A Faster CRuby interpreter with dynamically specialized IR

Vladimir Makarov

RedHat

Sep 10, 2022

Presentation outline

- Project motivation and initial expectations
- Dynamically specialized VM insns (IR)
- Current state of the project
- Microbenchmark performance comparison of
  - the base interpreter
  - the interpreter with specialized IR (SIR)
  - YJIT and MJIT
  - and early stage CRuby MIR JIT based on SIR
- Future plans

The project motivation

- MIR project to address shortcomings of MJIT
- MIR is an **universal light-weight** JIT compiler
  - already good for JITting **static** programming languages
  - still a lot of work to do to make it good for **dynamic** programming languages
    - a **generalized lazy basic block versioning** based on dynamic code properties
    - **trace** generation and optimization based on basic block cloning
    - ultimate goal is **meta-tracing** MIR C compiler
    - more details in my blog post "Code specialization for the MIR lightweight JIT compiler"
- Introduction of YJIT was a major disruption
  - need to use **more pragmatic** approach to use MIR in the current state
  - CRuby VM insn specialization instead of one in MIR itself
- Expectation register transfer language (RTL) with BB versioning can achieve YJIT performance for some benchmarks

Code specialization

- One Merriam-Webster definition of "specialization" – "design, train, or fit for one particular purpose"
- Code specialization is a common approach for faster code generation
- Specialized code already exists in CRuby,
  - VM insns for calling methods with particular name like `opt_plus`
- **Statically** and **dynamically** specialized code
- **Speculatively** specialized code and deoptimization
  - the more dynamic language is the more (speculative) specialization you need to be closer to static language performance

Dynamically specialized CRuby insns

- **Dynamic** specialization in a **lazy** way on VM insn **BB level**
  - usually a lot of versions of executed BB exists
- Specialization currently implemented:
  - **hybrid** stack-RTL insn specialization
  - type specialization based on **lazy basic block versioning**
    - Maxime Chevalier-Boisvert invention and the most important optimization technique of YJIT
  - different specialization based on **profile** info
    - type specialization based on profile info
    - specialized calls
    - specialized instance variable and attribute access
    - iterator specialization

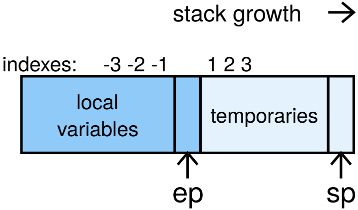
RTL

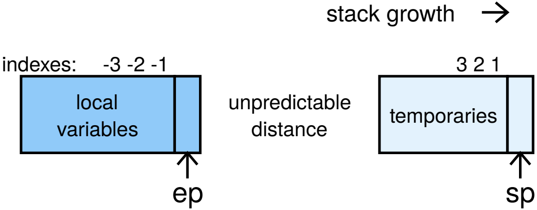
Stack insns	RTL insns

`getlocal v1   # push v1`
`getlocal v2   # push v2`
`opt_plus      # pop v1 and v2; push v1+v2`
`setlocal res # pop stack value and assign it to res`
	`sir_plusvvv res, v1, v2 # assign v1+v2 to res`

- RTL advantages
  - less insns, less insn dispatch code
  - less memory traffic
- RTL disadvantages
  - longer insn, more time in operand decoding
  - worse behaviour for working with values in stack ways (calls)

RTL (disjoint method frames)





- only ep can be used for addressing local variables (negative offset) and stack values (positive offset)

- ep should be used for addressing local variables and sp for stack values
  - a lot of ifs on offset values – a big performance impact on addressing

Hybrid stack-RTL insns

- Hybrid stack-RTL insns to overcome pure RTL disadvantages:

$$sir\_ \left\{ \begin{array}{c} plus \\ minus \\ mult \\ div \\ mod \\ or \\ and \\ eq \\ neq \\ lt \\ gt \\ le \\ ge \\ aref \\ aset \end{array} \right\} s \left\{ \begin{array}{c} s \\ v \\ i \end{array} \right\} \left\{ \begin{array}{c} s \\ v \\ i \end{array} \right\}$$

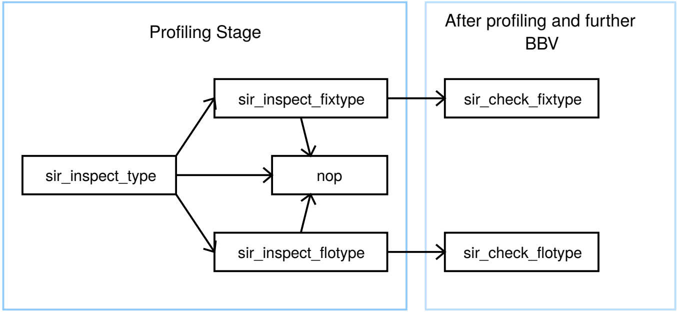
- `s` means value on stack
- `v` means value in a local variable
- `i` means immediate value
- insns `sir_aset..i` and insns with suffixes `sss`, `sii` are absent

Type specialized insns

$$
sir\_i \begin{Bmatrix} plus \\ minus \\ mult \\ div \\ mod \\ or \\ and \\ eq \\ neq \\ lt \\ gt \\ le \\ ge \\ aref \\ aset \end{Bmatrix} \begin{Bmatrix} s \\ v \end{Bmatrix} \begin{Bmatrix} s \\ v \\ i \end{Bmatrix} \begin{Bmatrix} s \\ v \\ i \end{Bmatrix}
\qquad
sir\_f \begin{Bmatrix} plus \\ minus \\ mult \\ div \\ mod \\ eq \\ neq \\ lt \\ gt \\ le \\ ge \end{Bmatrix} \begin{Bmatrix} s \\ v \end{Bmatrix} \begin{Bmatrix} s \\ v \\ i \end{Bmatrix} \begin{Bmatrix} s \\ v \\ i \end{Bmatrix}
$$

- Fixnum (prefix `sir_i`) and FP (prefix `sir_f`) type specialized insns:
  - Difference with non-type RTL insns: **result can be also a local variable**
  - Otherwise, strict correspondence between type-specialized insns and non-type ones is for safe deoptimization
- Many type specialized insns are generated by lazy BB versions
  - See numerous Maxime Maxime Chevalier-Boisvert presentations about BBV for details
- Rest of type specialized insns are generated from profile info

Profile-based specialization



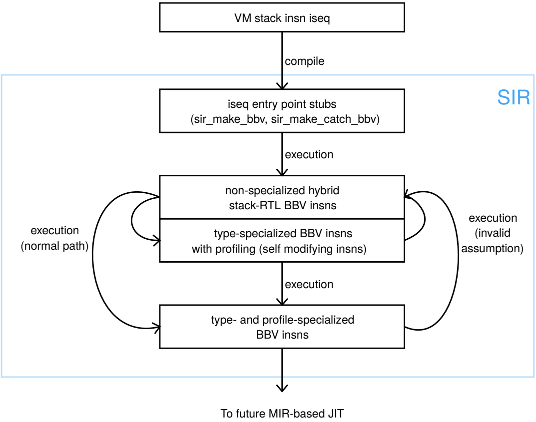
- Insns to inspect value types can not be deducted from BBV
  - After profiling, if inspect insns are transformed into type guards, further type specialization is done by BBV
- Specialized insns for calls and instance variable access also can be generated from profile info

Iterators

```
                                 sir_iter_start start_func
sir_cfunc_send   =>     Lcont: sir_iter_body Lexit, block_bbv, cond_func
                                 sir_iter_cont Lcont, arg_func
                        Lexit:
```

- A lot of Ruby standard methods are implemented on C, accept iseq blocks, and behave as iterators
- Calling the interpreter from C code is very expensive
- Such iterator method calls are changed by specialized insns avoiding the interpreter exits and enters
  - `sir_iter_start start_finc`, where start_func checks receiver type and setup block args
  - `sir_iter_body exit_label, block_bbv, cond_func`, where cond_func finishes iterator or calls block BBV
  - `sir_iter_cont cont_label, arg_func`, where arg_func updates block args and goto to given label

Dynamic flow of specialized insns



- Normal IR execution flow:
  - Start execution of BB with a stub
  - Stub execution generates hybrid stack-based RTL insns and type-specialized insns with profiling insns
  - Several executions of type-specialized insns results in type- and profile-specialized insns
  - Type- and profile-specialized insns are a source of the MIR-based JIT
- Exception IR execution flow:
  - Switching to non-type specialized stack-based RTL

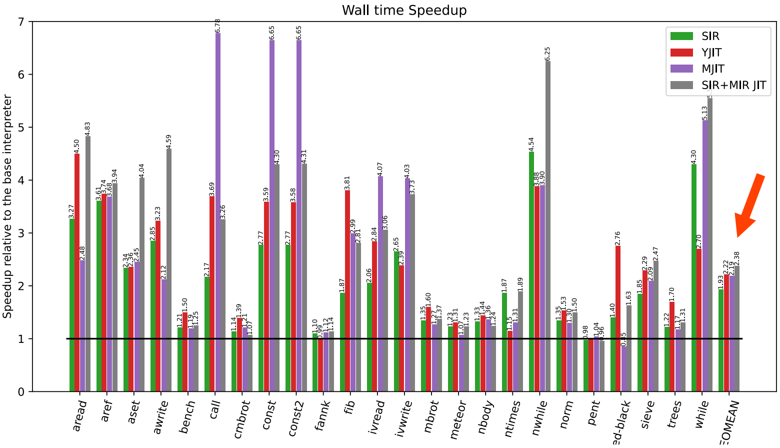
Implementation and the current state

- The code can be found in my github repository `https://github.com/vnmakarov/ruby`
- In the current state, SIR interpreter and MIR JIT are good only to run the micro-benchmarks
- Code for specialized IR generation and execution is about 3.5K lines of C
- Generator of C code for MIR is about 2.5K lines of C
- MIR-based JIT needs MIR library (from `bbv` branch) about 900KB of machine code
- Options to use the SIR interpreter: `--sir`, `--sir-debug`, `--sir-max-bb-versions=N`
- Option to use SIR interpreter and MIR JIT: `--mirjit`, `-mirjit-debug`

Benchmarking

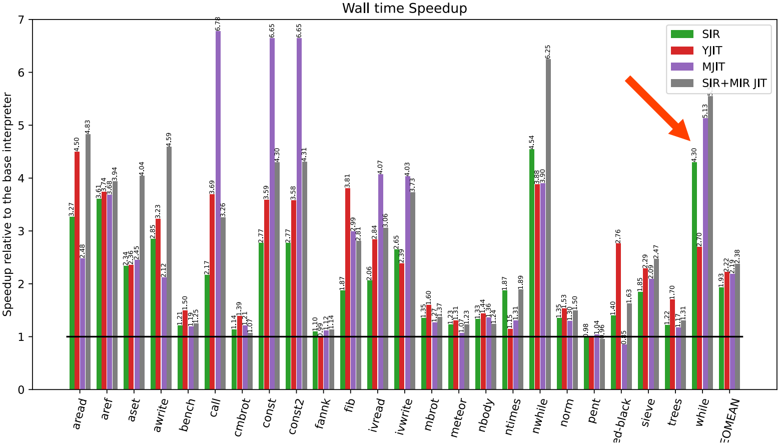
- Intel i7-9700K with 16GB memory under Linux FC 32
  - Base interpreter
  - Interpreter with SIR: –sir
  - YJIT: –yjit-call-threshold=1
  - MJIT: –jit-min-calls=1
  - SIR+MIR: –mirjit

Micro-benchmarks (Wall time)



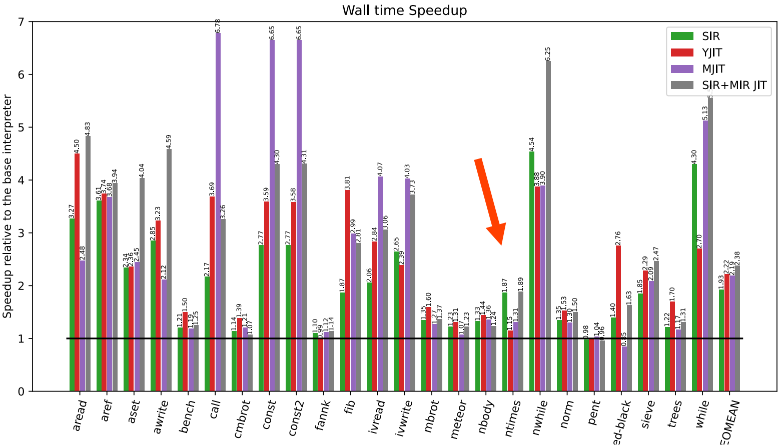
- 93% Geomean performance improvement for SIR

Micro-benchmarks (Wall time)



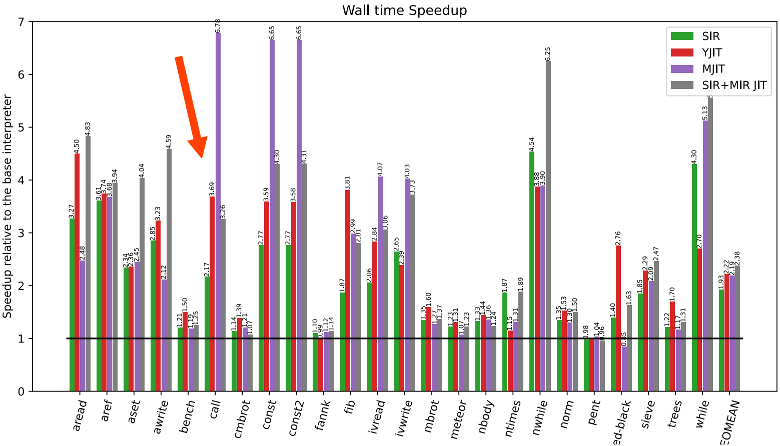
- SIR is faster YJIT on **while** benchmark because of RTL

Micro-benchmarks (Wall time)



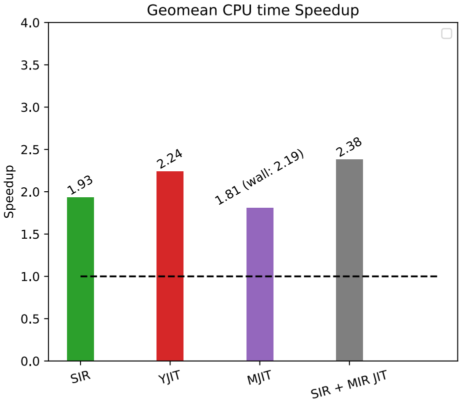
- SIR is faster YJIT on **nested times** benchmark because of iterator specialization

Micro-benchmarks (Wall time)



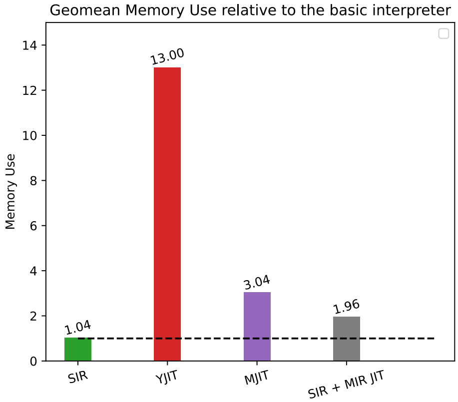
- YJIT is faster SIR on **call** benchmark (an example when YJIT specialization is better)

Micro-benchmarks (CPU time)



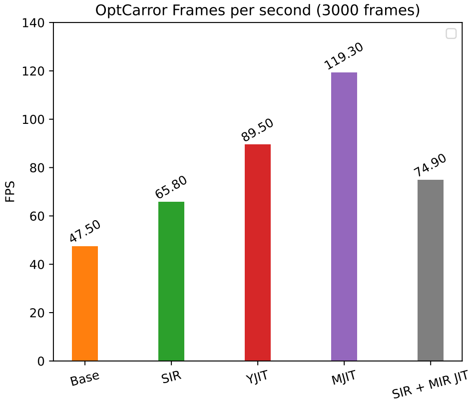
- CPU times are practically the same as wall times
- MJIT Exception: GCC run in parallel adds a lot of CPU time

Micro-benchmarks (Memory use)



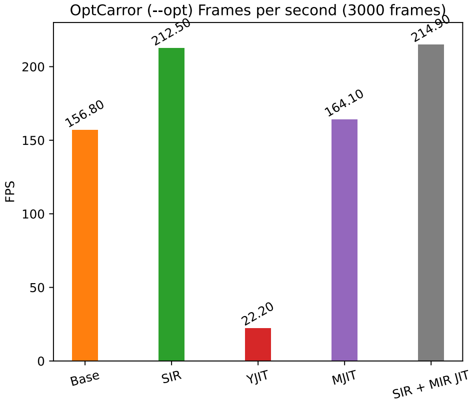
- Maximal resident memory size
- YJIT has the biggest memory consumption

OptCarrot



- The faster interpreter provides a modest 39% improvement

Optimized OptCarrot



- Huge method generation during execution (analog aggressive method inlining)
- YJIT behaviour is the worst

The future plans

- The faster interpreter is not ready yet
  - bug fixing and more optimization work
    - SIR is not API and there are no compatibility problems to change it
  - plans to finish it at the end of 2022
- MIR-based JIT is at the very early stage of the development
  - even more bug fixing and a lot of optimization work
  - plans for finish implementation in the next year
- Right now the faster interpreter and MIR-based JIT are more a research project
  - no commitment to submit it to CRuby
  - commitment only to support MIR project itself
- Adopting project ideas and/or code by Ruby developers is welcomed