Add GC.stat[:total_time] when GC profiling enabled

02/27/2015 11:19 PM - jasonrclark (Jason Clark)

Status: Open
Priority: Normal
Assignee:
Target version:

Description
This patch includes a :total_time value in GC.stat as a Fixnum of microseconds equivalent to GC::Profiler.total_time. A non-zero value is only provided if GC profiling is enabled.

This avoids problems with GC::Profiler’s API in the presence of multiple callers. Well-behaved clients of GC::Profiler are expected to call clear periodically to constrain memory usage of the profiling structures. However, this causes problems when multiple callers—unaware of each other—rely on the value of GC::Profiler.total_time being unaltered since their last clear.

Using a single value in GC.stat avoids this by providing a monotonically increasing count with every GC, not influenced by clear.

Considerations and Questions

- Because the individual GC times are potentially small, I tracked the total as a double and only convert to Fixnum when constructing GC.stat’s return. Is there something better I should be doing there, in light of GC.stat's insistence that the stats returned be of type size_t?

- What should I do (if anything) to cope with potential overflows? If Ruby is on a platform where size_t is 32 bits, we'd hit an overflow after 1.2 hours worth of cumulative GC time.

Future Directions

Is there any possibility of moving the GC timing (not the deeper profiling data) outside GC::Profiler.enable’s gating? I would love to avoid clients needing to make code changes to track their GC time with my gem (newrelic_rpm).

As it stands invocation start times are held on the GC profiling structures, so it felt like a much bigger change to alter that, and I thought I'd pitch this simpler change first.

Any advice on whether that would be possible? Is timing too slow to do by default, or is the gating just an artifact of how it was implemented with GC::Profiler?

History

#1 - 03/08/2015 10:21 PM - ko1 (Koichi Sasada)
With incremental GC (incremental marking and lazy sweep) consume very short time for each step.
I'm not sure we should measure such short time. At least, there are several BAD effect (especially on virtual machine environment).

If you want to measure only marking time, I can accept. But I'm not sure is it enough.

#2 - 03/09/2015 03:43 PM - jasonrclark (Jason Clark)
That makes sense, Koichi. I wondered if timing the small steps by default would be too heavy.

By only measuring mark time, do you mean https://github.com/ruby/ruby/blob/trunk/gc.c#L5184-L5212? For my own curiosity, I'll probably see how those timings look. Even if they're not complete, they might be useful.

Also, I apologize for mixing a patch and a separate question into one issue. The patch doesn't actually add any timings—it's just a new GC.stat counter, still gated by GC::Profiler.enable. Does that seem reasonable as it avoids issues with multiple clients calling GC::Profiler#clear?

#3 - 04/24/2015 10:18 PM - jasonrclark (Jason Clark)
Any thoughts on this patch Koichi? It just adds a GC.stat counter, and doesn't move anything outside the current GC::Profiler gating.

#4 - 08/06/2021 11:15 AM - byroot (Jean Boussier)
I'd like to restart the discussion on this as I recently hit the same issue.
With incremental GC (incremental marking and lazy sweep) consume very short time for each step.

Indeed. However when running a web service, even if individual pauses are short, it's not uncommon for the GC to represent a very substantial slice of the total request time (up to 30%), as such it is a very valuable information to collect and log as part of the request metrics.

At least, there are several BAD effect (especially on virtual machine environment).

Could you clarify what bad effects you have in mind? Is it a substantial overhead? If so I suppose it could have its own toggle.

Also right now the way to collect this information is to enable GC::Profiler, which as explained in the issue description really isn't very practical, and incur a more substantial overhead.

FWIW TruffleRuby and JRuby both have a :time value in GC.stat and it represents the total GC time, in milliseconds (GarbageCollectorMXBean#collectTime provides this data on the JVM) (No GC::Profiler needed to query it)
If this is added to CRuby I would suggest to match that for improved compatibility.

FWIW TruffleRuby and JRuby both have a :time value in GC.stat and it represents the total GC time, in milliseconds

Ah that's good to know. Thank you!

In this case I think the patch could look like this: https://github.com/Shopify/ruby/commit/7f02ac8d6906a4bde1bde154e958c1be30e451139bb

Ideally we'd return a float, but as noted by jasonrclark (Jason Clark), it's not easy to return anything else than size_t without breaking the size_t rb_gc_stat(VALUE) C API.

If that precision is needed (unsure), one possibility would be to add GC.total_time or so and return float seconds (or in nanoseconds, Floats have their own precision issues).

That patch contains too many unrelated changes.

And my concern is if :time is clear enough. Can you tell its unit by that name?

That patch contains too many unrelated changes.

You mean mine? Or the original one?

If mine, most of the diff is because I renamed a bunch of gc_prof_* in just gc_* as they are no longer GC::Profiler specific. Also the link I posted was just a rough first try, I improved it in https://github.com/ruby/ruby/pull/4724/files?w=1

And my concern is if :time is clear enough. Can you tell its unit by that name?

That's unfortunately the case with most time metrics. Some people tend to append _ms to the name to make it clearer, not sure if it's worth not being compatible with JRuby/TruffleRuby over this.

I agree. Need to consider performance cost.

Matz.
GC::Profiler is not maintained well and it doesn't calculate all of GC time now.

I rework it using clock_gettime (so it only works on some platforms).
https://github.com/ruby/ruby/pull/4757

```ruby
pooled = Array.new(100_000){ [] }
N = 50_000_000
h1 = h2 = nil
require 'benchmark'
Benchmark.bm{|x|
  x.report{
    h1 = GC.count
    N.times{
      a = "str".upcase
    }
    h2 = GC.count
  }
}
pool GC.stat
```

This script introduces lazy sweeping and each sweep consumes only a small time. So it is worst case.

```ruby
[RUBY_DEBUG_COUNTER] gc_enter_start 1,778
[RUBY_DEBUG_COUNTER] gc_enter_mark_continue 6
[RUBY_DEBUG_COUNTER] gc_enter_sweep_continue 49,547
[RUBY_DEBUG_COUNTER] gc_enter_rest 1
[RUBY_DEBUG_COUNTER] gc_enter_finalizer 1
```

It causes 1,778 GCs and 49,547 lazy sweeping. My patch counts the time about 50,000 times:

```ruby
master:
user system total real
3.332477 0.000425 3.332902 ( 3.332915)

with time:
user system total real
3.436177 0.000084 3.436261 ( 3.436298)
{:count=>1778,
 :time=>1286,
...}
```

and it slows down about 3%. Maybe it shows worst case.

```ruby
pooled = Array.new(100_000){ [] }
N = 50_000_000
h1 = h2 = nil
require 'benchmark'
Benchmark.bm{|x|
  x.report{
    h1 = GC.count
    N.times{
      a = "str".upcase
    }
    h2 = GC.count
  }
}
pool GC.stat
```

On the other hands, if the program contains meaningful task (in this case, String#upcase), the impact will be small.

```ruby
master:
user system total real
5.026109 0.003133 5.029242 ( 5.029301)

with time:
user system total real
4.995175 0.000336 4.995511 ( 4.995567) ## FASTER!!
{:count=>1423,
 :time=>1229,
...}
```

Summary:
• It can affect performance on some cases.
• Most of case I think the penalty is enough small.

Comments:
• It adds small times so it introduces errors.
• Adding flag like GC.measure_time = true/false (default: false) is one idea.

#12 - 08/21/2021 07:01 AM - byroot (Jean Boussier)

It can affect performance on some cases.

But less than GC::Profiler, so for users of this feature it would be a performance win.

It adds small times so it introduces errors.

I don't think it's a big problem if it isn't perfectly accurate.

Adding flag like GC.measure_time = true/false (default: false) is one idea.

That's acceptable I think. Application needing it can flip it up.

Files
| gc-total-time.patch | 3.43 KB | 02/27/2015 | jasonrclark (Jason Clark) |