Ruby master - Feature #12589

VM performance improvement proposal

07/18/2016 03:24 AM - vmakarov (Vladimir Makarov)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Hello. I'd like to start a big MRI project but I don't want to disrupt somebody else plans. Therefore I'd like to have MRI developer's opinion on the proposed project or information if somebody is already working on an analogous project.

Basically I want to improve overall MRI VM performance:

- First of all, I'd like to change VM insns and move from **stack-based** insns to **register transfer** ones. The idea behind it is to decrease VM dispatch overhead as approximately 2 times less RTL insns are necessary than stack based insns for the same program (for Ruby it is probably even less as a typical Ruby program contains a lot of method calls and the arguments are passed through the stack).

  But *decreasing memory traffic* is even more important advantage of RTL insns as an RTL insn can address temporaries (stack) and local variables in any combination. So there is no necessity to put an insn result on the stack and then move it to a local variable or put variable value on the stack and then use it as an insn operand. Insns doing more also provide a bigger scope for C compiler optimizations.

  The biggest changes will be in files compile.c and insns.def (they will be basically rewritten). **So the project is not a new VM machine. MRI VM is much more than these 2 files.**

  The disadvantage of RTL insns is a bigger insn memory footprint (which can be upto 30% more) although as I wrote there are fewer number of RTL insns.

  Another disadvantage of RTL insns *specifically* for Ruby is that insns for call sequences will be basically the same stack based ones but only bigger as they address the stack explicitly.

- Secondly, I'd like to **combine some frequent insn sequences** into bigger insns. Again it decreases insn dispatch overhead and memory traffic even more. Also it permits to remove some type checking.

  The first thing on my mind is a sequence of a compare insn and a branch and using immediate operands besides temporary (stack) and local variables. Also it is not a trivial task for Ruby as the compare can be implemented as a method.

I already did some experiments. RTL insns & combining insns permits to speed the following micro-benchmark in more 2 times:

```
i = 0
while i<30_000_000 # benchmark loop 1
  i += 1
end
```

The generated RTL insns for the benchmark are

```
== disasm: #<ISeq:<main>@while.rb>=====================================
== catch table
| catch type: break  st: 0007 ed: 0020 sp: 0000 cont: 0020
| catch type: next   st: 0007 ed: 0020 sp: 0000 cont: 0005
| catch type: redo   st: 0007 ed: 0020 sp: 0000 cont: 0007
|------------------------------------------------------------------------
local table (size: 2, temp: 1, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest:
-1])
[ 2] i
0000 set_local_val    2, 0                                          (   1)
0003 jump             13                                            (   2)
0005 jump             13
0007 plusi            <callcache>, 2, 2, 1, -1                      (   3)
0013 btlti            7, <callcache>, -1, 2, 30000000, -1           (   2)
0020 local_ret        2, 0                                          (   3)
```

In this experiment I ignored trace insns (that is another story) and a complication that a integer compare insn can be re-implemented as a Ruby method. Insn bflti is combination of LT immediate compare and branch true.

A modification of fib benchmark is sped up in 1.35 times:

```
def fib_m n
  if n < 1
    1
  else
    fib_m(n-1) * fib_m(n-2)
  end
end

fib_m(40)
```

The RTL code of fib_m looks like

```
== disasm: #<ISeq:fib_m@fm.rb>==========================================
local table (size: 2, temp: 3, argc: 1 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest:
-1])
[ 2] n<Arg>
0000 bflti            10, <callcache>, -1, 2, 1, -1                 (   2)
0007 val_ret          1, 16
0010 minusi           <callcache>, -2, 2, 1, -2                     (   5)
0016 simple_call_self <callinfo!mid:fib_m, argc:1, FCALL|ARGS_SIMPLE>, <callcache>, -1
0020 minusi           <callcache>, -3, 2, 2, -3
0026 simple_call_self <callinfo!mid:fib_m, argc:1, FCALL|ARGS_SIMPLE>, <callcache>, -2
0030 mult             <callcache>, -1, -1, -2, -1
0036 temp_ret         -1, 16
```

In reality, the improvement of most programs probably will be about 10%. That is because of very dynamic nature of Ruby (a lot of calls, checks for redefinition of basic type operations, checking overflows to switch to GMP numbers). For example, integer addition can not be less than about x86-64 17 insns out of the current 50 insns on the fast path. So even if you make the rest (33) insns 2 times faster, the improvement will be only 30%.

A very important part of MRI performance improvement is to make calls fast because there are a lot of them in Ruby but as I read in some Koichi Sasada's presentations he pays a lot of attention to it. So I don't want to touch it.

- Thirdly. I want to implement the insns as small inline functions for future AOT compiler, of course, if the projects described above are successful. It will permit easy AOT generation of C code which will be basically calls of the functions.

I'd like to implement AOT compiler which will generate a Ruby method code, call a C compiler to generate a binary shared code and load it into MRI for subsequent calls. The key is to minimize the compilation time. There are many approaches to do it but I don't want to discuss it right now.

C generation is easy and most portable implementation of AOT but in future it is possible to use GCC JIT plugin or LLVM IR to decrease overhead of C scanner/parser.

C compiler will see a bigger scope (all method insns) to do optimizations. I think using AOT can give another 10% improvement. It is not that big again because of dynamic nature of Ruby and any C compiler is not smart enough to figure out aliasing for typical generated C program.

The life with the performance point of view would be easy if Ruby did not permit to redefine basic operations for basic types, e.g. plus for integer. In this case we could evaluate types of operands and results using some data flow analysis and generate faster specialized insns. Still a gradual typing if it is introduced in future versions of Ruby would help to generate such faster insns.

Again I wrote this proposal for discussion as I don't want to be in a position to compete with somebody else ongoing big project. It might be counterproductive for MRI development. Especially I don't want it because the project is big and long and probably will have a lot of tehcnical obstacles and have a possibilty to be a failure.

Related issues:

Related to Ruby master - Feature #14235: Merge MJIT infrastructure with conse...	**Closed**

History

#1 - 07/18/2016 03:25 AM - vmakarov (Vladimir Makarov)

- Tracker changed from Bug to Feature

#2 - 07/18/2016 04:04 AM - matz (Yukihiro Matsumoto)

As for superoperators, shyouhei is working on it.

In any way, I'd suggest you take a YARV step for a big change like your proposal.
When the early stage of the development of YARV, Koichi created his virtual machine as a C extension.
After he brushed it up to almost complete, we replaced the VM.

I think Koichi would help you.

Matz.

#3 - 07/18/2016 02:24 PM - vmakarov (Vladimir Makarov)

Yukihiro Matsumoto wrote:

> As for superoperators, shyouhei is working on it.
>
> In any way, I'd suggest you take a YARV step for a big change like your proposal.
> When the early stage of the development of YARV, Koichi created his virtual machine as a C extension.
> After he brushed it up to almost complete, we replaced the VM.

Thank you for the advice. I investigate how to implement it as a C extension. Right now I just have a modified and hackish version of compile.c/insns.def of some year old Ruby version to get RTL code for the two cases I published. After getting some acceptable results I think I need to start to work more systematically and I would like to get some working prototype w/o AOT by the year end.

> I think Koichi would help you.

That would be great.

#4 - 07/19/2016 01:17 AM - shyouhei (Shyouhei Urabe)

FYI in current instruction set, there do exist bias between which instruction tends to follow which. A preexperimental result linked below shows there is clear tendency that a pop tends to follow a send. Not sure how to "fix" this though.

https://gist.github.com/anonymous/7ce9cb03b5bc6cfe6f96ec6c4940602e

#5 - 07/19/2016 02:26 AM - vmakarov (Vladimir Makarov)

Shyouhei Urabe wrote:

> FYI in current instruction set, there do exist bias between which instruction tends to follow which. A preexperimental result linked below shows there is clear tendency that a pop tends to follow a send. Not sure how to "fix" this though.
>
> https://gist.github.com/anonymous/7ce9cb03b5bc6cfe6f96ec6c4940602e

Thank you for the link. The results you got are interesting to me. You could add a new insn 'send_and_pop' but I suspect it will give only a tiny performance improvement. Pop is a low cost insn and especially when it goes with the send insn. The only performance improvement will be pop insn dispatch savings (it is only 2 x86-64 insns). Still it will give a visible insn memory saving.

RTL insns are better fit for optimizations (it is most frequent IR for optimizing compilers) including combining insns (code selection) but finding frequent combinations is more complicated because insns being combined should be dependent, e.g. result of the first insn is used as an operand of the 2nd insn (combining independent insns will again save VM insn dispatching and may be will result in improving a fine-grain parallelism by compiler insn scheduler or by logic of out-of-order execution CPU). It could be an interesting research what RTL insns should be combined for a particular code. I don't remember any article about this.

#6 - 07/19/2016 07:12 AM - ko1 (Koichi Sasada)

Hi!

Do you have interst to visit Japan and discuss Japanese ruby committers?
If you have interst, I will ask someone to pay your travel fare.

Thanks,
Koichi

#7 - 07/19/2016 02:18 PM - vmakarov (Vladimir Makarov)

Shyouhei Urabe wrote:

> FYI in current instruction set, there do exist bias between which instruction tends to follow which. A preexperimental result linked below shows there is clear tendency that a pop tends to follow a send. Not sure how to "fix" this though.
>
> https://gist.github.com/anonymous/7ce9cb03b5bc6cfe6f96ec6c4940602e

Sorry, I realized that I quickly jump to generalization about insn combining and did not write you all details how to implement send-and-pop. It needs a flag in a call frame and the return insn should use it. But imho, send-and-pop implementation has no sense as benefit of removing dispatch machine insns is eaten by insns dealing with the flag (it also slows down the regular send insn). Also increasing size of the call frame means decreasing maximal recursion depth although with some tricks you can add the flag (it is just one bit) w/o increasing call frame size.

#8 - 07/20/2016 11:22 AM - naruse (Yui NARUSE)

> Secondly, I'd like to combine some frequent insn sequences into
> bigger insns. Again it decreases insn dispatch overhead and
> memory traffic even more. Also it permits to remove some type checking.
>
> The first thing on my mind is a sequence of a compare insn and a
> branch and using immediate operands besides temporary (stack) and
> local variables. Also it is not a trivial task for Ruby as the
> compare can be implemented as a method.

I tried to unify "a sequence of a compare insn and a branch" as follows but 1.2x speed up:
https://github.com/nurse/ruby/commit/a0e8fe14652dbc0a9b830fe84c5db85378accfb7

If it can be written more simple and clean, it's worth to merge...

#9 - 07/20/2016 11:24 PM - vmakarov (Vladimir Makarov)

Yui NARUSE wrote:

> Secondly, I'd like to combine some frequent insn sequences into

> bigger insns. Again it decreases insn dispatch overhead and
> memory traffic even more. Also it permits to remove some type checking.
>
> The first thing on my mind is a sequence of a compare insn and a
> branch and using immediate operands besides temporary (stack) and
> local variables. Also it is not a trivial task for Ruby as the
> compare can be implemented as a method.

I tried to unify "a sequence of a compare insn and a branch" as follows but 1.2x speed up:
https://github.com/nurse/ruby/commit/a0e8fe14652dbc0a9b830fe84c5db85378accfb7

If it can be written more simple and clean, it's worth to merge...

Thank you for the link. Yes, imho, the code is worth to merge. Although RTL insns potentially can give a better improvement, the ETA is not known and even their success is not guaranteed (as I wrote Ruby has a specific feature -- a lot of calls. And calls require work with parameters in stack order anyway).

Using compare and branch is a no-brainer. Many modern processors contain such insns. Actually CPUs can be an inspiring source for what insns to unify. Some CPUs have branch and increment, madd (multiply and add), etc.

#10 - 03/28/2017 03:26 AM - vmakarov (Vladimir Makarov)

I think I've reached a state of the project to make its current code public. Most of the infrastructure for RTL insns and JIT has been implemented.

Although I did a lot of performance experiments to choose the current approach for the project, I did not focus at the performance yet. I wanted to get more solid performance first before publishing it. Unfortunately, I'll have no time for working on the project until May because of GCC7 release. So to get some feedback I decided to publish it earlier. Any comments, proposals, and questions are welcomed.

You can find the code on https://github.com/vnmakarov/ruby/tree/rtl_mjit_branch. Please, read file README.md about the project first.

The HEAD of the branch https://github.com/vnmakarov/ruby/tree/rtl_mjit_branch_base (currently trunk as of Jan) is and will be always the last merge point of branch rtl_mjit_branch with the trunk. To see all changes (the patch is big, more 20K lines), you can use the following link

https://github.com/vnmakarov/ruby/compare/rtl_mjit_branch_base...rtl_mjit_branch

The project is still at very early stages. I am planning to spend half of my work time on it at least for an year. I'll decide what to do with the project in about year depending on where it is going to.

#11 - 03/28/2017 04:21 AM - normalperson (Eric Wong)

vmakarov@redhat.com wrote:

> I think I've reached a state of the project to make its current
> code public. Most of the infrastructure for RTL insns and JIT has
> been implemented.

Thank you for the update! I was just rereading this thread last night (or was it today? I can't tell :<). Anyways I will try to look more deeply at this in a week or two.

#12 - 03/29/2017 04:12 AM - subtileos (Daniel Ferreira)

> I think I've reached a state of the project to make its current
> code public. Most of the infrastructure for RTL insns and JIT has
> been implemented.

Hi Vladimir,

Thank you very much for this post.
That README is priceless.
It is wonderful the kind of work you are doing with such a degree of
entry level details.
I believe that ruby core gets a lot from public posts like yours.
This sort of posts and PR's are the ones that I miss sometimes in
order to be able to understand in better detail the why's of doing
something in one way or another in terms of ruby core implementation.
In the README you explain very well all the surroundings around your
choices and the possibilities.
That makes me believe there may be space for collaboration from
someone that is willing to get deeper into the C level code.
If there is anyway I can be helpful please say so.

Once again thank you very much and keep up with your excellent
contribution making available to the rest of us the same level of
detail and conversation as much as possible.

Regards,

Daniel

P.S.

I was waiting a little bit to see the amount of reception this post
would have and surprisingly only Eric replied to you.
Why is that?

#13 - 03/29/2017 04:32 AM - subtileos (Daniel Ferreira)

Hi Vladimir,

On Tue, Mar 28, 2017 at 4:26 AM, vmakarov@redhat.com wrote:

> You can find the code on
> https://github.com/vnmakarov/ruby/tree/rtl_mjit_branch. Please, read
> file README.md about the project first.

Thank you very much for this post.
That README is priceless.
It is wonderful the kind of work you are doing with such a degree of
entry level details.
I believe that ruby core gets a lot from public posts like yours.
This sort of posts and PR's are the ones that I miss sometimes in
order to be able to understand in better detail the why's of doing
something in one way or another in terms of ruby core implementation.

> The HEAD of the branch
> https://github.com/vnmakarov/ruby/tree/rtl_mjit_branch_base (currently
> trunk as of Jan) is and will be always the last merge point of branch
> rtl_mjit_branch with the trunk. To see all changes (the patch is big,
> more 20K lines), you can use the following link
>
> https://github.com/vnmakarov/ruby/compare/rtl_mjit_branch_base...rtl_mjit_branch

What kind of feedback are you looking forward to get?
Can I help in any way?
Is the goal to try to compile your branch and get specific information
from the generated ruby?
If so what kind of information?

> The project is still at very early stages. I am planning to spend
> half of my work time on it at least for an year. I'll decide what to
> do with the project in about year depending on where it is going to.

In the README you explain very well all the surroundings around your
choices and the possibilities.
That makes me believe there may be space for collaboration from
someone that is willing to get deeper into the C level code.
If there is anyway I can be helpful please say so.

Once again thank you very much and keep up with your excellent

contribution making available to the rest of us the same level of detail and conversation as much as possible.

Regards,

Daniel

P.S.

I was waiting a little bit to see the amount of reception this post would have and surprisingly only Eric replied to you.
Why is that?

#14 - 03/29/2017 05:06 PM - vmakarov (Vladimir Makarov)

subtileos (Daniel Ferreira) wrote:

> Hi Vladimir,
>
> On Tue, Mar 28, 2017 at 4:26 AM, vmakarov@redhat.com wrote:
>
> > You can find the code on https://github.com/vnmakarov/ruby/tree/rtl_mjit_branch. Please, read file README.md about the project first.
>
> Thank you very much for this post.

You are welcomed.

> That README is priceless.
> It is wonderful the kind of work you are doing with such a degree of entry level details.
> I believe that ruby core gets a lot from public posts like yours.
> This sort of posts and PR's are the ones that I miss sometimes in order to be able to understand in better detail the why's of doing something in one way or another in terms of ruby core implementation.
>
> > The HEAD of the branch https://github.com/vnmakarov/ruby/tree/rtl_mjit_branch_base (currently trunk as of Jan) is and will be always the last merge point of branch rtl_mjit_branch with the trunk. To see all changes (the patch is big, more 20K lines), you can use the following link
> >
> > https://github.com/vnmakarov/ruby/compare/rtl_mjit_branch_base...rtl_mjit_branch
>
> What kind of feedback are you looking forward to get?

My approach to JIT is not traditional. I believe that implementing JIT in MRI should be more evolutional to be successful. The changes should be minimized but other ways should be still open. My second choice would be a specialized JIT with 3-4 faster compilation speed like luajit but it is in order magnitude bigger project (probably even more) than the current approach and for sure has a bigger chance to fail at the end. So the discussion of the current and other approaches would be helpful for me to better understand how reasonable my current approach is.

Another thing is to avoid a work duplication. My very first post in this thread was to figure out if somebody is already working on something analogous. I did not get an exact confirmation that I am doing a duplicative work. So I went ahead with the project.

For people who works on a Ruby JIT openly or in secret, posting info about my project would be helpful. At least my investigation of Oracle Graal and IBM OMR was very helpful.

Also I am pretty new to MRI sources (I started to work on it just about year ago). I found that MRI lacks documentation and comments. There is no document like GCC internals which could be helpful for a newbie. So I might be doing stupid things which can be done easier and I might not be following some implicit source code policies.

> Can I help in any way?
> Is the goal to try to compile your branch and get specific information from the generated ruby?
> If so what kind of information?

Trying the branch and informing what you like or don't like would be helpful. It could be anything, e.g. insn names. As I wrote RTL insns should work for serious Ruby programs. I definitely can not say the same about JIT. Still there is a chance that RTL breaks some code. Also RTL code might be slower because not all edge cases are implemented with the same level optimization as stack code (e.g. multiple assignment) and some Ruby code

can be better fit to the stack insns. It would be interesting to see such code.

MJIT is at very early stages of development. I think it will have a big chance to be successful if I achieve inlining on the path RUBY->C->Ruby for a reasonable compilation time. But even implementing this will not speed some Ruby code considerably (e.g. floating point benchmarks can not be speed up without changing representation of double/VALUE in MRI).

> The project is still at very early stages. I am planning to spend
> half of my work time on it at least for an year. I'll decide what to
> do with the project in about year depending on where it is going to.

> In the README you explain very well all the surroundings around your
> choices and the possibilities.

I omitted a few other pros and cons of the choices.

> That makes me believe there may be space for collaboration from
> someone that is willing to get deeper into the C level code.
> If there is anyway I can be helpful please say so.

Thank you. I guess if I get some considerable performance improvement, some help can be useful for more or less independent works. But unfortunately, I am not at this stage yet. I hope to get performance improvements I expect in a half year.

> Once again thank you very much and keep up with your excellent
> contribution making available to the rest of us the same level of
> detail and conversation as much as possible.

Thank you for kind words, Daniel and Eric.

> I was waiting a little bit to see the amount of reception this post
> would have and surprisingly only Eric replied to you.
> Why is that?

I think people need some time to evaluate the current state of the project and perspectives. It is not a traditional approach to JIT. This is what at least I would do myself. There are a lot of details in the new code. I would spend time to read sources to understand the approach better. And usually the concerned people are very busy. So it might need a few weeks.

#15 - 03/30/2017 03:12 AM - vmakarov (Vladimir Makarov)

Sorry, Matthew. I can not find your message on https://bugs.ruby-lang.org/issues/12589. So I am sending this message through email.

On 03/29/2017 04:36 PM, Matthew Gaudet wrote:

> Hi Vladimir,
>
> First and foremost, let me join in with others in thanking you for
> opening up your experimentation. I suspect that you'd be one of the
> 'secret' Ruby JITs Chris Seaton was talking about [1]. One more secret
> JIT to go :)
>
> Thank you. I would not call it a secret. I wrote about it couple times
> publicly. But it was a quite development. This is my first major update
> about the project.
>
> > I believe that implementing JIT in MRI should be more evolutional to
> > be successful.
> >
> > [...]
> >
> > Another thing is to avoid a work duplication.
>
> So far, evolutionary approaches have heavily dominated the work we've
> done with Ruby+OMR as well. I also recently wrote an article about what
> needs to happen with Ruby+OMR [2]. One thing in that article I want to
> call out is my belief that those of us working on JIT compilers for MRI
> have many opportunities to share ideas, implementation and features.
> My hope is that we can all keep each other in mind when working on
> things.

> I read your article. It was helpful. And I am agree with you about sharing the ideas.
> I haven't had a huge amount of time to go through your patches, though, I have gone through some of it. One comment I would make is that it seems you've got two very separate projects here: One is a re-design of YARV as an RTL machine, and the other is MJIT, your JIT that takes advantage of the structure of the RTL instructions. In my opinion, it is worth considering these two projects separately. My (offhand) guess would be that I could adapt Ruby+OMR to consume the RTL instructions in a couple of weeks, and other (secret) JITs may be in a similar place.

Yes, may be you are right about separating the project. For me it is just one project. I don't see MJIT development without RTL. I'll need a program analysis and RTL is more adequate approach for this than stack insns.

> Your approach to MJIT certainly seems interesting. I was quite impressed with the compile times you mentioned -- when I was first thinking about your approach I had thought they would be quite a bit higher.
>
> One question I have (and this is largely for the Ruby community to answer) is about how to measure impacts from JITs on non-performance metrics. In this case for example, should MJIT's dynamic memory footprint be computed as the total of the Ruby process and GCC, or can we maybe ignore the GCC costs -- at a cost to compilation time you could do the compiles elsewhere, and you have a clear path to Ahead-of-Time compilation in your mind.
>
> Yes we should measure memory footprint too to compare different JITs.

MJIT code itself is currently very small about 40KB. GCC code is pretty big about 20MB (LLVM library is even bigger) but code of multiple running instances of GCC (even hundred of them) will have the same 20MB in memory at least on Linux.

The data created in GCC is more important. GCC is not monstrous. As any optimizing compiler, it works by passes (in GCC there are more 300 of them): a pass gets IR, allocates the pass data, transforms IR, and frees the data. So the peak consumption is not big. I'd say the peak consumption for typical ISEQ with the compiled environment would be about couple megabytes.

GCC developers really care about data consumption and compiler speed.

There are some passes (GCSE and RA) which consume a lot of data (sometimes the data consumption is quadratic of IR size). Still GCC is every tunable and such behaviour can be avoided with particular options and parameters. I suspect, other JIT implementations will have analogous memory footprint for the data if they do inlining.

> My recollection is that one of the reasons rujit was abandoned was because its memory footprint was considered unacceptable, but, I don't don't know how that conclusion was drawn.
>
> It would be interesting to know all reasons why rujit was abandoned. I suspect it was more than just the data consumption.

You can not implement JIT without consuming additional memory. May be for some MRI environments like Heroku the additional memory consumption can be critical. And for such environment it might be better not to use JIT at all. Still there are other Ruby environments where people can spare memory consumption for faster code.

> At least my investigation of Oracle Graal and IBM OMR was very helpful.

> Glad we could help. One small note: The project is called Eclipse OMR, not IBM OMR. While IBM is the largest contributor right now, we're trying to build a community around the project, and it is run through the Eclipse foundation.

Thanks for the clarification.

I can also share my finding about Ruby OMR. I found that Ruby OMR is one thread program. So MRI waits for OMR to get a machine code and it hurts the performance. I think the compilation should be done in parallel with code execution in the interpreter as for Graal or JVM.

[1]: https://twitter.com/ChrisGSeaton/status/811303853488488448
[2]: https://developer.ibm.com/open/2017/03/01/ruby-omr-jit-compiler-whats-next/

#16 - 03/30/2017 04:41 AM - subtileos (Daniel Ferreira)

Hi Matthew,

https://developer.ibm.com/open/2017/03/01/ruby-omr-jit-compiler-whats-next/

I was reading your article, and I would like to say that what you present there is just fantastic in my point of view.
Why fantastic? Because having IBM embracing Ruby in that way can only give Ruby a brilliant future.
We have IBM and Oracle and Heroku and Redhat. How many companies more besides Japan (which also should be better exposed)? It is not just some developers. This is a powerful message for the world community and in my opinion Ruby needs to clearly present it to the wider audience.

This pleases me because I'm totally Ruby biased (for the better and the worst). (For me Ruby should be used everywhere. Even as a replacement for javascript. Opal needs more emphasis. I just love it.)

Ever since I heard about Ruby 3x3 in Matz announcement that I clearly saw it would be a major opportunity for Ruby to stand out from the crowd. A genius marketing move that well coordinated could have a very important impact in the coming future regarding the dynamic languages current competitive ecosystem.

I want to be part of it and have been trying to find a way to do that.
This is the reason I asked Vladimir what help could he be using from me.
I even asked about Ruby 3x3 to Eric regarding my symbols thread which is not dead.

It is also great that you agree that there is much room for collaboration.
I'm a newbie in terms of compilers and JITs and all that jazz but I'm willing to dig in and learn as much as possible and contribute the better I can.

For me it doesn't matter in what project.
What is important for me is a collaborative environment where we can communicate and learn things step-by-step throughout the way which seems what you have in your mind to offer.

Very glad you are creating the eclipse community.

You ask there what would be the best way to build that community.
I have a suggestion: Consider doing it by sharing the discussions with ruby-core like Vladimir is doing.
I was totally unaware of your current work if it not for this thread (I thought OMR was still closed code).
Anyone that do care about Ruby development subscribes to ruby-core.

I believe I can help also in terms of organisation.
I have clear ideas on how to improve ruby regarding communication and documentation.
And I'm very focused on architecture logic speaking about web development and DevOps but software design as a all.
I'm pretty sure I will learn tones working with you and being part of this endeavour but I can bring some added value in that regard.

Like Vladimir said Ruby lacks a way for new people to come on board in an easy way. When I develop code I always pay lots of emphasis to the

files organisation and design patterns being put in place, the tests and documentation so that it can be always easy to understand the architecture and reasons some options have been made.

Ruby 3x3 is for me a big opportunity to look at that problem and try to put some architecture documents in place.

This implies that for me each one of this projects should work in close form with ruby core developers. Again a reason to have OMR directly linked to ruby core issue tracker.

You mention as well that the existence of multiple JIT projects and that competition can only bring good things to Ruby itself.
Couldn't agree more. Important for me is to not let this projects to die.
One of the great things the ruby community has, is that ability to make each developer feel at home.
Matz was able to build that throughout the time.

Let me hear your thoughts on the matter.
If you are ready to bring me on board I'm ready to step in.

A note on that regard is that all my contribution for now will need to be out of work hours.
But in the future maybe I can convince my company to sponsorship me.
No promise as I didn't speak with them yet.

Regards,

Daniel

P.S.

(This text is pretty much some scattered thoughts but I will send it as it is anyway. Have so much things to say that I'm afraid if I start to better structure the text it will become to big for someone to read)

P.S.2

Sorry Vladimir for replying to Matthew on your thread. But I'm doing it to emphasise how much I think we should work together on this matter. (I could have sent a private email, think it is much better this way)

#17 - 03/31/2017 05:27 PM - magaudet (Matthew Gaudet)

vmakarov (Vladimir Makarov) wrote:

> Sorry, Matthew. I can not find your message on https://bugs.ruby-lang.org/issues/12589. So I am sending this message through email.

Very curious! I don't quite know what went wrong... so here I am writing a reply in Redmine to make sure it shows up for future searchers :)

> I read your article. It was helpful. And I am agree with you about sharing the ideas.

Glad to hear it. Let me know if there's any feature you'd like to see implemented that you'd like collaboration on. I've already submitted a patch for one feature we expect to be useful in the future (https://bugs.ruby-lang.org/issues/13265), and would be interested in helping to do more if desired.

> Yes, may be you are right about separating the project. For me it is just one project. I don't see MJIT development without RTL. I'll need a program analysis and RTL is more adequate approach for this than stack insns.

I totally understand. Especially for you, I can see how RTL feels like almost a means-to-an-end; I would just encourage you (and others in the Ruby community) to think of them separately, as if RTL is superior, it would be a shame to lose that progress if MJIT doesn't satisfy all its goals.

> Yes we should measure memory footprint too to compare different JITs.
>
> MJIT code itself is currently very small about 40KB. GCC code is pretty big about 20MB (LLVM library is even bigger) but code of multiple running instances of GCC (even hundred of them) will have the same 20MB in memory at least on Linux.
>
> The data created in GCC is more important. GCC is not monstrous. As any optimizing compiler, it works by passes (in GCC there are more 300 of them): a pass gets IR, allocates the pass data, transforms IR, and frees the data. So the peak consumption is not big. I'd say the peak consumption for typical ISEQ with the compiled environment would be about couple megabytes.

Kudos to the GCC developers (yourself included). That seems eminently reasonable.

> You can not implement JIT without consuming additional memory. May be for some MRI environments like Heroku the additional memory consumption can be critical. And for such environment it might be better not to use JIT at all. Still there are other Ruby environments where people can spare memory consumption for faster code.

Indeed. I spoke at Ruby Kaigi 2016 1 trying very hard to encouraging thinking about exactly what it is that 3x3 should accomplish, and how to measure. As I am sure you are aware, the selection of benchmark and benchmarking methodology is key to making sure you actually achieve your aims.

> I can also share my finding about Ruby OMR. I found that Ruby OMR is one thread program. So MRI waits for OMR to get a machine code and it hurts the performance. I think the compilation should be done in parallel with code execution in the interpreter as for Graal or JVM.

Absolutely agree. It's an item we've opened 2, but just haven't gotten around to implementing.

#18 - 03/31/2017 07:14 PM - vmakarov (Vladimir Makarov)

magaudet (Matthew Gaudet) wrote:

> > You can not implement JIT without consuming additional memory. May be for some MRI environments like Heroku the additional memory consumption can be critical. And for such environment it might be better not to use JIT at all. Still there are other Ruby environments where people can spare memory consumption for faster code.
>
> Indeed. I spoke at Ruby Kaigi 2016 [1] trying very hard to encouraging thinking about exactly what it is that 3x3 should accomplish, and how to measure. As I am sure you are aware, the selection of benchmark and benchmarking methodology is key to making sure you actually achieve your aims.
>
> [1]: http://rubykaigi.org/2016/presentations/MattStudies.html

By the way, I did some memory consumption measurements using size of max (peak) resident area for a small Ruby program (about 15 lines) and its sub-processes on free x86-64 machine with 32GB memory using (j)ruby --disable-gems. Here are the numbers:

```
Ruby trunk:                                   6.4MB
RTL:                                          6.5MB
RTL+GCC JIT:                                 26.9MB
RTL+LLVM JIT:                                52.1MB
OMR:                                          6.5MB
OMR+JIT:                                     18.0MB
jruby:                                      244.5MB
Graal:                                      771.0MB
```

It can give a rough idea what JIT memory consumption costs are.

The numbers should be taken with a grain of salt. It includes all code size too. As I wrote multiple running program copies

share the code. And in case of GCC (cc1) it is about 20MB (so in a case of 20 running GCC on a server, the average size of max resident area could be about 7.9MB).

I have no idea what is the code size of jruby and Graal as they use sub-processes and I know nothing about them.

#19 - 04/08/2017 11:41 PM - normalperson (Eric Wong)

vmakarov@redhat.com wrote:

> https://github.com/vnmakarov/ruby/compare/rtl_mjit_branch_base...rtl_mjit_branch

I've only taken a light look at it; but I think RTL seems interesting promise. I needed the following patch to remove "restrict" to compile on Debian stable:

```
https://80x24.org/spew/20170408231647.8664-1-e@80x24.org/raw
```

I also noted some rubyspec failures around break/while loops which might be RTL related (make update-rubyspec && make test-rubyspec):

```
https://80x24.org/spew/20170408231930.GA11999@starla/
```

(The Random.urandom can be ignored since you're on an old version)

I haven't tried JIT, yet, as I'm already unhappy with current Ruby memory usage; but if RTL alone can provide small speed improvements without significant footprint I can deal with it.

I'm currently running dtas-player with RTL to play music and it seems fine https://80x24.org/dtas/

Thanks.

Disclaimer: I do not use proprietary software (including JS) or GUI browsers; so I use "git fetch vnmakarov" and other normal git commands to fetch your changes after having the following entry in .git/config:

```
[remote "vnmakarov"]
    fetch = +refs/heads/*:refs/remotes/vnmakarov/*
    url = https://github.com/vnmakarov/ruby.git
```

#20 - 04/09/2017 03:17 PM - vmakarov (Vladimir Makarov)

normalperson (Eric Wong) wrote:

> vmakarov@redhat.com wrote:
>
> > https://github.com/vnmakarov/ruby/compare/rtl_mjit_branch_base...rtl_mjit_branch
>
> I've only taken a light look at it; but I think RTL seems interesting promise. I needed the following patch to remove "restrict" to compile on Debian stable:
>
> https://80x24.org/spew/20170408231647.8664-1-e@80x24.org/raw
>
> I also noted some rubyspec failures around break/while loops which might be RTL related (make update-rubyspec && make test-rubyspec):
>
> https://80x24.org/spew/20170408231930.GA11999@starla/
>
> (The Random.urandom can be ignored since you're on an old version)

Thank you for your feedback, Eric. I'll work on issues you found.

So far I spent about 80% of my MRI efforts on RTL. But probably it was because of the learning curve. I did not try RTL on serious Ruby applications yet. On small benchmarks, I got from 0% to 100% (for a simple while loop) improvement. I'd say the average improvement could be 10%. MRI has too many calls on which majority of time spent. So savings on less insn dispatching and memory traffic have a small impact. In some cases RTL can be even worse. For example, o.m(a1, a2, a3) has the following stack insns and RTL insns:

```
  push <o index>
  push <a1 index>
  push <a2 index>
  push <a3 index>
  send <callinfo> <cache>
```

```
  loc2temp -2, <a1 index>
  loc2temp -3, <a2 index>
  loc2temp -4, <a3 index>
  call_recv <call data>, <o index>, -1
```

RTL insns are 18% longer for this example. I am going to investigate what the overall length of executed stack insns vs RTL insns when I resume my work on the project.

> I haven't tried JIT, yet, as I'm already unhappy with current
> Ruby memory usage; but if RTL alone can provide small speed
> improvements without significant footprint I can deal with it.

I believe there would be no additional footprint for RTL insn or there would an insignificant increase (1-2%).

JIT is ready only for small benchmarks right now. My big worry is in using exec wrapper when we go from JITed code execution to interpreted code execution to another JITed code and so on. It might increase stack usage. But I am going to work on removing exec wrapper usage in some cases.

If you are not happy with the current MRI memory footprint, you will be definitely unhappy with any JIT because their work will require much more peak memory (at least in order of magnitude) than the current MRI footprint.

But I think with my approach I can use much less memory and CPUs (JITs might require more CPU usage because of the compilations) than jruby or Graal. My JIT will also have no startup delay which is huge for jruby and Graal. Still achieving a better performance (wall clock execution) should be the first priority of my JIT project.

By the way, I forgot to mention that my approach also opens a possibility in future to distribute gems in C code without binaries and it might help gems portability.

> I'm currently running dtas-player with RTL to play music and it
> seems fine https://80x24.org/dtas/

Great! Thank you for sharing this.

#21 - 04/09/2017 09:22 PM - normalperson (Eric Wong)

vmakarov@redhat.com wrote:

> ```
> **stack-based** insns to **register transfer** ones.  The idea behind
> it is to decrease VM dispatch overhead as approximately 2 times
> less RTL insns are necessary than stack based insns for the same
> program (for Ruby it is probably even less as a typical Ruby program
> contains a lot of method calls and the arguments are passed through
> the stack).
>
> But *decreasing memory traffic* is even more important advantage
> of RTL insns as an RTL insn can address temporaries (stack) and
> local variables in any combination.  So there is no necessity to
> put an insn result on the stack and then move it to a local
> variable or put variable value on the stack and then use it as an
> insn operand.  Insns doing more also provide a bigger scope for C
> compiler optimizations.
> ```

One optimization I'd like to add while remaining 100% compatible with existing code is to add a way to annotate read-only args for methods (at least those defined in C-API). That will allow delaying putstring instructions and giving them the same effect as putobject.

This would require having visibility into the resolved method at runtime; before putting its args on the stack.

One trivial example would be the following, where String#start_with? has been annotated(*) with the args being read-only:

```
foo.start_with?("/")
```

Instead of resolving the 'putstring "/"', first;
the method "start_with?" is resolved.

If start_with? is String#start_with? with a constant
annotation(*) for the arg(s); the 'putstring "/"'
instruction returns the string w/o resurrecting it
to avoid the allocation.

This would be a more generic way of doing things like
opt_aref_with/opt_aset_with; but without adding more global
redefinition flags.

(*) Defining a method may change from:

rb_define_method(rb_cString, "start_with?", rb_str_start_with, -1);

To something like:

rb_define_method2(rb_cString, "start_with?", rb_str_start_with,
"RO(*prefixes)");

But rb_define_method should continue to work as-is for old code;
but having a new rb_define_method2 would also allow us to fix
current inefficiencies in rb_scan_args and rb_get_kwargs.

#22 - 04/12/2017 02:17 AM - vmakarov (Vladimir Makarov)

normalperson (Eric Wong) wrote:

> One optimization I'd like to add while remaining 100% compatible
> with existing code is to add a way to annotate read-only args for
> methods (at least those defined in C-API). That will allow
> delaying putstring instructions and giving them the same effect
> as putobject.

Your idea is interesting. I guess the optimization would be very useful and help MRI memory system.

I'll think too how to implement it with RTL insns.

I wanted to try new call insns where the call arguments are call insn parameters, e.g. call2 recv, arg1, arg2 where recv, arg1, arg2 are location indexes or even values. If it works with the performance point of view, the optimization implementation would be pretty straightforward.

#23 - 06/01/2017 12:56 AM - vmakarov (Vladimir Makarov)

I've updated README.md of the project. I added performance (wall, CPU time, memory consumption) comparison of the current state of MJIT with some other MRI versions (v2.0, base) and implementations (JRuby, Graal, OMR) on different benchmarks including OPTCARROT.

I hope it will be interesting and save time if somebody decides to evaluate MJIT.

You can find the performance section on https://github.com/vnmakarov/ruby#update-31-may-2017

#24 - 06/03/2017 01:41 AM - normalperson (Eric Wong)

vmakarov@redhat.com wrote:

> I've updated README.md of the project. I added performance (wall, CPU time, memory consumption) comparison of the current state of MJIT with some other MRI versions (v2.0, base) and implementations (JRuby, Graal, OMR) on different benchmarks including OPTCARROT.

Thanks.

Btw, have you explored the GNU lightning JIT at all?
http://www.gnu.org/software/lightning/
I'm on the mailing list and it doesn't seem very active, though...

> I hope it will be interesting and save time if somebody decides to evaluate MJIT.
>
> You can find the performance section on https://github.com/vnmakarov/ruby#update-31-may-2017

I encountered a new compatibility problem with gcc 4.9 on
Debian stable with -Werror=incompatible-pointer-types not
being supported.

Also, my previous comment about C99 "restrict" not working on my setup still applies.

Sorry, I haven't had more time to look at your work; but I guess it's mostly ko1's job since I'm no a compiler/VM expert; just a *nix plumber.

Thanks again.

#25 - 06/04/2017 05:06 PM - vmakarov (Vladimir Makarov)

normalperson (Eric Wong) wrote:

> Btw, have you explored the GNU lightning JIT at all?
> http://www.gnu.org/software/lightning/
> I'm on the mailing list and it doesn't seem very active, though...

Yes, I know about GNULighting, Eric. It is an old project. It is just a portable assembler.

Using it for JIT, it is like building a car having only a wheel. To get a good performance result for JIT, you still need to write a lot of optimizations. To get at least 50% performance result of GCC or LLVM, somebody should spend many years to implement 10-20 most useful optimizations. Using tracing JITs could simplify the work as a compiled code has a very simple control flow graph (extended basic blocks). But still it is a lot of work (at least 10 man years) especially if you need achieve a good reliability and portability.

It is possible to make a port of GCC to GNUlighting to use GCC optimizations but it has no sense as GCC can directly generate a code to targets supported by GNUlighting and even to much more targets.

I've been studying JITs for many years and did some research on it and I don't see a better approach than using GCC (GCC became 30 year old this year) or LLVM. A huge amount of efforts of hundreds of developers were spent on these compilers to get a reliable, portable, and highly optimizing compilers.

There is a myth that JVM JIT creates a better code than GCC/LLVM. I saw reports saying that JVM JIT server compiler achieves only 40% of performance of GCC/LLVM on some code (e.g. a rendering code) of statically typed languages. That is why Azul (LLVM based java) exists despite legal issues.

I think Graal performance based on some articles I read is somewhere in the middle between JVM client and server JIT compilers. Probably OMR is approximately the same (although I know it less than Graal).

So saying using GCC/LLVM is the best option in my opinion, still there is an open question how to use them. GCC has libjit and LLVM has MCJIT. Their API might change in future. It is better to use C as the input because C definition is *stable*.

It looks like libjit and MCJIT is a shortcut in comparison with C compilation. But it is not that big as the biggest CPU consumers in GCC and LLVM are optimizations not lexical analysis or parsing. I minimize this difference even more with a few techniques, e.g. using precompiled headers for the environment (declarations and definitions needed for C code compiled from a Ruby method by JIT). By the way JVM uses analogous approach (class data sharing) for faster startup.

Using C as JIT input makes an easy switch from GCC to LLVM and vise verse. It makes JIT debugging easier. It simplifies the environment creation, e.g. libjit would need a huge number of tedious calls to the API to do the same. Libjit has no ability to do inlining too which would prevent inlining on Ruby->C->Ruby path.

So I see more upsides than downsides of my approach. The current performance are also encouraging -- **I have better performance on many tests than JRuby or Graal Ruby using much less computer resources** although I did not start yet to work on Ruby->Ruby inlining and Ruby->C->Ruby inlining.

> I encountered a new compatibility problem with gcc 4.9 on
> Debian stable with -Werror=incompatible-pointer-types not
> being supported.
>
> Also, my previous comment about C99 "restrict" not working on
> my setup still applies.

My project is just at the initial stages. There are a lot of things to do. When I implement inlining I will focus on JIT reliability and stability. I don't think MJIT can be used right now for more serious programs.

I should remove -Werror=incompatible-pointer-types from the script and restrict added by me. They are not important.

The code is currently tuned for my major environment (FC25 Linux). I very rarely check OSX. Some work should be done for configuring MRI to use right options depending on the environment.

Eric, thank you for trying my code and giving a feedback. I really appreciate it.

#26 - 06/06/2017 01:51 AM - normalperson (Eric Wong)

vmakarov@redhat.com wrote:

Thanks for detailed response.

> I should remove -Werror=incompatible-pointer-types from the script and
> restrict added by me. They are not important.

Actually, I've discovered AC_C_RESTRICT is convenient to add to configure.in and I would like us to be able to take advantage of useful C99 (and C1x) features as they become available:

https://80x24.org/spew/20170606012921.26806-1-e@80x24.org/raw

Perhaps -Werror=incompatible-pointer-types can be made a standard warning flag for building Ruby, too...

> The code is currently tuned for my major environment (FC25 Linux). I
> very rarely check OSX. Some work should be done for configuring MRI
> to use right options depending on the environment.

Heh, I never run non-Free systems like OSX. Anyways I've been using FreeBSD (via QEMU) sometimes and found Wshorten-64-to-32 errors in clang:

https://80x24.org/spew/20170606012944.26869-1-e@80x24.org/raw

I guess that will help clang testers on other systems, too.

> Eric, thank you for trying my code and giving a feedback. I really
> appreciate it.

No problem! I'm still learning VM and compiler stuff from all this and will do what I can to keep things running on the ancient crap I have :)

#27 - 06/12/2017 09:51 PM - normalperson (Eric Wong)

Eric Wong normalperson@yhbt.net wrote:

> vmakarov@redhat.com wrote:
>
> > I should remove -Werror=incompatible-pointer-types from the script and
> > restrict added by me. They are not important.
>
> Actually, I've discovered AC_C_RESTRICT is convenient to add to

> configure.in and I would like us to be able to take advantage of useful C99 (and C1x) features as they become available:
>
> https://80x24.org/spew/20170606012921.26806-1-e@80x24.org/raw

Ah, I noticed you've removed "restrict" from your branch. Technically, wouldn't that be a regression from an optimization standpoint? (of course you know far more about compiler optimization than I).

> Perhaps -Werror=incompatible-pointer-types can be made a standard warning flag for building Ruby, too...

That removal was fine by me.

Not a particularly focused review, just random stuff I'm spotting while taking breaks from other projects.

Mostly just mundane systems stuff, nothing about the actual mjit changes.

- I noticed mjit.c uses it's own custom doubly-linked list for rb_mjit_batch_list. For me, that places a little extra burden in having extra code to review. Any particular reason ccan/list isn't used?

Fwiw, the doubly linked list implementation in compile.c predated ccan/list; and I didn't want to:

a) risk throwing away known-working code

b) introduce a the teeny performance regression for loop-heavy code:

ccan/list is faster for insert/delete, but slightly slower iteration for loops from what I could tell.

- The pthread_* stuff can probably use portable stuff defined in thread.c and thread_*.h. (Unfortunately for me) Ruby needs to support non-Free platforms :<

- fopen should probably be replaced by something which sets cloexec; since the "e" flag of fopen is non-portable.

Perhaps rb_cloexec_open() + fdopen().

- It looks like meant to use fflush instead of fsync; fflush is all that's needed to ensure other processes can see the file changes (and it's done transparently by fclose). fsync is to ensure the file is committed to stable storage, and some folks still use stable storage for /tmp. fsync before the final fflush is wrong, even, as the kernel may not have all the data from userspace

- get_uniq_fname should respect alternate tmpdirs like Dir.tmpdir, does (in lib/dir.rb)

- we can use vfork + execve instead of fork to speed up process creation; just need to move the fopen (which can call malloc) into the parent. We've already used vfork for Process.spawn, system(), ``, IO.popen for a few years.

None of these are super important; and I can eventually take take some time to make send you patches or pull requests (via email/redmine)

rb_mjit_min_header-2.5.0.h takes forever to build...

Thank again for taking your time to work on Ruby!

#28 - 06/13/2017 03:04 PM - vmakarov (Vladimir Makarov)

normalperson (Eric Wong) wrote:

> Eric Wong normalperson@yhbt.net wrote:
>
> Ah, I noticed you've removed "restrict" from your branch.
> Technically, wouldn't that be a regression from an optimization
> standpoint? (of course you know far more about compiler
> optimization than I).

It was just a try to achieve desired aliasing. But it is hard to achieve this. There are too many VALUE * pointers in MRI VM. Removing restrict I added does not worsen the code. Aliasing is a weak point of C. Therefore many HPC developers still prefer Fortran in many cases.

I think changing type of pc might be more productive for achieving necessary aliasing.

> > Perhaps -Werror=incompatible-pointer-types can be made a
> > standard warning flag for building Ruby, too...
>
> That removal was fine by me.
>
> Not a particularly focused review, just random stuff I'm
> spotting while taking breaks from other projects.
>
> Mostly just mundane systems stuff, nothing about the actual
> mjit changes.

Although it is random. Still it took your time to do this and it is valuable to me. Thank you.

- I noticed mjit.c uses it's own custom doubly-linked list for rb_mjit_batch_list. For me, that places a little extra burden in having extra code to review. Any particular reason ccan/list isn't used?

> Fwiw, the doubly linked list implementation in compile.c
> predated ccan/list; and I didn't want to:

I remember MRI lists when I worked on changing compile.c. Uniformity of the code is important. I'll put it on my TODO list.

> a) risk throwing away known-working code
>
> b) introduce a the teeny performance regression for loop-heavy
> code:
>
> ccan/list is faster for insert/delete, but slightly
> slower iteration for loops from what I could tell.

- The pthread_* stuff can probably use portable stuff defined in thread.c and thread_*.h. (Unfortunately for me) Ruby needs to support non-Free platforms :<
- fopen should probably be replaced by something which sets cloexec; since the "e" flag of fopen is non-portable.

> Perhaps rb_cloexec_open() + fdopen().

- It looks like meant to use fflush instead of fsync; fflush is all that's needed to ensure other processes can see the file changes (and it's done transparently by fclose). fsync is to ensure the file is committed to stable storage, and some folks still use stable storage for /tmp. fsync before the final fflush is wrong, even, as the kernel may not have all the data from userspace

Yes, my mistake. I'll correct this. fsync is also worse with the performance point of view.

- get_uniq_fname should respect alternate tmpdirs like Dir.tmpdir, does (in lib/dir.rb)

I'll investigate this. For JIT performance the used temp files should be in a memory FS. If alternative tempdirs provide this, I should switch to it.

- we can use vfork + execve instead of fork to speed up process creation; just need to move the fopen (which can call malloc) into the parent.

> We've already used vfork for Process.spawn, system(), ``, IO.popen for a few years.

Yes, it can be a performance win although probably small one.

> None of these are super important; and I can eventually take
> take some time to make send you patches or pull requests (via
> email/redmine)

Only if it is not a burden for you. You already gave a fresh look
at the code and proposed valuable improvements.

I just focused on Linux and MacOS a bit. I ignored other OSes,
e.g. Windows. My major goal was to justify the approach with the
performance point of view and then work more on MJIT portability.

Now I can say it works although a lot of performance improvements still
can and should be done. I think the portability work already could
start.

> rb_mjit_min_header-2.5.0.h takes forever to build...

Yes, it is slow (about 75 sec on i3-7100). It is a ruby script trying
to remove unnecessary C definitions/declarations. After removing some
C code it calls C compiler to check that the code is valid.

I tried many things to speed it up, e.g. checking that the header will
be the same, removing several declarations at once, using special C
compiler options to speed up the check. But I got your message that it
is still slow.

I'll think about further speed up. May be I'll try running a few C
compilations in parallel or generating a bigger header as
loading/reading pre-compiled header takes a tiny part of even
a small method compilation.

> Thank again for taking your time to work on Ruby!

Eric, thank you for your time reviewing my code.

#29 - 06/13/2017 10:11 PM - normalperson (Eric Wong)

vmakarov@redhat.com wrote:

> normalperson (Eric Wong) wrote:
>
> > None of these are super important; and I can eventually take
> > take some time to make send you patches or pull requests (via
> > email/redmine)
>
> Only if it is not a burden for you. You already gave a fresh look
> at the code and proposed valuable improvements.

Alright; I've actually got plenty on my plate, but...

> > rb_mjit_min_header-2.5.0.h takes forever to build...
>
> Yes, it is slow (about 75 sec on i3-7100). It is a ruby script trying
> to remove unnecessary C definitions/declarations. After removing some
> C code it calls C compiler to check that the code is valid.

Yeah, that could actually be a blocker to potential contributors.

I force myself to work on ancient hardware to notice slow things
before others do; and sometimes I'm less inclined or get
distracted by other projects when builds take a long time.

Good to know you're also bothered by it :)

#30 - 10/18/2017 06:45 PM - k0kubun (Takashi Kokubun)

Hi Vladimir. I was happy to talk with you about JIT at RubyKaigi.

To help introducing RTL and MJIT to upstream Ruby core safely, I'm wondering if we might experimentally introduce optional (switchable by -j option) JIT infrastructure first and then separately introduce mandatory RTL instruction changes.

My experimental attempt for that goal, YARV-MJIT, is here: https://github.com/k0kubun/yarv-mjit
It's basically a fork of your project, but VM part is not changed from current Ruby and compiler is different. It's much slower than MJIT but meaningfully faster than Ruby 2.5.

If we take YARV-MJIT as intermediate step toward RTL-MJIT, we can keep major part of MJIT in upstream without introducing breaking instruction changes (as you know, VM instructions are already compiled by gems like bootsnap. So replacing instructions would be breaking even if it has no bug) possibly in 2.x. And I believe this approach will make it easier to maintain MJIT and make all Rubyists happy in final RTL+MJIT introduction at Ruby 3.

This is work in progress and this comment is just for sharing my plan. My project's current quality is much worse than yours (I found every part of MJIT is well considered and really great during development of YARV-MJIT), so I need to improve mine before I really propose to introduce it to core.

I want to hear your thoughts about this direction.

#31 - 10/19/2017 04:19 AM - vmakarov (Vladimir Makarov)

k0kubun (Takashi Kokubun) wrote:

> Hi Vladimir. I was happy to talk with you about JIT at RubyKaigi.

Hi. I am also glad that I visited RubyKaigi and I got a lot of feedback from my discussions with Koichi, Matz, and you.

> To help introducing RTL and MJIT to upstream Ruby core safely, I'm wondering if we might experimentally introduce optional (switchable by -j option) JIT infrastructure first and then separately introduce mandatory RTL instruction changes.

Yes. I guess it is possible. RTL and MJIT probably can be separated as projects. Matthew Gaudet already proposed to separate RTL. As I understand it could help OMR implementation in some way.

> My experimental attempt for that goal, YARV-MJIT, is here: https://github.com/k0kubun/yarv-mjit
> It's basically a fork of your project, but VM part is not changed from current Ruby and compiler is different. It's much slower than MJIT but meaningfully faster than Ruby 2.5.

I'll look at this. Thank you for pointing this out.

> If we take YARV-MJIT as intermediate step toward RTL-MJIT, we can keep major part of MJIT in upstream without introducing breaking instruction changes (as you know, VM instructions are already compiled by gems like bootsnap. So replacing instructions would be breaking even if it has no bug) possibly in 2.x. And I believe this approach will make it easier to maintain MJIT and make all Rubyists happy in final RTL+MJIT introduction at Ruby 3.

Saving stack insns is what we discussed with Koichi at RubyKaigi and after that. I promised to investigate another way of RTL generation through stack insns. I am now realizing that it might be a better way than generating RTL directly from MRI nodes because

- it will not break existing applications working with stack insns
- stack insns could be a stable existing interface to VM. RTL will be definitely changing as some new optimizations are implemented. So RTL will be unstable for long time and probably there is no sense to open RTL to Ruby programmers at all. Actually a similar approach is used by JVM: bytecode as an interface with JVM and another internal IR for JIT which is not visible for JVM users.
- it will make merging trunk into rtl-mjit branch much easier because the current RTL code generation means complete rewriting big compile.c file and any change to compile.c on the trunk will be a merge problem (now rtl-mjit-branch is almost 1 year behind the trunk).

Slowdown of nodes->stack insns->RTL path might be negligible in comparison with nodes->RTL path. And slowdown is a major disadvantage for stack insn -> RTL path.

So about week ago, I started to work on generation of RTL from stack insns. When the implementation starts working I'll make it public (it will be a separate branch). I hope it will happen in about a month. But it might be delayed if I am distracted by GCC work.

> This is work in progress and this comment is just for sharing my plan. My project's current quality is much worse than yours (I found every part of MJIT is well considered and really great during development of YARV-MJIT), so I need to improve mine before I really propose to introduce it to core.
>
> I want to hear your thoughts about this direction.

I am not against your plan. An alternative approach can be useful but it might be a waste of your time at the end. But any performance work requires a lot alternative implementations (e.g. the current global RA in GCC was actually one of my seven different RA implementations), some temporary

solutions might become permanent. who knows.

I still believe that RTL should exist at the end because GCC/LLVM optimizations will not solve all optimization problems.

For example, GCC/LLVM optimizes well int->fixnum->int->... conversions but they can not optimize well double->flonum->double->... conversions because tagged double representation as Ruby values is too complicated. Therefore fp benchmarks are not improved significantly by MJIT. Optimizing would be not a problem for non-tagged versions of values (e.g. (mode, int) or (mode, double)) but switching to another value representation is practically not possible as the current representation is already reflected in Ruby (objectid) and MRI C interface.

So the solution would be implementing analysis on RTL to use double values in JITted code of a method to avoid double->flonum and flonum->double conversions. RTL is a good fit to this.

Basic type inference could be another example for RTL necessity. I could find other examples.

MJIT itself is currently not stable. And I'd like to work on its stabilization after trying RTL generation from stack insns.

That is my major thoughts about your proposal. Thank you for asking.

#32 - 10/19/2017 08:23 AM - k0kubun (Takashi Kokubun)

> Saving stack insns is what we discussed with Koichi at RubyKaigi and after that. I promised to investigate another way of RTL generation through stack insns. I am now realizing that it might be a better way than generating RTL directly from MRI nodes because
>
> it will not break existing applications working with stack insns

Oh, I didn't know that plan. I like that approach.

> stack insns could be a stable existing interface to VM. RTL will be definitely changing as some new optimizations are implemented. So RTL will be unstable for long time and probably there is no sense to open RTL to Ruby programmers at all. Actually a similar approach is used by JVM: bytecode as an interface with JVM and another internal IR for JIT which is not visible for JVM users.

Good to know. That sounds a good way to introduce RTL insns for easy maintenance.

> Slowdown of nodes->stack insns->RTL path might be negligible in comparison with nodes->RTL path. And slowdown is a major disadvantage for stack insn -> RTL path.

I agree that the slowdown is negligible. For major disadvantage, compared to YARV-MJIT, implementation and debugging would be more complex. So maintainability and performance would be trade-off. Thus I think we need to decide approach comparing differences in implementation complexity and performance difference.

> An alternative approach can be useful but it might be a waste of your time at the end. But any performance work requires a lot alternative implementations (e.g. the current global RA in GCC was actually one of my seven different RA implementations), some temporary solutions might become permanent. who knows.

As I'm hacking Ruby as not work but just hobby to enjoy improving my Ruby core understanding, it wouldn't be a waste of time even if I end up with developing seven different JIT implementations :)

> So the solution would be implementing analysis on RTL to use double values in JITted code of a method to avoid double->flonum and flonum->double conversions. RTL is a good fit to this.

Question for my better understanding: Do you mean GCC and Clang can't optimize double<->flonum conversion well even if all necessary code is inlined? If so, having special effort to optimize it in Ruby core makes sense. I'm not sure why we can't do that with stack-based instructions or just in JIT-ed C code generation process. Can't we introduce instruction specialization (to avoid double<->flonum conversion, not sure its details) without having all instructions as register-based?

> Basic type inference could be another example for RTL necessity. I could find other examples.

Type inference at RTL instructions is interesting topic which I couldn't understand well from discussion with you at RubyKaigi. I'm looking forward to seeing the example!

#33 - 10/19/2017 03:38 PM - vmakarov (Vladimir Makarov)

k0kubun (Takashi Kokubun) wrote:

> > An alternative approach can be useful but it might be a waste of your time at the end. But any performance work requires a lot alternative implementations (e.g. the current global RA in GCC was actually one of my seven different RA implementations), some temporary solutions might become permanent. who knows.

> As I'm hacking Ruby as not work but just hobby to enjoy improving my Ruby core understanding, it wouldn't be a waste of time even if I end up with developing seven different JIT implementations :)

Sorry, Takashi. I was inaccurate. I am agree. Any serious problem solving (even if it does not result in MRI code change) makes anyone a better, more experienced MRI developer.

> > So the solution would be implementing analysis on RTL to use double values in JITted code of a method to avoid double->flonum and flonum->double conversions. RTL is a good fit to this.
>
> Question for my better understanding: Do you mean GCC and Clang can't optimize double<->flonum conversion well even if all necessary code is inlined?

Yes. It is too complicated for them. Tagging doubles manipulates with exponent and mantissa by constraining exponent range and using a part of exponent field to store few less significant bits of mantissa. Even worse, processing 0.0 makes it even more complicated. Optimizing compilers are not smart enough to see that untaging doubles is a reverse operation to tagging and vice versa.

> If so, having special effort to optimize it in Ruby core makes sense. I'm not sure why we can't do that with stack-based instructions or just in JIT-ed C code generation process. Can't we introduce instruction specialization (to avoid double<->flonum conversion, not sure its details) without having all instructions as register-based?

You can do the optimization with stack insns. You need to analyze all method(s) code and see where from operand values come. It is easier to do with RTL.

But actually the worst part with using stack insns for optimizations is that you can not easily transform a program on them (e.g. move an invariant expression from the loop -- you need to introduce new local vars for this) because they process values only in a stack mode and optimized code can process values in any order.

In any case, if we are going to do some optimizations by ourself (and I see such necessity in the future) not only by GCC/LLVM, we need a convenient IR for this. I tried to explain it in my presentation at RubyKaigi.

One simple case where we can avoid untagging is RTL insn with immediate operand (we can use double not VALUE for the immediate operand). It is actually on my TODO list.

> > Basic type inference could be another example for RTL necessity. I could find other examples.
>
> Type inference at RTL instructions is interesting topic which I couldn't understand well from discussion with you at RubyKaigi. I'm looking forward to seeing the example!

https://github.com/dino-lang/dino/blob/master/DINO/d_inference.c is an example how a basic type inference can be implemented on RTL-like language. It is a different approach to algorithm W in Hindley–Milner type system. The algorithm consists of the following steps

1. Building a control flow graph (CFG) consisting of basic blocks and control flow edges connecting them.
2. Calculating available results of RTL instructions – this is a forward data-flow problem on the CFG.
3. Using the availability information, building def-use chains connecting possible operands and results of RTL instructions and variables.
4. Calculating the types of RTL instruction operands and results – this is a forward data-flow problem on the def-use graph.

The definition of availability and def-use chains can be found practically in any book about optimizing compilers.

#34 - 10/20/2017 10:13 AM - k0kubun (Takashi Kokubun)

> In any case, if we are going to do some optimizations by ourself (and I see such necessity in the future) not only by GCC/LLVM, we need a convenient IR for this.

Yeah, I saw rtl_exec.c transforms ISeq dynamically and allows MJIT to have insn that can be inlined easily. I can imagine the same idea will work on stack->RTL->JIT version of your MJIT. For our shared goal (stack->JIT), I agree that having any IR might be a helpful tool.

> Tagging doubles manipulates with exponent and mantissa by constraining exponent range and using a part of exponent field to store few less significant bits of mantissa. Even worse, processing 0.0 makes it even more complicated. Optimizing compilers are not smart enough to see that untaging doubles is a reverse operation to tagging and vice versa.
>
> One simple case where we can avoid untagging is RTL insn with immediate operand (we can use double not VALUE for the immediate operand).

That makes sense. If compiler can't do that with inlined code, we need to do in MJIT level and it would decrease effort in some level if insns are RTL.

> https://github.com/dino-lang/dino/blob/master/DINO/d_inference.c is an example how a basic type inference can be implemented on RTL-like language. It is a different approach to algorithm W in Hindley–Milner type system. The algorithm consists of the following steps
>
> Building a control flow graph (CFG) consisting of basic blocks and control flow edges connecting them.
> Calculating available results of RTL instructions – this is a forward data-flow problem on the CFG.
> Using the availability information, building def-use chains connecting possible operands and results of RTL instructions and variables.
> Calculating the types of RTL instruction operands and results – this is a forward data-flow problem on the def-use graph.
> The definition of availability and def-use chains can be found practically in any book about optimizing compilers.

Thank you for pointing it out and summary. I'll take a look.

#35 - 12/26/2017 12:20 AM - k0kubun (Takashi Kokubun)

Happy Holidays, Vladimir. As our work has many duplicated things, I'm proposing to partially merge your work to upstream at https://bugs.ruby-lang.org/issues/14235. I would like your opinion on it.

#36 - 12/26/2017 12:21 AM - k0kubun (Takashi Kokubun)

- Related to Feature #14235: Merge MJIT infrastructure with conservative JIT compiler added

#37 - 12/26/2017 08:19 PM - vmakarov (Vladimir Makarov)

k0kubun (Takashi Kokubun) wrote:

> Happy Holidays, Vladimir. As our work has many duplicated things, I'm proposing to partially merge your work to upstream at https://bugs.ruby-lang.org/issues/14235. I would like your opinion on it.

Thank you, Takashi. Happy holidays to you and your family too.

Thank you very much for working on MJIT and trying alternative ways to use it. You did a great progress with this project.

First, I thought YARV-MJIT project is not worth to work but it seems already working and a working intermediate solution makes a lot of sense because it gives performance improvements and permits to debug and improve MJIT-engine on real applications. Working step by step is a good engineering approach.

So **I support your proposal** but I guess you should get other ruby developer opinions, especially Koichi's one. I did not check your code. I am not sure that MJIT in your pull request will work for all platforms (I used pthreads but to make it more portable MRI thread implementation should be used. Also how MJIT should work on Windows is a question for me). I guess the portability issues can be solved later during 2018 year.

Moreover YARV-MJIT might be a final solution. Although C code generated from RTL is better optimized by a C compiler and RTL is also more convenient for future optimizations in MRI itself, still stack insns can be optimized too although with more efforts and in a less effective way. So if I find that stack insns -> RTL translation has some serious problems, your approach can become a mainstream and I might switch to work on it too.

Right now I see that possible potential problems with stack insn -> RTL approach are big compilation time and the interpretation speed.

I am still working on RTL generation from stack insns. It is already a second version. It became a multipass algorithm because I need to provide the same state of emulated stack (depth and locations of the insn operands) on different CFG paths (it is a typical forward dataflow problem in compilers). So the translator might be slower than I originally thought.

Also a lot of things should be done for RTL to provide the same or better RTL interpretation speed.

In your pull request you are wondering what is the state of my stack insn->RTL generator. I planned to have a working generator before the end of the year but it takes more time than I thought. Now I hope to publish it sometime in February.

#38 - 12/27/2017 09:07 AM - k0kubun (Takashi Kokubun)

Thank you for sharing your thoughts and support.

> So I support your proposal but I guess you should get other ruby developer opinions, especially Koichi's one. I did not check your code.

Today I got code review from Koichi-san and mame-san. We found potential bugs in exception handling and TracePoint support, but it's considered to be not so hard to fix in our discussion.

And Koichi said: After fixing it and confirming that all tests pass on a mode that forces to synchronously compile all ISeqs and allows to compile infinite ISeqs, we can merge it.

> I am not sure that MJIT in your pull request will work for all platforms (I used pthreads but to make it more portable MRI thread implementation should be used. Also how MJIT should work on Windows is a question for me).

I fixed pthread part to use Windows native thread for Windows. So it can be compiled on mswin64. At initial merge I'm not going to support cl.exe (this is recognized by mswin64 maintainer). I'm going to fix mjit_init to disable MJIT if a compiler is not found especially for mswin64, but it'll be the only

change for platforms before merge. I have an idea for supporting cl.exe and it'll be done in early stage after merge.

I understand the mswin64 support allows us to cover all platforms that had been considered as tier1 in https://bugs.ruby-lang.org/projects/ruby-trunk/wiki/SupportedPlatforms (but tierX information is dropped recently). I believe tier2 ones work too (at least I confirmed it works on MinGW by https://github.com/vnmakarov/ruby/pull/4 work). So I think it should be sufficient for now.

> Moreover YARV-MJIT might be a final solution. Although C code generated from RTL is better optimized by a C compiler and RTL is also more convenient for future optimizations in MRI itself, still stack insns can be optimized too although with more efforts and in a less effective way. So if I find that stack insns -> RTL translation has some serious problems, your approach can become a mainstream and I might switch to work on it too.

I see. I'll continue to improve YARV-MJIT for the case to have serious problems in stack insns -> RTL, but I'm willing to see your new version of JIT compiler as I (and probably many Ruby users) want faster Ruby and I'm interested in technical differences between stack and RTL. I'll keep the mjit_compile function easy to replace.

> Right now I see that possible potential problems with stack insn -> RTL approach are big compilation time and the interpretation speed.
>
> I am still working on RTL generation from stack insns. It is already a second version. It became a multipass algorithm because I need to provide the same state of emulated stack (depth and locations of the insn operands) on different CFG paths (it is a typical forward dataflow problem in compilers). So the translator might be slower than I originally thought.

Interesting. Understanding your ideas and strategies from your code is always valuable experience, so I want to read it when it becomes ready to publish. Thank you for sharing the state.

I hope merging the patch will help your MJIT development by reducing the cost to rebase against trunk. Once it's merged, let's use the same MJIT infrastructure and please send patches you want to include to upstream for "working step by step" anytime.

#39 - 12/27/2017 10:50 AM - dsferreira (Daniel Ferreira)

These are great news for ruby and it’s community. Thank you both for your great work.
Things can only get better! Long live ruby!

#40 - 01/07/2018 09:26 AM - jwmittag (Jörg W Mittag)

vmakarov (Vladimir Makarov) wrote:

> For example, GCC/LLVM optimizes well int->fixnum->int->... conversions but they can not optimize well double->flonum->double->... conversions because tagged double representation as Ruby values is too complicated. […] switching to another value representation is practically not possible as the current representation is already reflected in Ruby (objectid) and MRI C interface.

I don't think objectid should be a stopper. There are exactly three things that objectid guarantees:

- Every object has exactly one objectid.
- An object has the same objectid for its entire lifetime.
- No two objects have the same objectid at the same time (but may have the same objectid at different times).

Any code (Ruby or C) that assumes anything more about objectids (such as specific values) is simply broken.

#41 - 02/19/2018 09:32 PM - vmakarov (Vladimir Makarov)

Last 4 months I've been working on generation of RTL from stack insns. The reason for this is that stack insns are already a part of CRuby. The current generation of RTL directly from the nodes actually would remove this interface.

Another reason for this work is to simplify future merging RTL and MJIT branches with the trunk.

I think I've reached a project state when I can make the branch public. But still there are lot of things to do for this project.

Generation of RTL from stack insns is a harder task than one from nodes. When we generate RTL from nodes we have a lot of context. When we generate RTL from stack insns we need to reconstruct this context (from different CFG paths in a stack insn sequence).

To reconstruct the context we emulate VM stack and can pass a stack insn sequence several times. First we calculate possible stack values on the label. It is a typical forward data flow problem in compilers (the final fixed point is only temporaries on the emulated stack). Then using this info we actually generate RTL insns on the last pass.

I was afraid that stack insn -> RTL generation might considerably slow done CRuby. Fortunately, it is not the case. This generation for optcarrot with one frame (it means practically no execution) takes about 0.2% of all CRuby run time. Running empty script takes about 2% more in comparison with using direct generation of RTL from the nodes.

I created a new branch in my repository for this project. The branch name is stack_rtl_mjit (https://github.com/vnmakarov/ruby/tree/stack-rtl-mjit-base). All my work including MJIT will be done on this branch. The previous branch rtl_mjit_branch is frozen.

The major code to generate RTL from stack insns is placed in a new file rtl_gen.c.

I am going to continue work on this branch. My next plans will be a merge with the trunk and fixing bugs. It is a big job as the branch is based on more than one year old trunk.

There were a lot of changes since then which will affect the code I am working on. The biggest one is Takashi Kokubun's work on MJIT for YARV. Another one is trace insns removal by Koichi Sasada.

I am planning to work on merging with trunk, unification of MJIT code on trunk and the branch, and fixing bugs till April/May. Sorry for a slow pacing but I have no much time for this work until gcc-8 release (probably middle of April).

After that I am going to work on MJIT optimizations including method inlining.

#42 - 02/19/2018 10:17 PM - sam.saffron (Sam Saffron)

I just measured your branch using Discourse bench at: https://github.com/discourse/discourse/blob/master/script/bench.rb

Looks like it is a bit slower than master:

RTL:

```
---
categories:
  50: 53
  75: 59
  90: 65
  99: 104
home:
  50: 59
  75: 69
  90: 82
  99: 130
topic:
  50: 60
  75: 67
  90: 78
  99: 108
categories_admin:
  50: 96
  75: 103
  90: 114
  99: 174
home_admin:
  50: 94
  75: 106
  90: 141
  99: 181
topic_admin:
  50: 110
  75: 119
  90: 136
  99: 197
timings:
  load_rails: 3749
ruby-version: 2.5.0-p-1
rss_kb: 247952
pss_kb: 236618
memorysize: 5.88 GB
```

```
virtual: vmware
architecture: amd64
operatingsystem: Ubuntu
processor0: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
physicalprocessorcount: 1
kernelversion: 4.4.0
```

Master

```
---
categories:
  50: 48
  75: 56
  90: 59
  99: 101
home:
  50: 56
  75: 66
  90: 105
  99: 127
topic:
  50: 54
  75: 65
  90: 80
  99: 122
categories_admin:
  50: 101
  75: 110
  90: 141
  99: 207
home_admin:
  50: 90
  75: 100
  90: 106
  99: 134
topic_admin:
  50: 101
  75: 108
  90: 118
  99: 172
timings:
  load_rails: 3789
ruby-version: 2.6.0-p-1
rss_kb: 276588
pss_kb: 265237
memorysize: 5.88 GB
virtual: vmware
architecture: amd64
operatingsystem: Ubuntu
processor0: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
physicalprocessorcount: 1
kernelversion: 4.4.0
```

Very interesting to see the significant memory improvement, is that expected? only env var I am running is:
RUBY_GLOBAL_METHOD_CACHE_SIZE: 131072

#43 - 02/20/2018 12:40 AM - k0kubun (Takashi Kokubun)

Great work on rtl_gen, Vladimir! Keeping both stack insns and RTL insns would be good for safe migration.

> There were a lot of changes since then which will affect the code I am working on. The biggest one is Takashi Kokubun's work on MJIT for YARV. Another one is trace insns removal by Koichi Sasada.

I hope your work on merging trunk to stack-rtl-mjit would be easy, which was the aim of Feature #14235. mjit_compile takes rb_iseq_constant_body and you should be able to read rtl_encoded from it. Merging it would make your work much portable, which is already running on many RubyCIs.

After merging it, which includes some fixes for JIT in test cases, could you try running "make test-all RUN_OPTS='--jit-wait --jit-min-calls=1'", and also "make test-all RUN_OPTS='--jit-wait --jit-min-calls=5'" if you're using some cache of method calls? The testing strategy was used for merging Feature #14235, and it can make sure that JIT-ed code and RTL insns translated from stack insns work for many cases (Some tests would fail by timeout though).
You're calling abort() for "Not implemented" insns like run_once, but I think it should just skip compiling the ISeq and work, like current trunk. At least it would be needed to pass the test.

#44 - 02/20/2018 04:32 AM - vmakarov (Vladimir Makarov)

On 02/19/2018 07:40 PM, takashikkbn@gmail.com wrote:

> Issue #12589 has been updated by k0kubun (Takashi Kokubun).
>
> Great work on rtl_gen, Vladimir! Keeping both stack insns and RTL insns would be good for safe migration.
> Thank you, Takashi.
>
> > There were a lot of changes since then which will affect the code I am
> > working on. The biggest one is Takashi Kokubun's work on MJIT for
> > YARV. Another one is trace insns removal by Koichi Sasada.
>
> I hope your work on merging trunk to stack-rtl-mjit would be easy, which was the aim of Feature #14235. mjit_compile takes rb_iseq_constant_body and you should be able to read rtl_encoded from it. Merging it would make your work much portable, which is already running on many RubyCIs.
>
> After merging it, which includes some fixes for JIT in test cases, could you try running "make test-all RUN_OPTS='--jit-wait --jit-min-calls=1'", and also "make test-all RUN_OPTS='--jit-wait --jit-min-calls=5'" if you're using some cache of method calls? The testing strategy was used for merging Feature #14235, and it can make sure that JIT-ed code and RTL insns translated from stack insns work for many cases.
> You're calling abort() for "Not implemented" insns like run_once, but I think it should just skip compiling the ISeq and work, like current trunk. At least it would be needed to pass the test.

Thank you for the tips. I am planing to start on merging trunk into
stack-rtl-mjit branch in a week or two.

#45 - 02/20/2018 04:42 AM - vmakarov (Vladimir Makarov)

On 02/19/2018 05:17 PM, sam.saffron@gmail.com wrote:

> Issue #12589 has been updated by sam.saffron (Sam Saffron).
>
> I just measured your branch using Discourse bench at: https://github.com/discourse/discourse/blob/master/script/bench.rb
>
> Looks like it is a bit slower than master:
> Trace insn are still generated on the branch. The current trunk does
> not generate them. Removing trace insns by Koichi improved performance
> by about 10%. I believe the branch will be faster when the trace insns
> are also removed there. But it is hard to predict what the actual
> improvement will be after that.
> RTL:
>
> ```
> timings:
>    load_rails: 3749
> ruby-version: 2.5.0-p-1
> rss_kb: 247952
> pss_kb: 236618
> memorysize: 5.88 GB
> virtual: vmware
> architecture: amd64
> operatingsystem: Ubuntu
> processor0: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
> physicalprocessorcount: 1
> kernelversion: 4.4.0
> ```
>
> Master
>
> ```
> timings:
>    load_rails: 3789
> ruby-version: 2.6.0-p-1
> rss_kb: 276588
> pss_kb: 265237
> memorysize: 5.88 GB
> virtual: vmware
> architecture: amd64
> operatingsystem: Ubuntu
> processor0: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
> physicalprocessorcount: 1
> kernelversion: 4.4.0
> ```
>
> Very interesting to see the significant memory improvement, is that expected?
> No. Actually I expected more memory usage for RTL. It is hard for me to

say a reason for memory improvement for now. When the current trunk will be merged into the branch, I could speculate more. Now the branch code is far behind (about 13 months) the current trunk.

only env var I am running is: RUBY_GLOBAL_METHOD_CACHE_SIZE: 131072

#46 - 02/20/2018 05:00 AM - sam.saffron (Sam Saffron)

No problems, thank you Vladimir, let me know when you are ready for me to test again!