to a C function on the LP coprocessor, in addition to a C function on the main processor. When it is called, it executes the C function on the main processor; instead, the compiled code calls the C function on the LP coprocessor. `Copro#gpio` and `Copro#delayMs` use this to execute on both processors.

5 Preliminary Evaluation

5.1 Evaluation Detail

We evaluate the code size and the wake-up/compile overheads by the LED blinking example (described in Listing 1) with the initial implementation. Currently, it supports:

- Integers, Booleans and `nil`
- Arithmetic operators
- A single call frame migration
- Calling methods defined in C
- Garbage Collection (however, not used)

and does not support:

- Objects including Arrays, Strings and Hashes
- Calling methods defined in Ruby
- Global variables
- Closures

Table 2 shows the evaluation environment. We evaluated under following configurations:

[2]Currently, we copied the original one.

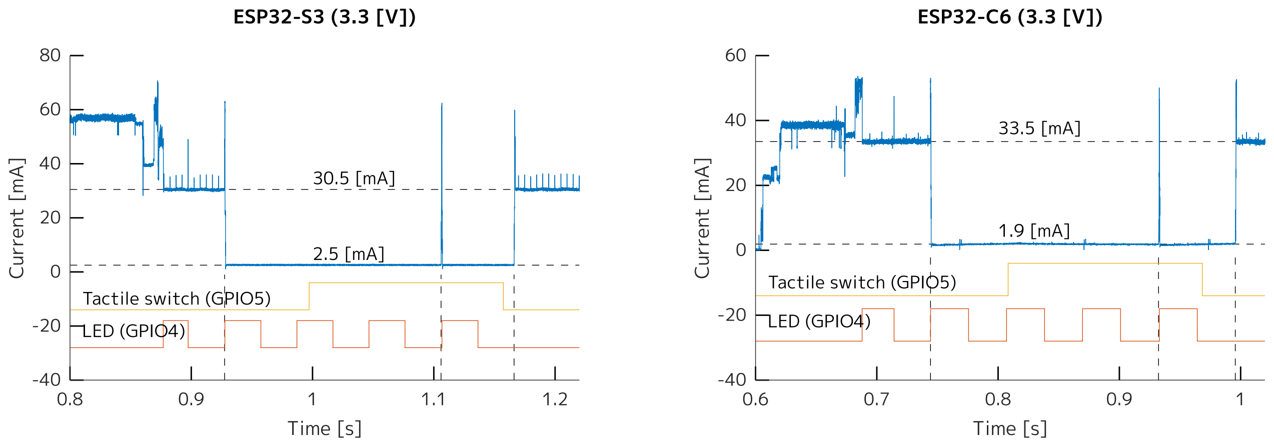

Figure 2. Current Consumption Results

Table 2. The Evaluation Environment

Role	Name and Revision
Evaluation boards	ESP32-S3-DevKitC-1 N8, v1.0 ESP32-C6-DevKitC-1 N8, v1.3 (Main processors freq. : 160 [MHz])
SDK	ESP-IDF v5.2.1
Ammeter	Nordic Power Profiler Kit 2 (PPK2)
Signal Generator	Nodemcu ESP8266 Ver 0.1

Table 3. The Runtime Code Size [B]

Target		Main Processor			Copro.
		.data+.bss	.text	.rodata	
ESP32-S3	Orig.	2582	48928	6511	0
	Ours	2614	68411	6864	1336
ESP32-C6	Orig.	2574	57456	6615	0
	Ours	2614	80822	6968	3072

- The signal generator simulates the tactile switch input. It makes GPIO5 (tactile switch) high for 160 [ms], 120 [ms] after GPIO4 (LED) is high at the first time.
- The sample rate of PPK2 is 100 [kHz]. It delivers the 3.3 [V] power to the targets and captures GPIO signals.
- On the ESP32-C6-DevkitC-1, the jumper is removed to disconnect the power-on indicator LED and the UART/USB controller [9]. However, on the ESP32-S3-DevkitC-1, there is no jumper so they are connected.
- The commit hash of mruby/c that we use is `73c1324f93`.
- The watchdog timers are disabled.
- We used the light-sleep state because the deep-sleep state cannot retain the main memory.

We measured the code size of the mruby runtime by `$ idf.py size-components` provided by ESP-IDF SDK. We also measured the wake-up time of the processors. GPIO1 becomes high before the running processor wakes the other processor and becomes low after the other processor is ready.

5.2 Result and Discussion

Table 3 shows the code size of the mruby runtime. The code of the LP coprocessor contains the garbage collector, the implementations of `Copro#delayMs`, `Copro#gpio`, and the bootstrap. The code size changes are about 20 [KiB]. Since we currently copy the original interpreter and modify it, these changes are not small compared to the interpreter code size (about 55 or 65 [KiB]). This suggests that the code should be shared between the interpreter and the JIT compiler when the ROM size is limited.

The memory overheads on both targets are:

- Profiling data on the main processor: 776 [B]
- Generated codes on the LP coprocessor: 220 [B]

Profiling data manages the register allocation and the typings. The generated code has many move instructions for constant values. This can be reduced if the copy-on-write register allocation is implemented (discussed in Section 6.1).

Figure 2 shows the current consumptions. In the figures, the horizontal dashed lines represent approximate values under steady-states. 30.5 [mA] and 33.5 [mA] are the power consumption of the main processor, and 2.5 [mA] and 1.9 [mA] are the power consumption of the LP coprocessor. By using the LP coprocessors, we can see that the power consumption is reduced by about 10 times. The vertical dashed lines in the figures represent when the main processors wake up or enter the sleep state. At the first vertical dashed line, the main processors enter the sleep state. At the second vertical dashed line, the main processors wake up and compile the following program for the LP coprocessors, then enter the sleep states. Since The tactile switch input changes its value, the taken branch is changed, and the uncompiled basic block is compiled. After the third dashed line, execution of `Copro#run`

Table 4. Wake-Up Time Overheads [ms]

	ESP32-S3	ESP32-C6
The Main Processor	0.51	0.58
The LP Coprocessor	0.18	0.02

finishes, and the main processors continue to work. Comparing before/after the first dashed line, we observed that the power consumption is reduced by the LP coprocessor. In this example, ten basic block versions are generated, but the main processor has only woken up twice. This is due to the tracing of the first execution by our method.

Table 4 shows the elapsed time of processor wake-ups. These overheads are inevitable when a processor wakes up, and are expected to be a problem for some applications. For example, I^2C standard mode operates at 100 [kHz] (*i.e.* 1-bit per 0.01 [ms]). If the processor wakes up during the I^2C communication by software, the timing is not met. We believe that entering the sleep state with a delay can alleviate this problem.

6 Future Work

6.1 Code Generation

Because the mruby bytecode is a register machine, the register allocation is simply done with one-pass algorithms. However, the instruction format is represented as (`R_i` is the *i*th register.):

```
ADD i # R_i = R_i + R_(i+1)
```

This frequently introduces move instructions despite the three-address code of RISC-V, the target (as follows).

```
add i, j, k # R_i = R_j + R_k
```

To reduce the code size, we should implement a copy-on-write register allocation algorithm. However, using a simple one-pass algorithm, each basic block version must have allocated register numbers for local variables because the allocated registers may be different with the code path. Using a two-pass algorithm, the generated code is minimal, but the code size of the compiler and the compilation time will be larger (The instruction format of the mruby bytecode is variable-sized). We must design the register allocation while considering the trade-off between the generated code size and the compilation overhead.

6.2 Object Management

Because objects are transferred on demand, read barriers are required, *e.g.*, before reading an instance variable. Before a pointer outside the memory region available for the LP coprocessor is copied into a register, the object pointed at must be transferred and the pointer must be translated. Instead of barriers, we consider that the hardware interrupts can be used. The handler for the bus error interrupts[3] searches an Least-Recently-Used (LRU) table. The table has combinations of addresses translated from and to. If the pointer is not found in that table, it may not have been transferred. The main processor wakes up to transfer the object. The main processor also has a complete (non-LRU) table. If the pointer is found in that table, the main processor tells the LP coprocessor the translated address. If it is not found, the object is transferred and then is told to the LP coprocessor. Fortunately, popular microcontrollers do not overlap memory regions among the LP and main processors.

6.3 I/O

In modern microcontrollers, the control registers of peripherals are memory-mapped. The problem is how to implement I/O functions. When the developer implements I/O functions in C language, additional machine codes have to be placed on the LP coprocessor. Even when the I/O functions are actually not used, these codes still have to be placed beforehand. It is a problem if live code updates happen. To avoid this, a relocation infrastructure for C functions is required.

To implement I/O functions in Ruby, we consider the differences in the memory-mapped addresses of the control registers between the processors. To solve this, we need to define an address translation function in C language, separately. Another solution is to perform the address translation on the bus error interrupts. However, guarding in the Ruby program cannot solve this problem because programs executed on the LP coprocessor must be executed on the main processor during compilation.

7 Concluding Remark

In this work-in-progress paper, we propose a method for utilizing low-power (LP) coprocessors in microcontroller SoCs using a dynamically typed managed language. The proposal introduces a JIT compiler for LP coprocessors that do not have sufficient memory and an inter-processor object management method. Our proposal enables seamless use of LP coprocessors in mruby scripts. We implemented a prototype based on mruby/c running on two microcontroller SoCs ESP32-S3 and ESP32-C6. The evaluation of their power consumption and the wake-up time of each processor shows that the proposed method has sufficient practical use.

Acknowledgments

This work was supported in part by JSPS KAKENHI Grant Numbers JP22K11967 and JP24K14892.

[3]It is thrown on the invalid memory accesses

References

[1] 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, New York, NY, USA, 105–116. https://doi.org/10.1145/3357390.3361032

[2] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 29, 16 pages. https://doi.org/10.1145/2741948.2741962

[3] Bernd Burgstaller, Bernhard Scholz, and Anton Ertl. 2006. An Embedded Systems Programming Environment for C. In *Euro-Par 2006 Parallel Processing*, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1204–1216.

[4] Maxime Chevalier-Boisvert and Marc Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 101–123. https://doi.org/10.4230/LIPIcs.ECOOP.2015.101

[5] Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. 2023. Evaluating YJIT's Performance in a Production Context: A Pragmatic Approach. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Cascais, Portugal) *(MPLR 2023)*. Association for Computing Machinery, New York, NY, USA, 20–33. https://doi.org/10.1145/3617651.3622982

[6] Espressif Systems 2023. *ESP32-S3 Technical Reference Manual*. Espressif Systems.

[7] Espressif Systems 2023. *ESP32-S3-WROOM-1, ESP32-S3-WROOM-1U Datasheet*. Espressif Systems.

[8] Espressif Systems 2024. *ESP32-C6 Technical Reference Manual*. Espressif Systems.

[9] Espressif Systems Accessed May 2024. *ESP32-C6-DevKitC-1 v1.2*. Espressif Systems. https://docs.espressif.com/projects/espressif-esp-dev-kits/en/latest/esp32c6/esp32-c6-devkitc-1/user_guide.html#current-measurement

[10] Andreas Gal, Christian W. Probst, and Michael Franz. 2006. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (Ottawa, Ontario, Canada) *(VEE '06)*. Association for Computing Machinery, New York, NY, USA, 144–153. https://doi.org/10.1145/1134760.1134780

[11] Edson Horta, Ho-Ren Chuang, Naarayanan Rao VSathish, Cesar Philippidis, Antonio Barbalace, Pierre Olivier, and Binoy Ravindran. 2021. Xar-trek: run-time execution migration among FPGAs and heterogeneous-ISA CPUs. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 104–118. https://doi.org/10.1145/3464298.3493388

[12] Infenion Technologies AG 2023. *PSoC 6 MCU: CY8C61x4, CY8C62x4 Architecture Technical Reference Manual (TRM)*. Infenion Technologies AG.

[13] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst.* 6, 1 (feb 1988), 109–133. https://doi.org/10.1145/35037.42182

[14] M5Stack. Accessed May 2024. UIFlow 2.0. https://uiflow2.m5stack.com/

[15] Nikolaos Mavrogeorgis, Christos Vasiladiotis, Pei Mu, Amir Khordadi, Björn Franke, and Antonio Barbalace. 2024. UNIFICO: Thread Migration in Heterogeneous-ISA CPUs without State Transformation. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction* (Edinburgh, United Kingdom) *(CC 2024)*. Association for Computing Machinery, New York, NY, USA, 86–99. https://doi.org/10.1145/3640537.3641565

[16] Microsoft. Accessed May 2024. DeviceScript. https://microsoft.github.io/devicescript/

[17] Moddable Tech, Inc. Accessed May 2024. Node-RED MCU Edition. https://github.com/phoddie/node-red-mcu

[18] Moddable Tech, Inc. Accessed May 2024. XS JavaScript. https://github.com/Moddable-OpenSource/moddable

[19] Konrad Moron and Stefan Wallentowitz. 2023. Support for Just-in-Time Compilation of WebAssembly for Embedded Systems. In *2023 12th Mediterranean Conference on Embedded Computing (MECO)*. 1–4. https://doi.org/10.1109/MECO58584.2023.10155088

[20] mruby developers. Accessed May 2024. mruby. https://github.com/mruby/mruby

[21] NXP Semiconductors N.V. 2019. *UM10503 LPC43xx/LPC43Sxx ARM Cortex(R)-M4/M0 multi-core microcontroller*. NXP Semiconductors N.V.

[22] NXP Semiconductors N.V. 2021. *i.MX RT1160 Processor Reference Manual*. NXP Semiconductors N.V.

[23] Kyushu Institute of Technology and Shimane IT Open-Innovation Center. Accessed May 2024. mruby/c. https://github.com/mrubyc/mrubyc

[24] Léonard Oest O'Leary, Mathis Laroche, and Marc Feeley. 2023. A R4RS Compliant REPL in 7 KB. arXiv:2310.13589 [cs.PL]

[25] George Robotics. Accessed Jan. 2024. MicroPython - Python for microcontrollers. https://micropython.org/

[26] SENSIRION 2019. *Datasheet SHT3x-DIS Humidity and Temperature Sensor*. SENSIRION.

[27] Nik Shaylor. 2002. A Just-in-Time Compiler for Memory-Constrained Low-Power Devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*. USENIX Association, USA, 119–126.

[28] Ruby Programming Shounendan. Accessed May 2024. Smalruby for mruby/c (SmT). https://github.com/gfd-dennou-club/smt-gui

[29] Kazuaki Tanaka, Avinash Dev Nagumanthri, and Yukihiro Matsumoto. 2015. mruby: Rapid Software Development for Embedded Systems. In *15th International Conference on Computational Science and Its Applications*. IEEE. https://doi.org/10.1109/ICCSA.2015.22

[30] Samuel Yvon and Marc Feeley. 2021. A small scheme VM, compiler, and REPL in 4k. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Chicago, IL, USA) *(VMIL 2021)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3486606.3486783

Received 2024-05-25; accepted 2024-06-24