Stories from YJIT Development

津市
Tsu City

Alan Wu

2022

Shopify

In this talk

- Past and present YJIT designs
- Problems we ran into
- Animations
- Tangents and other tangents
- No Rust related content. Sorry!
- Reading from my script 🔰

First design

very similar to the interpreter

Ruby code is translated to into small instructions first...

x = 2 + 2 →

```
putobject 2
putobject 2
opt_plus
setlocal :x
```

... then are run by instruction "handlers" in the executable

handlers jump to each other

```
add x26, x26, #0x10  ; advance Ruby program counter
str x26, [x21]       ; store new counter in memory
ldr x8, [x26]        ; load next handler address
br  x8               ; jump to the next handler
```

Ruby Code

x = 2 + 2

putobject 2
putobject 2
opt_plus
setlocal :x

Machine Code

address	code
10	; opt_plus
	...
	br
30	; putobject
	...
	br
40	; setlocal
	...
	br

①
②
③

Execution Log

machine program counter	machine instruction executed
30	; putobject
	...
	br
30	; putobject
	...
	br
10	; opt_plus
	...
	br
40	; setlocal
	...
	br

Ruby Code

x = 2 + 2

putobject 2
putobject 2
opt_plus
setlocal :x

Machine Code

address	code
10	; opt_plus
	...
	br
30	; putobject
	...
	br
40	; setlocal
	...
	br

①
②
③

Execution Log

machine program counter	machine instruction executed
30	; putobject
	...
	br
30	; putobject
	...
	br
10	; opt_plus
	...
	br
40	; setlocal
	...
	br

Two execution contexts

The first design is basically

```
output_code = ruby_instructions.map { _1.handler_code }
```

Output is similar to interpreter execution log

putobject 2
putobject 2
opt_plus
setlocal :x

; putobject
...
~~br~~
; putobject
...
~~br~~
; opt_plus
...
~~br~~
; setlocal
...
~~br~~

It was nice

Easy to explain

Easy to see how it could be faster

- Less jumping, less work!

```
putobject 2
putobject 2
opt_plus
setlocal :x
```

```
; putobject
...

; putobject
...

; opt_plus
...

; setlocal
...
```

slows down Rails :(

bench	speedup (%)
optcarrot	7.8
railsbench	-7.7

optcarrot **vs** railsbench

ファミコン
of course NES emulator ≠ web application

```
$ perf stat -I 1000 --topdown -a taskset -c 0 ruby ...
```

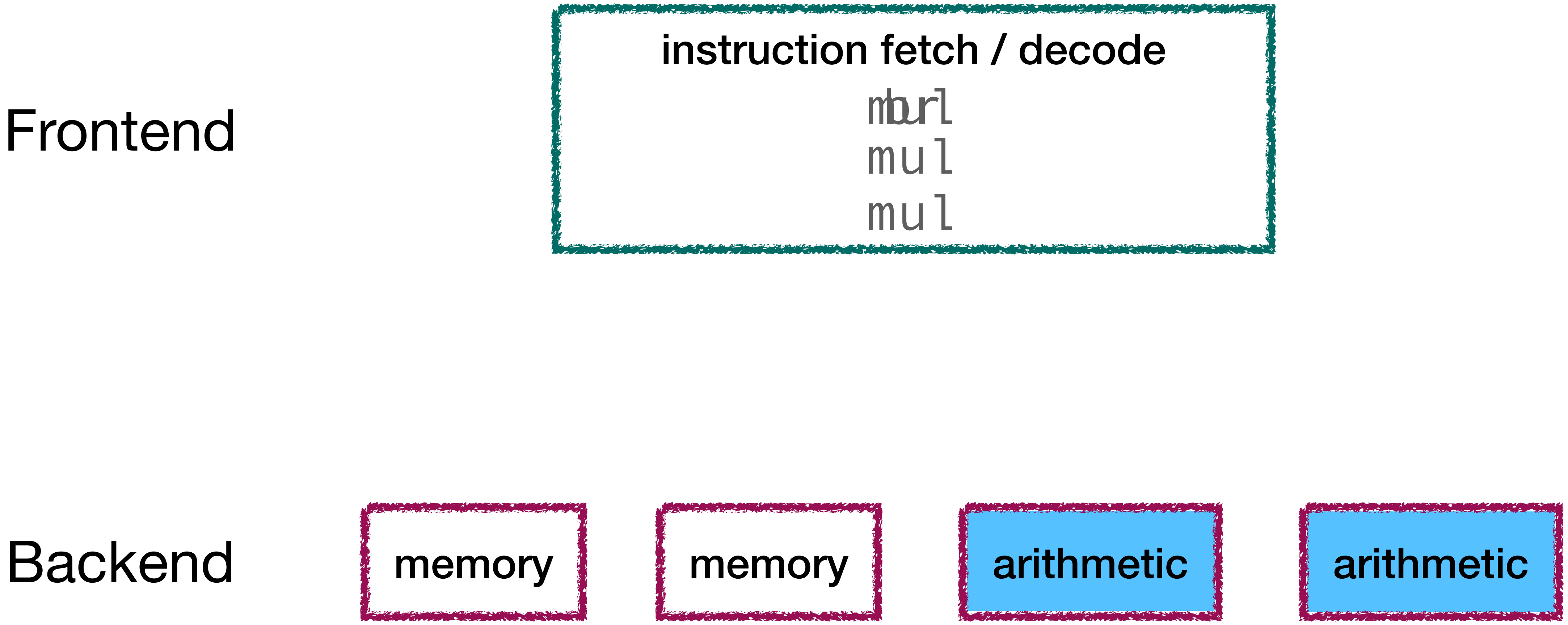
A Top-Down method for performance analysis and counters architecture

Publisher: IEEE Cite This PDF

Ahmad Yasin **All Authors**

Results without YJIT

	retiring	bad speculation	frontend bound	backend bound
optcarrot	60.3%	10.3%	14.3%	15.2%


Frontend
instruction fetch / decode
mul
mul
mul
Backend
memory
memory
arithmetic
arithmetic

For each opportunity to compute (slot)...

```
total_slots = total_cycles * max_ops_per_cycle
```

Frontend Bound: don't know what code to run

Backend Bound: don't have enough free execution units

Bad Speculation: ran wrong code

Retired: no problem

*** Maybe oversimplified. Sorry!**

Results without YJIT

	retiring	bad speculation	frontend bound	backend bound
optcarrot	60.3%	10.3%	14.3%	15.2%
railsbench				

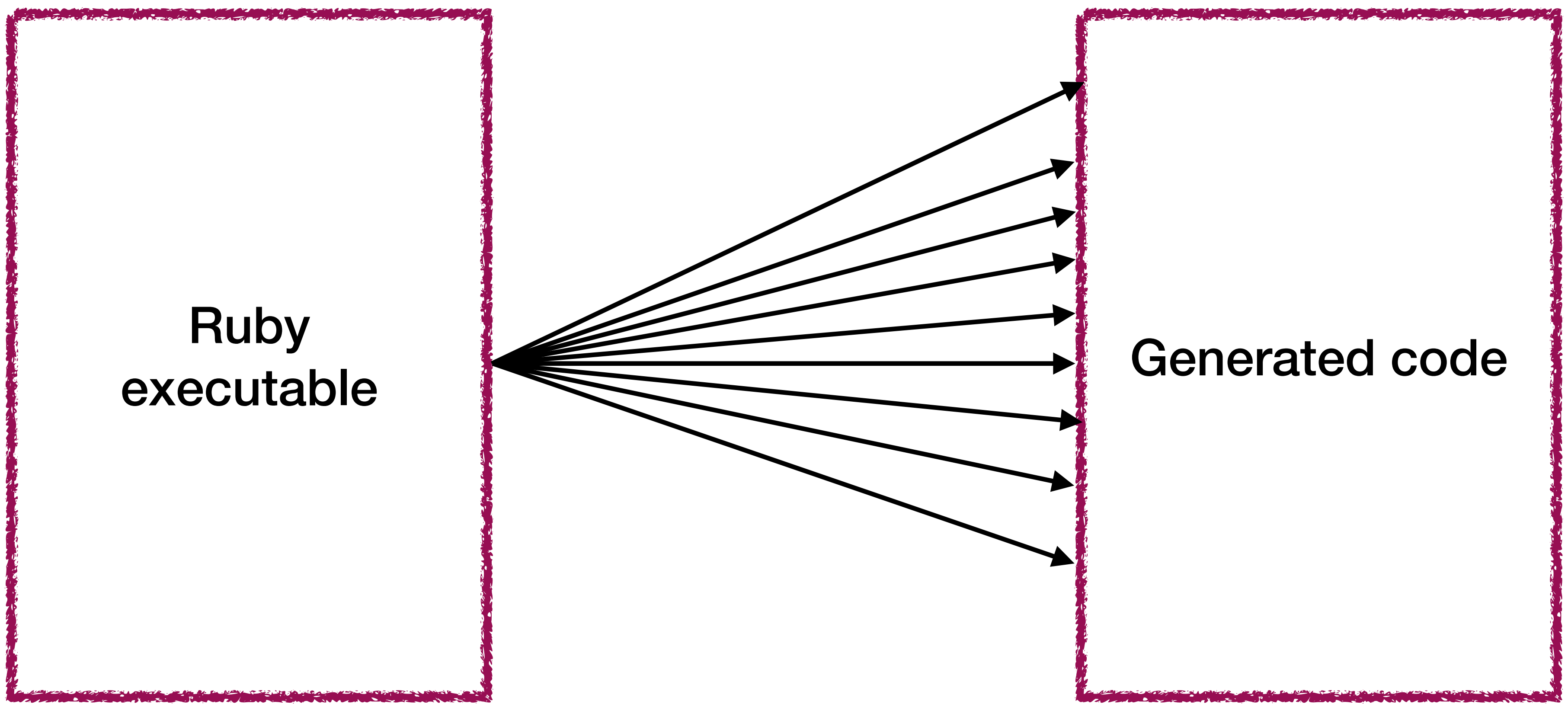
Why was it slower?

- Lots of frontend pressure already
- Jumping to generated code taxes the frontend even more
- Savings in the generated code not enough to offset the added work

```
# Samples: 4K of event 'br_misp_retired.all_branches_pebs:ppp'
# Event count (approx.): 1953243947
#
# Overhead  Command  Shared Object      Symbol
# ........  .......  ................   ...........................
#
    45.67%  ruby     ruby               [.] vm_exec_core
     2.72%  ruby     ruby               [.] ruby_sip_hash13
     2.52%  ruby     ruby               [.] vm_call_cfunc_with_frame
     2.01%  ruby     ruby               [.] obj_free
```

interpreter handlers

```
ldr   x8, [x26]     ; load next handler address
br    x8            ; jump to the next handler
```



- Branch history influences prediction (see Spectre & Meltdown)
- Generated code has many entry points -- hard to predict
- Initial YJIT design jumped from branches meant for which pollutes branch history

Want better output

Lazy Basic Block Versioning

Lazy Basic Block Versioning

Being lazy with "stubs"

- code that calls the compiler when run
- replaced with jump to the new code

```ruby
def call_itself(obj)
  obj.itself
end

custom = []

def custom.itself() = 2

call_itself(custom)
          # => 2
```

```
; getlocal :obj
; jump to(stub)
; jump unless obj
;         is custom
; setup custom.itself
;     return =
; putobject 2
; leave
; leave
```

stub
call to
itself

stub
call to
itself

stub
end of
call_itself

YJIT

```ruby
def call_itself(obj)
  obj.itself
end

custom = []

def custom.itself() = 2

call_itself(custom)

call_itself(custom)
```

```
; getlocal :obj
  jump ↓ (no-op)
; jump unless obj
;          is custom
; setup custom.itself
;       return =
; putobject 2
; leave
; leave
```

stub
call to
itself

YJIT

Interprocedural linearization as an emergent effect

```ruby
def call_itself(obj)
  obj.itself
end

custom = []

def custom.itself() = 2

call_itself(custom)

call_itself([3])
```

```
; getlocal :obj
  jump ↓ (no-op)
; jump unless obj
;        is custom
; setup custom.itself
;       return =
; putobject 2
; leave
; leave
; jump unless obj
;       is an Array
; setup Kernel#itself
```

stub
call to
itself

stub
call to
itself

YJIT

Speculating with the first call

- **Has relation to inline caching in the interpreter**
- **Each call site caches where the call goes to**
- **Monomorphic -- one target per site**

From Inline caching on WIKIPEDIA

Empirical measurements [3] show that in large Smalltalk programs about 1/3 of all send sites in active methods remain unlinked, and of the remaining 2/3, 90% are monomorphic, 9% polymorphic and 1% (0.9%) are megamorphic.

For CRuby, breakpoint counting with `bpftrace` in production showed hit rate is ~ 92%

```ruby
def call_itself(obj)
  obj.itself
end

custom = []

def custom.itself() = 2

call_itself(custom)

call_itself([3])
```

```
; getlocal :obj
  jump ↓ (no-op)
; jump unless obj
;        is custom
; setup custom.itself
;        return =
; putobject 2
; leave
; leave
; jump unless obj
;        is an Array
; setup Kernel#itself
```

stub
call to
itself

YJIT

General strategy

- Stay in generated code as much as possible
- For dynamic operations, use stubs for runtime data to guide speculation
- Use the interpreter for unimplemented operations and as a fallback

```
$ perf stat railsbench
```

	Instructions	Insn per cycle
Interpreter	104360039152	1.06
YJIT	87535747021	1.15

Fewer, easier to run instructions!

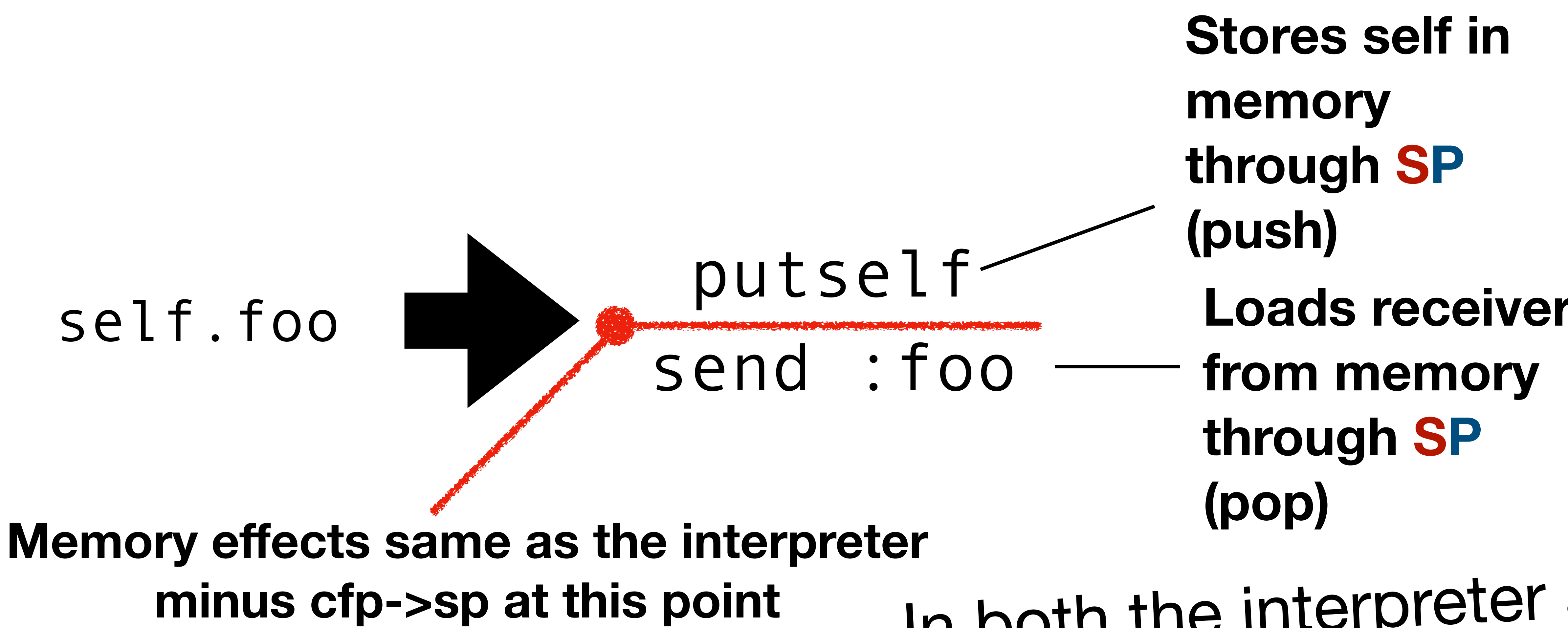
Reconstructing Interpreter State
("deoptimization")

1. Assign to cfp->pc
2. Assign to cfp->sp
3. That's it!

Lots of opportunities to optimize!

cfp->sp Stack Pointer

cfp: control frame pointer



In both the interpreter and YJIT output

Not to be confused with the call stack;
the "stack" here is for for transient intermediate values.
RubyVM/YARV is a "stack machine" (see Wikipedia).

```
# putself
ldur x11, [x19, #0x18]
stur x11, [x21]
# opt_send_without_block
ldur x11, [x21]
...
```

Not code one would write by hand 😅

Reconstructing Interpreter State

If we keep temporaries in registers

1. Assign to cfp->pc
2. Assign to cfp->sp
3. Flush stack temporaries to memory

Other concerns

- Make sure stubs can still access stack temporaries; they are read from memory at the moment
- Make sure when we point a jump to existing code that temporaries are in the right registers before the jump

cfp->pc Program Counter

YJIT	Updated on a need-to-know basis. e.g. before potential exceptions
Interpreter	Updated in each handler to know which instruction is next

```
add x26, x26, #0x10   ; advance Ruby program counter
str x26, [x21]        ; store new counter in memory
ldr x8, [x26]         ; load next handler address
br  x8                ; jump to the next handler
```

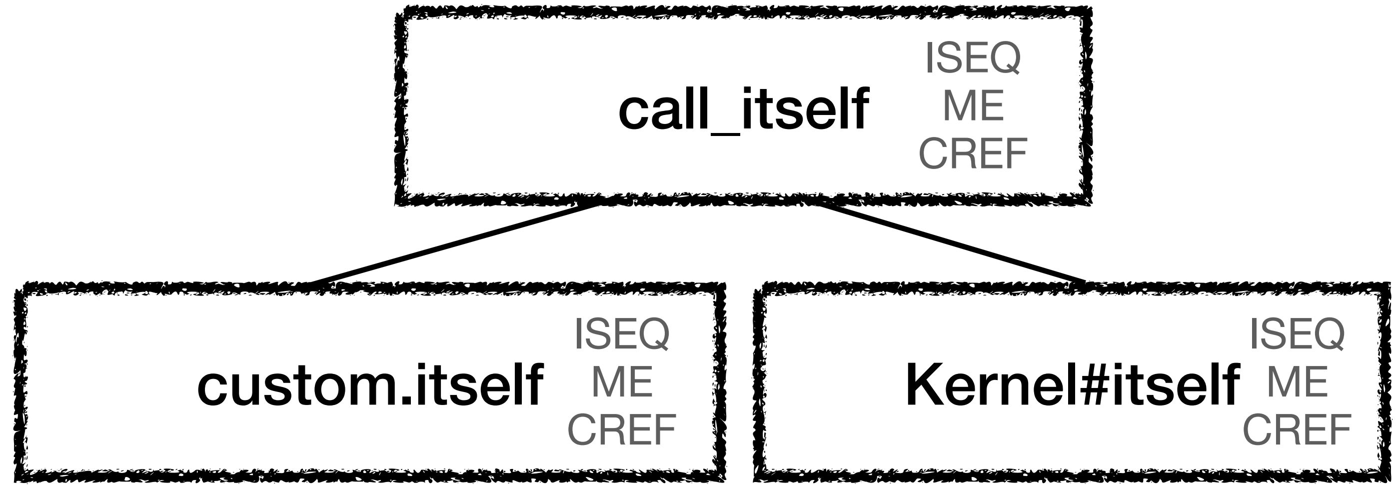
Other info that track execution progress

- ISEQ: houses VM instructions, exception table, other metadata
- ME: method entry object. Two methods can share the same ISEQ
- CREF: scope object for constant/refinement resolution
- ... and a few more!

YJIT metadata forms a call graph

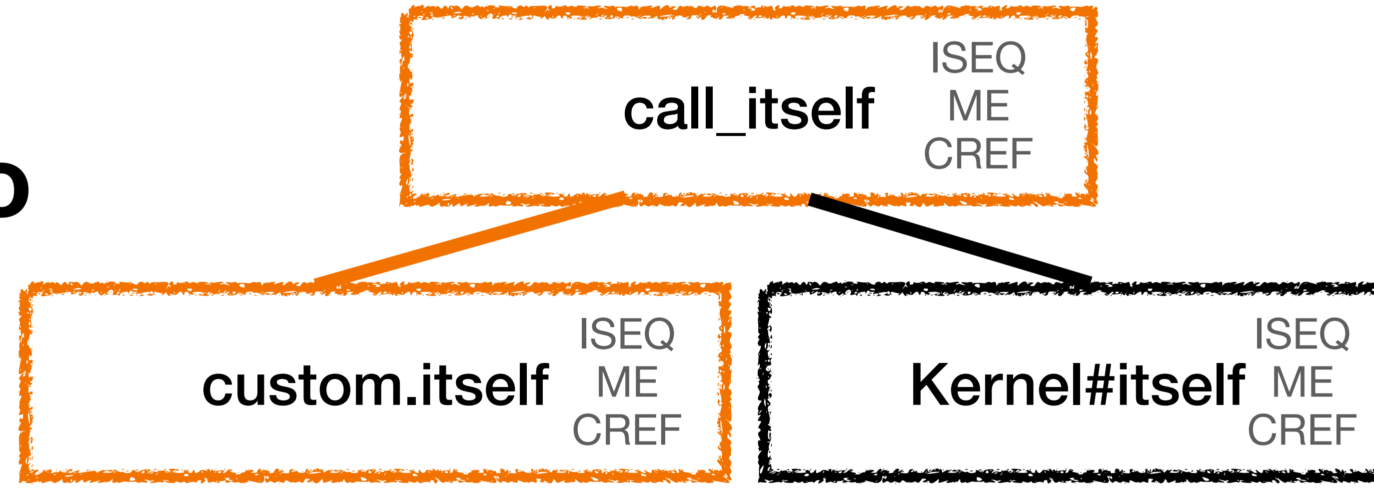
```
; getlocal :obj
  jump ↓ (no-op)
; jump unless obj
;        is custom
; setup custom.itself
;       return =
; putobject 2
; leave
; leave
; jump unless obj
;       is an Array
; setup Kernel#itself
```

stub
call to
itself



A weird inlining scheme

1. Speculate that we stay in the same call graph
2. On method call, point to a part of the metadata instead of pushing a frame and filling out ISEQ, CREF, ME, etc.



The metadata is like a bundle of stack frames!

3. Routines that want progress info can refer to the metadata (e.g. getting a backtrace)

Reconstructing Interpreter State

If we keep temporaries in registers

... and if we do weird inlining

1. Assign to cfp->pc
2. Assign to cfp->sp
3. 

Flush stack temporaries to memory
4. 

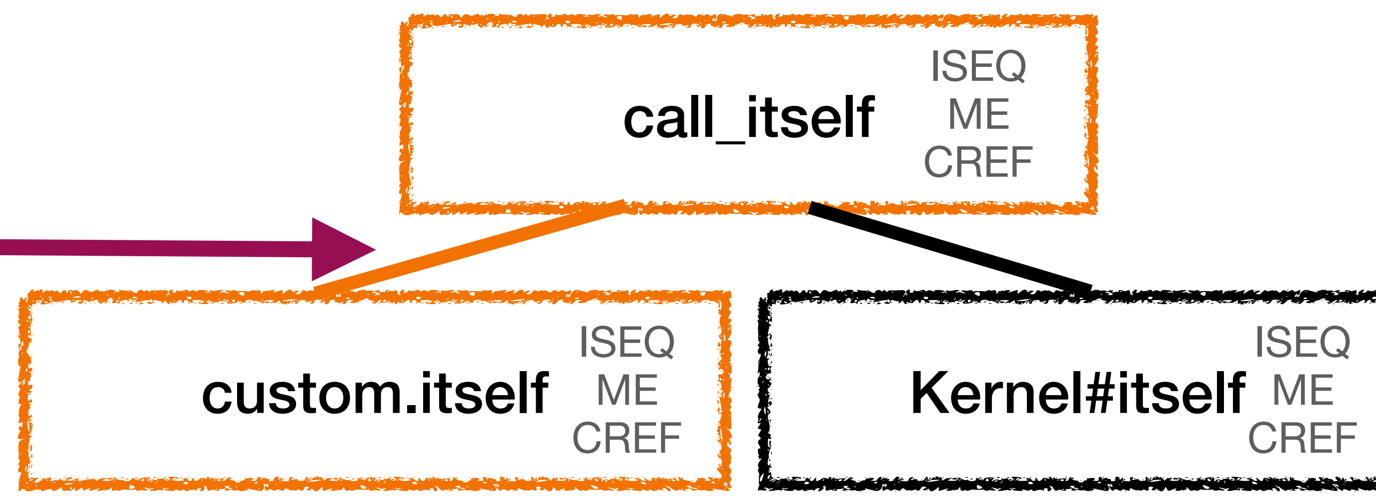
Unpack stack frames from the bundle

Turning work into metadata

```
# putself
ldur x11, [x19, #0x18]
stur x11, [x21]
# opt_send_without_block
ldur x11, [x21]
...
```

Which temporaries are in register?

Setting up stack frames



Challenges for JITs

- Speculative optimizations introduce slower-than-baseline deoptimization code paths
- Higher level transformations that give higher peak performance have higher risk of deoptimization
- Want to maintain a healthy level of minimum performance despite these risks
  - Our interpreter helps!

YJIT going forward

- **Provide a comprehensive performance solution for the entire lifetime of the process**
- **Quick boot, fair performance without compilation**
  - **Snappy** `bundle exec`**, low** `irb` **input latency**
- **Short and painless "warmup" period**
  - **Time spent compiling should pay for itself within seconds**
- **Good peak performance**
  - **Advanced transformations**
  - **Try hard to dodge slow deoptimizations**