## Ruby master - Feature #18589
### Finer-grained constant invalidation

02/16/2022 05:32 PM - kddeisz (Kevin Newton)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

This is related to [https://github.com/ruby/ruby/pull/5433](https://github.com/ruby/ruby/pull/5433).

### Current behavior

Caches depend on a global counter. All constant mutations cause all caches to be invalidated.

```ruby
class A
  B = 1
end

def foo
  A::B # inline cache depends on global counter
end

foo # populate inline cache
foo # hit inline cache

C = 1 # global counter increments, all caches are invalidated

foo # misses inline cache due to `C = 1`
```

### Proposed behavior

Caches depend on name components. Only constant mutations with corresponding names will invalidate the cache.

```ruby
class A
  B = 1
end

def foo
  A::B # inline cache depends constants named "A" and "B"
end

foo # populate inline cache
foo # hit inline cache

C = 1 # caches that depend on the name "C" are invalidated

foo # hits inline cache because IC only depends on "A" and "B"
```

### Examples of breaking the new cache:

```ruby
module C
  # Breaks `foo` cache because "A" constant is set and the cache in foo depends
  # on "A" and "B"
  class A; end
end

B = 1
```

We expect the new cache scheme to be invalidated less often because names aren't frequently reused. With the cache being invalidated less, we can rely on its stability more to keep our constant references fast and reduce the need to throw away generated code in YJIT.

03/12/2022
Performance benchmarks

The following benchmark (included in this pull request) performs about 2x faster than master.

CONSTANT1 = 1
CONSTANT2 = 1
CONSTANT3 = 1
CONSTANT4 = 1
CONSTANT5 = 1

def constants
    [CONSTANT1, CONSTANT2, CONSTANT3, CONSTANT4, CONSTANT5]
end

500_000.times do
    constants
    INVALIDATE = true
end

In terms of macro benchmarks, I ran with this code on railsbench and there was not a statistically significant different in startup time or overall runtime performance.

byroot (Jean Boussier) also ran performance benchmarks on our production application. He noticed that there were several cache busts related to Object#extend (from core libraries), ActiveRecord::Relation#extending (from Rails), and autoload (from various gems, both internal and external). After a lot of work, the cache busts went down:

but they're still frequent enough that it's a problem. These changes had a measurable performance difference in request speed:

Memory benchmarks

In terms of memory, this includes an increase in VM size by about 500KiB when running on railsbench. This is because we're now tracking cache associations ({ ID => IC[] }) on the VM to know how to invalidate specific caches when constants change.

I booted Shopify's core monolith with this branch as well. It increased proportional to the number of constant caches found in the application. For each constant cache 1 level deep (e.g., Foo) the increase is about 33 bytes. For a constant cache 2 levels deep (e.g., Foo::Bar) the increase is about 67 bytes. The overall increase was around 16Mb or about 1% of the total retained memory.

History

#1 - 02/16/2022 06:45 PM - ko1 (Koichi Sasada)
Current design is the global counter doesn't change frequently.
Do you have measurements about it on some apps?

#2 - 02/16/2022 07:10 PM - kddeisz (Kevin Newton)
At the moment on Shopify's core monolith we're seeing around 1 in 30 requests invalidate the global cache. We're still working out the source of the invalidations. But at the moment with the current design if anything changes anywhere everything is invalidated. So unfortunately it happens quite frequently.

#3 - 02/16/2022 07:34 PM - Eregon (Benoit Daloze)
During startup, global invalidation for constants also causes a lot of extra lookups, and with a JIT it throws away a lot of code during startup (or the JIT can't inline the value of the constant).

Global per-name constant invalidation is also what JRuby does IIRC.

This is what we did in TruffleRuby, it's per class and constant/method name so it's quite precise (same general approach for method & constant lookup):
https://medium.com/graalvm/precise-method-and-constant-invalidation-in-truffleruby-4dd56c6bac1a

It has the advantage to not invalidate needlessly when e.g. two modules have a constant with the same name.

#4 - 02/17/2022 12:30 PM - byroot (Jean Boussier)
Amusingly enough, this discussion led me to instrument our production environment to see what is bumping the cache, and one of the big offenders is open-uri: https://github.com/ruby/open-uri/blob/174a8eb7de357fc04c0675dd30073c0218f401a5/lib/open-uri.rb#L415-L417

Another big offender is tzinfo: https://github.com/tzinfo/tzinfo/pull/129

03/12/2022
Here's the modified Ruby I used to find the source of the bumps:
https://github.com/Shopify/ruby/commit/7aad79590dd62c05ba3b965d1964dc80f147441b6

#5 - 02/18/2022 08:47 AM - byroot (Jean Boussier)
I have a fix for open-uri: https://github.com/ruby/open-uri/pull/8

#6 - 02/18/2022 02:10 PM - Dan0042 (Daniel DeLorme)

   It increased total retained memory from 1.23Gb to 1.3Gb (about a 0.7% increase).

1.3 / 1.23 is a 5.7% increase, not 0.7%

#7 - 02/28/2022 02:46 PM - byroot (Jean Boussier)
So I've been digging more on the constant cache busts happening in our app, beside what I already mentioned, what I found is:

A large majority of the busts are caused by autoloaded constants in gems. A few shouldn't autoload at all, but in many case is because the gem provide multiple implementation of an interface, and doesn't want to load them all. So there isn't really a good fix for this. Even the rack gem is fully autoloaded. This means that any web application is bound to invalidate the constant cache on the first few requests it processes.

Then I found some more APIs using Object#extend like open-uri. These are more of a problem, because they bust the cache every time they're called, not just the first time.

And since currently Ruby doesn't offer much API to diagnose this problem, it's unlikely to get better any time soon. So maybe the extra memory usage is worth it?

#8 - 03/04/2022 03:05 PM - byroot (Jean Boussier)

   Do you have measurements about it on some apps?

So here's some metrics from the app I investigated. The metric is not directly RubyVM.stat(:global_constant_state), instead we emit an increment if the global_constant_state changed during a request cycle.

The straight line is the last 24h, and the dotted line is the same days 4 weeks prior:

The number one cause by far was Object#extend like in open-uri, aws-sdk as well as ActiveRecord::Relation#extending.

The second most common cause was gems using a lot of autoload, or having memoized class attribute like tzinfo.

Overall fixing these had a very significant impact on the service latency

I'm however worried that this situation will degrade over time, as there's very little way to actively prevent this kind of problems from cropping up.

#9 - 03/04/2022 05:49 PM - kddeisz (Kevin Newton)
   - Description updated

#10 - 03/04/2022 06:17 PM - kddeisz (Kevin Newton)
   - Description updated

#11 - 03/10/2022 02:17 AM - kddeisz (Kevin Newton)
   - Description updated

#12 - 03/10/2022 02:18 AM - kddeisz (Kevin Newton)
   Dan0042 (Daniel DeLorme) yeah sorry, I was looking at different numbers and got wires crossed.

#13 - 03/10/2022 02:47 AM - kddeisz (Kevin Newton)
   - Description updated

#14 - 03/10/2022 01:40 PM - Eregon (Benoit Daloze)
What's the memory overhead of this? (probably the biggest concern from CRuby's side)

A 5.7% increase does sound like a lot for this.
But it seems the description now says 1% for the monolith?
What's the percentage for railsbench?

#15 - 03/10/2022 01:45 PM - byroot (Jean Boussier)

kddeisz (Kevin Newton) is away for a few days, so I'll take the liberty to answer even though he may correct me later.

A 5.7% increase does sound like a lot for this. But it seems the description now says 1% for the monolith?

The initial measurement that was showing the 5.7% increase was flawed, it was actually much less. Sorry about that, it's not that trivial to measure. Also thanks to jhawthorn (John Hawthorn) the patch memory usage was reduced even further, hence the ~1%.

Also after chatting a bit yesterday, we believe that in practice this will likely save memory in forking environments, because the finer grained cache will mean less ISeq being written into when a single constant change, so less CoW invalidations. But of course that's heavily dependent on the app.

#16 - 03/10/2022 05:29 PM - jhawthorn (John Hawthorn)

Tested this patch out on GitHub's largest app and the size of the additional constant cache bookkeeping was only ~3MB (as measured by vm_memszie_constant_cache) for our ~950MB application.