Splitting:
The Crucial Optimization for Ruby Blocks

Benoit Daloze
RubyKaigi 2023

```ruby
1.sum_to(10)

sum_to

1.step(3) { |i| p i }

step

{ |i| sum += i }

{ |i| p i }
```
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
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 known 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:

```self
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"

```ruby
sumTo: upperBound = ( |
  sum <- 0|
  to: upperBound Do: [ |
    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

1.0.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?

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?

```ruby
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
```

```ruby
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)
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
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

```ruby
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
```
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

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
    end
    value += step
end
self
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"
            # ...
        else
            deopt
        end
    value += step
end
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

1. `sum_to(10)`

   `sum_to`

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

   `step`

   2. `{ |i| sum += i }`

   3. `{ |i| p i }`

   `p`
Splitting

1. `sum_to(10)`

   `sum_to`

   `step¹`

   `{ |i| sum += i }`

2. `step(3) { |i| p i }`

   `step²`

   `{ |i| p i }`

   `p`
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

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

- `sum_to`
- `step`
- `until value > limit`
- `Integer >`
- `Float >`
Recursive Splitting

1. sum_to(10)

\[ \text{sum_to}^1 \]

\[ \text{step}^1 \]

\[ \text{Integer} > \]

1.0 sum_to(10.0)

\[ \text{sum_to}^2 \]

\[ \text{step}^2 \]

\[ \text{Float} > \]
Compiling Integer#sum_to(Integer) (split)

# 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)

# 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
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
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

# arguments profile: limit is Integer, step is not passed

```ruby
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
```
def step(limit = nil, step = 1, &block)
    value = self
    return step_float(...) if [value, limit, 1].any?(Float)
    until value > limit
        yield value
        value += 1
    end
    self
end
Expand Float checks

```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
```
Fold .is_a?(Float) checks

```ruby
# 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)

```ruby
# 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
```

Copyright © 2022, Oracle and/or its affiliates
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#
        proc { |i| sum += i }.call(value)
        value += 1
    end
    sum
end
```
Let’s inline the block

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

```c
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

1.0.sum_to(10.0)
1.5.sum_to(10.0)
lr.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

<table>
<thead>
<tr>
<th></th>
<th>CRuby 3.1</th>
<th>TruffleRuby no splitting</th>
<th>TruffleRuby with splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup relative to CRuby</td>
<td>1</td>
<td>5</td>
<td>7.74</td>
</tr>
</tbody>
</table>

Copyright © 2022, Oracle and/or its affiliates
Benchmark results for RailsBench (from the yjit-bench suite)

Speedup relative to CRuby

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRuby 3.1</td>
<td>1</td>
</tr>
<tr>
<td>TruffleRuby no splitting</td>
<td>1.36</td>
</tr>
<tr>
<td>TruffleRuby with splitting</td>
<td>2.75</td>
</tr>
</tbody>
</table>
TruffleRuby: Peak performance on yjit-bench (14 benchmarks)

From https://eregon.me/blog/2022/01/06/benchmarking-cruby-mjit-yjit-jruby-truffleruby.html
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-milion line codebases in production. At the same time, dy-namic 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:
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

<table>
<thead>
<tr>
<th></th>
<th>Polymorphic Calls</th>
<th>Megamorphic Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>956,515 (6.9%)</td>
<td>63,319 (0.457%)</td>
</tr>
<tr>
<td>After 2-level inline cache</td>
<td>490,072 (3.5%)</td>
<td>557 (0.004%)</td>
</tr>
<tr>
<td>After Splitting</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

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:
  \texttt{Polyglot.eval('python', 'import matplotlib')}

• Regexp JIT Compiler and how to avoid ReDoS (RubyKaigi 2021 presentation)

• Parallel execution of Ruby code and soon of \texttt{RB_EXT_RACTOR_SAFE}-marked C extensions

• Tooling across 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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BlogRails</td>
<td>956,515</td>
<td>63,319</td>
<td>-48.8%</td>
<td>-99.1%</td>
</tr>
<tr>
<td>ChunkyCanvas*</td>
<td>322</td>
<td>98</td>
<td>-80.0%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>ChunkyColor*</td>
<td>320</td>
<td>98</td>
<td>-79.0%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>ChunkyDec</td>
<td>322</td>
<td>98</td>
<td>-79.5%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>ERubiRails</td>
<td>626,535</td>
<td>40,699</td>
<td>-37.4%</td>
<td>-98.6%</td>
</tr>
<tr>
<td>HexaPdfSmall</td>
<td>1,842,665</td>
<td>479,399</td>
<td>-21.7%</td>
<td>-99.6%</td>
</tr>
<tr>
<td>LiquidCartParse</td>
<td>821</td>
<td>280</td>
<td>-73.3%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>LiquidCartRender</td>
<td>12,598</td>
<td>280</td>
<td>-84.1%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>LiquidMiddleware</td>
<td>747</td>
<td>251</td>
<td>-68.8%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>LiquidParseAll</td>
<td>5,369</td>
<td>280</td>
<td>-87.4%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>LiquidRenderBibs</td>
<td>89,866</td>
<td>280</td>
<td>-73.7%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>MailBench</td>
<td>81,886</td>
<td>12,697</td>
<td>-77.6%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>PsdColor</td>
<td>14,053</td>
<td>233</td>
<td>-53.1%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>PsdCompose*</td>
<td>14,053</td>
<td>233</td>
<td>-53.0%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>PsdImage*</td>
<td>14,062</td>
<td>233</td>
<td>-53.0%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>PsdUtil*</td>
<td>14,048</td>
<td>233</td>
<td>-53.0%</td>
<td>-100.0%</td>
</tr>
<tr>
<td>Sinatra</td>
<td>7,909</td>
<td>3,911</td>
<td>-82.8%</td>
<td>-94.4%</td>
</tr>
</tbody>
</table>
The Effect of Splitting

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of calls</th>
<th>After splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlogRails</td>
<td>490,072</td>
<td>557</td>
</tr>
<tr>
<td>ChunkyCanvas*</td>
<td>66</td>
<td>0</td>
</tr>
<tr>
<td>ChunkyColor*</td>
<td>66</td>
<td>0</td>
</tr>
<tr>
<td>ChunkyDec</td>
<td>66</td>
<td>0</td>
</tr>
<tr>
<td>ERubiRails</td>
<td>391,997</td>
<td>553</td>
</tr>
<tr>
<td>HexaPdfSmall</td>
<td>1,443,211</td>
<td>2,066</td>
</tr>
<tr>
<td>LiquidCartParse</td>
<td>219</td>
<td>0</td>
</tr>
<tr>
<td>LiquidCartRender</td>
<td>2,000</td>
<td>0</td>
</tr>
<tr>
<td>LiquidMiddleware</td>
<td>233</td>
<td>0</td>
</tr>
<tr>
<td>LiquidParseAll</td>
<td>679</td>
<td>0</td>
</tr>
<tr>
<td>LiquidRenderBibs</td>
<td>23,633</td>
<td>0</td>
</tr>
<tr>
<td>MailBench</td>
<td>18,322</td>
<td>0</td>
</tr>
<tr>
<td>PsdColor</td>
<td>6,586</td>
<td>0</td>
</tr>
<tr>
<td>PsdCompose*</td>
<td>6,586</td>
<td>0</td>
</tr>
<tr>
<td>PsdImage*</td>
<td>6,588</td>
<td>0</td>
</tr>
<tr>
<td>PsdUtil*</td>
<td>6,584</td>
<td>0</td>
</tr>
<tr>
<td>Sinatra</td>
<td>1,362</td>
<td>220</td>
</tr>
</tbody>
</table>