Ruby 2.2: GC Improvement Report

Koichi Sasada
ko1@heroku.net
Background
RGenGC from Ruby 2.1: Micro-benchmark

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015
Background
RGenGC from Ruby 2.1: Pause time

Most of case, FASTER 😊

Background
RGenGC from Ruby 2.1: Pause time

Several peaks 😞

Today’s topic

• Use incremental GC algorithm for major GC to reduce long pause time
• Ruby 2.2 has it!!

<table>
<thead>
<tr>
<th></th>
<th>Before Ruby 2.1</th>
<th>Ruby 2.1 RGenGC</th>
<th>Incremental GC</th>
<th>Ruby 2.2 Gen+IncGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Pause time</td>
<td>Long</td>
<td>Long</td>
<td>Short</td>
<td>Short</td>
</tr>
</tbody>
</table>

Goal

Achievements: RGenGC+RincGC

Short pause time 😊

DANGER!!
Highly technical talk!

This is CONCLUSION
Who am I?
Koichi Sasada as a Programmer

• CRuby committer since 2007/01
• Original YARV developer since 2004/01
• From Japan
Who am I?
Koichi Sasada as a Employee
Who am I?
Koichi Sasada as a Employee

• A member of Matz team
  • Full-time CRuby developer
  • Working in Japan
• Mission of our team is to improve “QUALITY” of CRuby interpreter
  • Now we are working on Ruby 2.3 and Ruby 3.0
Who am I?
Koichi Sasada as a member of Matz team

Matz
Boss

Ko1
EDD developer

Nobu
Patch monster

Decide it

Improve it
Fix it

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015
Ruby 2.2
Current stable

http://www.flickr.com/photos/loginesta/5266114104

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015
Ruby 2.2
Syntax

• Symbol key of Hash literal can be quoted

```
{"foo-bar": baz}
#=> {:"foo-bar" => baz}
#=> not{"foo-bar" => baz} like JSON
```

TRAP!!
Easy to misunderstand
(I wrote a wrong code, already...)
Ruby 2.2
Classes and Methods

• Some methods are introduced
  • Kernel#itself
  • String#unicode_normalize
  • Method#curry
  • Binding#receiver
  • Enumerable#slice_after, slice_before
  • File.bithetime
  • Etc.nprocessors
  • ...

Ruby 2.2
Internal changes

• Remove obsolete C-APIs
• Hide internal definitions of data type
Ruby 2.2

Improvements

• Improve GC
  • Symbol GC
  • 4 ages generational GC
  • Incremental GC (today’s topic)

• Improve the performance of keyword parameters

• Use frozen string literals if possible
before = Symbol.all_symbols.size
1_000_000.times{|i| i.to_s.to_sym} # Make 1M symbols
after = Symbol.all_symbols.size; p [before, after]

# Ruby 2.1

#=> [2_378, 1_002_378] # not GCed 😞

# Ruby 2.2

#=> [2_456, 2_456] # GCed! 😊
Ruby 2.2
Symbol GC (cont.)

Memory (object) leak problem
  Symbols has corresponding String objects
  Symbols are collected, but Strings are not collected! (leak)

Ruby 2.2.1 solved this problem!!
  However, 2.2.1 also has problem (rarely you encounter BUG at the end of process [Bug #10933] ← not big issue, I want to believe)
  2.2.2 will solve this problem

TRAP!!
Ruby 2.2.0 has memory leak error!
Ruby 2.2
Fast keyword parameters

“Keyword parameters” introduced from Ruby 2.0 is useful, but slow!!

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015

![Graph showing execution time comparison between `foo6` and `foo_kw6` functions. The bar chart indicates that `foo_kw6` is x30 slower than `foo6` when repeated 10M times.](chart)
Ruby 2.2
Fast keyword parameters

Ruby 2.2 optimizes method dispatch with keyword parameters

Repeat 10M times

foo6(1, 2, 3, 4, 5, 6)  foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)

Execution time (sec)

Ruby 2.1  Ruby 2.2

But still x2 times slower compare with normal dispatch
Break

http://www.flickr.com/photos/donkeyhotey/8422065722

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015
Garbage collection
The automatic memory management

Fig. 109. — A Garbage Collector.
http://www.flickr.com/photos/circasassy/6817999189/

K. Sasada: "Ruby 2.2: GC Improvement Report",
Rubyconf Philippines 2015
History of CRuby’s GC

• 1993/12 Ruby 0.9: Conservative mark and sweep GC
  • Simple algorithm
  • Easy to implement C extensions
• 2011/10 Ruby 1.9.3: Lazy sweep
  • To reduce pause time on sweep
• 2013/02 Ruby 2.0: Bitmap marking
  • To make CoW friendly
• 2013/12 Ruby 2.1: RGenGC
  • To improve throughput
Since birth of Ruby
Simple Mark & Sweep

1. Mark reachable objects from root objects
2. Sweep unmarked objects (collection and de-allocation)

Since Ruby 2.1
RGenGC
• Weak generational hypothesis:
  “Most objects die young”

→ Concentrate reclamation effort only on the young objects

http://www.flickr.com/photos/ell-r-brown/5026593710

Since Ruby 2.1
RGenGC

• Separate young generations and old generations
  • Create objects as youngest generation
  • Promote to old generations after surviving GCs

• Many minor GC and rare major GC
  • Usually, GC on only young space (minor GC)
  • GC on both spaces if no memory (major/full GC)

→ Improve total throughput
Since Ruby 2.1
RGenGC [Minor M&S GC]

1st MinorGC

Root objects

- Mark reachable objects from root objects.
  - Mark and **promote to old generation**
  - Stop traversing after old objects

→ **Reduce mark overhead**

- Sweep not (marked or old) objects
- Can’t collect Some unreachable objects

Don’t collect old object even if it is unreachable.

Since Ruby 2.1
RGenGC [Minor M&S GC]

• Mark reachable objects from root objects.
  • Mark and promote to old generation
  • Assumption: “Old objects only refer old objects”
  • Stop traversing after old objects
  → Reduce mark overhead
  • Sweep not (marked or old) objects
  • Can’t collect Some unreachable objects

2nd MinorGC

Root objects

old

new/free

ignore

old

ignore

old

ignore

old

ignore

old

ignore

old

new/free

old

don’t collect old object even if it is unreachable

"K. S. Sasada: “Ruby 2.2: GC Improvement Report”, RubyconfPhilippines2015"
Since Ruby 2.1
RGenGC [Remember set]

- Assumption: “Old objects only refer old objects”
- However old objects can refer young objects by adding reference from old to new objects
  → Ignore traversal of old object
  → Minor GC causes marking leak!!
    - Because minor GC ignores referenced objects by old objects

Can’t mark new object! (Critical BUG)
Since Ruby 2.1
RGenGC [Remember set]

1. **Detect** creation of an [old->new] type reference
2. Add an [old object] into **Remember set (RSet)** if an old object refer new objects
Since Ruby 2.1
RGenGC [Remember set]

1. Mark reachable objects from root objects
   • **Remembered objects are also root objects**

2. Sweep not (marked or old) objects
Since Ruby 2.1
RGenGC [Write barrier]

• To detect \([\text{old} \rightarrow \text{new}]\) type references, we need to insert "Write-barrier" into interpreter for all "Write" operation

"Write barrier"
[Old->New] type reference detected!
Since Ruby 2.1
RGenGC: Challenge

• Trade-off of Speed and Compatibility
  • Introducing “Write barriers” completely is very hard
  • Can we achieve both speed-up w/ GenGC and keeping compatibility?
Since Ruby 2.1
RGenGC: Key idea

Introduce

WB unprotected objects
Since Ruby 2.1
RGenGC: Key idea

• Separate objects into two types
  • WB protected objects
  • WB unprotected objects

• Decide this type at creation time
  • A class care about WB → WB protected object
  • A class don’t care about WB → WB unprotected object
Since Ruby 2.1
RGenGC: **Key idea**

- Normal objects can be changed to WB unprotected objects
  - “WB unprotect operation”
  - C-exts which don’t care about WB, objects will be WB unprotected objects
- Example
  - `ptr = RARRAY_PTR(ary)`
  - In this case, we can’t insert WB for `ptr` operation, so VM shade “ary”

Now, WB unprotected object **can’t** change into WB p. object

Since Ruby 2.1
RGenGC: Rules

• Treat “WB unprotected objects” correctly
  • At Marking
    1. Don’t promote WB unprotected objects to old objects
    2. Remember WB unprotected objects pointed from old objects
  • At WB unprotect operation for old WB protected objects
    1. Demote objects
    2. Remember this unprotected objects
Since Ruby 2.1
RGenGC: [Minor M&S GC w/WB unp. objects]

Root objects

1st MinorGC

• Mark reachable objects from root objects
  • Mark WB unprotected objects, and *don’t promote* them to old gen objects
  • If WB unprotected objects pointed from old objects, then remember this WB unprotected objects by RSet.

→ Mark WB unprotected objects every minor GC!!

Since Ruby 2.1
RGenGC: [Minor M&S GC w/WB unp. objects]

- Mark reachable objects from root objects
  - Mark WB unprotected objects, and *don’t promote* them to old gen objects
  - If WB unprotected objects pointed from old objects, then **remember this WB unprotected objects** by RSet.

→ Mark WB unprotected objects every minor GC!!
Since Ruby 2.1
RGenGC: [Unprotect operation]

- Anytime Object can give up to keep write barriers
  → [Unprotect operation]
- Change old WB protected objects to WB unprotected objects
  - Example: RARRAY_PTR(ary)
    1. Demote object (old → new)
    2. Register it to Remember Set

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015
RGenGC: Micro-benchmark

K. Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015
RGenGC: Micro-benchmark

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015
RGenGC: Rdoc application

RGenGC: Rdoc application

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>total mark</td>
<td>8210.724461</td>
</tr>
<tr>
<td>total sweep</td>
<td>3524.425873</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>1049.27184</td>
</tr>
<tr>
<td>RGenGC</td>
<td>3579.584833</td>
</tr>
</tbody>
</table>

RGenGC: Rdoc application

RGenGC: Rdoc application

RGenGC: Rdoc application

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015
Since Ruby 2.1
RGenGC timing chart

2.0.0 GC (M&S w/lazy sweep)

w/RGenGC (Minor GC)

w/RGenGC (Major GC)

RGenGC: Rdoc application (again)

Issue of RGenGC: Long pause time

😊 RGenGC achieves *high throughput*
😊 Minor GC stops only *short pause time*
😊 Major GC still stops *long pause time*

→ *Introducing Incremental GC for major GC*

<table>
<thead>
<tr>
<th></th>
<th>Generational GC</th>
<th>Incremental GC</th>
<th>Gen+Inc GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Pause time</td>
<td>Long</td>
<td>Short</td>
<td>Short</td>
</tr>
</tbody>
</table>

RincGC: Restricted Incremental GC algorithms

RincGC algorithm is implemented for Ruby 2.2.

Incremental GC
Well-known GC algorithm to reduce pause time

• Do GC steps incrementally
  • Interleaving with Ruby’s execution (mutator) and GC process.
  • Lazy sweep is part of an incremental GC

STW GC
Ruby Mark Sweep Sweep Sweep
Stop the (Ruby) World

Inc GC
Ruby Mark Mark Mark Mark
Sweep Sweep Sweep Sweep

😊 Short pause time
😊 No total time change

Terminology: Tri-color GC

• Define three colors for objects
  • White objects is not traversed objects
  • Grey objects are marking objects
  • Black objects are marked objects
1. Color all objects “white”
2. Grey root objects
3. Choose a grey object and grey all reachable white objects, and black the chosen object (incremental marking)
4. Finish marking when no grey objects
5. Sweep white objects as *unmarked* objects

Incremental GC

Root objects

Collect white objects

Incremental GC requires WBs

Root objects

Mutator can add reference from a Black object to a white object!!
→ Mark miss!
No more traversal from black objects

Incremental GC requires WBs

Root objects

Use write barrier to detect an addition of references from black objects to white objects, and grey black objects

RincGC: Restricted Incremental GC using WB-unprotected objects

• Use WB unprotected objects like RGenGC
• Introducing a new rule: “Scan all black WB unprotected objects at the end of incremental GC at once”
  • WB unprotected objects can point white objects
  • Scan from all (“Black” and “WB unprotected objects”) at once (stop the world marking)
1. Color all objects “white”
2. Grey root objects
3. Choose a grey object and grey reachable white objects, and black the chosen object (incremental marking)
4. Finish marking when no grey objects
5. **Scan all black WB unprotected objects at once**
6. Sweep white objects as *unmarked* objects
RincGC: Discussion

• Long pause time than usual incremental GC step
  • This technique can introduce long pause time, relative to the number of WB unprotected objects at last. This is why this algorithm is named “Restricted”
• No problem, on most of cases
  • Similar/shorter pause time than “Minor GC” of RGenGC.

Implementation
Ruby’s implementation
State chart

Ruby program (mutators) → newobj() → garbage_collect()
  if (no pages)
    if (incremental_marking)
      marks_continue()
    else
      marks_start()
  else
    marks_step()
  if (sweep_pages)
    sweep_continue()
  else
    sweep_start()

State: marking → marks_step() → marks_finish()

State: sweeping → sweep_step() → sweep_finish()

State: none

Direct transition
To mutator

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015
Ruby’s implementation
WB protected/unprotected

• Make popular class instances WB protected
  • String, Array, Hash, and so on

• Implement “unprotect operation” for Array and so on

• Remain WB unprotected objects
  • Difficult to insert WBs: a part of Module, Proc (local variables) and so on.
  • Minor features
Ruby’s implementation
Data structure

- Introduce 2 bits age information to represent young and old objects
  - Age 0, 1, 2 is young object
  - Age 3 is old object
  - Surviving 1 GC increase one age
- Add 3 bits information for each objects (we already have mark bit)
  - WB unprotected bit
  - Long lived bit (old objects or remembered WB unprotected objects)
  - Remembered old object bit / Marking (Grey) bit
  - They can share 1 bit field because the former is used only at minor GC and the latter is used only at major GC (incremental GC)
Ruby’s implementation
Bitmap technique

• Each bits are managed by bitmap technique
  • Easy to manage remember set
  • Fast traversing
  • Easy to get a set
    • Remember set: (Remembered old object bitmap) | (Long lived bitmap & WB unp. Bitmap)
    • Living unprotected objects: (mark bitmap & WB unprotected bitmap)
RincGC: Evaluation

• Measure pause times for
  <https://github.com/tenderlove/ko1-test-app> by Aaron Patterson
Evaluation

K.Sasada: "Ruby 2.2: GC Improvement Report",
RubyconfPhilippines2015

65
Evaluation

2,500 GC's pause time since one major GC

- A long pause time of Major GC marking on RGenGC
- Short pause times of Incremental marking of Major GC on RincGC
- Lazy sweep pauses only short time
- Same pause time for Minor marking time
Evaluation

Maximum pause time

<table>
<thead>
<tr>
<th></th>
<th>rgengc</th>
<th>rincgc</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>0.015923438</td>
<td>0.004662491</td>
</tr>
</tbody>
</table>
NOTE: Incremental GC is not a silver bullet

• Incremental GC does not guarantee improving your application’s response time
  • Incremental GC does not reduce total GC time, so that a big task includes several major GC doesn’t improve its response time.
• Check GC counts with GC.stat(:major_gc_count) and GC.stat(:minor_gc_count) for each request.
References

• **Incremental Garbage Collection in Ruby 2.2**

• **Ruby Under a Microscope**
  - November 2013 by Pat Shaughnessy
  - [http://patshaughnessy.net/ruby-under-a-microscope](http://patshaughnessy.net/ruby-under-a-microscope)
Summary

- Introducing incremental GC algorithm into major GC to reduce long pause time
- Ruby 2.2 has it!!

<table>
<thead>
<tr>
<th></th>
<th>Before Ruby 2.1</th>
<th>Ruby 2.1 RGenGC</th>
<th>Incremental GC</th>
<th>Ruby 2.2 Gen+IncGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Pause time</td>
<td>Long</td>
<td>Long</td>
<td>Short</td>
<td>Small</td>
</tr>
</tbody>
</table>

Goal

Thank you for your attention

Koichi Sasada
<ko1@heroku.com>