This is for discussing an IR design for Ruby's JIT. This thread is spun out from [Feature #18229] to let it focus on the merge.

Original discussions

Updated by vmakarov (Vladimir Makarov): https://bugs.ruby-lang.org/issues/18229#note-8

I like very clever idea of lazy BB versioning on which YJIT is built. I've tried to solve problem of generation of type specialized code in original MJIT by dynamic changing VM insns to specialized variants of them and subsequent code generation. But it needs several (slow) code generation until the code is stabilized. It also doesn't remove redundant type checks because GCC and LLVM are not clever enough to remove checks when bitmasks operations are used for type tagging in CRuby (although recent development of ranger project https://gcc.gnu.org/wiki/AndrewMacLeod/Ranger might solve this problem). Lazy BBV has no such disadvantages.

In fact I'd like to try BBV in MIR project (https://github.com/vmakarov/mir) in machine independent way through new extensions on MIR and C level besides implementation of profiling and extensions pointing where to use it.

I also like YJIT approach for fast method calls and switching to the interpreter. I think the same approach might be implemented in MJIT (GCC naked functions might help).

I believe serious thinking should be done how to add YJIT to CRuby. I've been working on GCC for a long time and adding command line options to GCC and making them deprecated is a serious problem. Ideally by default CRuby should generate the best code without any options. I think YJIT should work by default and when a Ruby method run too many times MJIT should be used as in a standard approach for JVM.

I don't see currently a working alternative to YJIT as tier I JIT compiler for CRuby. This might stay as it for a long time. Saying that I also don't see a potential for big Ruby code performance improvement by YJIT without considerable redesign. YJIT does not optimize machine code generated for several VM insns. To solve the problem, adding IR and making classical optimizations on it is needed. Without IR YJIT can not move to another level of optimizations (interprocedural level, e.g. by using call inlining). But that is ok, YJIT does excellent work as tier I JIT compiler and can stay that way.

Updated by maximecb (Maxime Chevalier-Boisvert): https://bugs.ruby-lang.org/issues/18229#note-9

Without IR YJIT can not move to another level of optimizations (interprocedural level, e.g. by using call inlining). But that is ok, YJIT does excellent work as tier I JIT compiler and can stay that way.

we are starting to look at adding an IR to YJIT. The YARV bytecodes are big and have complex semantics, which makes it hard to build optimizations on top. As such we would like to translate YARV into a custom IR that is easier for us to optimize and do things like inlining. I have looked at MIR and it looks close to machine code, the kind of IR you would compile C code into. We are thinking of designing an IR that is maybe closer to Ruby semantics, so that Ruby-specific optimizations can be applied more easily. Open to discussion if you have input on the subject. We would appreciate your input.

Updated by vmakarov (Vladimir Makarov): https://bugs.ruby-lang.org/issues/18229#note-13

I have looked at MIR and it looks close to machine code, the kind of IR you would compile C code into. We are thinking of designing an IR that is maybe closer to Ruby semantics, so that Ruby-specific optimizations can be applied more easily. Open to discussion if you have input on the subject. We would appreciate your input.

MIR is designed to be used for different languages, including C. Standard ruby methods implemented on C, e.g. times, can be translated into MIR and user-defined Ruby block called by times can be translated into MIR too (may be through intermediate C translation), then MIR for times and the block can be intermixed (inlined) and optimized. So MIR permits optimization of code written on different languages. In this way MIR can be used not only for Ruby but for other dynamic language implementations (e.g. CPython).
MIR also makes easy implementation of classical compiler optimizations because it is an extension of tuple based IR.

You probably wrote about inlining methods (blocks) implemented on Ruby into another Ruby method. It is more constrained approach. Although if most standard Ruby methods like times will be rewritten on Ruby, it is less constraint approach but I am not sure that the overall machine code generated quality will be not worse. Probably it is also double approach if type information (type annotation can be used for this) and info about absence of integer overflow from standard methods rewritten from C to Ruby can be propagated and used.

Still for further improvement of YJIT you need some IR to optimize machine code generated from several VM insns. Right now YJIT is just a simple template code generator for given value types.

In any case, I am just at the very beginning to use MIR project for CRuby JIT and YJIT is a real thing. And it is the only thing that matters.

History

#1 - 09/30/2021 06:41 PM - k0kubun (Takashi Kokubun)
- Description updated

#2 - 09/30/2021 11:38 PM - k0kubun (Takashi Kokubun)

MIR is designed to be used for different languages, including C. Standard ruby methods implemented on C, e.g. times, can be translated into MIR and user-defined Ruby block called by times can be translated into MIR too (may be through intermediate C translation), then MIR for times and the block can be intermixed (inlined) and optimized. So MIR permits optimization of code written on different languages.

When #times calls a Ruby block, how does MIR associate a vm::yield call from the C function of #times with the MIR of the Ruby block? When it just calls a C function that does direct-threading with the iseq of the Ruby block, it's not necessarily straightforward to inline the MIR of the block iseq, which is just a pointer used by a C function. I guess you need to rely on a MIR-specific C language extension guarded by #ifdef? At least the approach doesn't seem to work for pure C code or MJIT.

However, if it's possible to inline Ruby methods called from a C method (called from a Ruby method) without rewriting the C method with Ruby, that would be fantastic. The performance impact of rewriting #times with Ruby seems trivial https://github.com/ruby/ruby/pull/3656, but it seems a bit harder to keep the interpreter performance when you rewrite #each and #map with Ruby https://github.com/ruby/ruby/pull/3658 https://github.com/ruby/ruby/pull/3666, which allows you to look at invokeblock and inline the iseq. If we have a tier-1 JIT that is enabled by default on all platforms and compiles all methods early, we could eventually ignore the interpreter performance and rewrite them with Ruby as long as JIT-ed performance (without inlining a block) is comparable to the original C code though.

#3 - 11/19/2021 05:35 PM - k0kubun (Takashi Kokubun)
- Status changed from Open to Closed