ORACLE

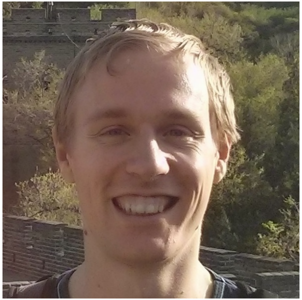
From Interpreting C Extensions to Compiling Them

Benoit Daloze

RubyKaigi 2024 MAY15-17 @NAHA,OKINAWA

TRUFFLE RUBY

Who am I?



Benoit Daloze

Mastodon: @eregon@ruby.social
Twitter: @eregontp
GitHub: @eregon
Website: `https://eregon.me`

- TruffleRuby lead at Oracle Labs, Zurich
- Worked on TruffleRuby since 2014
- PhD on parallelism in dynamic languages
- Maintainer of ruby/spec and ruby/setup-ruby
- CRuby (MRI) committer

TruffleRuby

- A high-performance Ruby implementation
- Uses the GraalVM JIT Compiler
- Targets full compatibility with CRuby 3.2, including C extensions
  e.g. Mastodon and Discourse can run on TruffleRuby
- GitHub: oracle/truffleruby, Twitter: @TruffleRuby, Mastodon: @truffleruby@ruby.social
  Website: `https://graalvm.org/ruby`

Why does Ruby have C Extensions?

C Extensions

Two main purposes:

- bindings to C libraries, e.g. zlib, openssl, mysql2, pg
  Alternative: the `ffi` gem (around 1100 gems depend on `ffi`)
  But `ffi` is impractical if library headers use lots of macros

- performance, e.g. json, msgpack
  Sort of discouraged these days, better to write Ruby and use a JIT.
  Also, TruffleRuby executes Ruby code much faster

Also known as *native extensions*, so not just C but C++, Rust, Go, etc.

C Extensions Usage

Native extensions are used in many gems, often no pure-Ruby replacement:

- 9 out of top 100 gems have a native extension (9%)
  69 out of top 1000 gems have a native extension (7%)
  416 out of top 10000 gems have a native extension (4%)

- Counting gems having a transitive runtime dependency on a native extension, excluding `ffi`:
  28 out of top 100 gems depend on a gem with a native extension (28%)
  295 out of top 1000 gems depend on a gem with a native extension (30%)
  4621 out of top 10000 gems depend on a gem with a native extension (46%)

- The same but also excluding `json` and `racc` (they have a pure-Ruby fallback):
  24 out of top 100 gems depend on a gem with a native extension (24%)
  281 out of top 1000 gems depend on a gem with a native extension (28%)
  4276 out of top 10000 gems depend on a gem with a native extension (43%)

- More complicated for JRuby since some gems have both native & jruby extensions

C Extensions API Compatibility

Group	CRuby 3.2	TruffleRuby dev	JRuby dev
RUBY_VERSION	3.2.4	3.2.2	3.1.4
TOTAL without C-API specs	100% 32534 passing in 1min 4s	97.4% 31722 passing in 3min 29s	96.0% 30731 passing in 13min 3s
C-API	100% of 1492 specs	98.3% of 1493 specs	0.0% of 1492 specs
TOTAL	100% 34026 passing in 1min 28s	97.4% 33190 passing in 3min 47s	91.7% 30731 passing in 13min 3s

From `https://eregon.me/rubyspec-stats/`, 2024-04-26

Various Ways to Implement C Extensions

Truffle

- A framework to build high-performance languages easily
- Get a JIT compiler for free by writing an AST interpreter or bytecode interpreter!
- The GraalVM compiler is able to partial evaluate a Truffle language's AST/bytecode, which produces efficient machine code specific to the language method being compiled.
- No need to duplicate logic between the interpreter and JIT compiler, which is a problem that most JIT compilers have

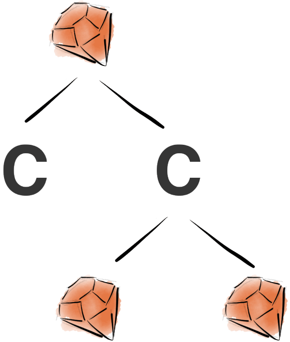
2014: Initial prototype using TruffleC

TruffleC interprets and JIT compiles C code!

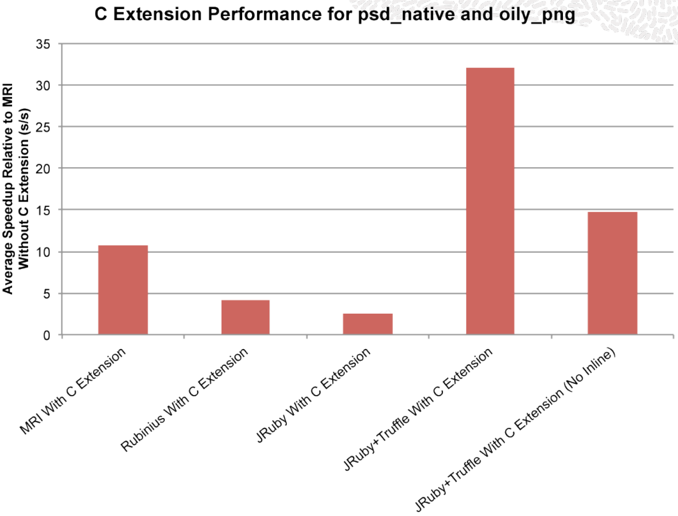
Why not simply using gcc/clang?

- To have freedom of representation and e.g. use a Ruby object to represent a C struct

- For performance, because the GraalVM JIT compiles both C and Ruby it can inline through both languages!



The Impact of Inlining Between C and Ruby



From `https://chrisseaton.com/truffleruby/cext/`

2016: Sulong, the successor of TruffleC

- TruffleC parsed C code, which is quite slow.
- Sulong parses bitcode (LLVM IR), faster because binary format.
- Sulong uses `clang` to compile from C to bitcode.
- So it works not only for C but C++ too.
- No longer a prototype and working for bigger C extensions.

Freedom of Representation

Why not simply using gcc/clang and run the code natively?

To have freedom of representation and e.g. storing Ruby objects in C `long` variables and avoiding any handles or wrappers!

```
// From ruby.h
typedef unsigned long VALUE;

// In some C extension
VALUE my_method(VALUE obj) {
  VALUE foo = obj;
  // foo is actually a Ruby object, which is a Java object!
  // The JVM GC can still move that Java object, no problem.
  ...
}
```

Freedom of Representation

Redirecting accesses to C structs:

```c
// From ruby.h
struct RBasic {
  VALUE flags;
  const VALUE klass;
};

// In some C extension
VALUE my_method(VALUE obj) {
  // Normally ->klass would force RBASIC() to return a native struct.
  // But with TruffleC/Sulong, this is actually the same as:
  // polyglot_read_member(RBASIC(obj), "klass")
  VALUE class_of_object = RBASIC(obj)->klass;
  ...
}
```


Freedom of Representation

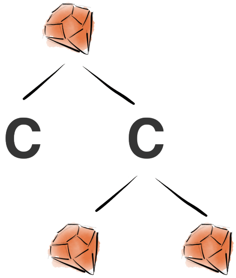
```
VALUE class_of_object = RBASIC(obj)->klass;
```

On TruffleRuby, `RBASIC(obj)` returns a Ruby object, which implements:

```ruby
# Called from the RBASIC C macro
def RBASIC(object)
  Truffle::CExt::RBasic.new(object)
end

class Truffle::CExt::RBasic
  def initialize(object)
    @object = object
  end

  def polyglot_read_member(name)
    raise unless name == 'klass'
    Primitive.metaclass(@object) # either .class or .singleton_class
  end
end
```

Inline Caches in Sulong AST nodes

```
VALUE my_method(VALUE obj) {
  // rb_funcall() in CRuby uses a global cache (no inline cache),
  // so it is still at least one hashtable lookup (slow).
  // But with Sulong we have an inline cache here and only need
  // to check that obj.class is the same as the one seen before.
  // The AST nodes of Sulong and TruffleRuby are mixed and inlined!
  rb_funcall(obj, rb_intern("foo"), 0);
}
```



The Need for Handles

- Having Ruby objects in C local variables or representing C structs is really cool
- But there are cases where a Ruby object is passed to a system library (e.g. libssl)
- That means we need handles, i.e. a way to associate a native pointer to a Ruby object
- We want to avoid handles because they cause extra indirections and extra cleanup
- Initially we change C extensions to create handles explicitly, but it does not scale
- So Sulong gains the `toNative()`/`asPointer()` interop messages to ask a handle automatically when needed. This way we only create handles where needed, and most objects passing through C extensions do not need a handle.

2024: Running C extensions natively

- Compile C/C++ code with the system toolchain (gcc/clang)
- Migrated from running C extensions on Sulong to natively (as machine code)
- `libtruffleruby.so` is still run on Sulong
- Faster startup (no need to parse bitcode), no warmup (no need to JIT compile C extensions)
- Able to run large extensions like `grpc` (never worked on Sulong)
- But handles are needed for every `VALUE` variable in C extensions code
- No more freedom of representation
- No more Ruby + C inlining and no more Sulong AST inline caches in C code

C Extensions API structs after running natively

- **`struct RBasic`** `{ flags, klass }`, `RBASIC(obj)`
  Replacement: `RBASIC_CLASS()`, `RBASIC_FLAGS()`, `RBASIC_SET_FLAGS()`

- **`struct rb_io_t`** `{ fd, mode, ... }`, **`struct RFile`** `{ rb_io_t *fptr }`
  Single native allocation for both, when requested, only some fields supported.
  Replacement: `rb_io_descriptor(VALUE io)`, `rb_io_mode(VALUE io)`, etc
  `rb_io_t` deprecated since `https://bugs.ruby-lang.org/issues/19057`

- **`struct rb_encoding`** `{ name, ... }`
  Native allocation when requested, only some fields supported.

Inline Caches with Macros: `rb_intern()`

From `ruby.h`, simplified:

```
#define rb_intern(str)
  (
    __builtin_constant_p(str) ?
    __extension__ ({
      static ID inline_cache;
      (inline_cache ?
        inline_cache :
        inline_cache = rb_intern(str));
    })
    :
    rb_intern(str)
  )

rb_funcall(obj, rb_intern("foo"), 0);
rb_funcall(obj, rb_intern("bar"), 0);
```

Inline Caches with Macros: `rb_funcall()`

Illustration of an idea:

```
struct rb_funcall_cache {
  VALUE klass;
  method* resolved_method;
};

#define rb_funcall(recv, method_id, argc, ...)
  __extension__ ({
      static struct rb_funcall_cache cache;
      method* m = (rb_class_of(recv) == cache.klass ?
        cache.resolved_method :
        lookup_method(recv, method_id, &cache));
      rb_funcall_cached(recv, m, argc, __VA_ARGS__);
  })

rb_funcall(obj, rb_intern("foo"), 0);
rb_funcall(obj, rb_intern("bar"), 0);
```

TruffleRuby C Extensions History: Interpreting, JIT, Compiling

From Interpreting C Extensions to Compiling Them

2014 Initial prototype using TruffleC.
TruffleC interprets and JIT compiles C code!

2016 Sulong, the successor of TruffleC.
Sulong interprets and JIT compiles LLVM bitcode!

2024 Running C extensions natively.
Compiling C code (AOT) with the system toolchain (gcc/clang).

C Extensions API: The Good & Bad Parts

C Extensions API: The Good Parts

- `VALUE`: pretty much an opaque pointer, makes it possible to use handles even though it was not designed as such!
- functions: great, can just reimplement them differently, works well.
- macros: can change them but changing any of them means not able to reuse extensions precompiled for CRuby. Also need to maintain a diff on top of CRuby headers.

C Extensions API: The Bad Parts

- Still a few `struct`, though many `struct` stopped being exposed
- `VALUE* RARRAY_PTR(ary)`, forces a flat native Array represenation
- `char* RSTRING_PTR(str)`, causes extra copying from `byte[]` to `char*`
- GC semantics, marker functions for RTypedData/RData: hard & expensive to emulate

Actually much better than CPython API which exposes way too many structs (so a new C API is needed for Python: HPy).

Benchmarks

Benchmark Configurations

All on `[x86_64-linux]`, measuring peak performance, i.e., after enough warmup:

- CRuby: `ruby 3.3.1 (2024-04-23 revision c56cd86388) +YJIT`
- Sulong: `truffleruby 23.1.2, like ruby 3.2.2, Oracle GraalVM JVM`
- LibFFI: TruffleNFI with the LibFFI backend, going through JNI and libffi: `truffleruby 24.1.0-dev-1727ac8b, like ruby 3.2.2, Oracle GraalVM JVM`
- Panama: TruffleNFI with the Panama backend, going only through Panama: `truffleruby 24.1.0-dev-1727ac8b, like ruby 3.2.2, Oracle GraalVM JVM`
- Pure: Pure-Ruby variant
- CExt: C Extension variant

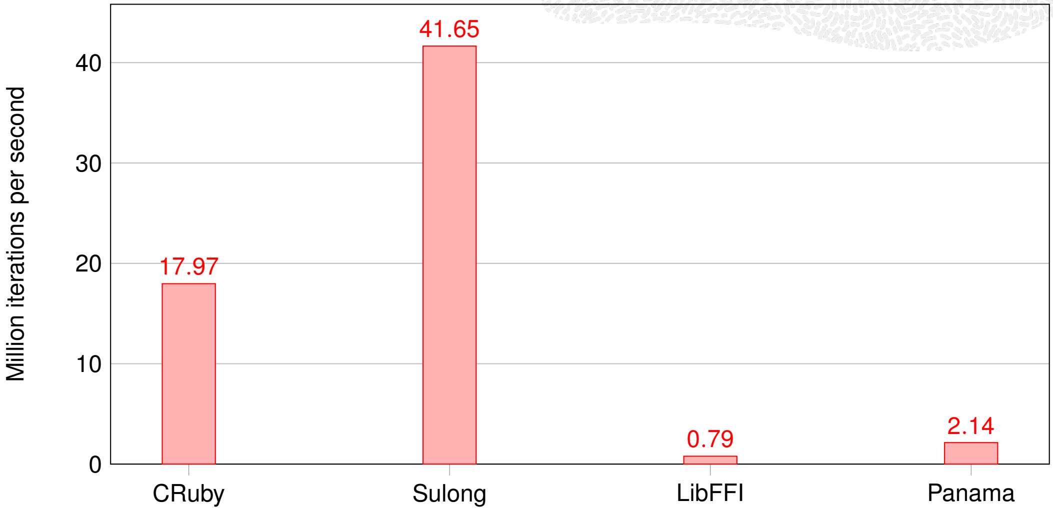
Upcall Benchmark

```c
#include "ruby.h"

static VALUE foo_itself(VALUE self) {
  return rb_funcall(self, rb_intern("itself"), 0);
}

void Init_cext(void) {
  VALUE cFoo = rb_define_class("Foo", rb_cObject);
  rb_define_singleton_method(cFoo, "foo", foo_itself, 0);
}
```

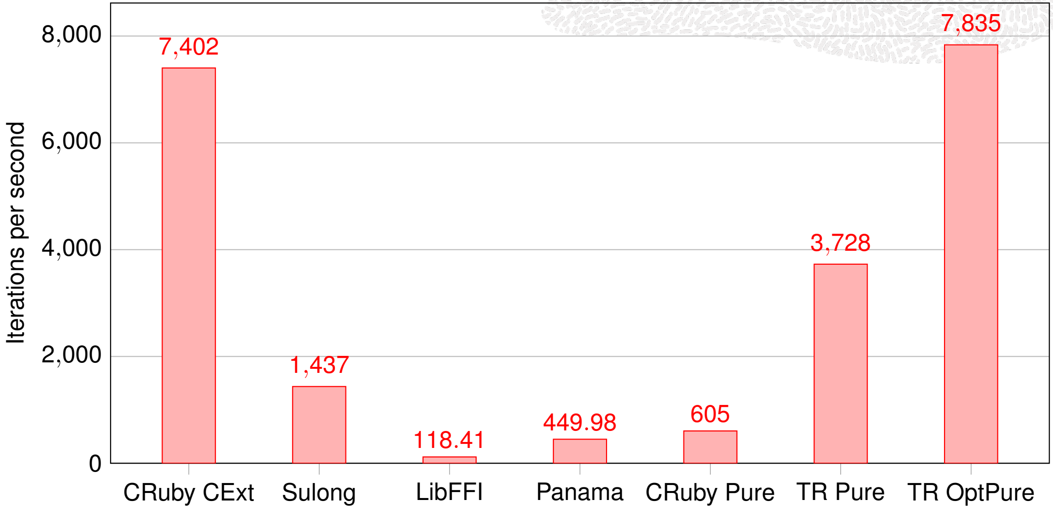
Ruby code:

```ruby
benchmark { Foo.foo }
```

Upcall Benchmark Results

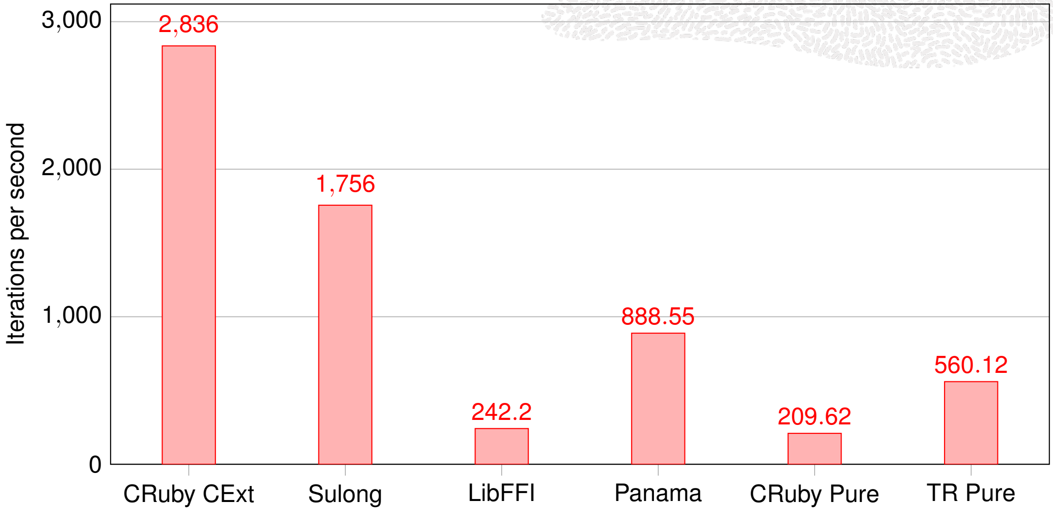


JSON.dump Benchmark Results



Benchmark from `https://github.com/flori/json/pull/580`

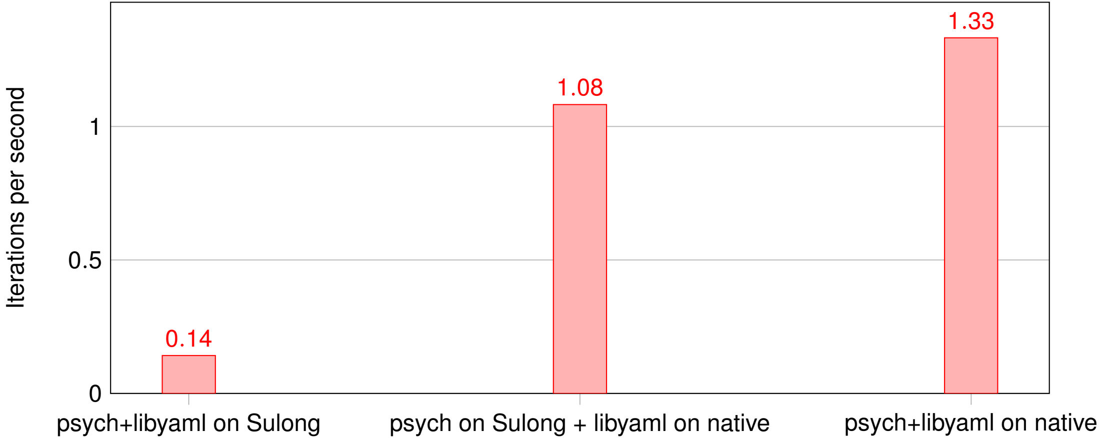
JSON.load Benchmark Results



Benchmark from `https://github.com/flori/json/pull/580`

YAML.load Benchmark Results (single iteration, no warmup)

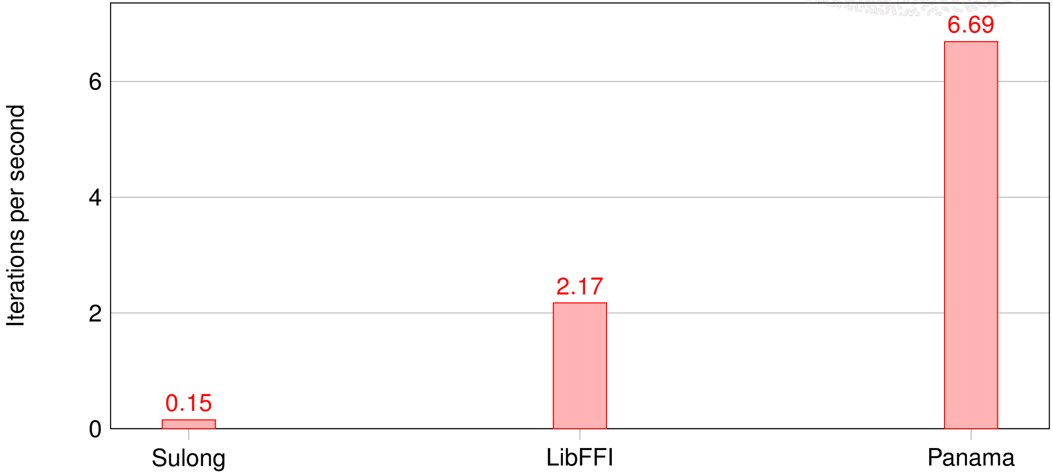
```
puts Benchmark.realtime {
  YAML.load(File.read("one_megabyte_file.yml"))
}
```



About 10x faster. These benchmarks are run on TruffleRuby Native instead of TruffleRuby JVM.

Ripper Benchmark Results

```
benchmark { Ripper.sexp(code) }
```



18x faster for LibFFI, 44x faster for Panama! Because `ripper_yyparse()` is too big (6686 lines)

Summary of Benchmarks

- Some extensions have better peak performance on Sulong, but they need a long warmup time
- Some extensions are faster on LibFFI/Panama and need no warmup
- Panama is 3-4 times faster than LibFFI, upcall (C->Ruby) performance matters a lot
- Ongoing research to automatically execute some libraries/functions natively when using Sulong

Conclusion

- About 43% of the top 10000 gems depend on a gem with a native extension
- The Ruby C extension API can be implemented by alternative Ruby implementations
- TruffleRuby first implemented the C API by JIT compiling C extensions, which enables inlining between Ruby and C and freedom of representation
- TruffleRuby 24.0 moved to running C extensions natively, which supports large extensions and no longer need startup and warmup for C extensions
- In general the API functions/macros returning a `struct` or pointer are problematic, because they force native allocations and extra copying.
  There are alternative functions/macros which do not have this issue.

Cool Things About TruffleRuby and GraalVM

- Interoperability with Java, Python, JS and other GraalVM languages:
  `Polyglot.eval('python', 'import matplotlib')`

- Regexp JIT Compiler and how to avoid ReDoS (RubyKaigi 2021 presentation)

- Parallel execution of Ruby code and soon of `RB_EXT_RACTOR_SAFE`-marked C extensions

- Most advanced Ruby JIT Compiler: Inlining Ruby/C/Java/etc, Splitting, Partial Evaluation, GraalVM Compiler optimizations like Partial Escape Analysis, etc

- Multiple GCs to choose from with various throughput and latency trade-offs (ParallelGC, G1, ZGC)

Trying TruffleRuby

Latest release: 24.0.1 (16 April 2024)
New: EA builds at `https://github.com/graalvm/oracle-graalvm-ea-builds`

Use your favorite Ruby manager/installer:

```
$ ruby-install truffleruby
$ ruby-install truffleruby-graalvm

$ ruby-build truffleruby-24.0.1
$ ruby-build truffleruby+graalvm-24.0.1
(or rbenv install instead of ruby-build)

$ rvm install truffleruby
```

See `https://github.com/oracle/truffleruby` for more details

Any question?