Write a Ruby interpreter in Ruby (& C) for Ruby 3

Koichi Sasada
Cookpad Inc.
<ko1@cookpad.com>

cookpad

This talk is about

- New proposal to write built-in method definitions in Ruby and C
  - MRI built-in libraries
  - Extension libraries

- Not about how to write VM/GC/… in Ruby

Today's talk

- Current problems with C-methods
- Proposal: Writing builtin methods in Ruby with C
- Performance hacks
  - Runtime-performance: New FFI instruction
  - Startup-time: New compiled binary features

Koichi Sasada

http://atdot.net/~ko1/

- A programmer
  - 2006-2012 Faculty
  - 2012-2017 Heroku, Inc.
  - 2017- Cookpad Inc.
- Job: MRI development
  - Core parts
    - VM, Threads, GC, etc

cookpad

Background
MRI built-in method definitions

- Well-known fact: MRI is written in C
- Most of built-in methods are written and defined in C
- A few methods written in Ruby, “prelude.rb” introduced from Ruby 1.9.0.
  - Unlike libraries, it is loaded by default and bundled with a MRI binary (you can’t edit it)

Built-in methods definitions in C

```
// quote from string.c Init_String()
rb_cString  = rb_define_class("String", rb_cObject);
…
rb_define_method(rb_cString, "<=>", rb_str_cmp_m, 1);
rb_define_method(rb_cString, "==", rb_str_equal, 1);
rb_define_method(rb_cString, "===", rb_str_equal, 1);
…
rb_define_method(rb_cString, "length", rb_str_length,
…
```

create String class

method name

C function name

arity

When the method called, corresponding C function will be called.

Built-in methods definitions in C
Method body function

```
# String#length impl.
VALUE
rb_str_length(VALUE str)
{
    return LONG2NUM(
      str_strlen(str, NULL));
}
```

How many classes/methods?

```
# ruby --disable-gems

require 'objspace'
p ObjectSpace.count_objects[:T_CLASS] +
  ObjectSpace.count_objects[:T_MODULE]
  #=> 526 # How many classes/modules?
p ObjectSpace.count_imemo_objects[:imemo_ment]
  #=> 2363 # How many method entries?
```

Problems

1. Annotation (meta data)
2. Performance
3. Productivity
4. API change for “Context”

Problem 1
Annotation (meta data)

Ruby's functionality issue

- Lack of meta-data compare with methods defined in Ruby (Ruby methods)
  - For example: "Method#parameters"

```
def hello(msg) puts "Hello #{msg}"; end
p method(:hello).parameters
#=> [[:req, :msg]]

p method(:is_a?).parameters
#=> [[:req]] # no parameter name
```

Other meta data
- Backtrace (stack-prof)
- Source (if exists)
- …

Problem 1
Annotation (meta data)

Ruby's runtime performance issue

- Need more information for further optimizations
- We can not know behaviors of C-methods unless analyzing "C-code" and it is feasible.
  - Can throw exceptions?
  - Has side-effect?
- If we know a method is "pure", we can apply more aggressive optimizations, such as passing "frozen" string instead of making new Strings.
  - `Ex: str.gsub("goodby", "hello")`
  - In this example, two strings can be frozen singleton objects (don't need "frozen-string-literal" pragma).

Problem 1
Annotation (meta data)

Ruby's loading time issue

- We can not know how many methods are defined before running.
  - We can not allocate method table at once.
  - Now we grow a method table on demand.

Problem 2
Performance

- Known fact: “C” is faster than Ruby
  - Most of case, it is true
  - Sometimes **it is false**
- Keyword parameters and exception handling are typical examples

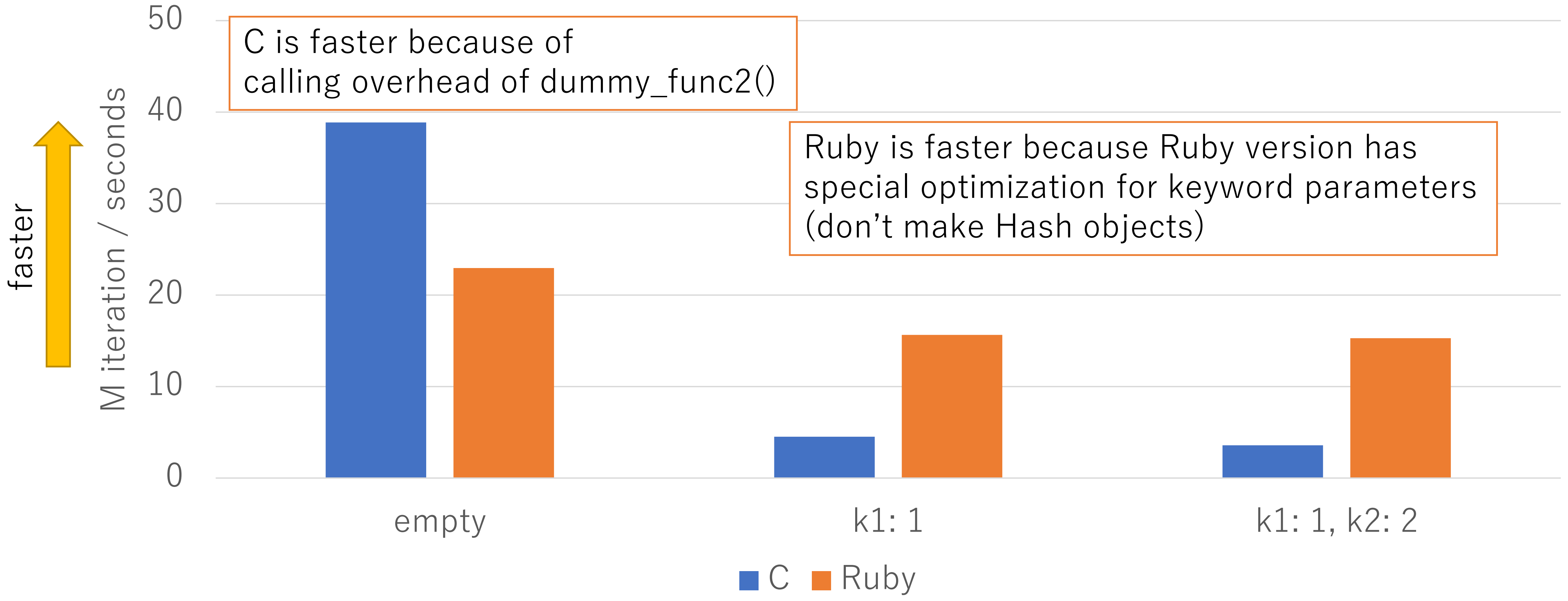
Problem 2
Performance: Keyword parameters in C

```
# Ruby
def dummy_func_kw(k1: 1, k2: 2)
  dummy_func2(k1, k2)
end
```

```
static VALUE
tdummy_func_kw(int argc, VALUE *argv, VALUE self)
{
    VALUE h;
    ID ids[2] = {rb_intern("k1"), rb_intern("k2")};
    VALUE vals[2];

    rb_scan_args(argc, argv, "0:", &h);
    rb_get_kwargs(h, ids, 0, 2, vals);
    return tdummy_func2(self,
                        vals[0] == Qundef ? INT2FIX(1) : vals[0],
                        vals[1] == Qundef ? INT2FIX(2) : vals[1]);
}
```

Performance problem on Keyword parameters in C

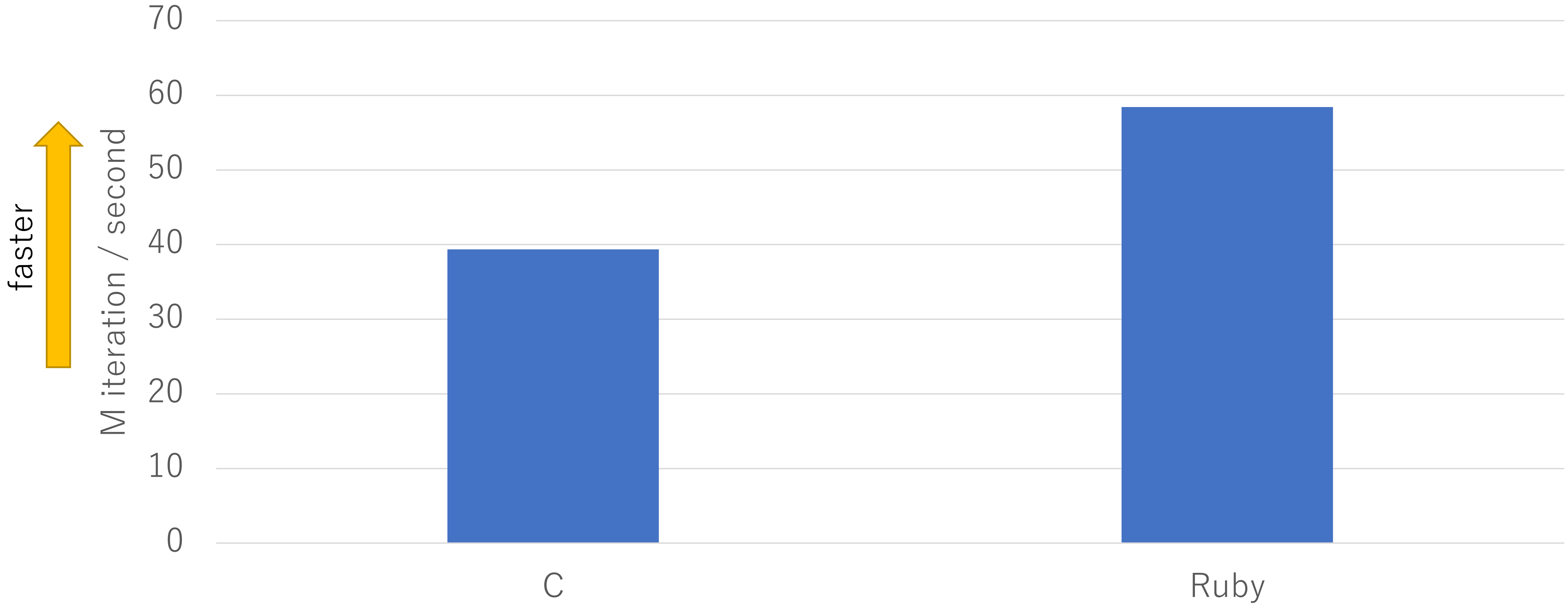


Problem 2
Performance: Exception in C

```c
static VALUE
dummy_body(VALUE self)
{
    return Qnil;
}
static VALUE
dummy_rescue(VALUE self)
{
    return Qnil;
}
static VALUE
tdummy_func_rescue(VALUE self)
{
    return rb_rescue(dummy_body, self,
                     dummy_rescue, self);
}
```

```ruby
# in Ruby
def em_dummy_func_rescue
  rescue
end
```

Problem 2
Performance: Exception in C



Problem 3
Productivity

- Ruby C-API
  - It is **easy** to make simple methods such as `String#length` in C
  - It is **difficult** to make methods using **complex features.**
    - Exception handling
    - Keyword parameters
    - Iterators

```
# String#length impl.
VALUE
rb_str_length(VALUE str)
{
    return LONG2NUM(
      str_strlen(str, NULL));
}
```

Problem 3
Productivity: Keyword parameters in C

```
# Ruby
def dummy_func_kw(k1: 1, k2: 2)
  dummy_func2(k1, k2)
end
```

```
static VALUE
tdummy_func_kw(int argc, VALUE *argv, VALUE self)
{
    VALUE h;
    ID ids[2] = {rb_intern("k1"), rb_intern("k2")};
    VALUE vals[2];

    rb_scan_args(argc, argv, "0:", &h);
    rb_get_kwargs(h, ids, 0, 2, vals);
    return tdummy_func2(self,
                        vals[0] == Qundef ? INT2FIX(1) : vals[0],
                        vals[1] == Qundef ? INT2FIX(2) : vals[1]);
}
```

Problem 3
Productivity: Exception in C

```c
static VALUE
dummy_body(VALUE self)
{
    return Qnil;
}
static VALUE
dummy_rescue(VALUE self)
{
    return Qnil;
}
static VALUE
tdummy_func_rescue(VALUE self)
{
    return rb_rescue(dummy_body, self,
                     dummy_rescue, self);
}
```

```ruby
# in Ruby
def dummy_func_rescue
  nil
rescue
  nil
end
```

Problem 3
Productivity

- Written in C is reasonable
  - if it is performance required (frequently used)
  - if it is easy to implement
  - if we cannot implement it in Ruby
- Written in Ruby is reasonable
  - if it is not frequently used, non-performance required features (such as TracePoint, and so on)
  - if it should try with proto-type

Problem 4
API changes for “Context”

- C API needs update to accept “Context”
  - To implement Guild system (or other parallel execution mechanism), we need to access “context” object.
  - Current API doesn’t accept a context object

```
# String#length impl.
VALUE
rb_str_length(VALUE str)
{
    return LONG2NUM(
      str_strlen(str, NULL));
}
```

mruby accepts “mrb_state” data as context!

```
# mruby String#length
static mrb_value
mrb_str_size(mrb_state *mrb, mrb_value self)
{
  mrb_int len = RSTRING_CHAR_LEN(self);
  return mrb_fixnum_value(len);
}
```

Problem 4
API changes for “Context”

- Getting “Context” from Thread-local-storage (TLS) is one idea
  - Good: We can keep current API. It is hard to bring new specification without carrots (飴).
  - Bad: Very slow to access TLS, especially from .so (.dll)

For details:

Sasada, et al. Parallel Processing on Multiple Virtual Machine for Ruby
笹田等, Ruby用マルチ仮想マシンによる並列処理の実現 (2012)

Problems and requests

1. Annotation (meta data)
   - We need **DSL** instead of C code
2. Performance
   - Sometimes **Ruby is faster**
3. Productivity
   - Sometimes **Ruby is enough** to implement
4. API changes for “Context”
   - We need **brand new** definition APIs

Problems and requests

1. Annotation (meta data)
   - We need **DSL** instead of C code
2. Performance
   - Sometimes **Ruby is faster**
3. Productivity
   - Sometimes **Ruby is enough** to implement
4. API changes for “Context”
   - We need **brand new** definition APIs

We should know good DSL supporting language!!

Solution

Write method definitions in

Ruby PROGRAMMING Language

with C

Solution
Writing definitions (declarations) in Ruby

- Write a definitions or declarations in Ruby
  - We can analyze Ruby code in advance
  - We can embed meta-data (method attribute) in Ruby DSL
- Call C functions if needed
  - We only need to call already implemented C functions (w/ small modifications)
- We don’t need to replace all of defs
  - We can move new defs gradually.

Solution
Writing definitions (declarations) in Ruby

1. Annotation (meta data)
   - Write a code in Ruby and analyze it
   - Add additional annotation method
2. Performance
   - Introduce **new FFI feature** to call C function
   - Sometimes pure-Ruby is enough
3. Productivity
   - Some cases (kwparams, and so on) are definitely easy to write in Ruby
4. API changes for “Context”
   - New FFI passes “Context” parameter “ec”

Where should we write definitions?

Current definitions

```
// string.c
str_length() {…}

Init_String(){
  …
 rb_define_method(…);
 rb_define_method(…);
 rb_define_method(…);
 …
}
```

compile

Ruby binary

Proposed definitions

```
# string.rb
class String
  def …; end
  def length
    …
  end
end
```

```
// string.c
str_length()
{
    …
}
…
```

analysis and combine

Ruby binary

New FFI feature to call C functions

```
# string.rb
class String
  …
  def length
    __ATTR__.pure
    __C__.str_length
  end
end
```

Declare no side effect

call

```
# String#length impl. with new FFI
static VALUE
str_length(rb_ec_t *ec, VALUE str)
{
    return LONG2NUM(
      str_strlen(str, NULL));
}
```

added!

NOTE
Keywords are not fixed yet
These keywords are enabled on special compile mode (not a syntax proposal for ordinal Ruby)

Programming with new FFI

- You can use Ruby features as you want ☺
  - Parameter analyzing (opt, kw, …)
  - Complex feature like exception handling, iterators, …
- You can write C code ☺
  - Reuse existing C implementation
  - Write fast code with C
- You **need** to care a bit more😒
  - GVL release (interrupt) timing, GC timing…

Questions:
Is it slow than C methods?

- Runtime overhead concerns
  - FFI can be runtime overhead ☹
    - Primitive methods like `String#length` can affect this kind of overhead
- Startup time concerns
  - Compile time can increase a loading time ☹

Today’s technical achievements

- Runtime overhead
→ Fast FFI implementation by new instructions

- Loading time
→ Improve compiled binary format

Fast FFI implementation by new instructions

FFI (Foreign Function call) instruction "invokecfunc"

- Introduce a new instruction **"invokecfunc"** in the virtual machine

```
# string.rb
class String
  def length
    __C__.str_length
  end
end
```

```
== disasm: #<ISeq:length@string.rb:10>
0000 invokecfunc
0002 leave
```

Optimize “invokecfunc”

- In fact, “invokecfunc” is slow compare with C methods
  - Additional overhead for VM stack manipulation
  - Additional overhead for VM frame manipulation

- Policy: DO EVERYTHING I CAN DO

Optimize “invokecfunc”

- Most of cases, methods become delegator type definitions. Passing same parameters.

```
def dummy_func2 a, b
  __C__.dummy_func2(a, b)
end
```

```
0000 getlocal a@0, 0
0003 getlocal b@1, 0
0006 invokecfunc <dummy_func2/2>
0008 leave
```

a, b: Same as parameters

Optimize “invokecfunc”

- Most of cases, methods become delegator type definitions. Passing same parameters.

→ Special instruction: **invokecfuncwparam**

```
def dummy_func2 a, b
  __C__.dummy_func2(a, b)
end
```

```
0000 invokecfuncwparam<dummy_func2/2>
0002 leave
```

Optimize “invokecfunc”

- Most of cases, methods become delegator type definitions. “call&return”

```
def dummy_func2 a, b
  __C__.dummy_func2(a, b)
end
```

```
0000 invokecfuncwparam<dummy_func2/2>
0002 leave
```

Optimize “invokecfunc”

- Most of cases, methods become delegator type definitions. “call&return”

→ Special instruction: `invokecfuncwparamandleave`

```
def dummy_func2 a, b
  __C__.dummy_func2(a, b)
end
```

```
0000 invokecfuncwparamandleave …
0002 leave
```

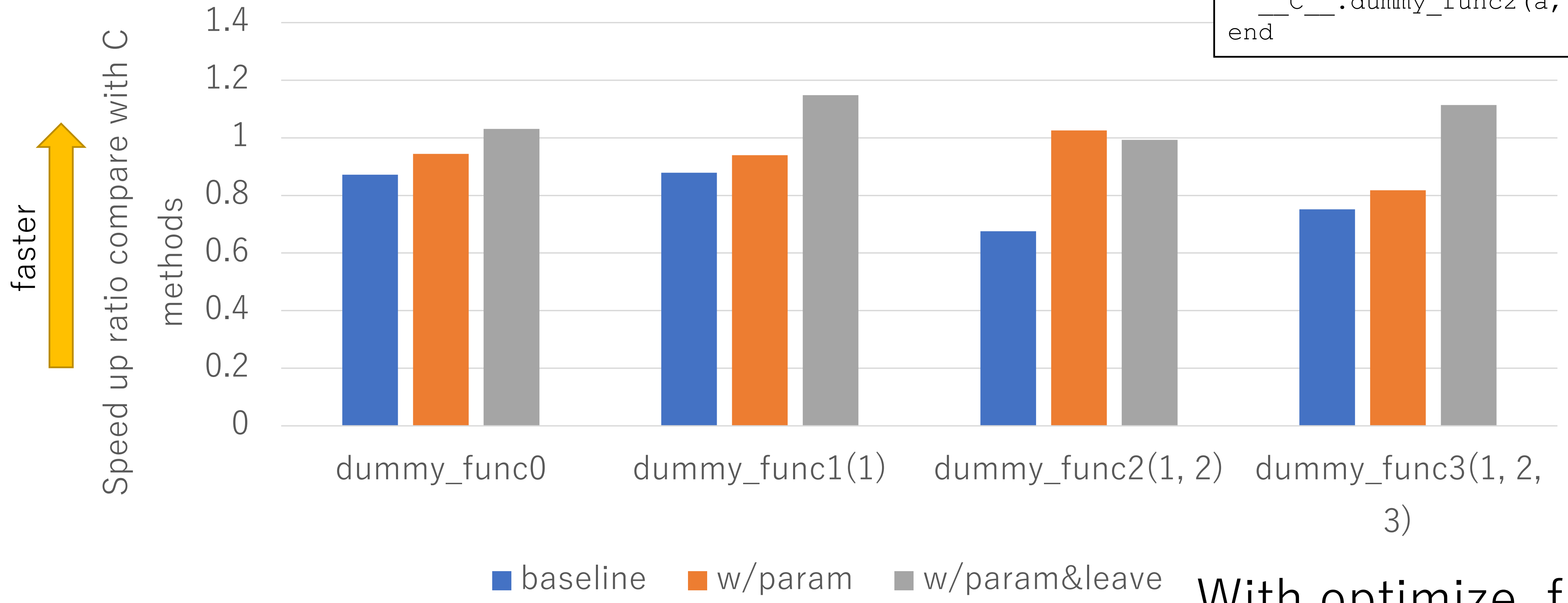
NOTE: To support TracePoint, “leave” is required yet.

Evaluation

- Prepare empty methods and compare them
  - Traditional C methods
    - rb_define_method(…)
  - Ruby methods calls empty C function
    - def dummy_func; __C__.dummy_func(); end
- Apply optimizations
  - baseline: invokecfunc
  - w/param: invokecfuncwparm
  - w/param&leave: invokecfuncwparamandleave

Evaluation
With positional arguments

```
def dummy_func0
  __C__.dummy_func0
end
def dummy_func1 a
  __C__.dummy_func1(a)
end
def dummy_func2 a, b
  __C__.dummy_func2(a, b)
end
```

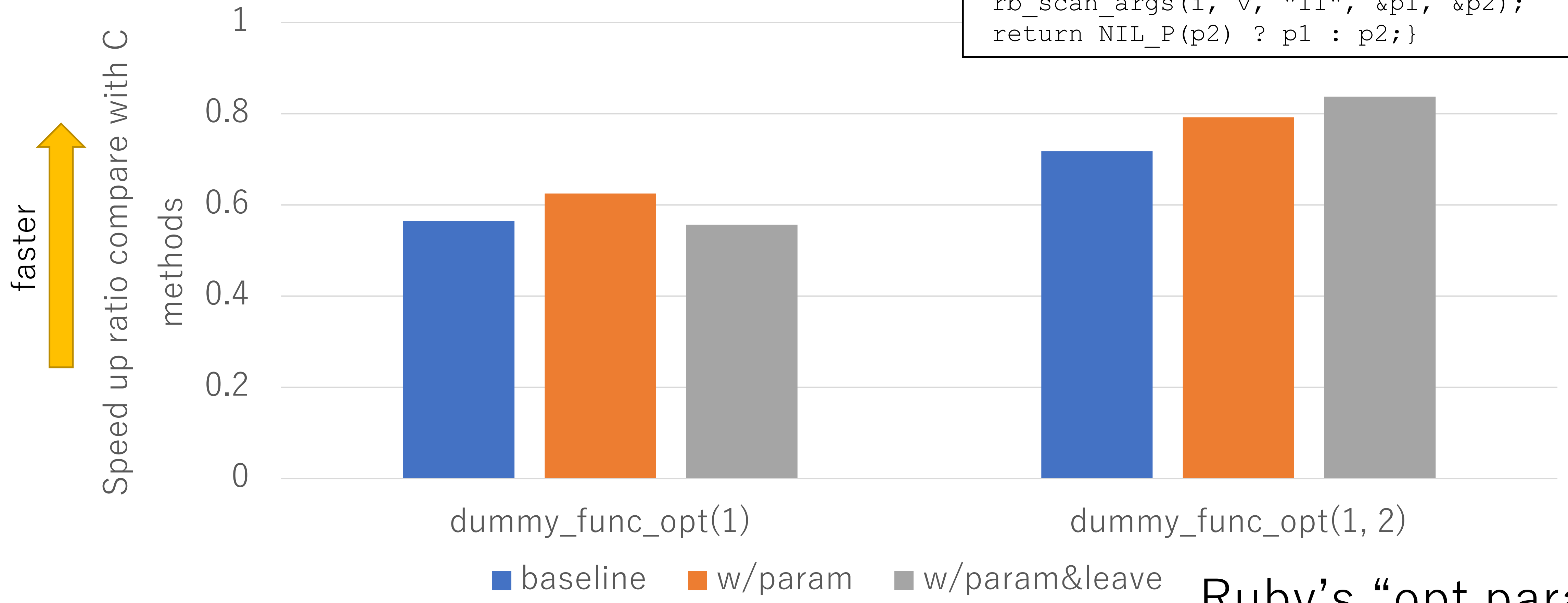


With optimize, faster than C methods! ☺

Evaluation
With optional arguments

```
def dummy_func_opt a, b=nil
  __C__.dummy_func_opt(a, b)
end
```

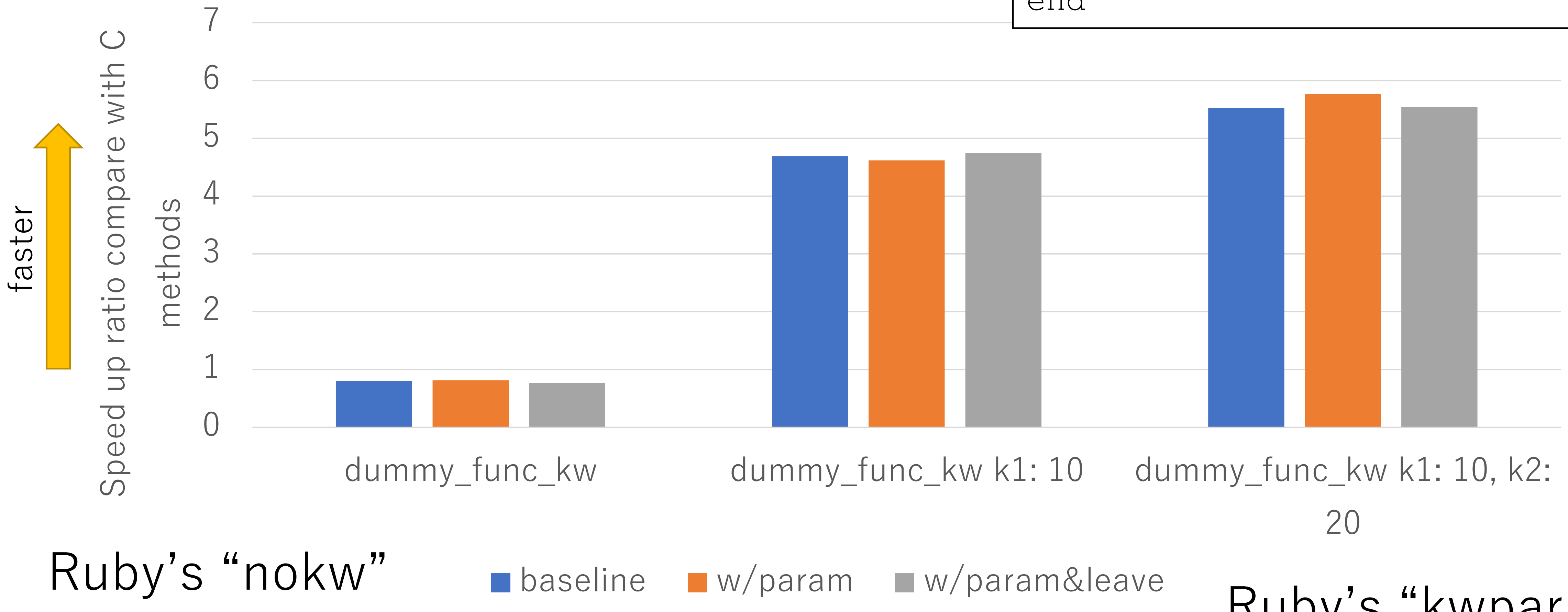
```
static VALUE
dummy_func_opt(int i, VALUE *v, VALUE self)
{VALUE p1, p2;
 rb_scan_args(i, v, "11", &p1, &p2);
 return NIL_P(p2) ? p1 : p2;}
```



Ruby’s “opt params” is slower ☹

Evaluation
With keyword arguments

```
def dummy_func_kw k1:1, k2:2
  __C__.dummy_func2(k1, k2)
end
```



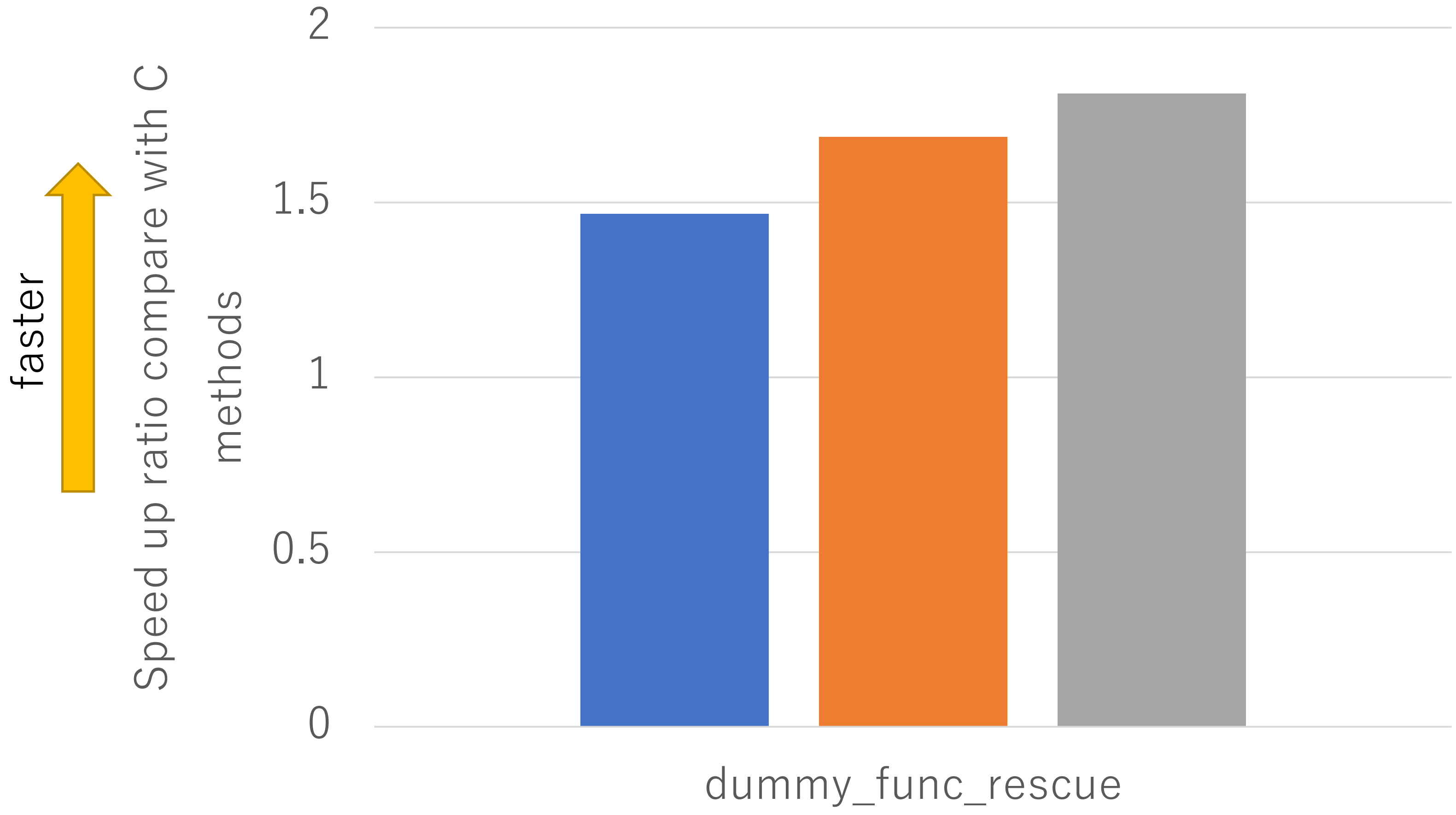
Ruby's "nokw" is slower ☹

Ruby's "kwparms" is 4~5 faster! ☺

Evaluation
Rescue

```
def dummy_func_rescue
  __C__.dummy_func0
rescue
  __C__.dummy_func0
end
```

```
static VALUE
dummy_func_rescue(VALUE s)
{
return
  rb_rescue(dummy_func0, self,
            dummy_func0, self);
}
```



Ruby's "rescue" is 1.5 faster! ☺

FFI instruction “invokecfunc” Summary

- Good performance ☺
  - Many cases, “__C__.func” new FFI calls are faster than C methods
  - Some cases, significant improvements (keyword parameters / exception handling)
  - Optional parameters are slow ☹
    - Built-in methods have many optional arguments so it is important problem
- Enjoy Ruby’s productivity ☺

FFI instruction “invokecfunc”
Future work

- Introduce arity overloading for slow opt params

```
# example syntax
overload def foo(a)
  __C__.foo1(a)
end
overload def foo(a, b)
  __C__.foo2(a, b)
end
```

At method dispatch, we can find appropriate method body and we can cache it in inline cache!

FFI instruction “invokecfunc” Related work

Koichi Sasada: Ricsin: A System for “C Mix-in to Ruby” (Ricsin: RubyにCを埋め込むシステム) (2009.3)

```
# Writing C in Ruby code
def open_fd(path)
  fd = __C__(%q{ // passing string literals to __C__ methods
    /* C implementation */
    return INT2FIX(open(RSTRING_PTR(path), O_RDONLY));
  })
  raise 'open error' if fd == -1
  yield fd
ensure
  raise 'close error' if -1 == __C__(%q{
    /* C implmentation */
    return INT2FIX(close(FIX2INT(fd)));
  })
end
```

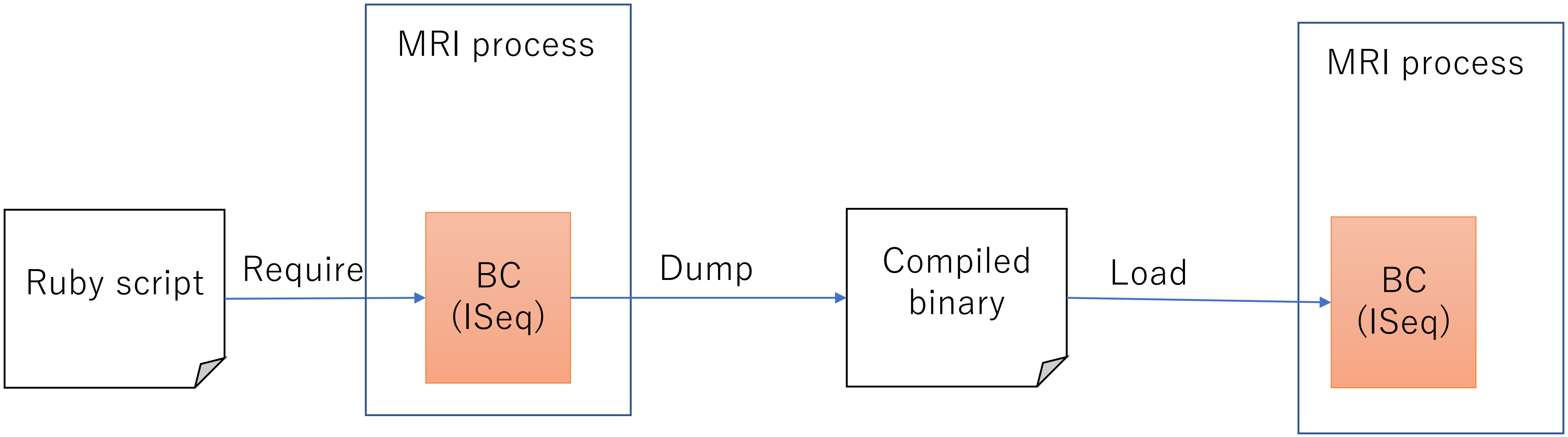
Interestingly, C code can refer Ruby variables ☺

Improve compiled binary format

Compiled binary format?

- Bytecode dumped binary
  - `bin = RubyVM::InstructionSequence#to_binary`
  - `RubyVM::InstructionSequence.load_from_binary(bin)`
  - Introduced from Ruby 2.3 (by me ☺)
- AOT compiling feature
  - bootsnap (maybe) use it

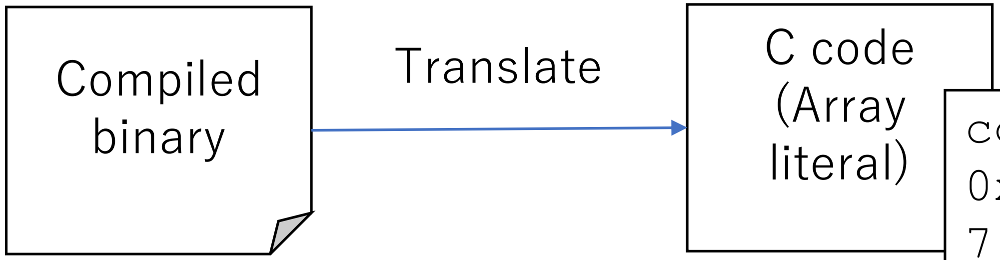
Compiled binary: creation and usage



Faster loading because
we don’t need to parse/compile.

Embed compiled binary into MRI

- For short startup time, we can bundle compiled binary with MRI binary




```c
const char compiled_binary[] = {
0x59,0x41,0x52,0x42,0x02,0x00,0x00,0x00,0x0
7,0x00,0x00,0x00,0xbc,0x56,0xf8,
0x00,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x0
0,0xf0,0x87,0x00,0x00,0xce,0x0b,
0x00,0x00,0x8d,0x9f,0x00,0x00,0xf8,0x6a,0xe
1,0x00,0x38,0x4b,0xdf,0x00,0x00,
0x6b,0xe1,0x00,0x88,0xd8,0xf5,0x00,0x78,0x3
8,0x36,0x5f,0x36,0x34,0x2d,0x6c,c
```

NOTE
We can use mmap at startup.

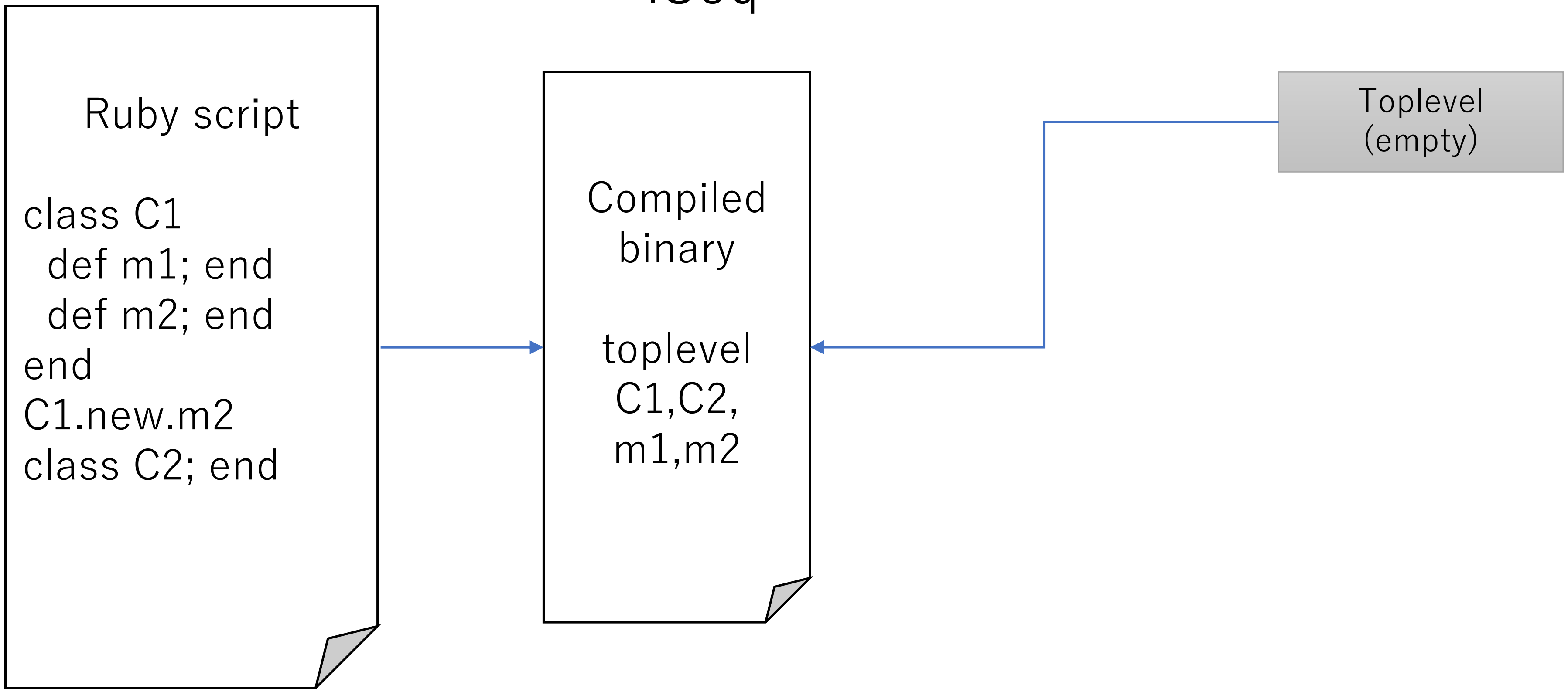
Further optimizations for startup-time

- Lazy loading
  - RubyKaigi 2015, but not enabled yet on MRI
- Multiple file supported binary

Technique
Lazy loading

Idea: Load only really used ISeq
Most of methods are not used in a process

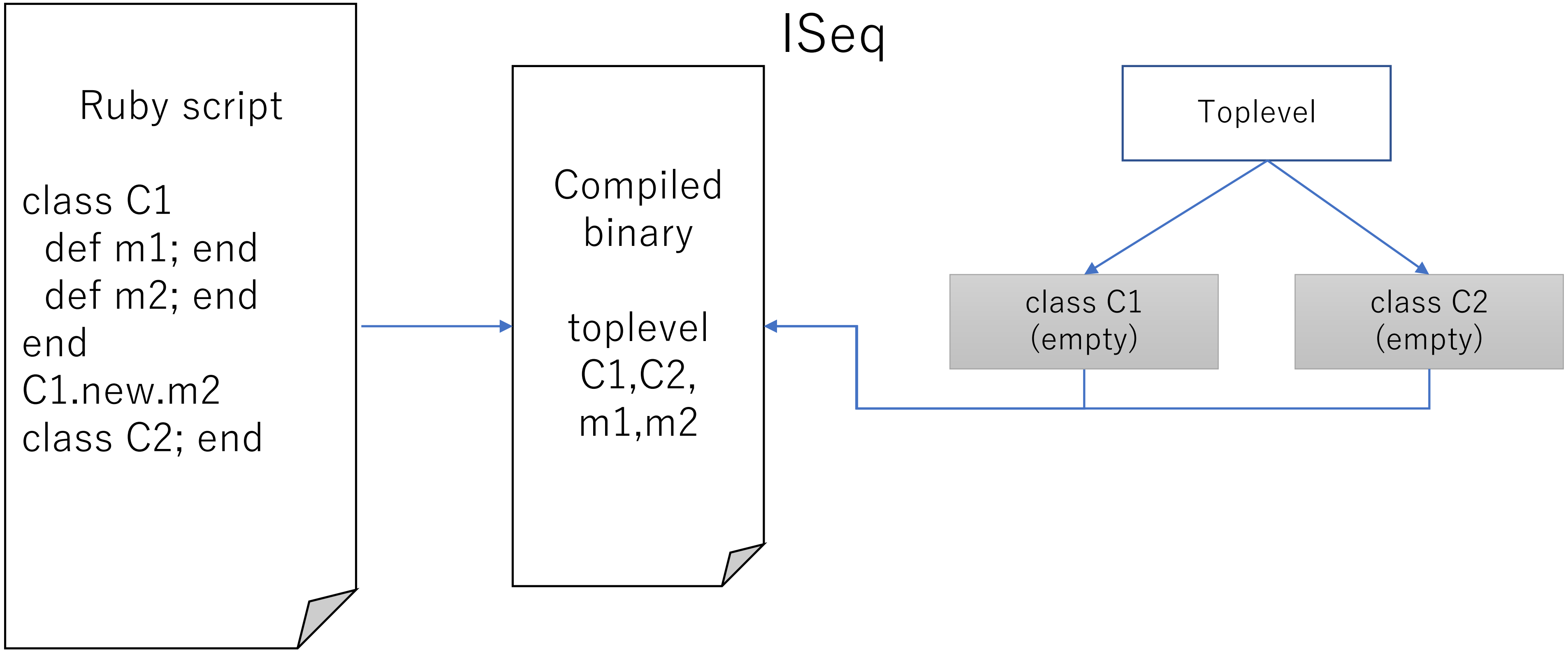
(1)Load and make an empty toplevel ISeq



Technique
Lazy loading

Idea: Load only really used ISeq
Most of methods are not used in a process

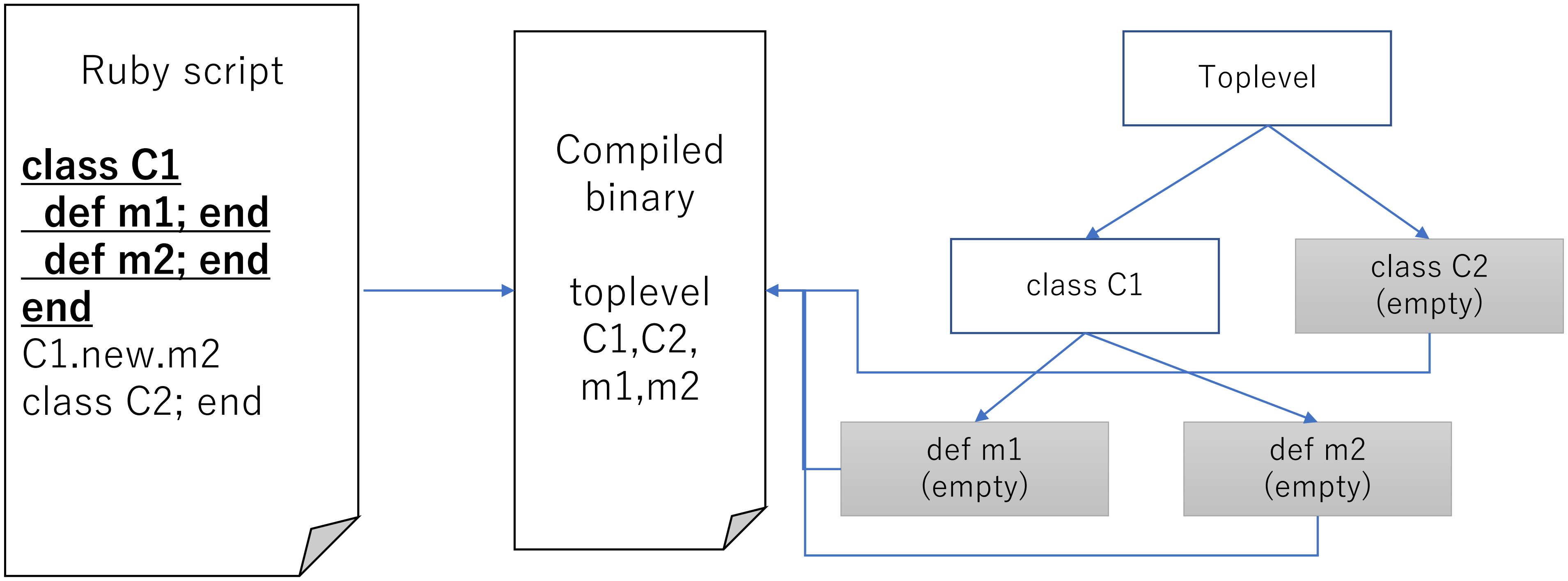
(2) Load toplevel ISeq and make empty C1, C2 empty ISeq and evaluate toplevel ISeq



Technique
Lazy loading

Idea: Load only really used ISeq
Most of methods are not used in a process

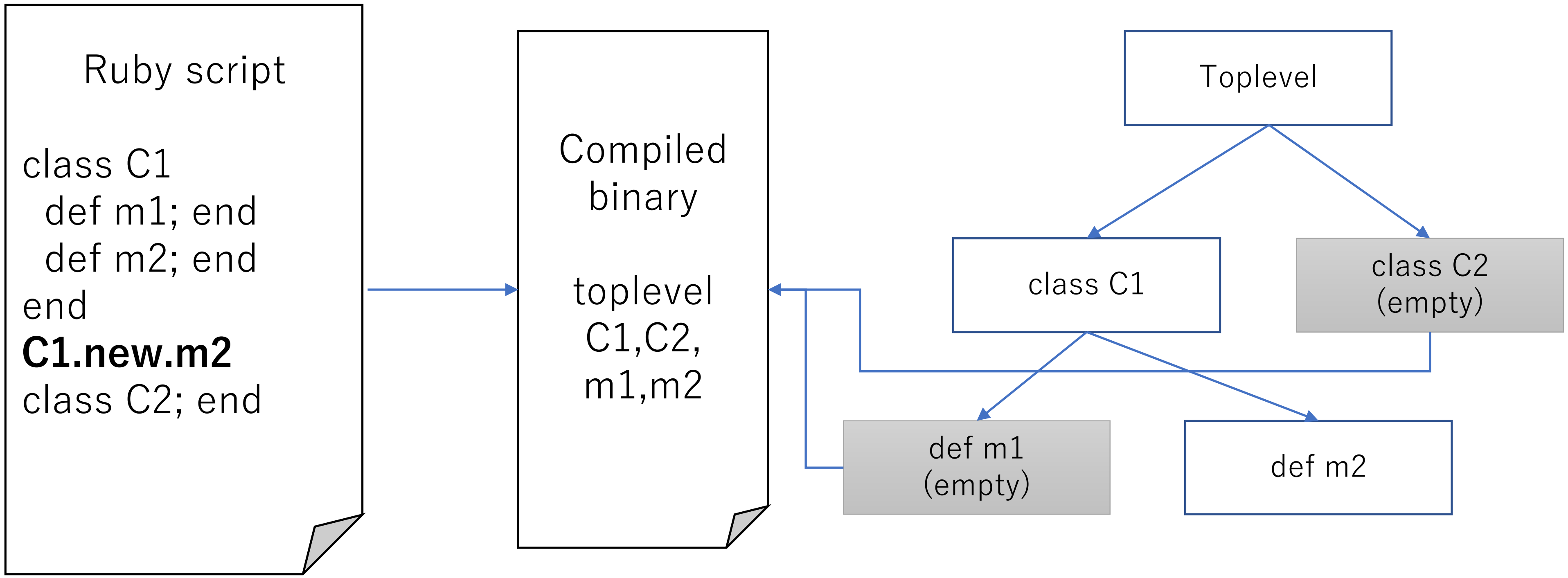
(3) Load C1 and evaluate C1
Define m1 and m2 with
empty ISeqs



Technique
Lazy loading

Idea: Load only really used ISeq
Most of methods are not used in a process

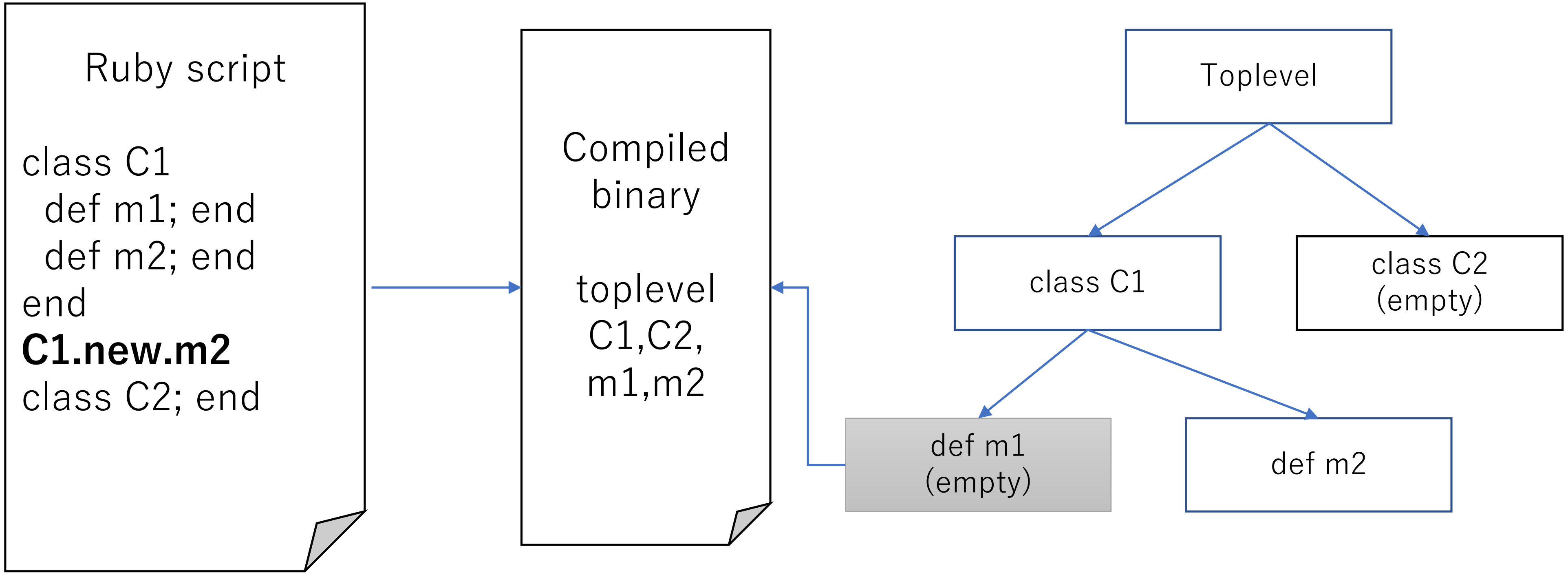
(4) Load m2 and invoke m2



Technique
Lazy loading

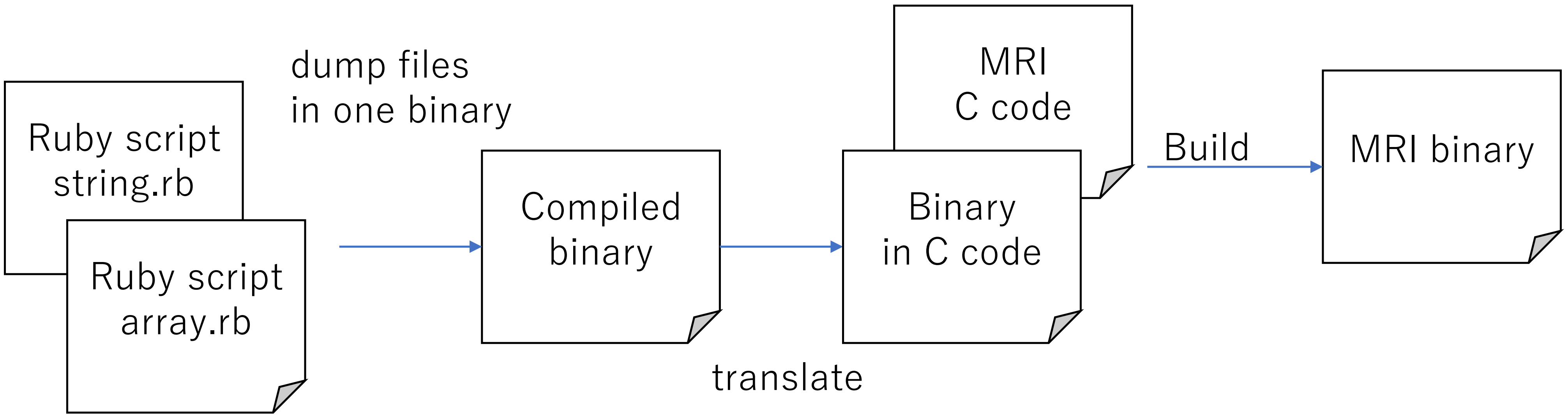
Idea: Load only really used ISeq
Most of methods are not used in a process

(4) Load C2 and evaluate C2



Technique
Multiple file supported binary

- Current compiled binary can contain 1 file
- Expand it to support multiple files
  - We can share resources more.



Evaluation

- Create 3000 classes, they have 1~20 methods
  - One file
    - def.rb: 582KB (3000 classes definitions)
    - compiled binary: 16MB
    - translated C code: 79MB
  - Separate files
    - .rb: 3000 files
    - compiled binary in 1 file: 17MB
    - translated C code: 86MB
  - Corresponding C code: 4.2MB
    - using rb_deifne_method()

```
class C0
  def m0; m; end
  def m1; m; end
  def m2; m; end
  def m3; m; end
  def m4; m; end
  def m5; m; end
  def m6; m; end
  def m7; m; end
  def m8; m; end
  def m9; m; end
  def m10; m; end
  def m11; m; end
  def m12; m; end
end

class C1
  …
```

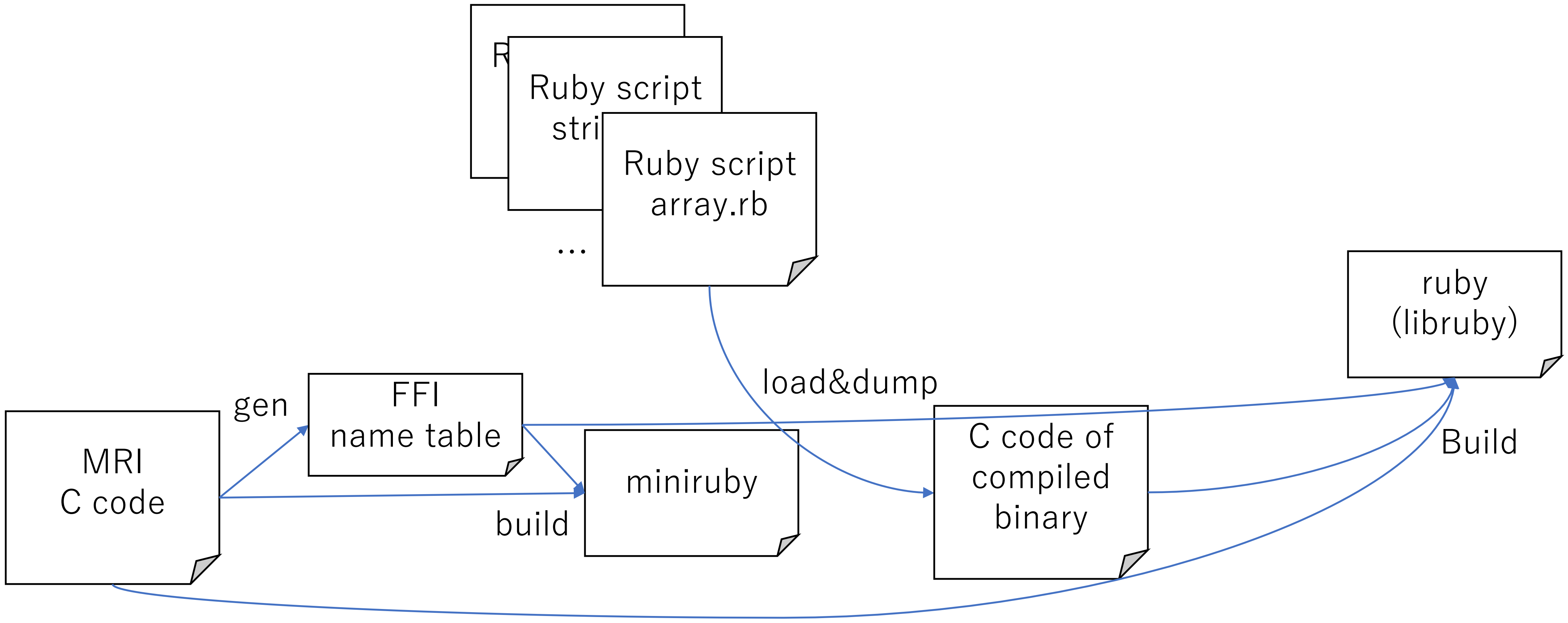
Evaluation
Startup time w/ additonal 3000 classes

	Startup time (ms)	Compare w/ C inits
Miniruby (w/o 3k classes)	6.4	
C inits	**27.5**	
binary/one	87.1	x3.2
binary/sep	82.1	x3.0
binary/one/lazy	47.1	x1.7
binary/sep/lazy	51.1	x1.9
require/one	161.3	x5.9
require/sep	425.2	x15.5

Future work

- Pre-allocated method table
  - We can know which methods are defined at startup, so we can pre-allocate method table in TEXT area (reduce making table overhead)
- Bulk-define instruction
  - Multiple definitions can be done at once with special bulk definition instruction
- Make compact binary format
  - Now it consume huge bytes. Can anyone try?

MRI build Bootstrap



Today's talk

- Current problems with C-methods
- Proposal: Writing builtin methods in Ruby with C
- Performance hacks
  - Runtime-performance: New FFI instruction
  - Startup-time: New compiled binary features

Thank you for your attention

Write a Ruby interpreter in Ruby for Ruby 3

“Ask the speaker” at Cookpad booth (3F) next break

Koichi Sasada
Cookpad Inc.
<ko1@cookpad.com>