Tiny JITs for a Faster FFI

2025-02-12 • Aaron Patterson

Can we have a faster FFI for CRuby? Yes.

Can we have a faster FFI for CRuby?

I love programming in Ruby, and I advocate for people to write as much Ruby as possible. But sometimes you really really must call out to native code. Even in those cases, I encourage people to *write as much Ruby as possible*, especially because YJIT can optimize Ruby code but not C code.

Taken to its logical extreme, this guidance means that if you want to call a native library, you should write a native extension with a very very limited API where most work is done in Ruby. Any native code would be a very thin wrapper around the function *we actually want to call* that just converts Ruby types in to the types required by the native function.

Of course such a simplistic API would be well suited to work with a library like FFI.

Now, usually I steer clear of FFI, and to be honest the reason is simply that it doesn't provide the same performance as a native extension.

Lets take a look at a very simple example benchmark to better understand what I mean. In this benchmark, we're going to wrap the `strlen` C function with FFI. We'll compare the FFI implementation with a C extension that does the same thing (using the `strlen` Ruby Gem that yours truly wrote just for this post). We'll also include a comparison with indirectly calling the `String#bytesize`, as well as directly calling `String#bytesize`.

```
require "ffi"
require "strlen"
require "benchmark/ips"

module A
  extend FFI::Library
  ffi_lib 'c'
  attach_function :strlen, [:string], :int
end

module B
  def self.strlen(x)
    x.bytesize
  end
```

```ruby
end

str = "foo"

Benchmark.ips do |x|
  x.report("strlen-ffi")  { A.strlen(str) }
  x.report("strlen-ruby") { B.strlen(str) }
  x.report("strlen-cext") { Strlen.strlen(str) }
  x.report("ruby-direct") { str.bytesize }
  x.compare!
end
```

Here is the output from the benchmark:

```
ruby 3.5.0dev (2025-02-11T16:42:26Z master 4ac75f6f64) +PRISM [arm64-darwin24]
Warming up --------------------------------------
          strlen-ffi     1.557M i/100ms
         strlen-ruby     2.875M i/100ms
         strlen-cext     3.047M i/100ms
         ruby-direct     4.048M i/100ms
Calculating --------------------------------------
          strlen-ffi     15.682M (± 0.5%) i/s   (63.77 ns/i) -     79.398M in   5
         strlen-ruby     28.697M (± 0.3%) i/s   (34.85 ns/i) -    143.747M in   5
         strlen-cext     30.661M (± 0.8%) i/s   (32.61 ns/i) -    155.406M in   5
         ruby-direct     39.879M (± 0.6%) i/s   (25.08 ns/i) -    202.412M in   5

Comparison:
         ruby-direct: 39878845.7 i/s
         strlen-cext: 30661398.4 i/s - 1.30x  slower
         strlen-ruby: 28697184.3 i/s - 1.39x  slower
          strlen-ffi: 15681971.0 i/s - 2.54x  slower
```

First, directly calling `String#bytesize` is the fastest, and we can think of it as our baseline. Any indirection we add will necessarily add more overhead, and we probably can't "beat" this number. Calling `strlen` via C extension is second fastest, followed by indirectly calling `String#bytesize`, and finally the FFI implementation is slowest.

These benchmark results can teach us a couple interesting things.

First, the difference between the "ruby-direct" benchmark and the "strlen-ruby" benchmark shows that there definitely is overhead in pushing and popping stack frames. Eliminating this overhead is one of the things that JIT compilers like YJIT specialize in.

Second, the difference between the "strlen-cext" benchmark and the "strlen-ffi" benchmark shows that there is significant overhead incurred when calling a native function via FFI. Calling

the C extension is slower than directly calling `String#bytesize`, but calling `strlen` via FFI adds *even more* overhead than the C extension does.

In other words, if Ruby provides a method to do something you need, then just use the method that Ruby provides. But if you need to call a foreign function, a small C extension wrapper will generally have less overhead than an FFI wrapper.

I've not avoided FFI because I think it's *intrinsically worse* than a C extension. Rather, paying the FFI tax is just a reality I've tried to avoid.

Can we change reality?

A few years ago Chris Seaton gave me an idea that's been rattling around in my head ever since. Rather than calling out to a 3rd party library, could we just JIT the code required to call the external function?

Lets take a look at the FFI wrapper example:

```
module A
  extend FFI::Library
  ffi_lib 'c'
  attach_function :strlen, [:string], :int
end
```

The call to `attach_function` tells us the name of the function we need to call (`strlen`) as well as the parameter types (a string) and the return type (an int). Since we know these types at the time we're defining the wrapper function, we could generate the machine code required to wrap and unwrap Ruby types, as well as call in to the foreign function.

For years I've been scheming for a way to do this, and I think the stars will finally align with the release of Ruby 3.5 later this year.

In order to make this dream happen, we need a few things to come together.

First, we need a way to generate machine code. This is why I wrote the AArch64 gem as well as the Fisk gem which can generate ARM64 and x86_64 machine code respectively.

Second, we need a way to allocate executable memory so that we can actually *execute* the machine code. Assembling machine code isn't good enough, we have to place that machine code in memory that's marked as "executable". That is why I wrote the creatively named JITBuffer gem.

With these utilities, we have a way to generate executable machine code. Unfortunately, we have one more hurdle to overcome, and that is trying to *get Ruby to jump in to the machine code*.

It's not good enough to just generate executable machine code. Any rag-tag team of misfits can

do that. We also need to get Ruby to jump in to that machine code so that we can *skip the FFI overhead*.

Leveraging RJIT

For those that don't know, RJIT is a JIT compiler for Ruby that is itself written in Ruby, and also it ships with Ruby. Its internal structure is quite similar to YJIT, but it wasn't intended for production use, which is why most people have probably heard of YJIT but not RJIT.

Kokubun, the author of RJIT, recently filed a feature request to extract RJIT as a gem. The major feature provided by this extraction is that people will be able to more easily write JIT compilers for Ruby as 3rd party gems. The proposed feature does 2 important things.

First, it extracts RJIT as a gem. RJIT uses a mechanism similar to bindgen from Rust, where it **generates Ruby data structures** that map out all of Ruby's internal types (you can see some of that generated code here). This means that 3rd party JIT compilers can get the information they need to *wrap and unwrap Ruby data types*.

The second important thing it does is *always execute the JIT entry function pointer if there is one*. This is important because it means that 3rd party JITs will have a way to register their machine code and Ruby will automatically jump to that machine code.

With these two pieces in place, we can write a very small-scale, single-purpose JIT compiler that acts as an FFI interface.

Proof of Concept

I created a very small proof of concept called "FJIT". "FJIT" is short for "FFI JIT" and does what it says on the tin. Namely, it generates machine code at runtime that can call a foreign function. In this case we're going to use it to call the `strlen` function.

I'm not going to put the entire program in this post because even though it's "small", it still contains a whole JIT compiler. The important part is the benchmark:

```
module A
  extend FFI::Library
  ffi_lib 'c'
  attach_function :strlen, [:string], :int
end

module B
  def self.strlen(x)
    x.bytesize
  end
```

```ruby
end

module C
  extend FJIT
  attach_function :strlen, [:string], :int
end

str = "foo"

Benchmark.ips do |x|
  x.report("strlen-ffi")  { A.strlen(str) }
  x.report("strlen-ruby") { B.strlen(str) }
  x.report("strlen-cext") { Strlen.strlen(str) }
  x.report("ruby-direct") { str.bytesize }
  x.report("strlen-fjit") { C.strlen(str) }
  x.compare!
end
```

Module `C` in this updated benchmark uses an `FJIT` module, and you can see that its interface is very similar to that of FFI. When `attach_function` is called, FJIT will generate the machine code required to unwrap the Ruby string, call the `strlen` function, and return the length of the string as a Ruby object.

Here are the benchmark results:

```
ruby 3.5.0dev (2025-02-11T16:42:26Z master 4ac75f6f64) +RJIT +PRISM [arm64-darwin
Warming up --------------------------------------
          strlen-ffi     1.558M i/100ms
         strlen-ruby     2.953M i/100ms
         strlen-cext     2.981M i/100ms
         ruby-direct     4.142M i/100ms
         strlen-fjit     3.206M i/100ms
Calculating --------------------------------------
          strlen-ffi     15.629M (± 0.7%) i/s   (63.98 ns/i) -     79.455M in   5
         strlen-ruby     28.851M (± 0.3%) i/s   (34.66 ns/i) -    144.704M in   5
         strlen-cext     29.778M (± 2.8%) i/s   (33.58 ns/i) -    149.025M in   5
         ruby-direct     41.907M (± 0.8%) i/s   (23.86 ns/i) -    211.219M in   5
         strlen-fjit     32.508M (± 0.9%) i/s   (30.76 ns/i) -    163.504M in   5

Comparison:
         ruby-direct: 41907248.7 i/s
         strlen-fjit: 32507961.2 i/s - 1.29x  slower
         strlen-cext: 29778234.0 i/s - 1.41x  slower
         strlen-ruby: 28850712.3 i/s - 1.45x  slower
          strlen-ffi: 15629443.7 i/s - 2.68x  slower
```

Of course directly calling `String#bytesize` is still the fastest. However, the machine code generated by FJIT is second fastest. Surprisingly, it's slightly faster than the `strlen` C extension. But even more promising is that it's faster than the indirect Ruby call, and more than 2x faster than calling via FFI!

Conclusion

I think this is very exciting because it means that we can achieve the same speeds (or even better) than a C extension while maintaining the "write as much Ruby as possible" philosophy. I've been very jealous of programming languages like Zig, that are able to support calling native code without the use of FFI. If we can get all of these moving parts to settle, then I think Ruby can have the same advantages.

Caveats

I know the conclusion is supposed to come last, since it is a "conclusion". But I didn't want people to wade through the current caveats before getting to the good stuff.

First, the JIT compiler I wrote in the proof of concept is limited to ARM64 platforms. If we want to make this "for real", we need to add an x86_64 backend. Of course this is possible, it just needs to be done. Second, it currently doesn't handle all parameter types and return types. I am confident we can support all parameter types and the work would not be onerous. Third, it also only handles functions that take a single parameter and return a single parameter. Again, I think it's just a matter of fleshing out the rest of the compiler. Fourth, you have to run Ruby with `--rjit --rjit-disable` flags at the moment. Once Kokubun's feature lands, that shouldn't be the case anymore. Last, this proof of concept only runs with current Ruby head at the moment.

Whew, I know its a fairly long list of restrictions, but it is shorter than the average EULA, and also nothing we can't overcome.

Anyway, that's the end. Have a good day!

Update: Ruby head moves quickly, and RJIT has been removed! If you want to play with this script, you'll need to check out Ruby at f32d5071b7b01f258eb45cf533496d82d5c0f6a1.