Co-operative JIT Compilation for Resource-Constrained Low-Power Coprocessors

Go Suzuki, Takuo Watanabe, Sosuke Moriguchi

Department of Computer Science, School of Computing

Institute of Science Tokyo

Managed Programming Languages and Runtimes 2025

Target: Low-Power (LP) Coprocessor

SoCs with LP coprocessors are found in a variety of embedded devices.

20 mA

Main Power Domain

Main Processor

520 KiB Main Memory

Interconnect

≧ 4 MiB Flash/ROM

Peripheral Controller (I²C, SPI, etc.)

AXI?

2 mA

LP Power Domain

16 KiB LP Memory

LP Coprocessor

LP Peripheral Controller (I²C, GPIO)

LP : Low-Power

Challenges in LP Coprocessor Programming

- **No Memory Protection** on LP coprocessor

  Access to invalid pointer occurs undefined behavior (random value/system freeze).

- **Separated Memory Space**

  To share data between processors, developers must transfer them.

- **Limited Memory**

  This requires only code to be executed must be loaded on the LP memory.

- **Different ISAs** in some targets

  Developers must compile a program for each processor.

ISA: Instruction Set Architecture

Challenges in LP Coprocessor Programming

- No Memory Protection in LP coprocessor
- Separated Memory Space

→ Our runtime system checks pointer validity and moves objects, automatically.

- Limited Memory
- Different ISAs in some targets

→ Our JIT compiler generates code to be executed.

Overview of this talk

Purpose: Utilize LP coprocessors in managed languages.

1. Design considerations & Use-case: **Sensing and buffering tasks**
2. Key idea: **Co-operative JIT compilation**
   - For small footprint, and to load only code to be actually executed.
   - The main processor compiles the executed path into type-specialized basic blocks (LLBV) for the LP coprocessor, reusing the interpreter (trace-based JIT).
3. Object Management: **Different format on each processor**
4. Experiment: four mruby applications on ESP32-C6
   - achieves **power savings** comparable to **hand-written C**.
   - shows that the **resulting code size** is **2.4x to 6.6x larger than C**.
5. Discussion & Related Work: Other languages, Comparison with systems for LP coprocessor

Use-case: IoT Sensor

Copro#run offloads to the LP coprocessor.

Ruby

```ruby
sensor = SHT30.new(I2C.new())
result = Array.new(COUNT)
Copro.run do
 i = 0
 while i < COUNT do
  v = sensor.read()# Read from the sensor.
  break if v.nil?
  # if the read failed, v will be nil.
  result[i] = v
  i += 1
  Copro.delayMs(60*1000) # Sleep for 1 min.
 end
end
# on Main Processor
Network.send(result) # Send the result.
```

Object-oriented features simplify I/O device abstraction.

Only code to be executed is allocated on the LP's memory.

Objects are automatically transferred to the LP coprocessor.

Design Considerations: Target Tasks

Sensor readings and buffering tasks are suitable for LP coprocessors.

Table. Experimental and analytical investigation the efficient use of LP coprocessors (summarized "PLATFORM DESIGN IMPLICATIONS" in [43])

	Sensor Reading, Buffering Tasks	Signal Processing Tasks
Frequency	High	Low
Complexity	Simple	Complex
Power Saving	**Significant**	**Marginal**

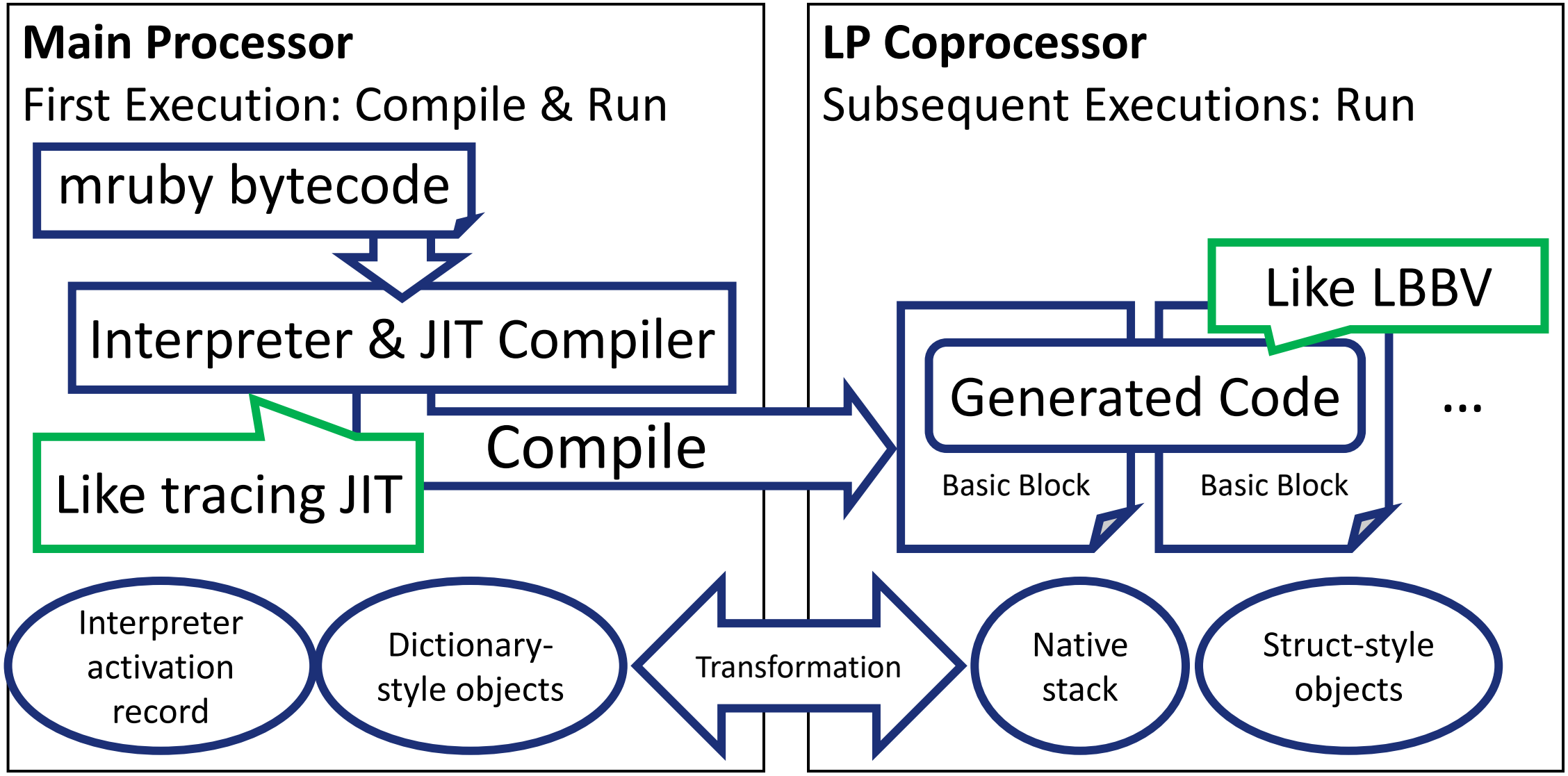
[43] Moo-Ryong Ra, Bodhi Priyantha, Aman Kansal, and Jie Liu. **Improving Energy Efficiency of Personal Sensing Applications with Heterogeneous Multi-Processors**. ACM UbiComp '12

Design Considerations: Why JIT?

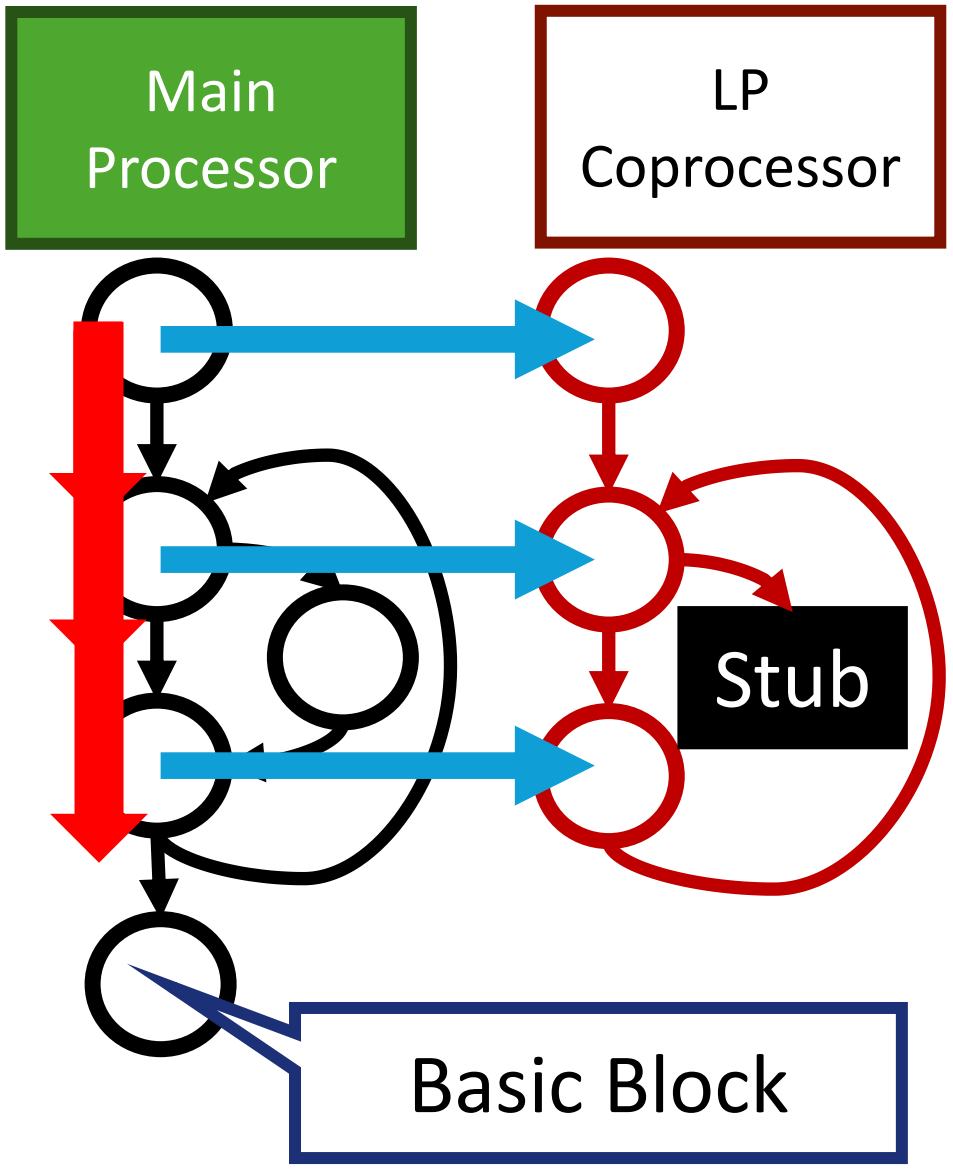
To allocate only code to be actually executed

- Managed languages have features that lead to larger code sizes.
  - **Dynamic dispatches**: Difficult to know which code to be executed, Vtables
  - **Null-checks** on each load/store (except for Rust, Swift, etc.)
  - **Type-checks** on dynamically-typed languages
- Naïve AOT compilation would have to include all possible code paths.

Key Idea: Co-operative JIT compilation



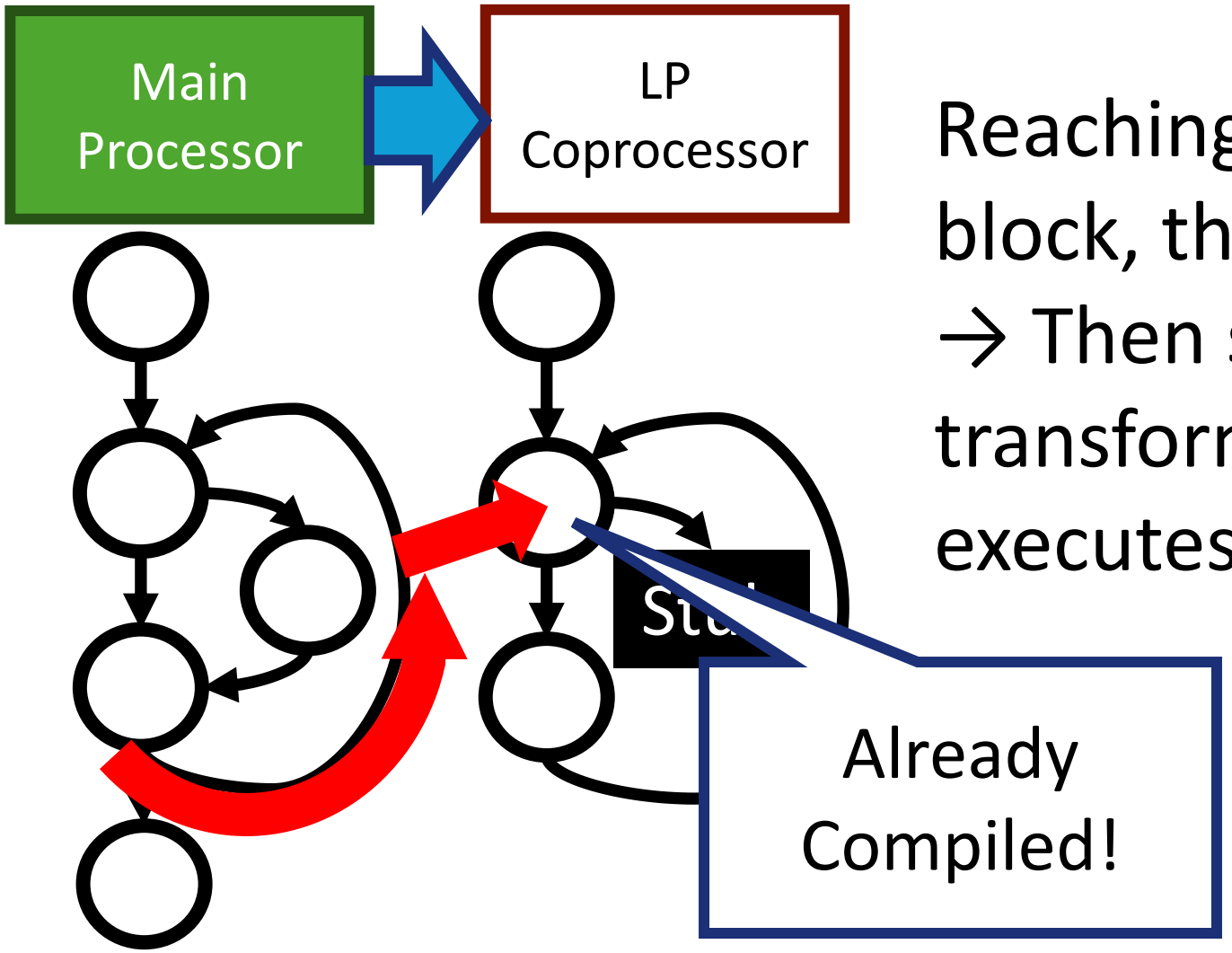
How the JIT Works: First Execution



The interpreter on Main **executes and compiles** like tracing JIT.

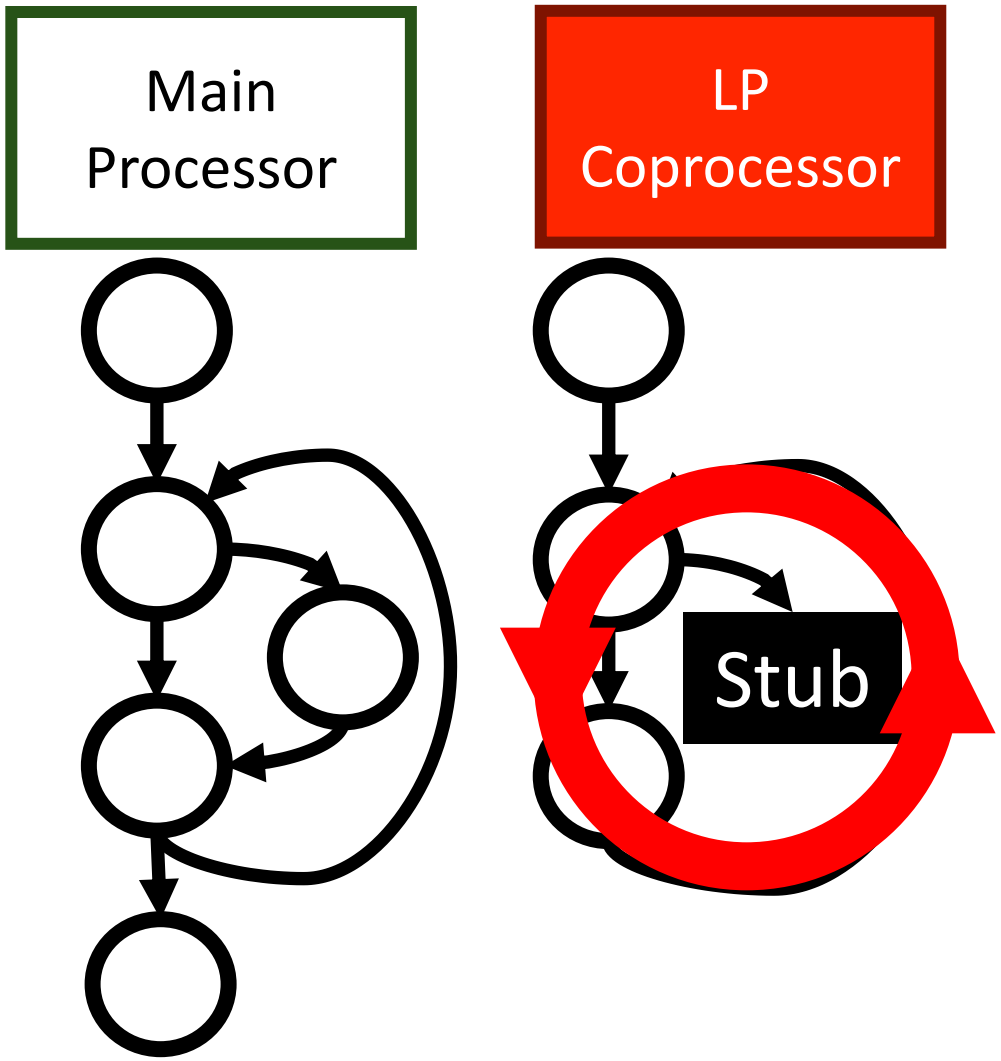
Specialized with observed types like lazy basic block versioning (LBBV).

How the JIT Works: Migration



Reaching any compiled basic block, the LP wakes up.
→ Then stack are transformed, and the LP executes the generated code.

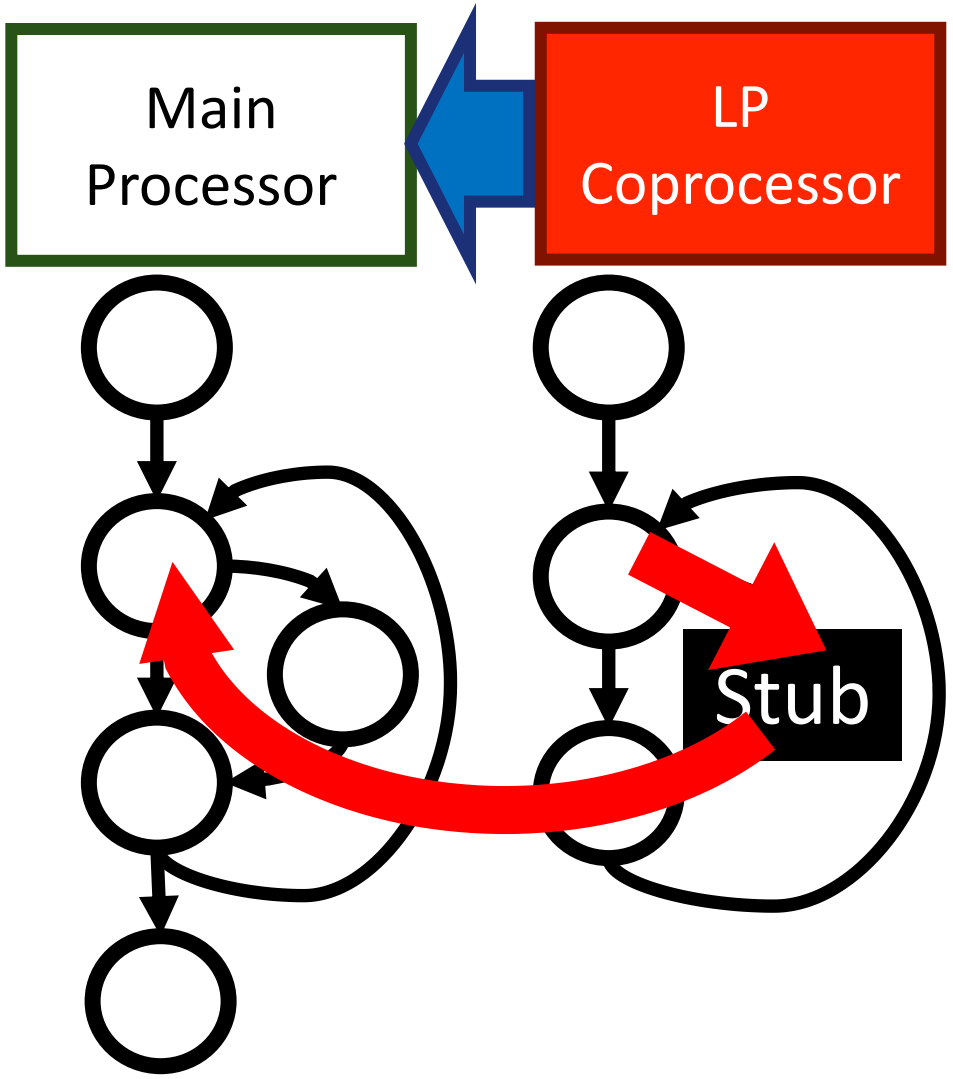
How the JIT Works: Execution on LP coproc.



Reaching any compiled basic block, the LP wakes up.
→ Then stack are transformed, and the LP executes the generated code.

To reduce migrations, we delay them, while the original LBBV migrates after it compiles each basic block.

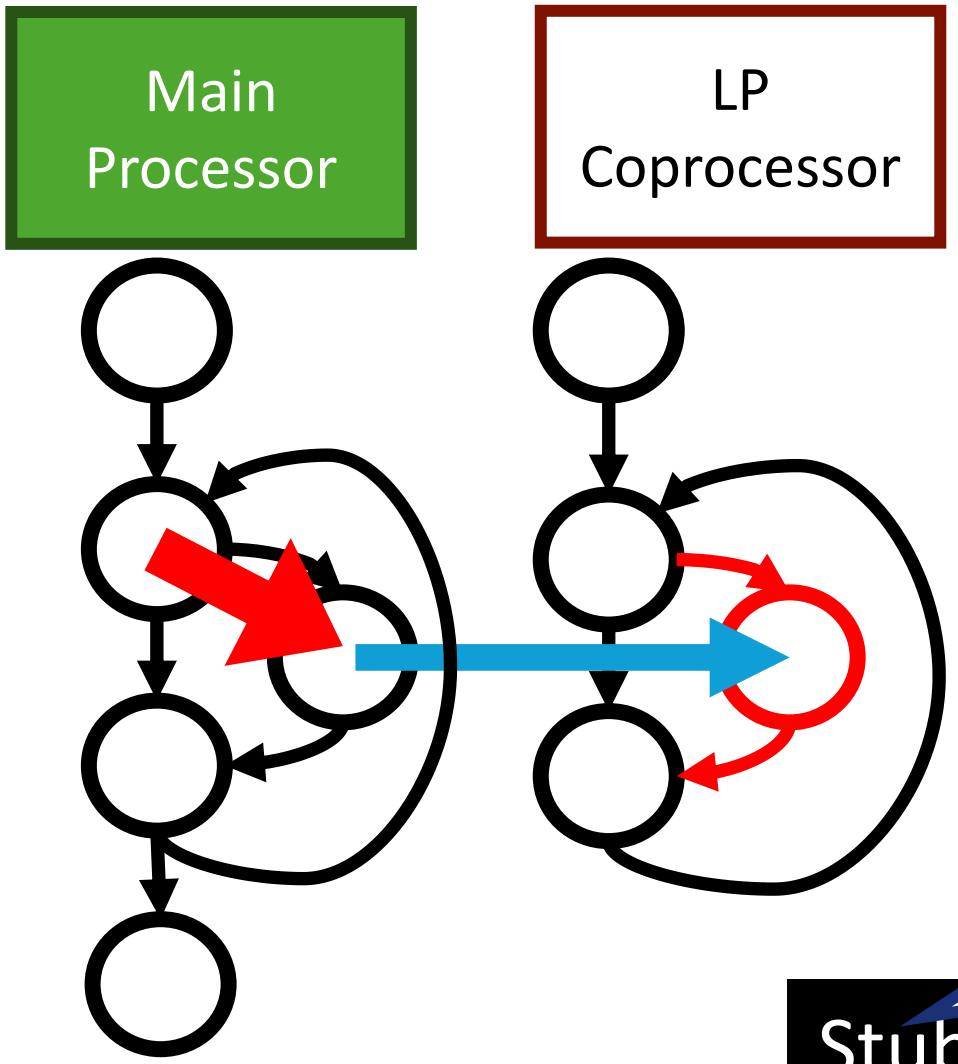
How the JIT Works: Return to the main



When an uncompiled basic block is encountered, it migrates back to the main processor (**Deoptimization**).

It jumps to a single stub function, which dumps all registers and wakes the main processor.

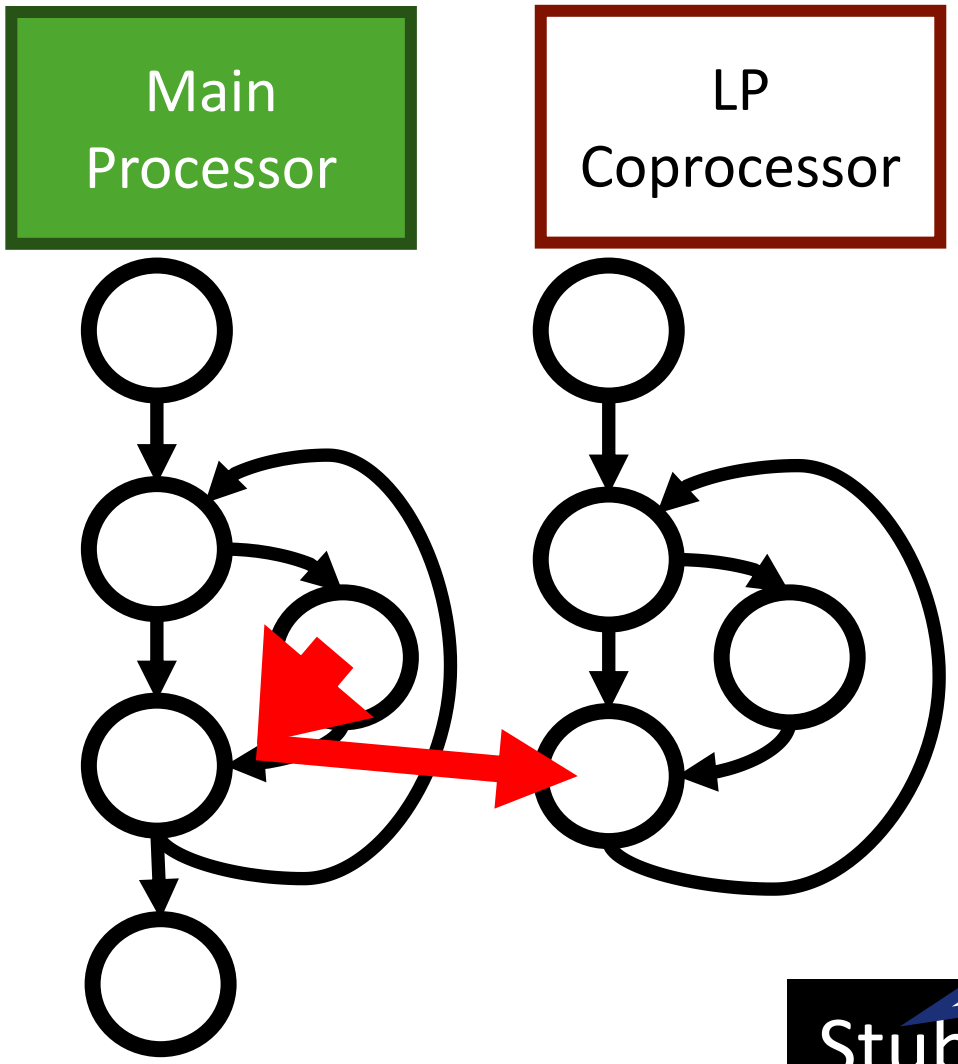
How the JIT Works: Return to the main



The basic block is compiled on the main processor, and the jump to the stub function is patched.

All uncompiled branches jump to this single stub function.

How the JIT Works: Return to the main



The basic block is compiled on the main processor, and the jump to the stub function is patched.

All uncompiled branches jump to this single stub function.

Object Management

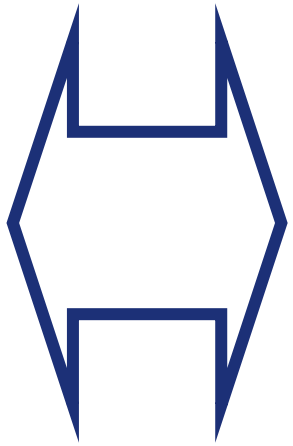
Object formats on the main and the LP are different.

- Main Processor: Dictionary for dynamic features
- LP coprocessor: Compact fixed-layout

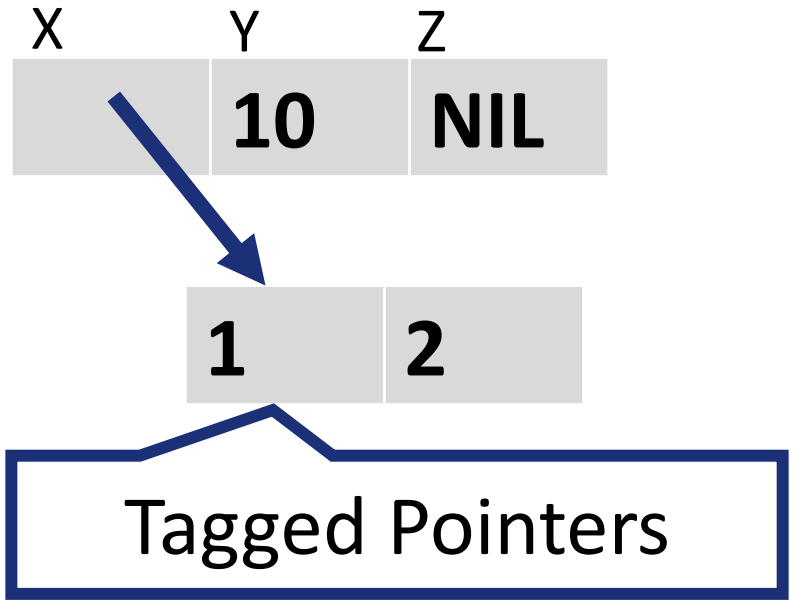
Main Processor

Key	Value
X	
Y	10
Z	NIL

Key	Value
A	1
B	2



LP Coprocessor



Object Movement

A key invariant:

Objects referenced from the stack are copied to LP and type-checked.

For this invariant, checks are performed on reading non-local variables (instance variables, global variables, elements of collections) and transforming the stack.

- To simplify the implementation.
  (No checks on types on each instructions on the code generation)
- To remove runtime checks on writes to instance variables.
  (e.g., checks for foo in “foo.bar = 12”).
- To reduce main processor wake-ups; we assume that objects on the stack are likely to be used soon.

Experiments

We implemented a prototype based on mruby/c.
4 Applications

TEMP
Temperature and humidity measuring

WL
Water Level measuring with ultrasonic sensor

LOC
Acceleration Measuring

BLED
Breathing LED

Frequency

1 samples/min — 20 s/min — 750 s/min — High (Always)

Processing

Constructing Objects — **Calculating Average** — **Sorting and Calculating Median** — Toggling GPIO

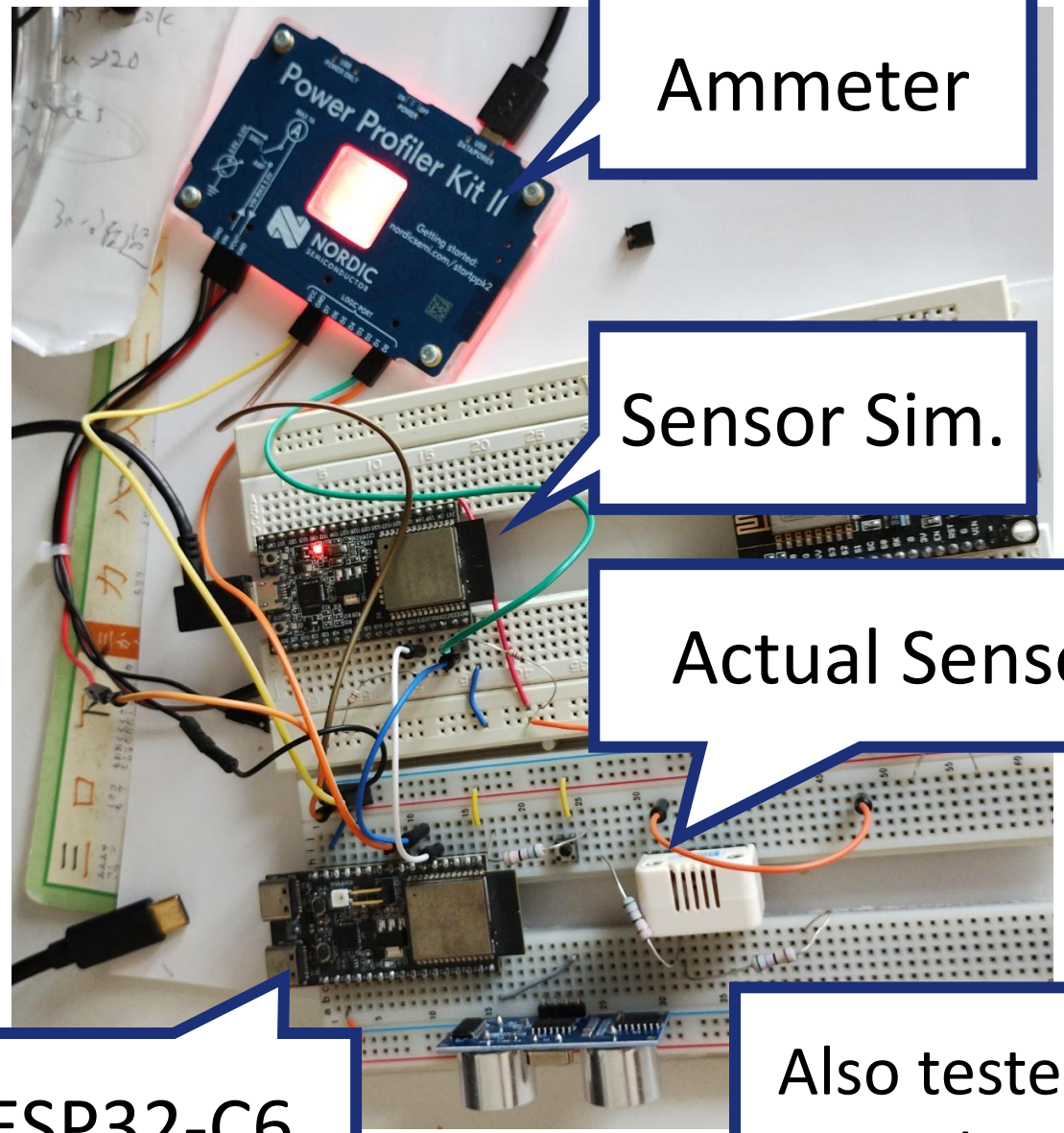
Configurations

Configurations

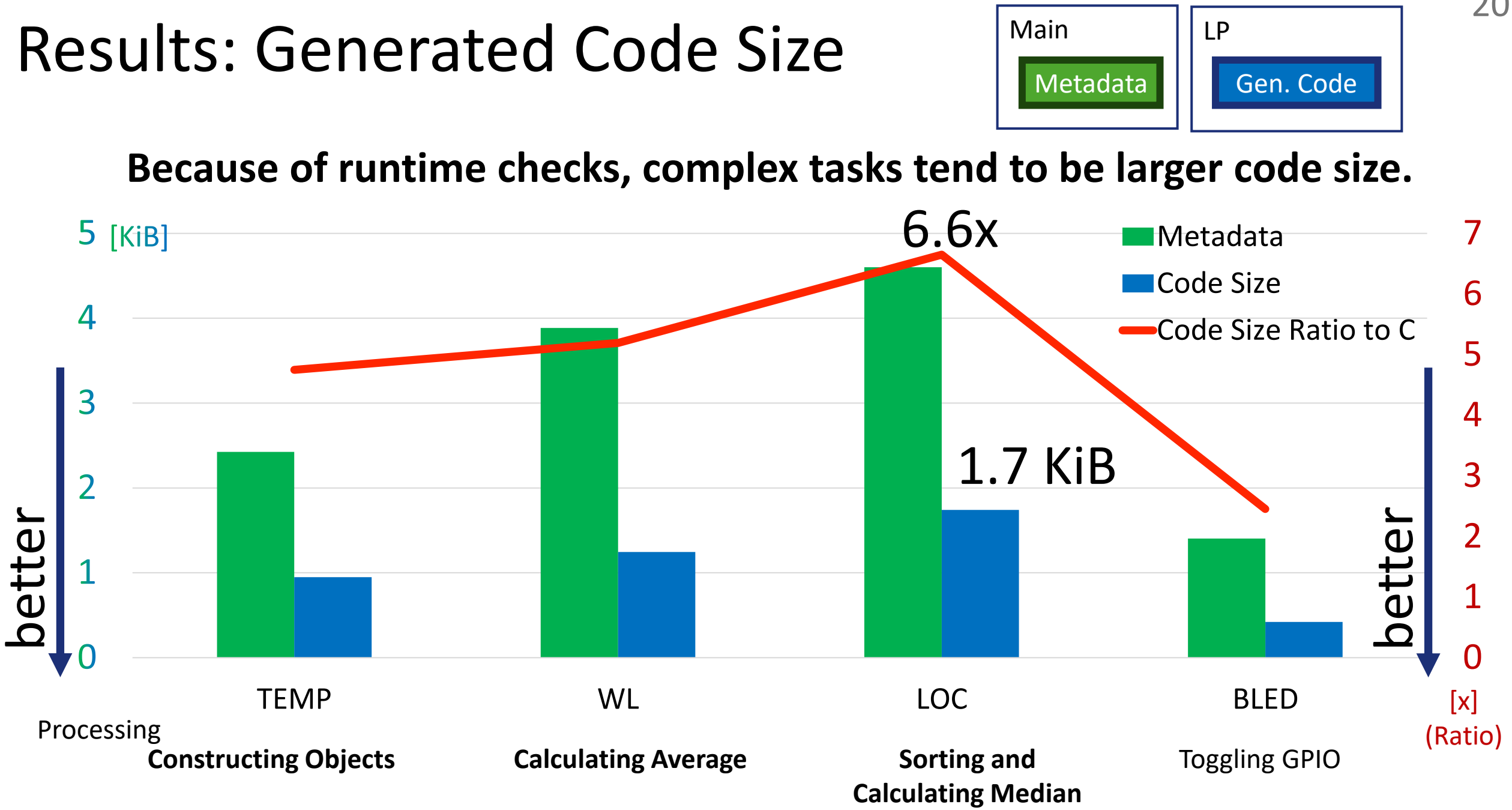
Configuration	Language	Processor
RM (original mruby/c)	mruby	Main
RL (ours)	mruby	LP
CM	C	Main
CL	C	LP

The Evaluated Environment

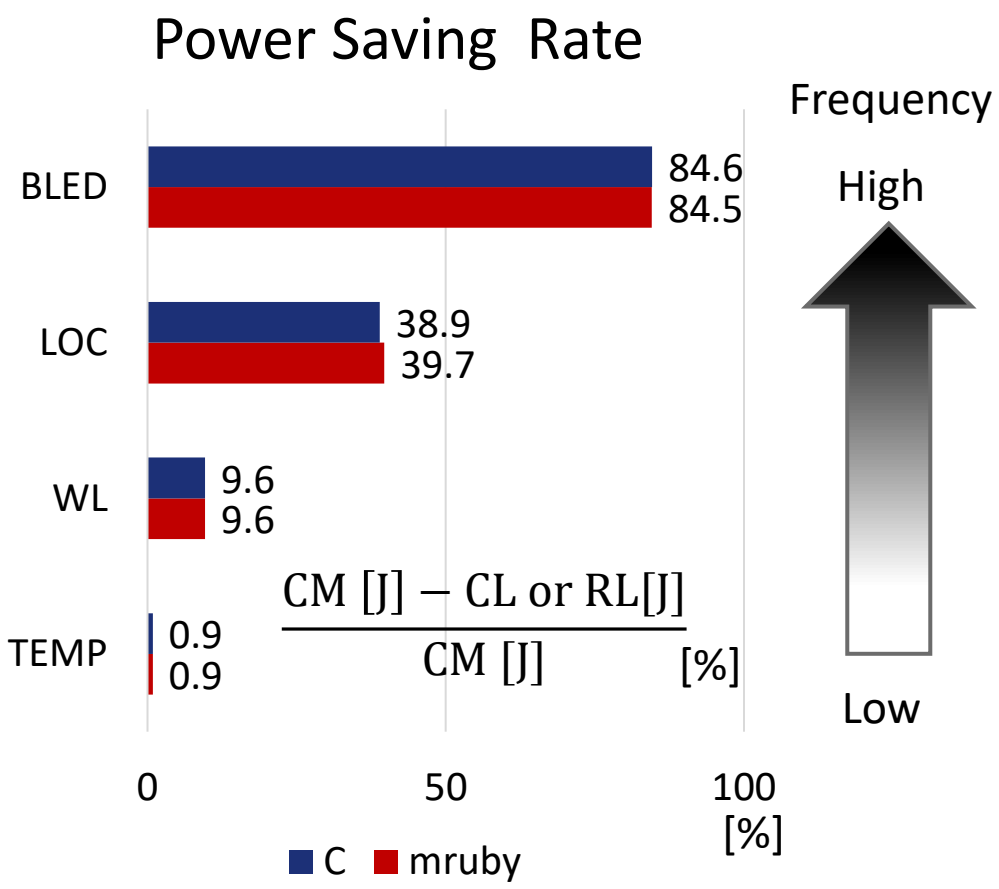
	Value
Main Freq.	80 MHz (minimum)
LP Freq.	20 MHz
Operating Voltage	3.3 V
Sample Rate	100 kHz
Ammeter & Source	Nordic Power Profiler Kit II
Evaluation Board	ESP32-C6-DevkitC-1 N8, v1.3
Sensors	Simulated by ESP32

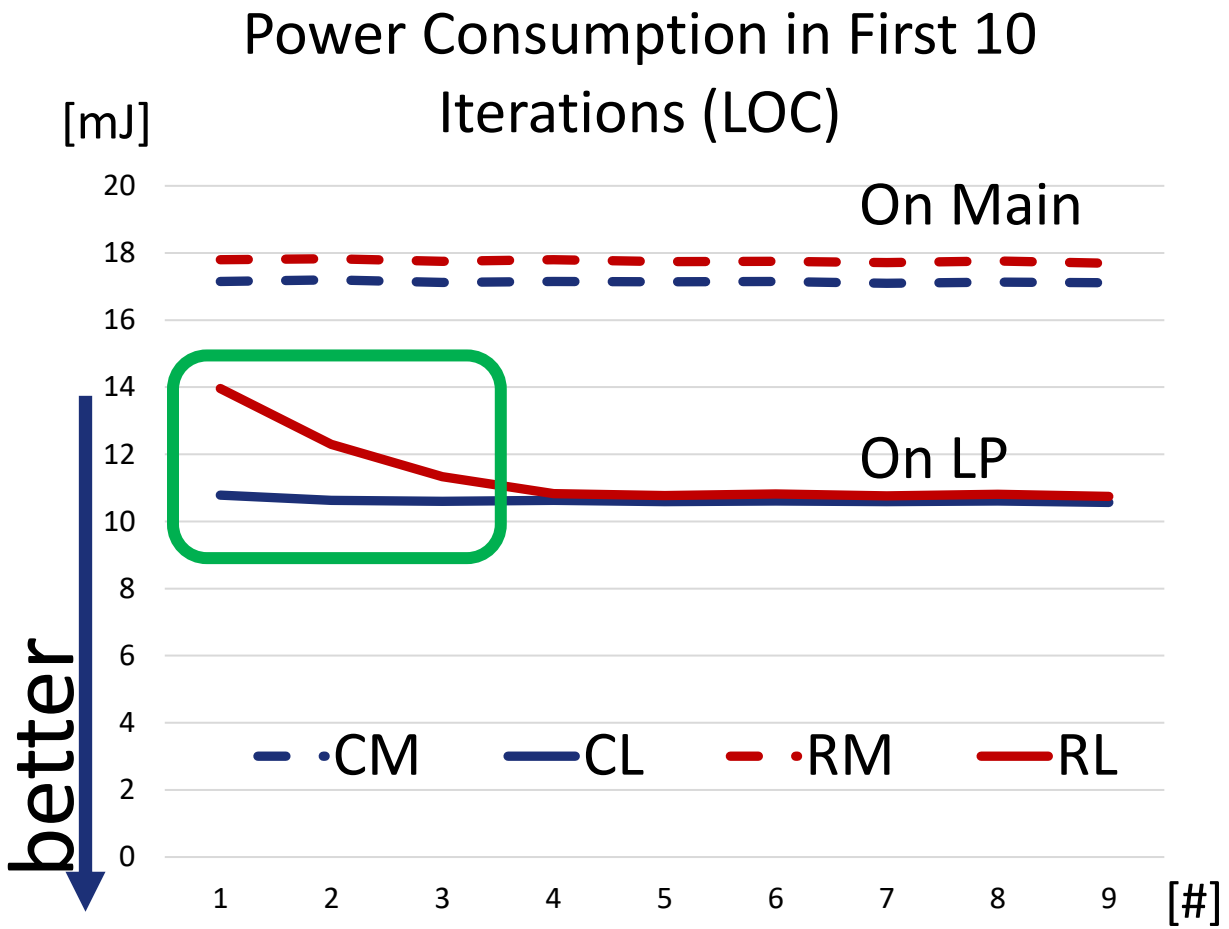


Results: Generated Code Size



Results: Power Consumption





Power Saving is Comparable to C ($\pm 0.8\%$)

We observed an initial overhead during the first few iterations, caused by JIT compilation and object transfers.

Results: Runtime Size

(RISC-V, RV32IMAC)

Main Processor

80.8 KiB (+35 %)

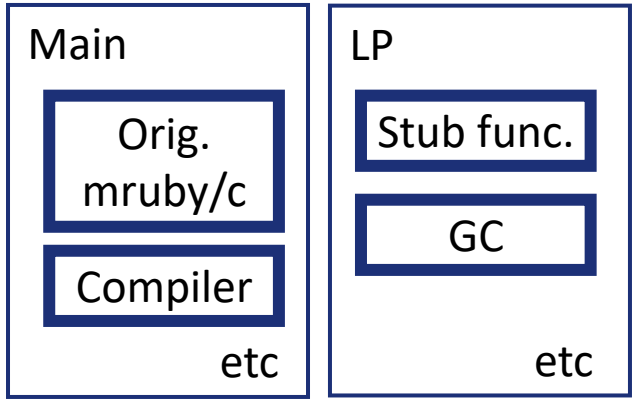
(mruby/c: 60 KiB → Ours: 80.8 KiB)

Ranking:
1st JIT Compiler (+9.8 KiB)
2nd Object Management (+2.7 KiB)
3rd Stack Transformer (+2.3 KiB)

LP Coprocessor

5.24 KiB (of 16 KiB LP SRAM)

ESP32 Runtimes	2.2
ESP32 (I2C related)	0.9
Garbage Collector	1.0
Runtime (Stub func. etc.)	0.6
Peripheral Methods	0.5



Original mruby/c runtime is over 40 KiB.
Our runtime on the LP coprocessor achieves the smaller size.

Discussion: Applicability to Other Languages

Object Management Challenges

Before dereferencing, runtime checks are required for object movement even in no GC languages and present challenges.

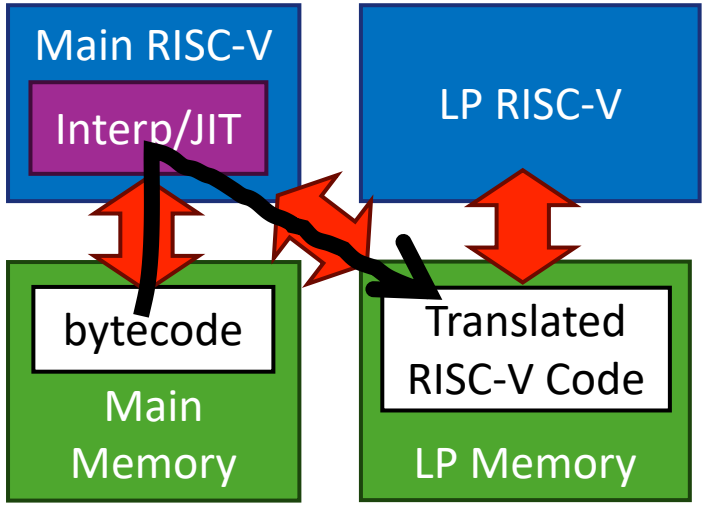
- C: limiting pointer arithmetic to guarantee the safety. (cf. Reflex [25])
- Rust: adapting Rust's ownership model (implementation) to a distributed memory environment. (cf. DRust [28])
- WebAssembly: Linear memory model makes object-level coherence difficult. WasmGC may mitigate this problem.

Statically Typed Languages: Further evaluation is required.

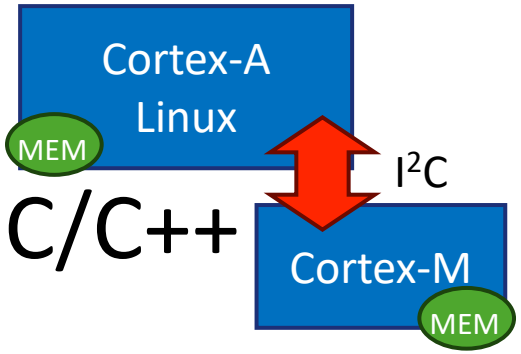
- Lazy Basic Block Versioning may be less critical, as these languages require fewer runtime checks (e.g., for arithmetic operations).
- However, we believe that dynamic loading and linking are feasible because runtime checks for object movement remains.

Related Work

This Work

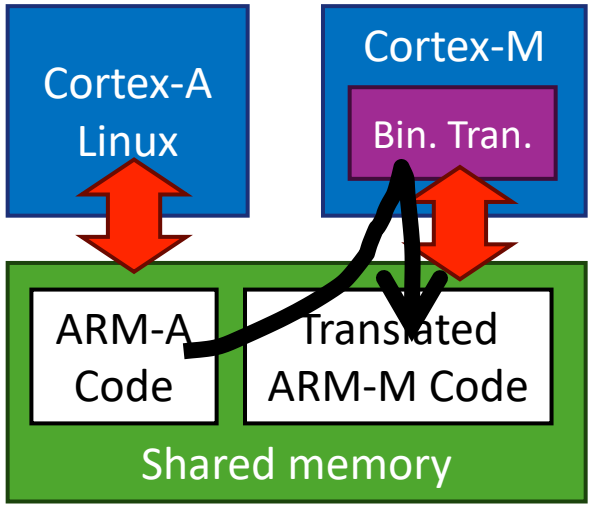


Reflex [25]

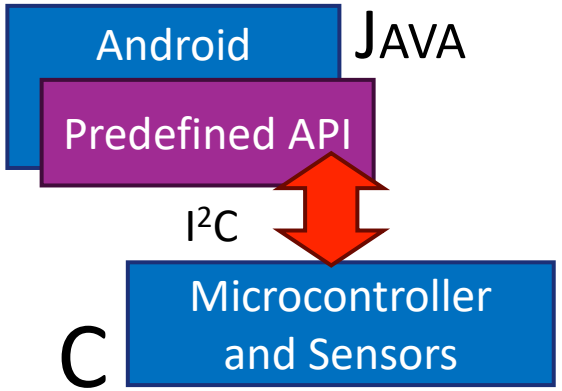


C with Software DSM
(Distributed Shared Memory)

Transkernel [17]



SideWinder [24]



This Work	Reflex		Transkernel	SideWinder
Yes	Yes	Programmable	Only Kernel	Limited
No	Yes	Processors Work Simultaneously	No	No
JIT on Main	Static	Compile?	JIT on LP	Interpreter on LP
mruby	C/C++	Language	Any (Bin. Trans.)	Java (Main) / C(LP)
ESP32-C6	OMAP4 + MSP430	Target Example	OMAP4	Nexus 4 + MSP430

Conclusion

- **The Challenge: Bridging the Gap between Productivity and Power Efficiency**
  - High-level managed languages are productive but have been impractical for resource-constrained LP coprocessors due to their large code footprint.
- **Our Contribution: A Co-operative JIT Compilation**
  - We proposed a novel JIT design where the main CPU acts as a compilation server, generating specialized, compact native code for the LP coprocessor.
  - This approach minimizes code size by compiling only executed paths and automates complex object management across heterogeneous cores.
- **Key Result: Achieving Efficiency without Sacrificing Productivity**
  - Our (subset) mruby prototype demonstrates that it is possible to gain the productivity and safety of a managed language...
  - ...while achieving power savings **comparable to a low-level, handwritten C implementation.**
- **Impact: Enabling Simpler, Safer Development for Low-Power IoT**
  - This work makes it practical for developers to build ultra-low power applications more productively, paving the way for the next generation of sophisticated IoT devices.

References

[17] Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin. **Transkernel**: Bridging Monolithic Kernels to Peripheral Cores. USENIX ATX '19

[24] Daniyal Liaqat, Silviu Jingoi, Eyal de Lara, Ashvin Goel, Wilson To, Kevin Lee, Italo De Moraes Garcia, and Manuel Saldana. **Sidewinder**: An Energy Efficient and Developer Friendly Heterogeneous Architecture for Continuous Mobile Sensing. ASPLOS '16

[25] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. **Reflex**: Using Low-Power Processors in Smartphones without Knowing Them. ASPLOS XVII (2012)

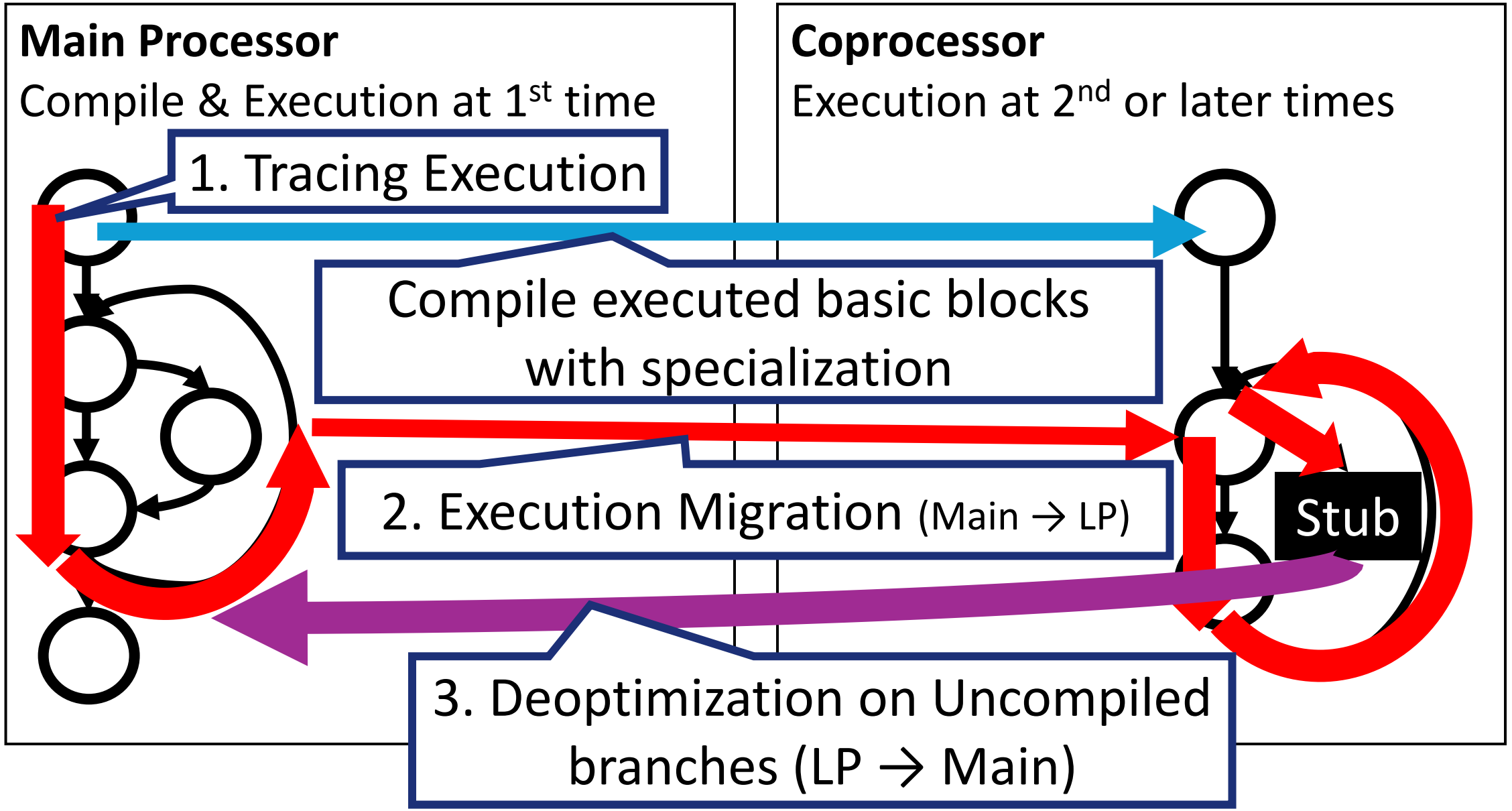
[26] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. **K2**: A Mobile Operating System for Heterogeneous Coherence Domains. ASPLOS '14

[31] Ross McIlroy and Joe Sventek. **Hera-JVM**: A Runtime System for Heterogeneous Multi-Core Architectures. OOPSLA '10

[36] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. **Helios**: Heterogeneous Multiprocessing with Satellite Kernels. SOSP '09

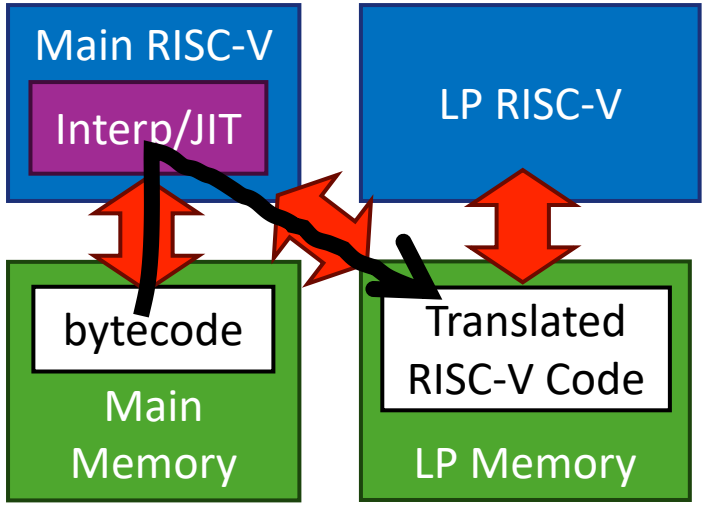
[43] Moo-Ryong Ra, Bodhi Priyantha, Aman Kansal, and Jie Liu. **Improving Energy Efficiency of Personal Sensing Applications with Heterogeneous Multi-Processors**. UbiComp '12

Key Idea: Co-operative JIT compilation

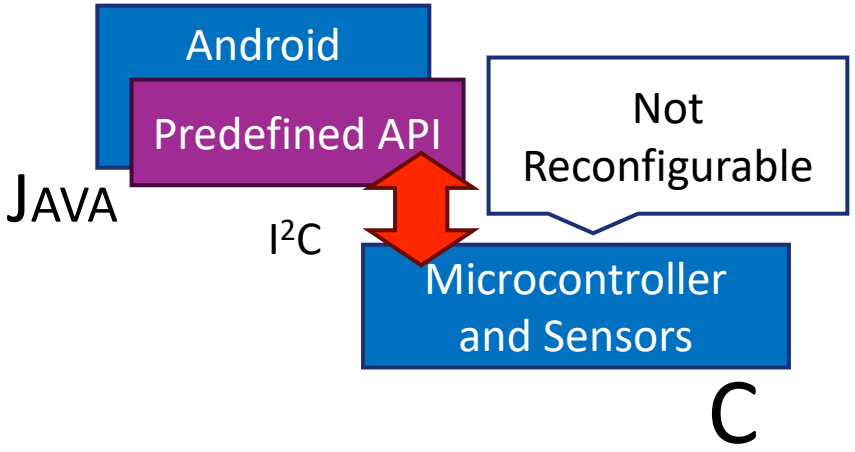


Related Work

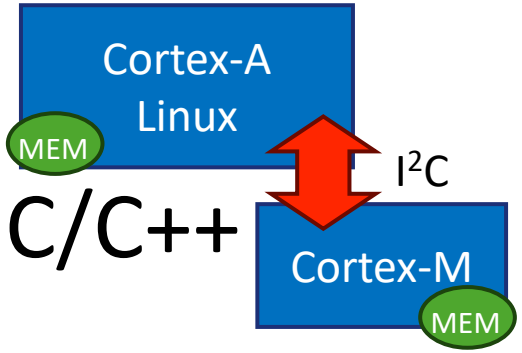
This Work



SideWinder [24]

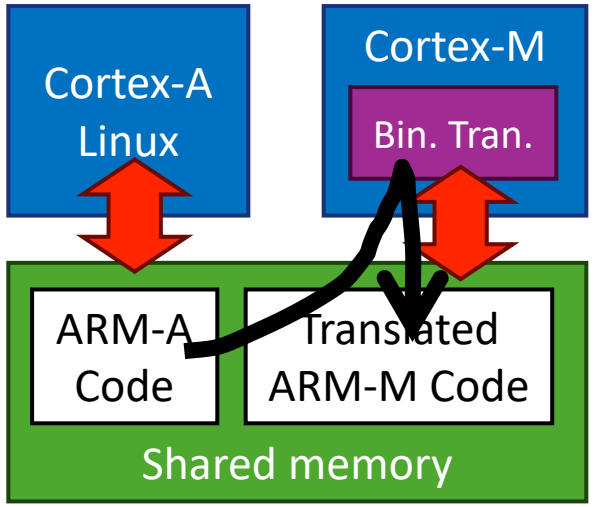


Reflex [25]

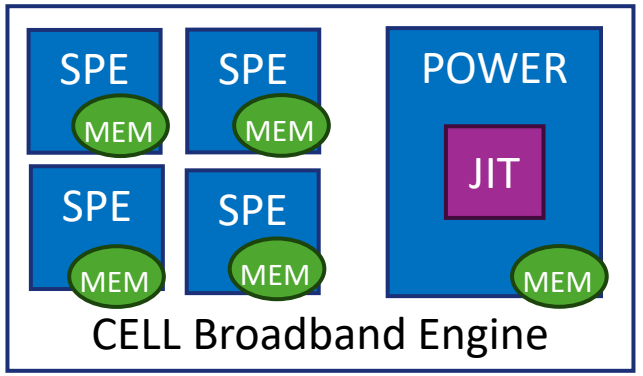


C with Software DSM
(Distributed Shared Memory)

Transkernel [17]

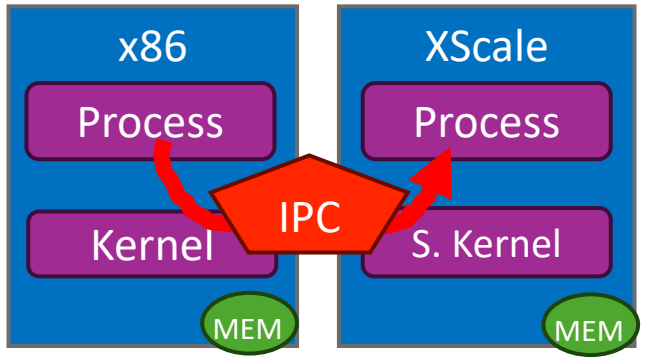


Hera-JVM [31]



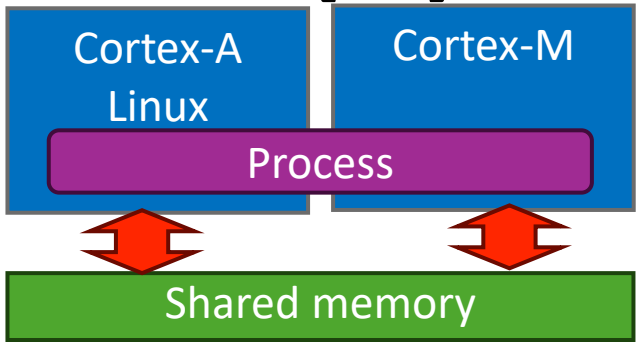
JAVA

Helios [36] (Singularity)



Sing#
(C#)

K2 [26]



Discussion: Limitation

- **If the generated code exceeds the LP memory, the code is discarded, and the compiler recompiles.**

  → Too large or pathological programs occur continuous recompilation (“thrashing”).
  → Developers must be mindful of this constraint.

- Threats to Validity
  - Implemented prototype supports a subset of mruby.
  - ESP-IDF SDK is unstable, currently.
- Runtime functions (including peripheral drivers) written in C must be pre-allocated. (not on demand)

  → Requires a dedicated dynamic loading infrastructure.

Stack Transformation

The LP coprocessor uses different stack to reduce code and stack size.

- Main Processor: original mruby/c VM activation record
- LP Coprocessor: native stack

VM Registers (mruby is a register machine) are one-to-one mapped to LP Coprocessors' RISC-V registers.

Note: We do not currently support closures (blocks).

Application Examples

```
def read()
    @i2c.write(DEV_ADDR, CMD_MID_REPEAT_READ)
    Copro.delayMs(6)
    data = @i2c.read(DEV_ADDR, 6)
    if data.size == 0 then
      return nil
    end
    t_ticks = (data.getbyte(0) << 8) + data.getbyte(1)
    rh_ticks = (data.getbyte(3) << 8) + data.getbyte(4)
    t_degC = ((t_ticks * 175)/0xFFFF) – 45 # temprerature
    rh_pRH = ((rh_ticks * 100)/0xFFFF) # humidity
    …
```

Decode the received packet.

Bubble sort

```
def sort!(ary)
  # bubble sort
  swapped = true
  while (swapped) do
    swapped = false
    i = 0
    while i < ary.length-1 do
      aryi = ary[i]
      aryi1 = ary[i+1]
      if aryi > aryi1 then
        ary[i] = aryi1
```

Constructing objects

```
  def read()
    @i2c.write(DEV_ADDR, CMD_FIFO)
    Copro.delayMs(5)
    val = @i2c.read(DEV_ADDR, 6)
    return nil if val.size == 0
    ADXLResult.new(conv(val, 0), conv(val, 2), conv(val, 4))
  end
```

Heterogeneous Microcontrollers

	Main Proc	Main Memory	LP Coproc	Memory	Interconnect
SG2002	Cortex-A/RISC-V	DRAM	8051	8 KiB	AXI?
Arduino Uno Q	Cortex-A	DRAM	Cortex-M	786 KiB	UART?
NXP i.MX93	Cortex-A	DRAM	Cortex-M	256 KiB	AXI
Infenion PSoC 62	Cortex-M4	128 KiB – 1MiB	Cortex-M0	Min. 32 KiB	AXI
ESP32	Xtensa	520 KiB	Original	8 KiB	AXI?
ESP32-S3	Xtensa	512 KiB	RV32IMC	8 KiB	AXI?
ESP32-C5	RV32IMAC	364 KiB	RV32IMAC	16 KiB	AXI?
ESP32-P4	RV32IMACF…	768 KiB	RV32IMAC	24 Ki / 20 Ki	AXI?
LPC4350	Cortex-M4	264 KiB	Cortex-M0	Min. 72 KiB	AXI
i.MX RT 1160	Cortex-M7	1 MiB	Cortex-M4	256 KiB	AXI
nRF54L15	Cortex-M33	256 KiB	RV32EMC	?	?
Analog Devices MAX32655	Cortex-M4F	128 KiB	RV32I	16-48 KiB	AXI

Implementations for Resource-constrained Devices

	NanoVM	KVM	µLISP	Ribbit	PikaPython	mruby/c
Source	(Java)	(Java)	Lisp	Scheme	(Python)	(Ruby)
Intermediate	JVM bc.	JVM bc.	S-exp.	Ribbit	Original	mruby
Target	AVR	StrongARM	AVR etc.	Linux	ARM, Xtensa	PIC24 etc.
Implementation	Interpreter	MethodJIT	Interpreter	Interpreter	Interpreter	Interpreter
Req. RAM [KiB]	1	8	2	≧ 8 ?	4	small
Req. ROM [KiB]	8	60	32	4 - 7 (lies on syscalls)	32	50

NanoVM: Till Harbaum-Impressum, http://www.harbaum.org/till/nanovm/index.shtml
KVM: Nik Shaylor (Sun Microsystems), A Just-In-Time compiler for memory constrained low-power devices. JVM'02
µLisp: David Johnson-Davies, http://www.ulisp.com/
Ribbit: Leonard Oest O'Leary, Mathis Laroche and Marc Feeley, An R4RS compliant REPL in 7KB in SCHEME Workshop
PikaPython: lyon 李昂, https://github.com/pikasTech/pikaPython

Other Languages

	Typing	Object Metadata	Expando	Unmanaged Pointers
Ruby	Dynamic (Nominal)	Yes	△	× (FFI)
Python	Dynamic (Nominal)	Yes	○	× (FFI)
TypeScript	Dynamic (Structual)	Yes	○	× (FFI)
Static TypeScript	Both (Nominal)	Yes	△ (Hash)	× (FFI)
C#	Both (Nominal) (dynamic)	Yes	△ (ExpandoObject)	△ (unsafe)
Java	Static	Yes	×	× (FFI)
Go	Static	No	×	× (JNI)
Rust	Static	No	×	△ (unsafe)
Desired	-	Yes	△/×	△/×

Implementations for ESP32 (No port work is required)

Ruby, Python, JS, C#, Go, Rust : Ported

Java : FlintESPJVM (2024/5?-) MicroEJ = closed? , NanoVM = not ported yet.

Expando

Python

```python
class Foo:
  def __init__(self)
    self.bar = 0

h = Foo ()
h.buzz = 0 # Expando
```

Ruby

```ruby
class Foo
  def initialize()
    @bar = 0; end; end
h = Foo.new ()
h.@buzz = 0  # SyntaxError
h.buzz = 0   # NameError
# not expandable.
```

In Ruby, expanding requires special class/method syntax.

Cf. https://developer.mozilla.org/ja/docs/Glossary/Expando

Overhead on migration

- Time
  - Processor wake-ups: main 0.58 ms and LP 0.02 ms (in WIP paper)
  - Object transformation: Proportional to the heap size
  - Stack transformation: Proportional to the call depth
- Power
  - Less than the WiP paper: less than 25 mA × elapsed time.

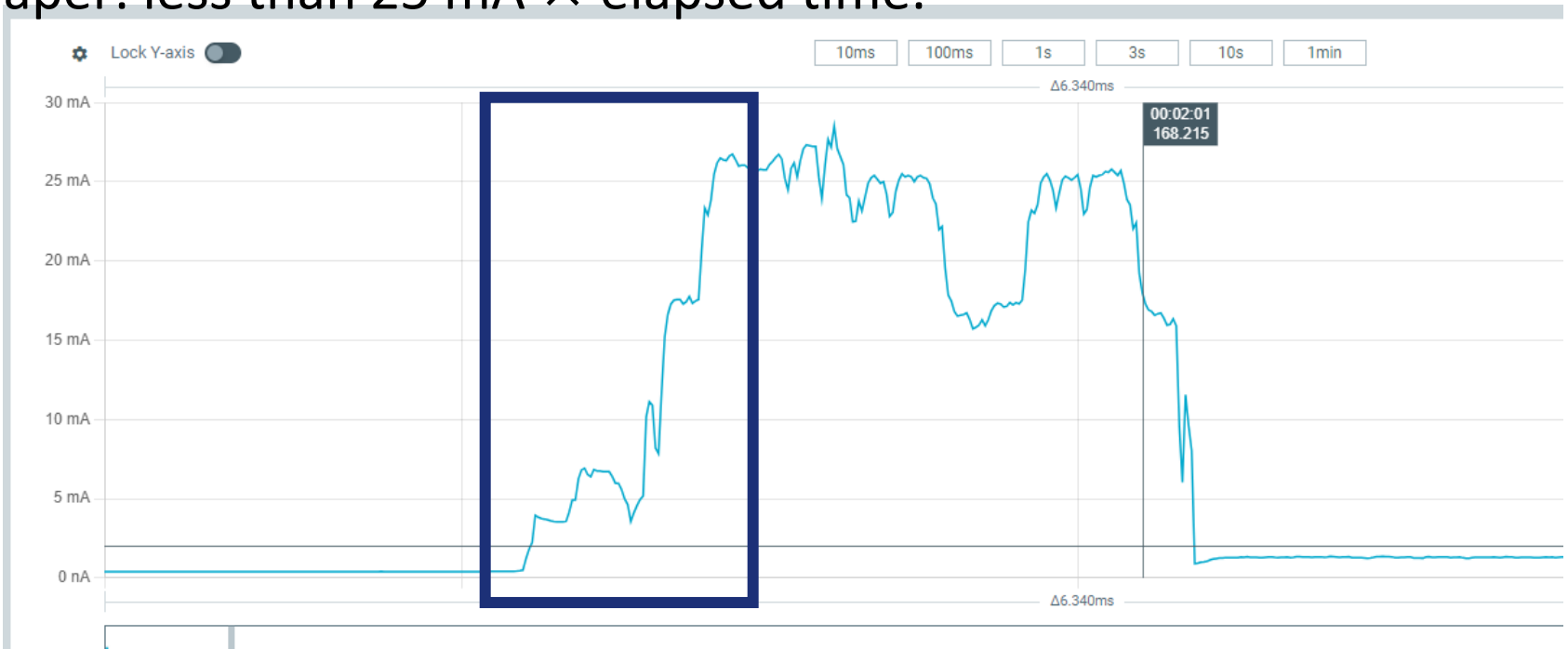


Table 4. Runtime Code Size on the LP coprocessor [KiB]

	Size	Description
ESP Runtime	2.23	Functions by ESP-IDF
ESP I^2C	0.86	Functions for I^2C
Our GC	1.03	Mark and sweep GC
Our Runtime	0.64	stub, `get_instance_variable`, etc.
`Copro` methods	0.48	I^2C, `Copro#delayMs`, etc.
Sum	5.24	

Table 5. Runtime Code Size on the Main Processor [KiB]

	.text & .rodata	.data & .bss
Ours	78.12	2.60
Original	57.49	2.50

Table 6. Generated Code and Metadata Size [B]

	RL		CL
	Generated Code	Metadata	Code Size
BLED	422	1403	172
LOC	1742	4600	262
WL	1246	3886	240
TEMP	950	2425	200

Table 7. Power Consumption [J]

	RM	RL	CM	CL
BLED	156	5.49	160	5.47
BLED*	35.5	N/A	35.7	N/A
LOC	3.93	2.37	3.85	2.40
WL	2.49	2.25	2.51	2.25
TEMP	2.29	2.27	2.28	2.27

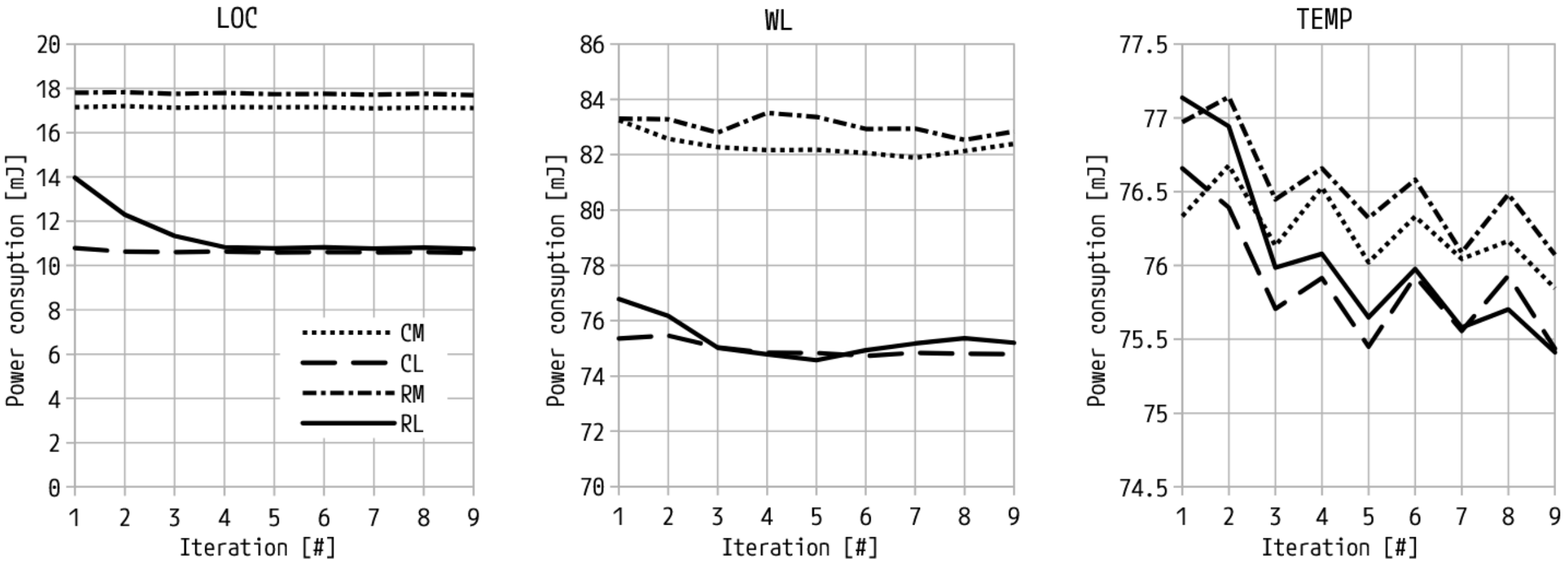


Figure 4. Power consumptions in the first 10 iterations

Runtime Size Comparison with LBBV (preliminary)

- Ours: 22.49 KiB
  - A single interpreter: 22.49 KiB
- LBBV-based (preliminary evaluation): 28.78 KiB
  - Original mruby/c interpreter: 12.69 KiB
  - Compiler (stripping down our modified interpreter to its code-generation logic): 16.09 KiB
- Size Reduction: 22% (6.29 KiB)

Experiments