Ruby master - Bug #9607

Change the full GC timing

03/07/2014 08:56 AM - ko1 (Koichi Sasada)

Status:	Closed		
Priority:	Normal		
Assignee:	ko1 (Koichi Sasada)		
Target version:	2.2.0		
ruby -v:	2.2	**Backport:**	2.0.0: DONTNEED, 2.1: DONE

Description

Abstract

Generational GC (called RGenGC) was introduced from Ruby 2.1.0. RGenGC
reduces marking time dramatically (about x10 faster). However, RGenGC
introduce huge memory consumption. This problem has impact especially
for small memory machines.

Ruby 2.1.1 introduced new environment variable
RUBY_GC_HEAP_OLDOBJECT_LIMIT_FACTOR to control full GC timing. However,
this solution is not solve problem completely.

To solve this issue, we modify **Full GC timing strategy**:
(1) Always invoke full GC before extending the heap.
(2) Increase the heap if not enough old-objects space.
This modification introduces a bit slow down, but reduce memory
consumption.

Background and problem

RGenGC algorithm

Ruby 2.0 and earlier versions uses simple mark and sweep. Long marking
time had been an big issue. To solve this issue, Ruby 2.1.0 introduced
new generational GC called RGenGC (restricted generational GC).

RGenGC algorithm enables to introduce partial marking (called minor GC'),
which marks only newer created objects, and skips marking fof old
objects (*1). Sometime, this marks all objects (calledmajor GC' or
`full GC'). Many minor GC and small number of major GC makes GC faster.

(*1) RGenGC doesn't skip sweeping for old-objects. This is another issue.

Full GC timing

There is a question: "When should we invoke invoke full GC?".

Usually, generational GC uses the strategy that "when a space for old
objects is full, then invoke full GC".

Ruby 2.1.0 defines the size of old space for old objects with
old_object_limit' and old_object_limit is doubled by the old objects
number (old_object_count') at the last full GC.

Before the GC, we determine minor or major by comparing
old_object_limit' and current old objects number (old_object_count')
if we compare current old object number and old_object_limit, and do
full GC if old_object_count > old_object_limit.

Here is a pseudo code of RGenGC:

```
def gc
  if old_object_count > old_object_limit
    major_gc = false
    minor_mark()
  else
    major_gc = true
    major_mark()
  end
  sweep() # Actually it is lazy sweep.

  # double `old_object_count' here when it is major GC
  old_object_limit = old_object_count * 2 if major_gc
end
```

This strategy works fine for memory rich machines, because only a few
full GCs are invoked.

However, this strategy causes more and more memory consumption.

Fig.1 is a result of (modified) discourse benchmark (Thanks Sam
Saffron!!). X-axis is GC count and Y-axis represents a number of slots
(objects). total_slots' is avaialbe slots to use,old_object' is
old_object_count.

ruby2_2.JPG
As you can see, old_object_limit is too high and total_slots are
expanded (x1.8, specified by GC_HEAP_GROWTH_FACTOR) before full GC.

Full GC timing tuning from Ruby 2.1.1

To solve this issue, Ruby 2.1.1 introduced an environment variable
"RUBY_GC_HEAP_OLDOBJECT_LIMIT_FACTOR" (use `old_object_limit_factor' for
short).

This variable control how to extend `old_object_limit'.

In pseudo code, we changed from

```
# double `old_object_count' here when it is major GC
old_object_limit = old_object_count * 2 if major_gc
```

to

```
# double `old_object_count' here when it is major GC
old_object_limit = old_object_count * old_object_limit_factor if major_gc
```

The default value of this environment variable is 2. So it is same
behavior on default.

With RUBY_GC_HEAP_OLDOBJECT_LIMIT_FACTOR=1.3, the benchmark result is
Fig.2.

ruby_2_2_factor_1_3.JPG
We can observe that the total slots doesn't grow than the default
behavior.

Try this environment variable if you have trouble with memory usage.

Note that if you want to disable generational garbage collection, you
can specify 0.9 (any number lesser than 1.0) for
RUBY_GC_HEAP_OLDOBJECT_LIMIT_FACTOR. With this technique, on every GC
"old_object_count > old_object_limit" is true and do major GC.

BTW, this variable should be noted on NEWS file. I missed to add it.

More intelligent approach

"RUBY_GC_HEAP_OLDOBJECT_LIMIT_FACTOR" with small number can solve this issue, but we need to specify correct value for each application. It is tough work for us.

Proposal

With these graphes, we find two insights.

(1) We need to invoke full GC becore expanding heaps. If we invoke full GC, it is possible to stop expanding heaps.
(2) Increasing speed of old objects is completely slow.

To invoke full GC before expanding, we set a upper bount for old_object_limit as "total_slots * 0.7". This value is same as the threshold to determin expanding heaps or not.

After full GC, it is possible that "old_object_count > old_object_limit" is true, but only a few differences. This situation causes many of full GC. To avoid such situation, we add a few slots if "old_object_limit * 0.7 < old_object_count). In this case, "old_object_limit * 0.7" is a minimum space for old objects.

In pseudo code:

```
def gc
  if old_object_count > old_object_limit
    major_gc = false
    minor_mark()
  else
    major_gc = true
    major_mark()
  end

  sweep() # Actually it is lazy sweep.

  if major_gc
    if total_slots * 0.7 < using_slots
      # not enough space
      extend_heap(total_slots * (1.8 - 1)) # 1.8 is growth_factor
    elsif old_object_limit * 0.7 < old_object_count
      # not enough old object count
      extend_heap(old_object_count - object_limit * 0.7)
    end
  else
    do_major_gc_at_next_gc = true
  end

  if major_gc
    a = old_object_count * old_object_limit_factor
    b = total_slots * 0.7
    old_object_limit = [a, b].min
  end
end
```

With this proposal, we can reduce total_slots consumption (Fig3, Fig4).

proposed.JPG
proposed_factor_1_3.JPG
However, more and more GC invoking time. It is trade-off because reducing total_slots introduces more frequent GC. We can solve this issue by making condition parameter 0.7 as tunable.

Future work

(1) Promotion strategy

Current growing speed of old object number is too high. So we need to
consider about promotion strategy. Current strategy is "promote young
objects when young objects survive one garbage collection". We already
implemented "RGENGC_THREEGEN" mode, which enable to filter unexpected
promotion.

NOTE: THREEGEN = 3gen is strange name because generation is only two. We
will change this mode name to AGE2PROMOTION and so on.

(2) Partial sweep

We successed to use partial marking on minor GC. However, everytime
sweep all available slots. Sweeping time is not so big, but there is a
space to optimize it.

(3) Incremental major GC

With this proposal, we increase major GC count. To avoid long major GC
pausing time, we need to implement incremental marking on full GC.

Associated revisions

Revision 5b2a7458 - 06/09/2014 11:43 AM - ko1 (Koichi Sasada)

- gc.c: change full GC timing to keep lower memory usage. Extend heap only at (1) after major GC or (2) after several (two times, at current) minor GC Details in https://bugs.ruby-lang.org/issues/9607#note-9 [Bug #9607]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@46387 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 46387 - 06/09/2014 11:43 AM - ko1 (Koichi Sasada)

- gc.c: change full GC timing to keep lower memory usage. Extend heap only at (1) after major GC or (2) after several (two times, at current) minor GC Details in https://bugs.ruby-lang.org/issues/9607#note-9 [Bug #9607]

Revision 46387 - 06/09/2014 11:43 AM - ko1 (Koichi Sasada)

- gc.c: change full GC timing to keep lower memory usage. Extend heap only at (1) after major GC or (2) after several (two times, at current) minor GC Details in https://bugs.ruby-lang.org/issues/9607#note-9 [Bug #9607]

Revision 46387 - 06/09/2014 11:43 AM - ko1 (Koichi Sasada)

- gc.c: change full GC timing to keep lower memory usage. Extend heap only at (1) after major GC or (2) after several (two times, at current) minor GC Details in https://bugs.ruby-lang.org/issues/9607#note-9 [Bug #9607]

Revision 46387 - 06/09/2014 11:43 AM - ko1 (Koichi Sasada)

- gc.c: change full GC timing to keep lower memory usage. Extend heap only at (1) after major GC or (2) after several (two times, at current) minor GC Details in https://bugs.ruby-lang.org/issues/9607#note-9 [Bug #9607]

Revision 46387 - 06/09/2014 11:43 AM - ko1 (Koichi Sasada)

- gc.c: change full GC timing to keep lower memory usage. Extend heap only at (1) after major GC or (2) after several (two times, at current) minor GC Details in https://bugs.ruby-lang.org/issues/9607#note-9 [Bug #9607]

Revision 46387 - 06/09/2014 11:43 AM - ko1 (Koichi Sasada)

- gc.c: change full GC timing to keep lower memory usage. Extend heap only at (1) after major GC or (2) after several (two times, at current) minor GC Details in https://bugs.ruby-lang.org/issues/9607#note-9 [Bug #9607]

Revision 478a0180 - 08/30/2014 04:29 PM - nagachika (Tomoyuki Chikanaga)

merge revision(s) r46387: [Backport #9607]

```
    * gc.c: change full GC timing to keep lower memory usage.
      Extend heap only at
      (1) after major GC
      or
      (2) after several (two times, at current) minor GC
      Details in https://bugs.ruby-lang.org/issues/9607#note-9
      [Bug #9607]
```

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/branches/ruby_2_1@47326 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 47326 - 08/30/2014 04:29 PM - nagachika (Tomoyuki Chikanaga)

merge revision(s) r46387: [Backport #9607]

```
* gc.c: change full GC timing to keep lower memory usage.
  Extend heap only at
  (1) after major GC
  or
  (2) after several (two times, at current) minor GC
  Details in https://bugs.ruby-lang.org/issues/9607#note-9
  [Bug #9607]
```

History

#1 - 03/07/2014 09:01 AM - ko1 (Koichi Sasada)

- Description updated

#2 - 03/07/2014 09:05 AM - ko1 (Koichi Sasada)

- Description updated

#3 - 03/07/2014 09:07 AM - ko1 (Koichi Sasada)

- File gc.patch added

Patch is added.

#4 - 03/16/2014 09:19 PM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> File gc.patch added

Running this (on top of current trunk) to serve my (mostly static sites)
on yhbt.net.   Memory usage seems stable at ~31M (from ~49M)

I noticed vm1_gc_short_with_complex_long got very slow with this
patch: ~10s => ~118s
I ran this several times to be sure. 2.0.0 only took around ~13s

#5 - 03/17/2014 04:09 AM - ko1 (Koichi Sasada)

Eric Wong wrote:

> Running this (on top of current trunk) to serve my (mostly static sites)
> on yhbt.net.   Memory usage seems stable at ~31M (from ~49M)

Thanks!

> I noticed vm1_gc_short_with_complex_long got very slow with this
> patch: ~10s => ~118s
> I ran this several times to be sure. 2.0.0 only took around ~13s

OMG. Thank you for reporting.

Yes. It will do useless marking (minor marking) because of there are no empty spaces.

mmm.

#6 - 03/21/2014 09:58 PM - normalperson (Eric Wong)

Eric Wong normalperson@yhbt.net wrote:

> ko1@atdot.net wrote:

File gc.patch added

Running this (on top of current trunk) to serve my (mostly static sites)
on yhbt.net. Memory usage seems stable at ~31M (from ~49M)

Actually, not, it hit ~89M(!). It could be I got a traffic surge (but I
did not pay attention to that).

I also did not set MALLOC_MMAP_THRESHOLD_ for eglibc malloc;
only MALLOC_ARENA_MAX=1 MALLOC_ARENA_CHECK=1. So this may be
a problem of malloc fragmentation, too.

#7 - 04/10/2014 04:05 AM - Student (Nathan Zook)

I wonder if it might not be better to give the user control? Specifically, consider a web application. It would make sense to hold off promoting objects created during a call-response cycle until the end of the cycle. But there is no way for the GC to know when that might be. This is trivial for the application to know.

More generally, applications know when they are entering periods with lots of mid-term objects being created. Adding GC.no_promote would allow an application to tune this behavior. I would suggest allowing calls with and without blocks, the blockless form would be reveresed by GC.allow_promote.

#8 - 04/10/2014 05:38 AM - normalperson (Eric Wong)

GC.promote/allow_promote can work in some cases, but I consider it too
ugly; as ugly as OobGC. It would also be error prone and hard to work
in multithreaded situations.

I would rather have work towards automatic run-time optimizations
(perhaps via online profiling/escape-analysis) than encourage users to
do brittle/ugly things in their code for short-term benefit.

#9 - 06/09/2014 11:37 AM - ko1 (Koichi Sasada)

- File discourse_benchmark.png added

- File young_objects.png added

After introducing AGE2_PROMOTION patch, only a few old objects are added for each minor GC.
The impact of this issue (too many slots) are not critical now. But some application can not protect increasing total slots.

I rewrite algorithm more simple:

```
def gc
  if do_major_gc_at_next_gc
    major_gc = false
    minor_mark()
  else
    major_gc = true
    major_mark()
    last_major_gc = GC.count
  end

  sweep() # Actually it is lazy sweep.

  if total_slots * 0.7 < using_slots
    # not enough space
    if major_gc ||
       GC.count - last_major_gc > 2 # (A) extend heap at least 2 minor GC run
      extend_heap
    else
      do_major_gc_at_next_gc = true
    end
  end
end
```

This algorithm simply do:

(1) If not enough slots after minor GC, do major GC at next GC
(2) If not enough slots after major GC, extend heap
(3) If not enough slots after minor GC *and* only a few minor GC until last major GC (it should be not enough slots), extend heap ((A) in the above algorithm)

This strategy keeps lower total slots.

discourse_benchmark.png
This picture shows the (a) total_slots (b) old_objects (c) ru_minflt (used memory pages given by getrusage) for current trunk and modified (above algorithm introduced) trunk using gc_tracer.

You can see:

(1) ru_minflt (rough memory usage) is linear to total_slots.
(2) heap_slots/modified is fewer than heap_slots. It has impact.
(3) old_objects/modified is reduced periodically. Do major GC aggressively compare with current trunk.
(4) Modified trunk invokes higher number of GC events (similar to GC count). This is because total_slots number is fewer than current trunk. It is intentional.

You can increment consuming slot number using environment variables. So that I will introduce this patch. It means that default GC parameter is "low memory consuming/low performance" compare with current trunk/Ruby 2.1.

The patch is also small:

```
Index: gc.c
===================================================================
--- gc.c    (revision 46386)
+++ gc.c    (working copy)
@@ -531,6 +531,9 @@
     int parent_object_is_old;

     int need_major_gc;
+
+    size_t last_major_gc;
+
     size_t remembered_shady_object_count;
     size_t remembered_shady_object_limit;
     size_t old_object_count;
@@ -3035,15 +3038,13 @@
            (int)heap->total_slots, (int)heap_pages_swept_slots, (int)heap_pages_min_free_slots);

      if (heap_pages_swept_slots < heap_pages_min_free_slots) {
-    heap_set_increment(objspace, heap_extend_pages(objspace));
-    heap_increment(objspace, heap);
-
-#if USE_RGENGC
-    if (objspace->rgengc.remembered_shady_object_count + objspace->rgengc.old_object_count > (heap_pages_lengt
h * HEAP_OBJ_LIMIT) / 2) {
-        /* if [old]+[remembered shady] > [all object count]/2, then do major GC */
-        objspace->rgengc.need_major_gc = GPR_FLAG_MAJOR_BY_RESCAN;
+    if (objspace->rgengc.during_minor_gc && objspace->profile.count - objspace->rgengc.last_major_gc > 2 /* ma
gic number */) {
+        objspace->rgengc.need_major_gc = GPR_FLAG_MAJOR_BY_NOFREE;
     }
-#endif
+    else {
+        heap_set_increment(objspace, heap_extend_pages(objspace));
+        heap_increment(objspace, heap);
+    }
      }

      gc_prof_set_heap_info(objspace);
@@ -4256,6 +4257,7 @@
     objspace->profile.major_gc_count++;
     objspace->rgengc.remembered_shady_object_count = 0;
     objspace->rgengc.old_object_count = 0;
+    objspace->rgengc.last_major_gc = objspace->profile.count;
     rgengc_mark_and_rememberset_clear(objspace, heap_eden);
      }
 #endif
```

PS. BTW, AGE2PROMOTION strategy introducing young objects, which is promoted from infant objects and will be promoted to old objects. I also counts them and I found that they are only a few objects in this program.

young_objects.png

#10 - 06/09/2014 11:43 AM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

- % Done changed from 0 to 100

Applied in changeset r46387.

- gc.c: change full GC timing to keep lower memory usage. Extend heap only at (1) after major GC or (2) after several (two times, at current) minor GC Details in https://bugs.ruby-lang.org/issues/9607#note-9 [Bug #9607]

#11 - 06/30/2014 06:57 AM - ko1 (Koichi Sasada)

- Backport changed from 1.9.3: UNKNOWN, 2.0.0: UNKNOWN, 2.1: UNKNOWN to 1.9.3: UNKNOWN, 2.0.0: UNKNOWN, 2.1: REQUIRED

The following patch is for current Ruby 2.1 branch.

Can anyone (who has memory consuming trouble) try this patch on Ruby 2.1 trunk?

I think this patch will decrease memory consumption on Ruby 2.1 (but increase major GC counts).

Chikanaga-san:
Please consider this patch for next Ruby 2.1 release.

```
Index: gc.c
===================================================================
--- gc.c    (revision 46622)
+++ gc.c    (working copy)
@@ -519,6 +519,9 @@
    int parent_object_is_old;

    int need_major_gc;
+
+   size_t last_major_gc;
+
    size_t remembered_shady_object_count;
    size_t remembered_shady_object_limit;
    size_t old_object_count;
@@ -2954,14 +2957,17 @@
          (int)heap->total_slots, (int)heap_pages_swept_slots, (int)heap_pages_min_free_slots);

     if (heap_pages_swept_slots < heap_pages_min_free_slots) {
+#if USE_RGENGC
+   if (objspace->rgengc.during_minor_gc && objspace->profile.count - objspace->rgengc.last_major_gc > 2 /* ma
gic number */) {
+       objspace->rgengc.need_major_gc = GPR_FLAG_MAJOR_BY_NOFREE;
+   }
+   else {
+       heap_set_increment(objspace, (heap_pages_min_free_slots - heap_pages_swept_slots) / HEAP_OBJ_LIMIT);
+       heap_increment(objspace, heap);
+   }
+#else
    heap_set_increment(objspace, (heap_pages_min_free_slots - heap_pages_swept_slots) / HEAP_OBJ_LIMIT);
    heap_increment(objspace, heap);
-
-#if USE_RGENGC
-   if (objspace->rgengc.remembered_shady_object_count + objspace->rgengc.old_object_count > (heap_pages_lengt
h * HEAP_OBJ_LIMIT) / 2) {
-       /* if [old]+[remembered shady] > [all object count]/2, then do major GC */
-       objspace->rgengc.need_major_gc = GPR_FLAG_MAJOR_BY_RESCAN;
-   }
 #endif
     }
```

#12 - 08/21/2014 09:15 AM - ko1 (Koichi Sasada)

I checked benchmark results and no big regression are observed comapre with current edge of Ruby 2.1.
http://www.atdot.net/sp/raw/ywfnan (ruby_2_1a is same as ruby_2_1. to check accuracy (ideally, there should be no difference)).

I understand that the last patch doesn't work. Please try it:
http://www.atdot.net/sp/raw/sabnan

or

```
Index: gc.c
===================================================================
--- gc.c    (revision 47240)
+++ gc.c    (working copy)
@@ -519,6 +519,9 @@
    int parent_object_is_old;
    int need_major_gc;
+
+   size_t last_major_gc;
+
    size_t remembered_shady_object_count;
    size_t remembered_shady_object_limit;
    size_t old_object_count;
@@ -2954,14 +2957,17 @@
          (int)heap->total_slots, (int)heap_pages_swept_slots, (int)heap_pages_min_free_slots);

     if (heap_pages_swept_slots < heap_pages_min_free_slots) {
+#if USE_RGENGC
+   if (objspace->rgengc.during_minor_gc && objspace->profile.count - objspace->rgengc.last_major_gc > 2 /* ma
gic number */) {
+       objspace->rgengc.need_major_gc = GPR_FLAG_MAJOR_BY_NOFREE;
+   }
+   else {
+       heap_set_increment(objspace, (heap_pages_min_free_slots - heap_pages_swept_slots) / HEAP_OBJ_LIMIT);
+       heap_increment(objspace, heap);
+   }
+#else
    heap_set_increment(objspace, (heap_pages_min_free_slots - heap_pages_swept_slots) / HEAP_OBJ_LIMIT);
    heap_increment(objspace, heap);
-
-#if USE_RGENGC
-   if (objspace->rgengc.remembered_shady_object_count + objspace->rgengc.old_object_count > (heap_pages_lengt
h * HEAP_OBJ_LIMIT) / 2) {
-       /* if [old]+[remembered shady] > [all object count]/2, then do major GC */
-       objspace->rgengc.need_major_gc = GPR_FLAG_MAJOR_BY_RESCAN;
-   }
 #endif
     }
```

#13 - 08/21/2014 09:58 PM - normalperson (Eric Wong)

Thanks, I can confirm good results on 2.1.2!

#14 - 08/30/2014 04:32 PM - nagachika (Tomoyuki Chikanaga)

- Backport changed from 1.9.3: UNKNOWN, 2.0.0: UNKNOWN, 2.1: REQUIRED to 1.9.3: UNKNOWN, 2.0.0: UNKNOWN, 2.1: DONE

Thank you ko1 for providing a patch to ruby_2_1. I can apply it cleanly.
Ant thank you eric for your confirmation.

Backported into ruby_2_1 branch at r47326.

#15 - 09/05/2014 05:17 AM - usa (Usaku NAKAMURA)

- Backport changed from 1.9.3: UNKNOWN, 2.0.0: UNKNOWN, 2.1: DONE to 2.0.0: DONTNEED, 2.1: DONE

Files

ruby2_2.JPG	64.4 KB	03/07/2014	ko1 (Koichi Sasada)
ruby_2_2_factor_1_3.JPG	72.4 KB	03/07/2014	ko1 (Koichi Sasada)
proposed.JPG	69.7 KB	03/07/2014	ko1 (Koichi Sasada)
proposed_factor_1_3.JPG	70.3 KB	03/07/2014	ko1 (Koichi Sasada)
gc.patch	7.48 KB	03/07/2014	ko1 (Koichi Sasada)
discourse_benchmark.png	18.4 KB	06/09/2014	ko1 (Koichi Sasada)
young_objects.png	8.17 KB	06/09/2014	ko1 (Koichi Sasada)