Incremental GC
for Ruby interpreter

Koichi Sasada
ko1@heroku.net
2014

Very important year for me
10th Anniversary
10th Anniversary

YARV development (2004/01-)
First presentation at RubyConf 2004
Garbage Collection Improvements

Good throughput and short pause time

Ruby 2.2 will be released soon.

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
RGenGC: Micro-benchmark

K. Sasada: Incremental GC for Ruby interpreter, RubyConf2014
RGenGC: Pause time
Today’s topic

• Use incremental GC algorithm for major GC to reduce long pause time
• Ruby 2.2 will have 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

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
Achievements: RGenGC+RincGC
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
Upcoming Ruby 2.2

What’s next?

http://www.flickr.com/photos/adafruit/8483990604

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
Ruby 2.2
Syntax

• No notable changes (maybe)

• Symbol key of Hash literal can be quoted
  • {“foo-bar”: baz} #=> {:"foo-bar" => baz}
  #=> not {“foo-bar” => baz} like JSON

TRAP: easy to misunderstand
Ruby 2.2
Classes and Methods

• Some methods are introduced
  • Kernel#itself
  • String#unicode_normalize
  • 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
Ruby 2.2
Symbol GC

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 (dev)

=> [2_456, 2_456] # GCed! 😊
Ruby 2.2
Fast keyword parameters

# time of this program

def foo(k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil)
10_000_000.times{foo(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)}

![Graph showing execution time comparison before and after optimization]

x15 faster!

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
Break

http://www.flickr.com/photos/donkeyhotey/8422065722
Garbage collection
The automatic memory management

http://www.flickr.com/photos/circasassy/6817999189/
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

K. Sasada: Incremental GC for Ruby interpreter, RubyConf2014
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]

• 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.

K. Sasada: Incremental GC for Ruby interpreter, RubyConf2014
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

K. Sasada, Incremental GC for Ruby interpreter, RubyConf2014
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!**
→ **Sweeping living 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
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 [old→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]

- 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]

Root 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
RGenGC: Micro-benchmark

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
RGenGC: Micro-benchmark

K. Sasada: Incremental GC for Ruby interpreter, RubyConf2014
RGenGC: Rdoc application
RGenGC: Rdoc application

<table>
<thead>
<tr>
<th></th>
<th>Total Mark</th>
<th>Total Sweep</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>8210.724461</td>
<td>3524.425873</td>
</tr>
<tr>
<td>RGenGC</td>
<td>1049.27184</td>
<td>3579.584833</td>
</tr>
</tbody>
</table>

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
RGenGC: Rdoc application

<table>
<thead>
<tr>
<th></th>
<th>no</th>
<th>RGenGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>total sweep</td>
<td>3524.425873</td>
<td>3579.584833</td>
</tr>
<tr>
<td>total mark</td>
<td>8210.724461</td>
<td>1049.27184</td>
</tr>
<tr>
<td>others</td>
<td>100907.4225</td>
<td>100640.8472</td>
</tr>
</tbody>
</table>

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
RGenGC: Rdoc application

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
RGenGC: Rdoc application

K. Sasada: Incremental GC for Ruby interpreter, RubyConf2014
Since Ruby 2.1
RGenGC timing chart

2.0.0 GC (M&S w/lazy sweep)

w/RGenGC (Minor GC)

w/RGenGC (Major GC)

Stop the (Ruby) World

Mark

Sweep

Sweep

Sweep

Sweep

Sweep

Sweep

Sweep

Sweep

Sweep

Sweep

Sweep

Sweep

Sweep

Sweep

Mark

Sweep

Sweep

Sweep

Sweep

Sweep

Stop the (Ruby) World

K. Sasada: Incremental GC for Ruby interpreter, RubyConf2014
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><strong>Throughput</strong></td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td><strong>Pause time</strong></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.

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
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

🎉 Short pause time 😞 No total time change

K. Sasada: Incremental GC for Ruby interpreter, RubyConf2014
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 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)
RincGC
Restricted Incremental GC using WB-unprotected objects

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”
  - Similar/shorter pause time than “Minor GC” of RGenGC.
Implementation
Ruby's implementation
State chart

Ruby program (mutators)

garbage_collect()

if (no pages)

marks_start()

State: marking

marks_step()

marks_finish()

sweep_start()

State: sweeping

sweep_step()

sweep_continue()

State: none

newobj()

if (incremental_marking)

if (sweep_pages)

marks_continue()

sweep_continue()

Direct transition
To mutator

State: none
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
Evaluation

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

- Long major marking time of RGenGC
- Incremental Major GC of RincGC
- Lazy sweep pauses only short time
- Same pause time for Minor marking time

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
Evaluation

maximum pause time

<table>
<thead>
<tr>
<th></th>
<th>maximum pause time</th>
</tr>
</thead>
<tbody>
<tr>
<td>rgengc</td>
<td>0.015923438</td>
</tr>
<tr>
<td>rincgc</td>
<td>0.004662491</td>
</tr>
</tbody>
</table>

K.Sasada: Incremental GC for Ruby interpreter, RubyConf2014
NOTE: Incremental GC is not 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.
Summary

• Introducing incremental GC algorithm into major GC to reduce long pause time
• Ruby 2.2 will have 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>
Thank you for your attention

Koichi Sasada
<ko1@heroku.com>