Ruby master - Bug #17031

`Kernel#caller_locations(m, n)` should be optimized

07/14/2020 07:07 PM - marcandre (Marc-Andre Lafortune)

Status:	Closed		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:		**Backport:**	2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN

Description

Kernel#caller_locations(1, 1) currently appears to needlessly allocate memory for the whole backtrace.

It allocates ~20kB for a 800-deep stacktrace, vs 1.6 kB for a shallow backtrace.
It is also much slower for long stacktraces: about 7x slower for a 800-deep backtrace than for a shallow one.

Test used:

```
def do_something
  location = caller_locations(1, 1).first
end

def test(depth, trigger)
  do_something if depth == trigger

  test(depth - 1, trigger) unless depth == 0
end

require 'benchmark/ips'

Benchmark.ips do |x|
  x.report (:short_backtrace    )    {test(800,800)}
  x.report (:long_backtrace     )    {test(800,  0)}
  x.report (:no_caller_locations)    {test(800, -1)}
end

require 'memory_profiler'

MemoryProfiler.report { test(800,800) }.pretty_print(scale_bytes: true, detailed_report: false)
MemoryProfiler.report { test(800,  0) }.pretty_print(scale_bytes: true, detailed_report: false)
```

Found when checking memory usage on RuboCop.

Associated revisions

Revision f2d7461e - 08/12/2020 06:03 PM - jeremyevans (Jeremy Evans)

Improve performance of partial backtraces

Previously, backtrace_each fully populated the rb_backtrace_t with all
backtrace frames, even if caller only requested a partial backtrace
(e.g. Kernel#caller_locations(1, 1)). This changes backtrace_each to
only add the requested frames to the rb_backtrace_t.

To do this, backtrace_each needs to be passed the starting frame and
number of frames values passed to Kernel#caller or #caller_locations.

backtrace_each works from the top of the stack to the bottom, where the
bottom is the current frame. Due to how the location for cfuncs is
tracked using the location of the previous iseq, we need to store an
extra frame for the previous iseq if we are limiting the backtrace and
final backtrace frame (the first one stored) would be a cfunc and not
an iseq.

To limit the amount of work in this case, while scanning until the start

of the requested backtrace, for each iseq, store the cfp. If the first
backtrace frame we care about is a cfunc, use the stored cfp to find the
related iseq. Use a function pointer to handle the storage of the cfp
in the iteration arg, and also store the location of the extra frame
in the iteration arg.

backtrace_each needs to return int instead of void in order to signal
when a starting frame larger than backtrace size is given, as caller
and caller_locations needs to return nil and not the empty array in
these cases.

To handle cases where a range is provided with a negative end, and the
backtrace size is needed to calculate the result to pass to
rb_range_beg_len, add a backtrace_size static function to calculate
the size, which copies the logic from backtrace_each.

As backtrace_each only adds the backtrace lines requested,
backtrace_to_*_ary can be simplified to always operate on the entire
backtrace.

Previously, caller_locations(1,1) was about 6.2 times slower for an
800 deep callstack compared to an empty callstack. With this new
approach, it is only 1.3 times slower. It will always be somewhat
slower as it still needs to scan the cfps from the top of the stack
until it finds the first requested backtrace frame.

Fixes [Bug #17031]

Revision 3b24b791 - 08/27/2020 10:17 PM - jeremyevans (Jeremy Evans)

Improve performance of partial backtraces

Previously, backtrace_each fully populated the rb_backtrace_t with all
backtrace frames, even if caller only requested a partial backtrace
(e.g. Kernel#caller_locations(1, 1)). This changes backtrace_each to
only add the requested frames to the rb_backtrace_t.

To do this, backtrace_each needs to be passed the starting frame and
number of frames values passed to Kernel#caller or #caller_locations.

backtrace_each works from the top of the stack to the bottom, where the
bottom is the current frame. Due to how the location for cfuncs is
tracked using the location of the previous iseq, we need to store an
extra frame for the previous iseq if we are limiting the backtrace and
final backtrace frame (the first one stored) would be a cfunc and not
an iseq.

To limit the amount of work in this case, while scanning until the start
of the requested backtrace, for each iseq, store the cfp. If the first
backtrace frame we care about is a cfunc, use the stored cfp to find the
related iseq. Use a function pointer to handle the storage of the cfp
in the iteration arg, and also store the location of the extra frame
in the iteration arg.

backtrace_each needs to return int instead of void in order to signal
when a starting frame larger than backtrace size is given, as caller
and caller_locations needs to return nil and not the empty array in
these cases.

To handle cases where a range is provided with a negative end, and the
backtrace size is needed to calculate the result to pass to
rb_range_beg_len, add a backtrace_size static function to calculate
the size, which copies the logic from backtrace_each.

As backtrace_each only adds the backtrace lines requested,
backtrace_to_*_ary can be simplified to always operate on the entire
backtrace.

Previously, caller_locations(1,1) was about 6.2 times slower for an
800 deep callstack compared to an empty callstack. With this new
approach, it is only 1.3 times slower. It will always be somewhat
slower as it still needs to scan the cfps from the top of the stack
until it finds the first requested backtrace frame.

This initializes the backtrace memory to zero. I do not think this is

necessary, as from my analysis, nothing during the setting of the
backtrace entries can cause a garbage collection, but it seems the
safest approach, and it's unlikely the performance decrease is
significant.

This removes the rb_backtrace_t backtrace_base member. backtrace
and backtrace_base were initialized to the same value, and neither
is modified, so it doesn't make sense to have two pointers.

This also removes LOCATION_TYPE_IFUNC from vm_backtrace.c, as
the value is never set.

Fixes [Bug #17031]

History

#1 - 07/15/2020 02:30 PM - Eregon (Benoit Daloze)

Could you post the results of running that on your computer?
Then it's easier to see your point without needing to reproduce.

#2 - 07/15/2020 04:11 PM - marcandre (Marc-Andre Lafortune)

Sure

```
Calculating -------------------------------------
     short_backtrace     28.315k (± 7.1%) i/s -    141.984k in   5.044733s
      long_backtrace     24.168k (± 8.7%) i/s -    120.900k in   5.050243s
 no_caller_locations     29.288k (± 2.5%) i/s -    148.359k in   5.068723s
Total allocated: 1.58 kB (3 objects)
Total retained:  0 B (0 objects)

Total allocated: 19.58 kB (3 objects)
Total retained:  0 B (0 objects)
```

I got a factor 6.2 this time: (1/24.168-1/29.288)/(1/28.315-1/29.288)

#3 - 07/17/2020 11:09 PM - jeremyevans0 (Jeremy Evans)

Reviewing the related code in vm_backtrace.c, you are correct. This occurs both for caller and caller_locations. The entire internal backtrace object is generated by rb_ec_backtrace_object, and then passed to a function that looks at specific parts of it to generate the strings or Thread::Backtrace::Location objects. To fix this would require changing the logic so that rb_ec_backtrace_object was passed the starting level and number of frames.

#4 - 07/23/2020 10:25 PM - jeremyevans0 (Jeremy Evans)

I've added a pull request that addresses this issue: https://github.com/ruby/ruby/pull/3357

#5 - 08/12/2020 07:14 AM - ko1 (Koichi Sasada)

Thank you Jeremy!
Great patch!

#6 - 08/12/2020 06:03 PM - jeremyevans (Jeremy Evans)

- Status changed from Open to Closed

Applied in changeset git|f2d7461e85053cb084e10999b0b8019b0c29e66e.

Improve performance of partial backtraces

Previously, backtrace_each fully populated the rb_backtrace_t with all
backtrace frames, even if caller only requested a partial backtrace
(e.g. Kernel#caller_locations(1, 1)). This changes backtrace_each to
only add the requested frames to the rb_backtrace_t.

To do this, backtrace_each needs to be passed the starting frame and
number of frames values passed to Kernel#caller or #caller_locations.

backtrace_each works from the top of the stack to the bottom, where the
bottom is the current frame. Due to how the location for cfuncs is
tracked using the location of the previous iseq, we need to store an
extra frame for the previous iseq if we are limiting the backtrace and
final backtrace frame (the first one stored) would be a cfunc and not

an iseq.

To limit the amount of work in this case, while scanning until the start
of the requested backtrace, for each iseq, store the cfp. If the first
backtrace frame we care about is a cfunc, use the stored cfp to find the
related iseq. Use a function pointer to handle the storage of the cfp
in the iteration arg, and also store the location of the extra frame
in the iteration arg.

backtrace_each needs to return int instead of void in order to signal
when a starting frame larger than backtrace size is given, as caller
and caller_locations needs to return nil and not the empty array in
these cases.

To handle cases where a range is provided with a negative end, and the
backtrace size is needed to calculate the result to pass to
rb_range_beg_len, add a backtrace_size static function to calculate
the size, which copies the logic from backtrace_each.

As backtrace_each only adds the backtrace lines requested,
backtrace_to_*_ary can be simplified to always operate on the entire
backtrace.

Previously, caller_locations(1,1) was about 6.2 times slower for an
800 deep callstack compared to an empty callstack. With this new
approach, it is only 1.3 times slower. It will always be somewhat
slower as it still needs to scan the cfps from the top of the stack
until it finds the first requested backtrace frame.

Fixes [Bug #17031]

#7 - 08/12/2020 06:50 PM - jeremyevans0 (Jeremy Evans)

- Status changed from Closed to Open

Reopening this as the commit had to be reverted as CI showed issues in backtrace_mark. I can't work on debugging this right away, so someone is welcome to look into fixing this in the meantime.

#8 - 08/21/2020 06:01 PM - jeremyevans0 (Jeremy Evans)

Most of the CI issues were when running with asserts, so I compiled with asserts enabled and ran tests. It took quite a while, and no problems were noted:

```
Finished tests in 82924.634324s, 0.2518 tests/s, 67.0770 assertions/s.
20884 tests, 5562336 assertions, 0 failures, 0 errors, 79 skips
```

Here are some backtraces for the crashed CI processes before the commit was reverted:

```
/tmp/ruby/v3/build/trunk-test/libruby.so.2.8.0(rb_bug+0xe4) [0x7fd5f7149c73] /tmp/ruby/v3/src/trunk-test/error
.c:660
/tmp/ruby/v3/build/trunk-test/libruby.so.2.8.0(gc_mark_ptr+0xee) [0x7fd5f7202a3e] /tmp/ruby/v3/src/trunk-test/
gc.c:5301
/tmp/ruby/v3/build/trunk-test/libruby.so.2.8.0(backtrace_mark+0x89) [0x7fd5f73b7859] /tmp/ruby/v3/src/trunk-te
st/vm_backtrace.c:129
/tmp/ruby/v3/build/trunk-test/libruby.so.2.8.0(gc_mark_children+0x5d7) [0x7fd5f7203447] /tmp/ruby/v3/src/trunk
-test/gc.c:5544

/tmp/ruby/v3/build/trunk-gc-asserts/libruby.so.2.8.0(sigsegv+0x4b) [0x7f998c3acc8b] /tmp/ruby/v3/src/trunk-gc-
asserts/signal.c:959
/lib/x86_64-linux-gnu/libc.so.6(0x7f998bdecf20) [0x7f998bdecf20]
/tmp/ruby/v3/build/trunk-gc-asserts/libruby.so.2.8.0(backtrace_mark+0x27) [0x7f998c441167] /tmp/ruby/v3/src/tr
unk-gc-asserts/vm_backtrace.c:425
/tmp/ruby/v3/build/trunk-gc-asserts/libruby.so.2.8.0(gc_mark_children+0x5e7) [0x7f998c28d607] /tmp/ruby/v3/src
/trunk-gc-asserts/gc.c:5544

/tmp/ruby/v3/build/trunk-asserts/libruby.so.2.8.0(check_rvalue_consistency_force+0x5c) [0x7f939d46ecac] /tmp/r
uby/v3/src/trunk-asserts/gc.c:1291
/tmp/ruby/v3/build/trunk-asserts/libruby.so.2.8.0(gc_mark+0x2c) [0x7f939d4730ec] /tmp/ruby/v3/src/trunk-assert
s/gc.c:1307
/tmp/ruby/v3/build/trunk-asserts/libruby.so.2.8.0(backtrace_mark+0x89) [0x7f939d62a309] /tmp/ruby/v3/src/trunk
-asserts/vm_backtrace.c:129
/tmp/ruby/v3/build/trunk-asserts/libruby.so.2.8.0(gc_mark_children+0x74f) [0x7f939d4738bf] /tmp/ruby/v3/src/tr
unk-asserts/gc.c:5544
```

Unfortunately, not being able to recreate the issue or get a backtrace with debugging information, I'm not sure what the root cause is. My best guess at this point is it may be caused by this chunk:

```
@@ -126,6 +129,10 @@ location_mark_entry(rb_backtrace_location_t *fi)
        rb_gc_mark_movable((VALUE)fi->body.iseq.iseq);
        break;
       case LOCATION_TYPE_CFUNC:
+        if (fi->body.cfunc.prev_loc) {
+            location_mark_entry(fi->body.cfunc.prev_loc);
+        }
+        break;
       case LOCATION_TYPE_IFUNC:
       default:
        break;
@@ -484,22 +491,47 @@ backtrace_alloc(VALUE klass)
     return obj;
 }
```

I originally thought that was necessary to handle the location for a cfunc where the previous iseq frame was not part of the backtrace, but on reflection it should not be, because that iseq frame should be separately marked as it is the last entry in rb_backtrace_t.backtrace. I'm not sure why it would cause problems, but since it is new and shouldn't be necessary, it might be best to eliminate it. I submitted this change as a pull request (https://github.com/ruby/ruby/pull/3441).

#9 - 08/27/2020 10:17 PM - jeremyevans (Jeremy Evans)

- Status changed from Open to Closed

Applied in changeset git|3b24b7914c16930bfadc89d6aff6326a51c54295.

Improve performance of partial backtraces

Previously, backtrace_each fully populated the rb_backtrace_t with all
backtrace frames, even if caller only requested a partial backtrace
(e.g. Kernel#caller_locations(1, 1)). This changes backtrace_each to
only add the requested frames to the rb_backtrace_t.

To do this, backtrace_each needs to be passed the starting frame and
number of frames values passed to Kernel#caller or #caller_locations.

backtrace_each works from the top of the stack to the bottom, where the
bottom is the current frame. Due to how the location for cfuncs is
tracked using the location of the previous iseq, we need to store an
extra frame for the previous iseq if we are limiting the backtrace and
final backtrace frame (the first one stored) would be a cfunc and not
an iseq.

To limit the amount of work in this case, while scanning until the start
of the requested backtrace, for each iseq, store the cfp. If the first
backtrace frame we care about is a cfunc, use the stored cfp to find the
related iseq. Use a function pointer to handle the storage of the cfp
in the iteration arg, and also store the location of the extra frame
in the iteration arg.

backtrace_each needs to return int instead of void in order to signal
when a starting frame larger than backtrace size is given, as caller
and caller_locations needs to return nil and not the empty array in
these cases.

To handle cases where a range is provided with a negative end, and the
backtrace size is needed to calculate the result to pass to
rb_range_beg_len, add a backtrace_size static function to calculate
the size, which copies the logic from backtrace_each.

As backtrace_each only adds the backtrace lines requested,
backtrace_to_*_ary can be simplified to always operate on the entire
backtrace.

Previously, caller_locations(1,1) was about 6.2 times slower for an
800 deep callstack compared to an empty callstack. With this new
approach, it is only 1.3 times slower. It will always be somewhat
slower as it still needs to scan the cfps from the top of the stack
until it finds the first requested backtrace frame.

This initializes the backtrace memory to zero. I do not think this is
necessary, as from my analysis, nothing during the setting of the

backtrace entries can cause a garbage collection, but it seems the
safest approach, and it's unlikely the performance decrease is
significant.

This removes the rb_backtrace_t backtrace_base member. backtrace
and backtrace_base were initialized to the same value, and neither
is modified, so it doesn't make sense to have two pointers.

This also removes LOCATION_TYPE_IFUNC from vm_backtrace.c, as
the value is never set.

Fixes [Bug #17031]