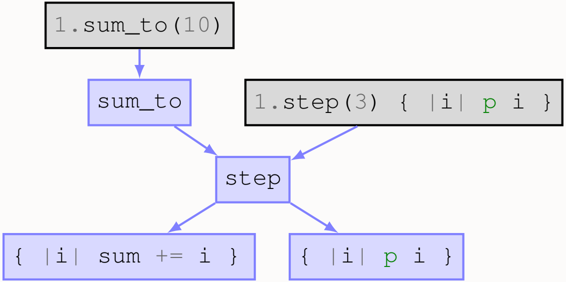
ORACLE

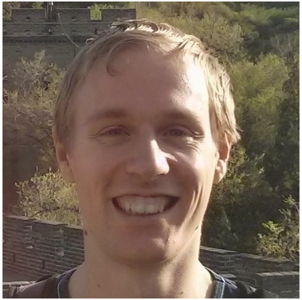
Splitting: The Crucial Optimization for Ruby Blocks



Benoit Daloze

RubyKaigi 2023

Who am I?



Benoit Daloze

Mastodon: @eregon@ruby.social
Twitter: @eregontp
GitHub: @eregon
Website: `https://eregon.me`

- TruffleRuby lead at Oracle Labs, Zurich
- Worked on TruffleRuby since 2014
- PhD on parallelism in dynamic languages
- Maintainer of ruby/spec
- CRuby (MRI) committer

TruffleRuby

- A high-performance Ruby implementation
- Uses the GraalVM™ JIT Compiler
- Targets full compatibility with CRuby 3.1, including C extensions
  e.g. Mastodon and Discourse can run on TruffleRuby
- GitHub: oracle/truffleruby, Twitter: @TruffleRuby, Mastodon: @truffleruby@ruby.social
  Website: `https://graalvm.org/ruby`

Splitting

SELF, the source of many dynamic language optimizations

- Similar to Smalltalk, but prototype-based, created in 1986

Many research breakthrough, used by dynamic languages nowadays:

- maps/Shapes to represent objects efficiently (used by TruffleRuby and CRuby since 3.2)
- Deoptimization: from JITed code to the interpreter and reoptimize
- Polymorphic Inline Caches (generalized as dispatch chains in Truffle)
- Splitting

The Customization / Splitting paper (July 1989)

Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language*

Craig Chambers
David Ungar
Stanford University

Abstract

Dynamically-typed object-oriented languages please programmers, but their lack of static type information penalizes performance. Our new implementation techniques extract static type information from declaration-free programs. Our system compiles several copies of a given procedure, each *customized* for one receiver type, so that the type of the receiver is bound at compile time. The compiler *predicts* types that are statically unknown but likely, and inserts run-time type tests to verify its predictions. It *splits* calls, compiling a copy on each control path, optimized to the specific types on that path. Coupling these new techniques with compile-time message lookup, aggressive procedure inlining, and traditional optimizations has doubled the performance of dynamically-typed object-oriented languages.

Some languages minimize message passing by including static procedure calls and built-in operators for non-object-oriented code. For example, C++ [Str86] includes C's repertoire of built-in operators and control structures, and some object-oriented Lisps [Moo86, Bob88] include normal Lisp functions, such as car and cdr which only work on cons-cells. While these impure object-oriented languages can avoid the cost of message passing with non-object-oriented constructs, the resulting programs are significantly restricted in flexibility and reusability.

To reduce the cost of message passing, some object-oriented languages, such as C++, Trellis/Owl [Sch86], and Eiffel [Mey86], include explicit type declarations. This allows the implementation to reduce the cost of a message send (or virtual function call) to no more than

Splitting Example in SELF

2.1. An Example

Let's look at a small piece of SELF code; we will come back later to this example to illustrate the compiler's optimizations and transformations. This example sums up the numbers from the receiver to some upper bound, and is defined in a parent object inherited by all numbers:*

```
sumTo: upperBound = (
        | sum <- 0 |
        to: upperBound Do: [
                | :index |
                sum: sum + index ].
        sum )
```

Splitting Example Translated to Ruby and Similarities

```ruby
class Numeric
  def sum_to(upper_bound)
    sum = 0
    self.step(upper_bound) do |i|
      sum += i
    end
    sum
  end
end
```

```
"Defined on Number"
sumTo: upperBound = (
    |sum <- 0|
    to: upperBound Do: [ |:index|
        sum: sum + index
    ].
    sum
)
```

Note we don't use `upto` because that's only available on Integer, and `step` is closer to the SELF example.

Example Call Sites for sum_to

```
1.sum_to(10) # => 55

1.0.sum_to(10.0) # => 55.0

1.5.sum_to(10.0) # => 49.5 (1.5 + 2.5 + ... + 9.5)

1r.sum_to(10r) # => (55/1)

(2**80).sum_to(2**81)
```

Compiling sum_to: can we inline step?

```ruby
class Numeric
  def sum_to(upper_bound)
    sum = 0
    # self is a Numeric, we would like to inline Numeric#step
    # but maybe some code added Integer#step or Float#step
    self.step(upper_bound) do |i|
      sum += i
    end
    sum
  end
end

1.sum_to(10)
1.0.sum_to(10.0)
```

Compiling sum_to: can we inline step?

```
class Numeric
  def sum_to(upper_bound)
    sum = 0
    # Inline cache with all seen receiver types/classes
    # [Integer => Numeric#step, Float => Numeric#step]
    self.step(upper_bound) do |i|
      sum += i
    end
    sum
  end
end

1.sum_to(10)
1.0.sum_to(10.0)
```

Compiling sum_to: can we inline step?

```
class Numeric
  def sum_to(upper_bound)
    sum = 0
    # 2 levels of inline cache: lookup cache and call target cache
    # lookup cache: [Integer => Numeric#step, Float => Numeric#step]
    # call target cache: [Numeric#step]
    self.step(upper_bound) do |i|
      sum += i
    end
    sum
  end
end

1.sum_to(10)
1.0.sum_to(10.0)
```

Numeric#step, simplified (no keyword arguments, etc)

```ruby
def step(limit = nil, step = 1, &block)
  return create_step_enumerator(limit, step) unless block_given?
  raise TypeError, 'step must be numeric' if Primitive.nil? step
  raise ArgumentError, "step can't be 0" if step == 0

  value = self
  descending = step < 0
  limit ||= descending ? -Float::INFINITY : Float::INFINITY
  if value.is_a?(Float) or limit.is_a?(Float) or step.is_a?(Float)
    step_float(self, limit, step, descending, &block)
  else
    if descending
      until value < limit
        yield value
        value += step
      end
    else
      until value > limit
        yield value
        value += step
      end
    end
  end
  self
```

Example Call Sites for Numeric#step

```
1.step(3) { |i| p i } # 1, 2, 3
1.0.step(3.0) { |i| p i } # 1.0, 2.0, 3.0

1.step(7, 2) { |i| p i } # 1, 3, 5, 7
7.step(1, -2) { |i| p i } # 7, 5, 3, 1

1.step(to: 7, by: 2) { ... } # keyword arguments

1.step(by: 2) { ... } # no upper limit

1.step(5) # => an Enumerator
```


Numeric#step, without Enumerator and early step checks

```
def step(limit = nil, step = 1, &block)
  return create_step_enumerator(limit, step) unless block_given?
  raise TypeError, 'step must be numeric' if Primitive.nil? step
  raise ArgumentError, "step can't be 0" if step == 0

  value = self
  descending = step < 0
  limit ||= descending ? -Float::INFINITY : Float::INFINITY
  if value.is_a?(Float) or limit.is_a?(Float) or step.is_a?(Float)
    step_float(self, limit, step, descending, &block)
  else
    if descending
      until value < limit
        yield value
        value += step
      end
    else
      until value > limit
        yield value
        value += step
      end
    end
  end
  self
```

Numeric#step, with descending logic in another method

```
def step(limit = nil, step = 1, &block)
  value = self
  descending = step < 0
  limit ||= descending ? -Float::INFINITY : Float::INFINITY
  return step_float(...) if value.is_a?(Float) or limit.is_a?(Float) or step.is_a?(Float)

  if descending
    until value < limit
      yield value
      value += step
    end
  else
    until value > limit
      yield value
      value += step
    end
  end

  self
end
```

Numeric#step, with descending logic in another method

```
def step(limit = nil, step = 1, &block)
  value = self
  descending = step < 0
  limit ||= descending ? -Float::INFINITY : Float::INFINITY
  return step_float(...) if [value, limit, step].any?(Float)
  return step_descending(...) if descending

  until value > limit
    yield value
    value += step
  end

  self
end
```

Compiling step: the main loop

```ruby
def step(limit = nil, step = 1, &block)
  # ...
  until value > limit
    # inline cache: [block in sum_to, block in main]
    yield value
    value += step
  end
  self
end

1.sum_to(10)
1.step(3) { |i| p i }
```


Compiling step: inline both blocks?

```
def step(limit = nil, step = 1, &block)
  # ...
  until value > limit
    if block is "block in sum_to" # { |i| sum += i }
      block.outer_variables[:sum] += value
    elsif block is "block in main" # { |i| p i }
      p value
    else
      deopt
    end
    value += step
  end
  self
end
```

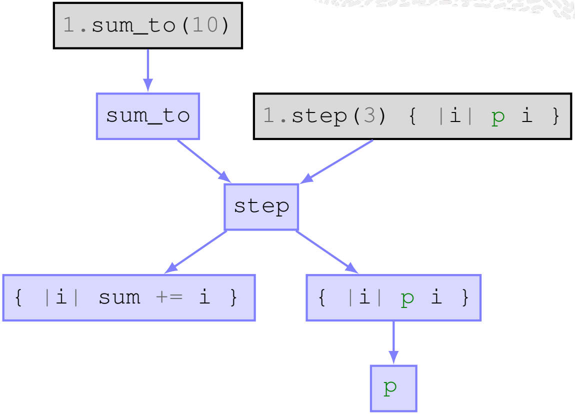

Compiling step: inline N blocks?

```
def step(limit = nil, step = 1, &block)
  # ...
  until value > limit
    if block is "block in sum_to" # { |i| sum += i }
      block.outer_variables[:sum] += value
    elsif block is "block in main" # { |i| p i }
      p value
    elsif block is "block 3"
      # ...
    elsif block is "block 4"
      # ...
    elsif block is "block 5"
      # ...
    elsif block is "block 6"
      # ...
    elsif block is "block 7"
```

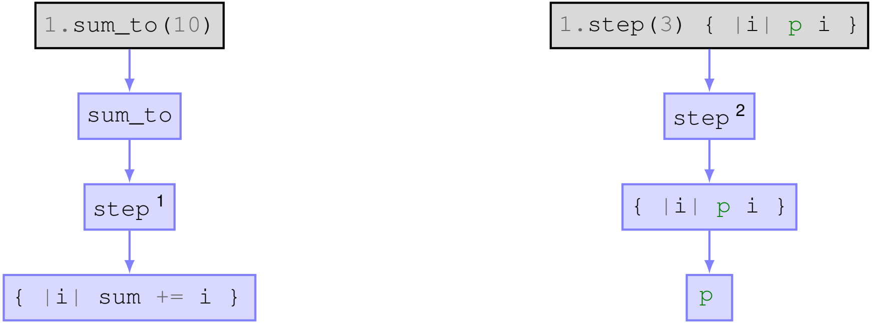
Solution: compile multiple copies of step

```ruby
def step1(limit = nil, step = 1, &block) # copy for block in sum_to
  # ...
  until value > limit
    deopt unless block is "block in sum_to" # { |i| sum += i }
    block.outer_variables[:sum] += value
    value += step
  end
end

def step2(limit = nil, step = 1, &block) # copy for block in main
  # ...
  until value > limit
    deopt unless block is "block in main" # { |i| p i }
    p value
    value += step
  end
```

Splitting



Splitting



Splitting

- What we just did is called *splitting*
- We split the method `step` so there is a copy of *step* for each caller
- Those copies or *splits* can then be optimized further by having more information from the caller through inline caches and profiling information

Splitting in TruffleRuby and Truffle: a more generic approach

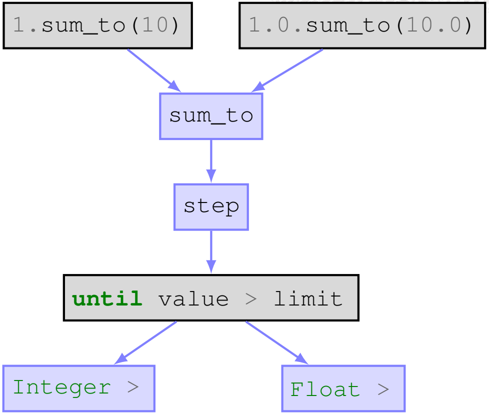
An inline cache or call site can be:

- Monomorphic: single entry, for a call site it always calls the same method
- Polymorphic: 2+ entries (in TruffleRuby currently up to 8)
- Megamorphic: too many entries to cache

Everytime TruffleRuby detects polymorphism or megamorphism, it uses splitting to try to make it monomorphic again.

- In TruffleRuby, once we decided to split we will split *for each call site*
- More than that, if we still see polymorphism we might decide to split callers (e.g., sum_to)

Recursive Splitting



Recursive Splitting

Compiling Integer#sum_to(Integer) (split)

```ruby
# arguments profile: upper_bound is always seen as Integer
def sum_to(upper_bound)
  sum = 0
  # [Integer => Numeric#step], let's inline
  self.step(upper_bound) do |i|
    sum += i
  end
  sum
end

1.sum_to(10)
```


Compiling Numeric#step split for Integer#sum_to(Integer)

```ruby
# arguments profile: limit is Integer, step is not passed
def step(limit = nil, step = 1, &block)
  value = self
  descending = step < 0 # step is not passed, so step is 1
  limit ||= descending ? -Float::INFINITY : Float::INFINITY
  return step_float(...) if [value, limit, step].any?(Float)
  return step_descending(...) if descending

  until value > limit
    yield value
    value += step
  end

  self
end
```

step is always 1, fold 1 < 0

```
# arguments profile: limit is Integer, step is not passed
def step(limit = nil, step = 1, &block)
  value = self
  descending = 1 < 0 # step is not passed, so step is 1
  limit ||= descending ? -Float::INFINITY : Float::INFINITY
  return step_float(...) if [value, limit, 1].any?(Float)
  return step_descending(...) if descending

  until value > limit
    yield value
    value += 1
  end

  self
end
```

Propagate descending=false

```ruby
# arguments profile: limit is Integer, step is not passed
def step(limit = nil, step = 1, &block)
  value = self
  descending = false
  limit ||= descending ? -Float::INFINITY : Float::INFINITY
  return step_float(...) if [value, limit, 1].any?(Float)
  return step_descending(...) if descending

  until value > limit
    yield value
    value += 1
  end

  self
end
```


limit is Integer

```ruby
# arguments profile: limit is Integer, step is not passed
def step(limit = nil, step = 1, &block)
  value = self
  limit ||= Float::INFINITY
  return step_float(...) if [value, limit, 1].any?(Float)

  until value > limit
    yield value
    value += 1
  end

  self
end
```


self is Integer

```ruby
# arguments profile: self is Integer, limit is Integer, step not passed
def step(limit = nil, step = 1, &block)
  value = self # Integer
  return step_float(...) if [value, limit, 1].any?(Float)

  until value > limit # Integer#>
    yield value
    value += 1 # Integer#+
  end

  self
end
```


Expand Float checks

```
# arguments profile: self is Integer, limit is Integer, step not passed
def step(limit = nil, step = 1, &block)
  value = self # Integer
  return step_float(...) if [value, limit, 1].any?(Float)

  until value > limit # Integer#>
    yield value
    value += 1 # Integer#+
  end

  self
end
```


Fold .is_a?(Float) checks

```
# arguments profile: self is Integer, limit is Integer, step not passed
def step(limit = nil, step = 1, &block)
  value = self # Integer
  if value.is_a?(Float) or limit.is_a?(Float) or 1.is_a?(Float)
    return step_float(...)
  end

  until value > limit # Integer#>
    yield value
    value += 1 # Integer#+
  end

  self
end
```

Compiled Numeric#step split for Integer#sum_to(Integer)

```
# arguments profile: self is Integer, limit is Integer, step not passed
def step(limit = nil, step = 1, &block)
  value = self

  until value > limit # Integer#>
    yield value
    value += 1 # Integer#+
  end

  self
end
```


Let’s inline step in sum_to

```
def sum_to(upper_bound)
  sum = 0
  self.step(upper_bound) do |i|
    sum += i
  end
  sum
end
def step(limit = nil, step = 1, &block)
  value = self
  until value > limit # Integer#>
    yield value
    value += 1 # Integer#+
  end
  self
end
```

Let’s inline step in sum_to

```ruby
def sum_to(upper_bound)
  sum = 0
  value = self
  until value > upper_bound # Integer#>
    proc { |i| sum += i }.call(value)
    value += 1 # Integer#+
  end
  sum
end
```

Let’s inline the block

```ruby
def sum_to(upper_bound)
  sum = 0
  value = self
  until value > upper_bound # Integer#>
    sum += value # Integer#+
    value += 1 # Integer#+
  end
  sum
end
```

Final result

`sum_to` was compiled as efficiently as this C code:

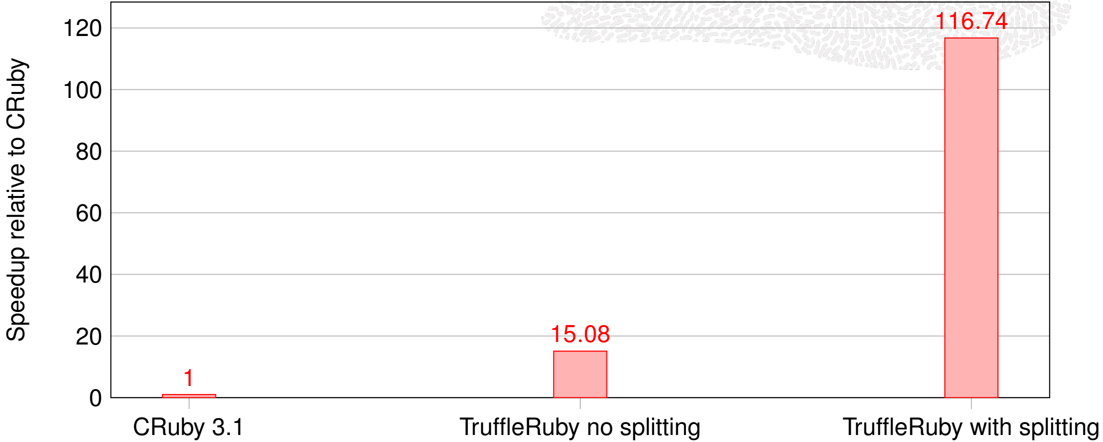
```
int sum_to(int self, int upper_bound) {
  int sum = 0;
  int value = self;
  while (value <= upper_bound) {
    sum += value; // + overflow check (CPU flag check like jo)
    value++;      // + overflow check (CPU flag check like jo)
  }
  return sum;
}
```

but it works for Float, Rational, Bignums and has no overflow!

Benchmark sum_to

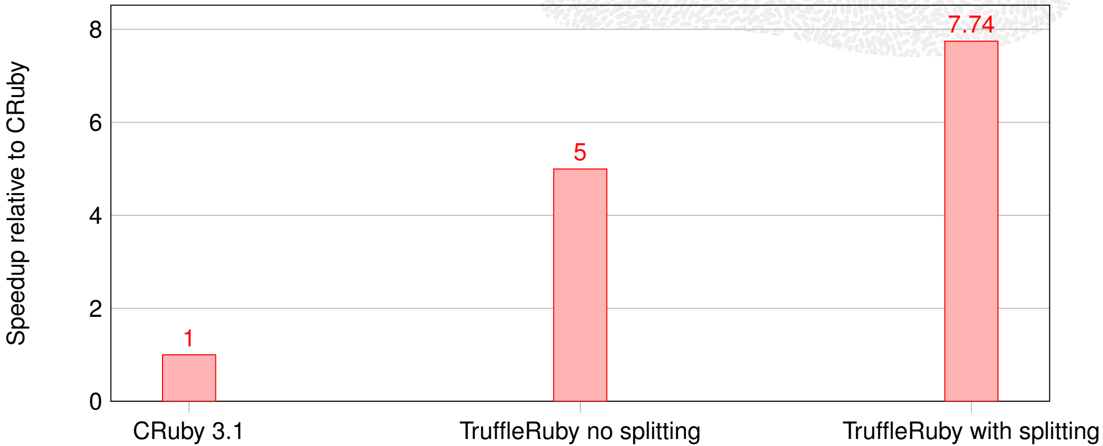
```ruby
1.sum_to(10)
1.0.sum_to(10.0)
1.5.sum_to(10.0)
1r.sum_to(10r)
1.step(7, 2) { |i| p i }
1.step(to: 7, by: 2) { }
1.step(5)
p 1.sum_to(1000)

benchmark do
  1.sum_to(1000)
end
```

Benchmark results for sum_to

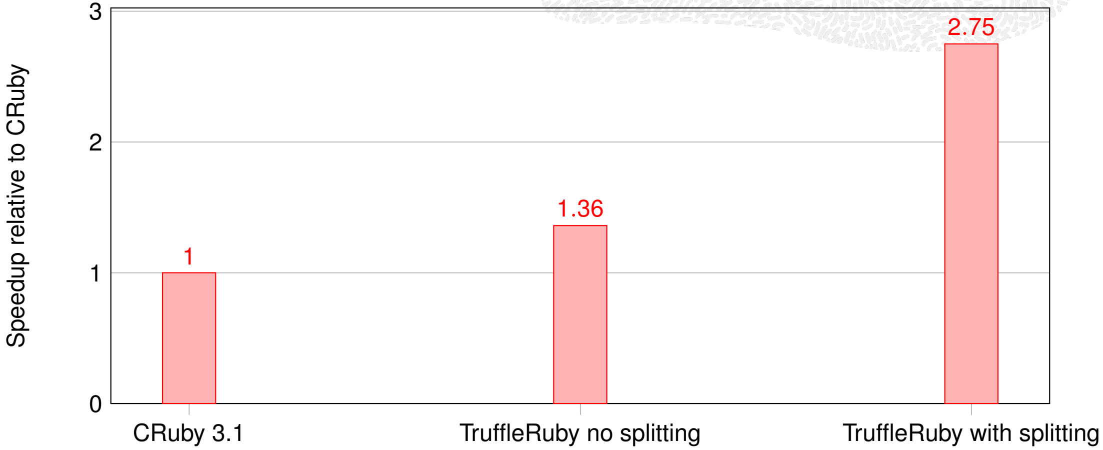


TruffleRuby JIT makes sum_to 15x faster, and splitting makes sum_to 7.7x faster on top of that!

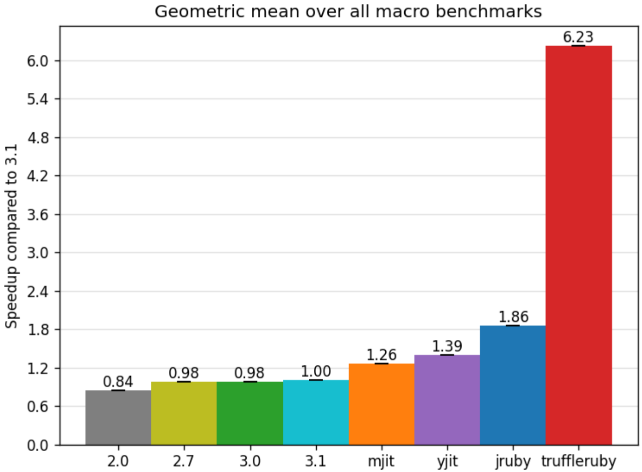
Benchmark results for OptCarrot



Benchmark results for RailsBench (from the yjit-bench suite)



TruffleRuby: Peak performance on yjit-bench (14 benchmarks)



From `https://eregon.me/blog/2022/01/06/benchmarking-cruby-mjit-yjit-jruby-truffleruby.html`

Analyzing Ruby Call-Site Behavior paper

Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications

Sophie Kaleba
S.Kaleba@kent.ac.uk
University of Kent
United Kingdom

Octave Larose
O.Larose@kent.ac.uk
University of Kent
United Kingdom

Richard Jones
R.E.Jones@kent.ac.uk
University of Kent
United Kingdom

Stefan Marr
s.marr@kent.ac.uk
University of Kent
United Kingdom

Abstract

Applications written in dynamic languages are becoming larger and larger and companies increasingly use multi-million line codebases in production. At the same time, dynamic languages rely heavily on dynamic optimizations, particularly those that reduce the overhead of method calls.

In this work, we study the call-site behavior of Ruby benchmarks that are being used to guide the development of upcoming Ruby implementations such as TruffleRuby and YJIT.

ACM Reference Format:
Sophie Kaleba, Octave Larose, Richard Jones, and Stefan Marr. 2022. Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications. In *Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '22), December 07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 16 pages.
https://doi.org/10.1145/3563834.3567538

1 Introduction

Analyzing Ruby Call-Site Behavior paper

- Research by Sophie Kaleba, Octave Larose, Stefan Marr and Prof. Richard Jones
- The paper uses TruffleRuby to analyze the behavior of call sites on various Ruby benchmarks
- They find that TruffleRuby has two main ways to reduce polymorphism and megamorphism:
  - 2-level inline cache for method calls (lookup cache and call target cache)
  - Splitting
- There is also a blog post at `https://stefan-marr.de/`

Analyzing Calls in RailsBench

	Polymorphic Calls	Megamorphic Calls
Initial	956,515 (6.9%)	63,319 (0.457%)
After 2-level inline cache	490,072 (3.5%)	557 (0.004%)
After Splitting	0%	0%

The 2-level inline cache for method calls and Splitting . . .
completely remove polymorphism and megamorphism in all 44 benchmarks used in the paper!

Conclusion

- *Splitting* is a technique from the SELF VM research, invented in 1989 (34 years ago)
- It applies well to Ruby, for methods taking blocks and also for other forms of polymorphism
- It completely removes polymorphism and megamorphism on all 44 benchmarks (Kaleba et al.)
- Splitting gives speedups of 7.7x on sum_to, 1.5x on OptCarrot and 2x on RailsBench

Cool Things About TruffleRuby and GraalVM

- Interoperability with Java, Python, JS and other GraalVM languages:
  `Polyglot.eval('python', 'import matplotlib')`
- Regexp JIT Compiler and how to avoid ReDoS (RubyKaigi 2021 presentation)
- Parallel execution of Ruby code and soon of `RB_EXT_RACTOR_SAFE`-marked C extensions
- Tooling accross multiple languages (LSP, DAP, backtraces mixing C and Ruby, etc)
- Most advanced Ruby JIT Compiler: Inlining Ruby/C/Java/etc, Splitting, Partial Evaluation, GraalVM Compiler optimizations like Partial Escape Analysis, etc
- Multiple GCs to choose from with various throughput and latency trade-offs (ParallelGC, G1, ZGC)

Trying TruffleRuby

Latest release: 23.0.0-preview1
23.0.0 planned for June 13 (in one month)

Use your favorite Ruby manager/installer:

```
$ ruby-install truffleruby

$ ruby-build truffleruby-23.0.0-preview1
$ ruby-build truffleruby+graalvm-23.0.0-preview1
(or rbenv install instead of ruby-build)

$ rvm install truffleruby
```

See `https://github.com/oracle/truffleruby` for more details

Any question?

Polymorphic and Megamorphic Calls

Benchmark	Stmts	Stmts Cov.	Fns	Fns Cov.	kCalls	Poly+ Mega. calls
BlogRails	118,717	48%	37,595	38%	13,863	7.4%
ChunkyCanvas*	19,279	32%	5,082	20%	11,323	0.0%
ChunkyColor*	19,266	32%	5,077	20%	19	2.0%
ChunkyDec	19,289	32%	5,083	20%	21	2.0%
ERubiRails	117,922	45%	37,328	35%	12,309	5.4%
HexaPdfSmall	26,624	44%	6,990	35%	31,246	7.4%
LiquidCartParse	23,531	37%	6,259	27%	87	1.3%
LiquidCartRender	23,562	39%	6,269	30%	236	5.5%
LiquidMiddleware	22,374	37%	5,939	27%	70	1.4%
LiquidParseAll	23,276	37%	6,186	27%	295	1.9%
LiquidRenderBibs	23,277	39%	6,185	29%	385	23.4%
MailBench	31,857	40%	8,392	32%	2,756	3.4%
PsdColor	27,498	40%	7,724	28%	352	4.1%
PsdCompose*	27,498	40%	7,724	28%	352	4.0%
PsdImage*	27,531	40%	7,736	28%	5,509	0.0%
PsdUtil*	27,496	40%	7,724	28%	351	4.0%
Sinatra	31,187	40%	8,492	29%	172	6.9%

The Effect of 2-level Inline Cache for Method Calls

	Number of calls		After eliminating target duplicates	
Benchmark	Poly.	Mega.	Poly.	Mega.
BlogRails	956,515	63,319	-48.8%	-99.1%
ChunkyCanvas*	322	98	-80.0%	-100.0%
ChunkyColor*	320	98	-79.0%	-100.0%
ChunkyDec	322	98	-79.5%	-100.0%
ERubiRails	626,535	40,699	-37.4%	-98.6%
HexaPdfSmall	1,842,665	479,399	-21.7%	-99.6%
LiquidCartParse	821	280	-73.3%	-100.0%
LiquidCartRender	12,598	280	-84.1%	-100.0%
LiquidMiddleware	747	251	-68.8%	-100.0%
LiquidParseAll	5,369	280	-87.4%	-100.0%
LiquidRenderBibs	89,866	280	-73.7%	-100.0%
MailBench	81,886	12,697	-77.6%	-100.0%
PsdColor	14,053	233	-53.1%	-100.0%
PsdCompose*	14,053	233	-53.0%	-100.0%
PsdImage*	14,062	233	-53.0%	-100.0%
PsdUtil*	14,048	233	-53.0%	-100.0%
Sinatra	7,909	3,911	-82.8%	-94.4%

The Effect of Splitting

	Number of calls		After splitting	
Benchmark	**Poly.**	**Mega.**	**Poly.**	**Mega.**
BlogRails	490,072	557	-100%	-100%
ChunkyCanvas*	66	0	-100%	0%
ChunkyColor*	66	0	-100%	0%
ChunkyDec	66	0	-100%	0%
ERubiRails	391,997	553	-100%	-100%
HexaPdfSmall	1,443,211	2,066	-100%	-100%
LiquidCartParse	219	0	-100%	0%
LiquidCartRender	2,000	0	-100%	0%
LiquidMiddleware	233	0	-100%	0%
LiquidParseAll	679	0	-100%	0%
LiquidRenderBibs	23,633	0	-100%	0%
MailBench	18,322	0	-100%	0%
PsdColor	6,586	0	-100%	0%
PsdCompose*	6,586	0	-100%	0%
PsdImage*	6,588	0	-100%	0%
PsdUtil*	6,584	0	-100%	0%
Sinatra	1,362	220	-100%	-100%