Object management on Ruby 2.1

Koichi Sasada
Heroku, Inc.
ko1@heroku.com
Summary of this talk

• Ruby 2.1.0 will be released soon!
  • 2013/12/25
  • Some new features and internal performance improvements
• Rewrite object management to improve performance
  • “gc.c” → 3414 insertions, 1121 deletions
  • Allocation/Deletion trace mechanism
  • Introduce generational mechanism
  • Tuning GC parameters
  • Optimize object allocation path
  • Refactoring (terminology, method path, etc)
Who am I?

- Koichi Sasada a.k.a. ko1
- From Japan
- 笹田 (family) 耕一 (given) in Kanji character
  - “Ichi” (Kanji character “一”) means “1” or first
  - This naming rule represents I’m the first son of my parents
  - Ko”ichi” → ko1
Who am I?

- Koichi Sasada a.k.a. ko1
- Matz team at Heroku, Inc.
  - Full-time CRuby developer
  - Working in Japan
- CRuby/MRI committer
  - Virtual machine (YARV) from Ruby 1.9
  - YARV development since 2004/1/1
- Director of Ruby Association
• Foundation to encourage Ruby developments and communities
  • Chairman is Matz
  • Located at Matsue-city, Japan

• Activities
  • Maintenance of Ruby (Cruby) interpreter
    • Now, it is for Ruby 1.9.3
    • Ruby 2.0.0 in the future?
  • Events, especially RubyWorld Conference
  • Ruby Prize
    • 3 nominates
  • Grant. We have selected 3 proposals in 2013
    • Win32Utils Support, Conductor, Smalruby - smalruby-editor
  • Making an appeal for contribution
• Heroku, Inc. <http://www.heroku.com>
  • You should know about Heroku!

• Heroku supports Ruby development
  • Many talents for Ruby (and also other languages)
  • Especially Heroku employs three Ruby interpreter core developers
    • Matz
    • Nobu
    • Ko1 (me)

• We name our group “Matz team”
Mission of Matz team

• Improve quality of next version of CRuby
  • Matz decides a spec finally
  • Nobu fixed (huge number of) bugs
  • Ko1 improves the performance

Current target is “Ruby 2.1”
Ruby 2.1
Next version
Ruby 2.1 release plan announcement

“I, Naruse, take over the release manager of Ruby 2.1.0 from mame. **Ruby 2.1.0 is planed to release in 2013-12-25.** I’m planning to call for feature proposals soon like 2.0.0 [ruby-core:45474], so if you have a suggestion you should begin preparing the proposal.”

- [ruby-core:54726] Announce take over the release manager of Ruby 2.1.0

by NARUSE, Yui
2013/12/25!

http://www.flickr.com/photos/htakashi/5285103341/ by Takashi Hososhima
Object management on Ruby 2.1
by Koichi Sasada at RubyConf2013
Ruby 2.1 schedule

2013/02
Ruby 2.0.0

We are here!

2013/12/25
Ruby 2.1.0

RubyKaigi2013
5/30, 31, 6/1

Euruko2013
6/28, 29

RubyConf2013
11/8-10

Events are important for EDD (Event Driven Development) Developers

Object management on Ruby 2.1 by Koichi Sasada at RubyConf2013
Ruby 2.1 schedule (more)

- **2013/06**: Call for Feature Proposal (CFP)
- **2013/07**: Dev-meeting w/ Matz
- **2013/09**: Feature freeze
- **2013/10**: Preview1
- **2013/11**: Preview2
- **2013/12**: RC
- **2013/12/25**: Ruby 2.1.0

We are here!

Ruby 2.1

New features
Now, much smaller than Ruby 2.0

Object management on Ruby 2.1 by Koichi Sasada at RubyConf2013
Ruby 2.1 new features

• New Numeric syntax (1/2r => Rational(1, 2), etc.)
• “def” returns a symbol of method name
• Refine features introduced from Ruby 2.0
  • Keyword arguments
  • Refinements
  • Module#prepend
• New methods
  • Refine m17n introduced from Ruby 1.9
    • String#scrub, String#scrub!
    • Verify and fix invalid byte sequence.
  • Enumerable#to_h
• Frozen objects
  • All symbols
  • Frozen string is discussed now
Ruby 2.1 Internal improvements

- Profiling supports
  - Additional internal hooks for object allocation and deallocation
  - Profiling API
- More sophisticated garbage collection
  - RGenGC: Introduce generational GC into CRuby
  - GC Parameter tuning
  - Other tuning
- More sophisticated method caching
- Bignum/Integer improvements
- ...

Today’s topic
Object management
New Profiling support

• Internal hooks for object management
• Profiling API to get backtrace information without huge overhead
Internal hooks for object management

What’s nice?

• You can collect more detailed analysis

• Examples
  • Collect object allocation site information
  • Collect usage of allocated objects
  • Measure GC performance from outside
Internal hooks for object management

• 4 added events
  • RUBY_INTERNAL_EVENT_NEWOBJ
    • When object is created
  • RUBY_INTERNAL_EVENT_FREEOBJ
    • When object is freed
  • RUBY_INTERNAL_EVENT_GC_START
    • When GC is started
  • RUBY_INTERNAL_EVENT_GC_END
    • When GC is finished

Object management on Ruby 2.1
by Koichi Sasada at RubyConf2013
Internal hooks for object management

*Caution*

- You can *NOT* trace these events using TracePoint (introduced from 2.0)
- You need to write C-ext to use them, because events are invoked during GC, etc
- Use new “postponed job” API
Internal hooks for object management
Sample feature using new hooks

• **ObjectSpace. trace_object_allocations**
  • Trace object allocation and record allocation-site
    • Record filename, line number, creator method’s id and class
  • Usage:
    ObjectSpace.trace_object_allocations{ # record only in the block
      o = Object.new
      file = ObjectSpace.allocation_sourcefile(o) #=> __FILE__
      line = ObjectSpace.allocation_sourceline(o) #=> __LINE__ -2
    }

• Demonstration
Ruby 2.1
To be more sophisticated object management
Better Object management

• Refactoring object management code
  • Object management code is in “gc.c” in CRuby
  • I have rewritten (am rewriting) gc.c many parts
  • “gc.c” → 3414 insertions, 1121 deletions

• GC parameter tuning

• New GC algorithm called “RGenGC”
  • Generational garbage collection
  • Keep compatibility and performance
GC parameter tuning
Introduce only one case
GC parameter tuning

• When GC occur?
  (1) There are no slot to allocate object
  (2) Exceed threshold of memory allocation

• (1) is easy to understand
• Introduce (2) more details
GC parameter tuning
malloc_increase and malloc_limit

• Memory allocation → GC
  • Every time allocate “n” size memory (call malloc(n))
    increase “malloc_increase” with “n”
  • If malloc_increase > malloc_limit, then cause GC

• Default parameter of “malloc_limit” is “8MB”!!
  • Too small!!
    • String read from 8MB file
    • An array which has 1M entry on 64bit CPU
  • I ask Matz “why such small value?”
  • His reply is “I used 10MB machine at 20 years old”
GC parameter tuning
Dynamic tuning of “malloc_limit”

• Default: 8MB → 16MB

• Adaptive tuning
  • If “malloc_increase” exceeds “malloc_limit”, then increase “malloc_limit”
    • Increase “malloc_limit” by a factor of environment variable “GC_MALLOC_LIMIT_GROWTH_FACTOR” (default is 1.4)
    • Maximum value of “malloc_limit” can be set with environment variable “GC_MALLOC_LIMIT_MAX” (default is 32MB)
  • If “malloc_increase” doesn’t exceed limit, then decrease “malloc_limit”
GC parameter tuning
Dynamic tuning of “malloc_limit”

Grow limit aggressively on Ruby 2.1

Small limit cause many GCs
Introduce Generational GC into CRuby
RGenGC: Summary

- **RGenGC: Restricted Generational GC**
  - New generational GC algorithm allows mixing “Write-barrier protected objects” and “WB unprotected objects”
  - **No (mostly) compatibility issue** with C-exts

- Inserting WBs gradually
  - We can concentrate WB insertion efforts for major objects and major methods
  - Now, **Array, String, Hash, Object, Numeric** objects are WB protected
    - Array, Hash, Object, String objects are very popular in Ruby
    - Array objects using `RARRAY_PTR()` change to WB unprotected objects (called as Shady objects), so existing codes still works.
RGenGC: Previous talk

• Algorithm is introduced at
  • RubyKaigi2013
  • Euruko2013

• See also these slides/movie for details
RGenGC: Agenda

• Background
  • Generational GC
  • Ruby’s GC strategy

• Proposal: RGenGC
  • Separating into normal objects and shady objects
  • Shady objects at marking
  • Shade operation

• Implementation
RGenGC: Background
Current CRuby’s GC

• Mark & Sweep
  • Conservative
  • Lazy sweep
  • Bitmap marking
  • Non-recursive marking

• C-friendly strategy
  • Don’t need magical macros in C source codes
  • Many many C-extensions under this strategy
RGenGC
Restriction of CRuby’s GC

1. Because of “C-friendly” strategy:
   • We can’t know object relation changing timing
   • We can’t use “Moving GC algorithm” (such as copying/compacting)

2. Because of “Object data structure”:
   • We can’t measure exact memory consumption
   • Based on assumption: “malloc” library may be smarter than our hack
     • We rely on “malloc” library for memory allocations
     • GC only manage “object” allocation/deallocation
RGenGC: Background

Mark & Sweep

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

Collect unreachable objects

Object management on Ruby 2.1
by Koichi Sasada at RubyConf2013
RGenGC: Background
Generational GC (GenGC)

• Weak generational hypothesis:
  “Most objects die young”

→ Concentrate reclamation effort only on the young objects
RGenGC: Background
Generational hypothesis

Object lifetime in RDoc
(How many GCs surviving?)

95% of objects dead by the first GC
RGenGC: Background
Generational hypothesis

Object lifetime in RDoc
(How many GCs survive?)

Some type of objects (like Class) has long lifetime

Object management on Ruby 2.1
by Koichi Sasada at RubyConf2013
RGenGC: Background
Generational GC (GenGC)

• Separate young generation and old generation
  • Create objects as young generation
  • Promote to old generation after surviving $n$-th GC
  • In CRuby, $n == 1$ (after 1 GC, objects become old)

• Usually, GC on young space (minor GC)
• GC on both spaces if no memory (major/full GC)
RGenGC: Background
Generational GC (GenGC)

• Minor GC and Major GC can use different GC algorithm
  • Popular combination is:
    Minor GC: Copy GC, Major GC: M&S
  • On the CRuby, we choose:
    Minor GC: M&S, Major GC: M&S
  • Because of CRuby’s restriction (we can’t use moving algorithm)
RGenGC: Background: GenGC

[Minor M&S GC]

1\(^{st}\) 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.
RGenGC: Background: GenGC
[Minor M&S GC]

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.
RGenGC: Background: GenGC

[Major M&S GC]

- Normal M&S
- Mark reachable objects from root objects
  - Mark and **promote to old gen**
- Sweep unmarked objects

**Sweep all unreachable (unused) objects**
RGenGC: Background: GenGC
Problem: mark miss

- Old objects refer young 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)
RGenGC: Background: GenGC
Introduce Remember set (Rset)

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
RGenGC: Background: GenGC
[Minor M&S GC] w/ RSet

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

2. Sweep not (marked or old) objects
RGenGC: Background: GenGC

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!
RGenGC
Back to Ruby’s specific issue
RGenGC: CRuby’s case
Write barriers in Ruby

• Write barrier (WB) example in Ruby world
  • (Ruby) old_ary[0] = new0 # [old_ary → new0]
  • (Ruby) old_obj.foo = new1 # [old_obj → new1]
RGenGC: CRuby's case
Difficulty of inserting write barriers

• To introduce generational garbage collector, WBs are necessary to detect [old→new] type reference

• “Write-barrier miss” causes terrible failure
  1. WB miss
  2. Remember-set registration miss
  3. (minor GC) marking-miss
  4. Collect live object → Terrible GC BUG!!
RGenGC: Problem
Inserting WBs into C-extensions (C-exts)

• All of C-extensions need perfect Write-barriers
  • C-exts manipulate objects with Ruby’s C API
  • C-level WBs are needed

• Problem: How to insert WBs into C-exts?
  • There are many WB required programs in C-exts
    • Example (C): RARRAY_PTR(old0)[0] = new1
    • Ruby C-API doesn’t require WB before
  • CRuby interpreter itself also uses C-APIs

• How to deal with?
  • We can rewrite all of source code of CRuby interpreter to add WB, with huge debugging effort!!
  • We can’t rewrite all of C-exts which are written by 3rd party
**RGenGC: Problem**

Inserting WBs into C-extensions (C-exts)

Two options

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Give up GenGC</td>
<td>Poor</td>
</tr>
<tr>
<td>2</td>
<td>GenGC with re-writing all of C exts</td>
<td>Good</td>
</tr>
</tbody>
</table>

Trade-off of Speed and Compatibility

2.0 and earlier conservative choice

Object management on Ruby 2.1 by Koichi Sasada at RubyConf2013
RGenGC: Challenge

• Trade-off of Speed and Compatibility
  • Can we achieve both speed-up w/ GenGC and keeping compatibility?

• Several possible approaches
  • Separate heaps into the WB world and non-WB world
    • Need to re-write whole of Ruby interpreter
    • Need huge development effort
  • WB auto-insertion
    • Modify C-compiler
    • Need huge development effort
RGenGC: Our approach

• Create **new generational GC algorithm** permits WB protected objects **AND** WB un-protected object in the same heap

RGenGC: Restricted Generational Garbage Collection
## RGenGC: Invent 3rd option

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Give up GenGC</td>
<td>Poor</td>
</tr>
<tr>
<td>2</td>
<td>GenGC with re-writing all of C codes</td>
<td>Good</td>
</tr>
<tr>
<td>3</td>
<td>Use new RGenGC</td>
<td>Good</td>
</tr>
</tbody>
</table>

Breaking the trade off. You can praise us!!

Object management on Ruby 2.1 by Koichi Sasada at RubyConf2013
RGenGC: Key idea

• Introduce **Shady object**
  • I use the word “Shady” as questionable, doubtful, ...
  • Something feeling dark
  • 日陰者, in Japanese
RGenGC: Key Idea

• Separate objects into two types
  • Normal Object: WB Protected
  • Shady Object: WB Unprotected

• We are not sure that a shady object points new objects or not

• Decide this type at creation time
  • A class care about WB → Normal object
  • A class don’t care about WB → Shady object
RGenGC: Key Idea

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

Now, Shady object can’t change into normal object
RGenGC

Key Idea: Rule

• Treat “Shady objects” correctly
  • At Marking
    1. Don’t promote shady objects to old objects
    2. Remember shady objects pointed from old objects
  • At Shade operation for old normal objects
    1. Demote objects
    2. Remember shaded shady objects
RGenGC

[Minor M&S GC w/Shady object]

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

→ Mark shady objects every minor GC!!
RGenGC
[Minor M&S GC w/Shady object]

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

→ Mark shady objects every minor GC!!
RGenGC
[Shade operation]

• Anytime Object can give up to keep write barriers
  → [Shade operation]
• Change old normal objects to shade objects
  • Example: RARRAY_PTR(ary)
    (1) Demote object (old → new)
    (2) Register it to Remember Set
RGenGC
Timing chart

2.0.0 GC (M&S w/lazy sweep)

w/RGenGC (Minor GC)

• Shorter mark time (good)
• Same sweep time (not good)
• (little) Longer execution time b/c WB (bad)
RGenGC
Number of objects

2.0.0 GC (M&S)

# of Living objects  # of Freed objects

w/RGenGC (Minor GC)

# of Living objects  # of Freed objects

# of old objects (#old)  # of new objects (#new)  # of freed but remembered objects

(a) # of old objects by WB  (b) # of shady objects pointed by old
(c) # of old but shady objects

Object management on Ruby 2.1
by Koichi Sasada at RubyConf2013
Object management on Ruby 2.1
by Koichi Sasada at RubyConf2013

RGenGC
Number of objects

<table>
<thead>
<tr>
<th></th>
<th>Marking space</th>
<th>Number of unused, uncollected objs</th>
<th>Sweeping space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark&amp;Sweep GC</td>
<td># of Living objects</td>
<td>0</td>
<td>Full heap</td>
</tr>
<tr>
<td>Traditional GenGC</td>
<td>#new + (a)</td>
<td>(a)</td>
<td>#new</td>
</tr>
<tr>
<td>RGenGC</td>
<td>#new + (a) + (b) + (c)</td>
<td>(a) + (b)</td>
<td>Full heap</td>
</tr>
</tbody>
</table>

(a) # of old objects by WB
(b) # of shady objects pointed by old
(c) # of old but shady objects
RGenGC
Discussion: Pros. and Cons.

• Pros.
  • Allow WB unprotected objects (shady objects)
    • 100% compatible w/ existing extensions which don’t care about WB
    • A part of CRuby interpreter which doesn’t care about WB
  • Inserting WBs step by step, and increase performance gradually
    • We don’t need to insert all WBs into interpreter core at a time
    • We can concentrate into popular (effective) classes/methods.
    • We can ignore minor classes/methods.
  • Simple algorithm, easy to develop (already done!)
RGenGC
Discussion: Pros. and Cons.

• Cons.
  • Increasing “unused, but not collected objects until full/major GC
    • Remembered normal objects (caused by traditional GenGC algorithm)
    • Remembered shady objects (caused by RGenGC algorithm)
  • WB insertion bugs (GC development issue)
    • RGenGC permit shady objects, but sunny objects need correct/perfect WBs. But inserting correct/perfect WBs is difficult.
    • This issue is out of scope. We have another idea against this problem (out of scope).
  • Can’t reduce Sweeping time
    • But many (and easy) well-known techniques to reduce sweeping time (out of scope).
  • Increase complexity
    • Additional tuning parameters
RGenGC
Implementation: WB support status

<table>
<thead>
<tr>
<th>Type name</th>
<th>Status</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_OBJECT</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>T_CLASS</td>
<td>Supported</td>
<td>Possible to change into shady</td>
</tr>
<tr>
<td>T_ICLASS</td>
<td>Supported</td>
<td>Possible to change into shady</td>
</tr>
<tr>
<td>T_MODULE</td>
<td>Supported</td>
<td>Possible to change into shady</td>
</tr>
<tr>
<td>T_FLOAT</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>T_STRING</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>T_REGEXP</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>T_ARRAY</td>
<td>Supported</td>
<td>Possible to change into shady / more efforts are needed</td>
</tr>
<tr>
<td>T_HASH</td>
<td>Supported</td>
<td>Possible to change into shady</td>
</tr>
<tr>
<td>T_STRUCT</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>T_BIGNUM</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>T_FILE</td>
<td>Unsupported</td>
<td>Not yet</td>
</tr>
<tr>
<td>T_DATA</td>
<td>Supported</td>
<td>Only InstructionSequence objects are supported</td>
</tr>
<tr>
<td>T_MATCH</td>
<td>Unsupported</td>
<td>Most of MatchData objects are short-lived</td>
</tr>
<tr>
<td>T_RATIONAL</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>T_COMPLEX</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>T_NODE</td>
<td>Unsupported</td>
<td>Most of Node objects are short-lived</td>
</tr>
</tbody>
</table>
RGenGC
Implementation
• Introduce two flags into RBasic
  • FL_KEEP_WB: WB protected or not protected
    • 0 → unprotected → Shady object
    • 1 → protected → Sunny object
    • Usage: NEWOBJ_OF(ary, struct RArray, klass, T_ARRAY | FL_KEEP_WB);
  • FL_PROMOTED: Promoted or not
    • 0 → Young gen
    • 1 → Old gen
    • Don’t need to touch by user program
• Remember set is represented by bitmaps
  • Same as marking bitmap
  • heap_slot::rememberset_bits
  • Traverse all object area with this bitmap at first
RGenGC
Implementation: WB operation API

• OBJ_WRITE(a, &a->x, b)
  • Declare ‘a’ aggregates ‘b’
  • Write: *&a->x = b
  • Write barrier
  • OBJ_WRITE(a, b) returns “a”

• OBJ_WRITTEN(a, oldv, b)
  • Declare ‘a’ aggregates ‘b’ and old value is ‘oldv’
  • Non-write operation
  • Write barrier
RGenGC
Implementation: WB operation API

• T_ARRAY
  • \texttt{RARRAY\_PTR(ary)} causes shade operation
    • Can’t get RGenGC performance improvement
    • But works well 😊 (Do not need to modify codes)

  • Instead of \texttt{RARRAY\_PTR(ary)}, use alternatives
    • \texttt{RARRAY\_AREF(ary, n)} → \texttt{RARRAY\_PTR(ary)[n]}
    • \texttt{RARRAY\_ASET(ary, n, obj)} → \texttt{RARRAY\_PTR(ary)[n] = obj w/ Write-barrier}
    • \texttt{RARRAY\_PTR\_USE(ary, ptrname, {...block...})}
      • Only in block, pointers can be accessed by `ptrname’ variable (VALUE*).
      • \textbf{Programmers need to insert collect WBs (miss causes BUG).}

Important!!

Object management on Ruby 2.1
by Koichi Sasada at RubyConf2013
RGenGC
Incompatibility

• Make RBasic::klass “const”
  • Need WBs for a reference from an object to a klass.
  • Only few cases (zero-clear and restore it)
• Provide alternative APIs
  • Now, RBASIC_SET_CLASS(obj, klass) and
    RBASIC_CLEAR_CLASS(obj) is added. But they should be
    internal APIs (removed soon).
  • rb_obj_hide() and rb_obj_reveal() is provided.
RGenGC
Performance evaluation

• Ideal micro-benchmark for RGenGC
  • Create many old objects at first
  • Many new objects (many minor GC, no major GC)

• RDoc
  • Same “make doc” task from trunk
RGenGC
Performance evaluation (micro)

- Shorter mark time (good)
- Same sweep time (not good)

Object management on Ruby 2.1 by Koichi Sasada at RubyConf2013
RGenGC
Performance evaluation (RDoc)

Compare with M&S and RGenGC

- **Faster minor GC**
- **Major/full GC peaks**

* Disabled lazy sweep to measure correctly.
RGenGC
Performance evaluation (RDoc)

About x15 speedup!

* Disabled lazy sweep to measure correctly.

Object management on Ruby 2.1
by Koichi Sasada at RubyConf2013
RGenGC
Performance evaluation (RDoc)

* 12% improvements compare with w/ and w/o RGenGC
* Disabled lazy sweep to measure correctly.

<table>
<thead>
<tr>
<th>Total execution time (sec)</th>
<th>w/o RGenGC</th>
<th>RGenGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other than GC</td>
<td>103.7627479</td>
<td>102.3799865</td>
</tr>
<tr>
<td>GC</td>
<td>16.04393815</td>
<td>4.946003494</td>
</tr>
</tbody>
</table>

* 12% improvements compare with w/ and w/o RGenGC
* Disabled lazy sweep to measure correctly.
RGenGC: Summary

• RGenGC: Restricted Generational GC
  • New GC algorithm allow mixing “Write-barrier protected objects” and “WB unprotected objects”
  • (mostly) **No compatibility issue** with C-exts

• Inserting WBs gradually
  • We can concentrate WB insertion efforts for major objects and major methods
RGenGC
Remaining task

• **Reduce old objects**
  • Short lived objects promotion old-gen accidentally is harmful

• **Minor GC / Major GC timing tuning**
  • Too many major GC → slow down
  • Too few major GC → memory consumption issue

• **Inserting WBs w/ application profiling**
  • Profiling system
  • Benchmark programs

• **Debugging/Detecting system for WBs bugs**

• **Improve sweeping performance**
Remaining task

• To reduce old objects
  → 3 Generations GC (3GenGC)

• Minor GC / Major GC timing tuning
  → Count oldgen objects count
  → Estimate oldgen space
Three generational GC (3GenGC) Problem

• RGenGC introduces two generation “Young” and “Old”
• Some “short-lived” young object will be promoted as old-gen accidentally
  • Ex: loop{a = Object.new; b = Object.new}
• If such “short-lived” objects consumes huge memory, we need to free such objects
  • Ex: loop{Array.new(1_000_000)} # 1M entries
Three generational GC (3GenGC)

Idea

• Add new generation “Infant” generation
  • Before: Young → Old
  • After 3gen GC: Infant → Young → Old

• Most of objects died in infant, and also young gen
  • Good: Avoid short-lived old-gen objects
  • Good: Reduce full-GC timing
  • Bad: Some overhead

• We implemented this feature and evaluating it now
Estimated oldgen space

Problem

• We can not measure how oldgen objects consume memories collectly
  • A few oldgen objects can grab huge memory
  • Major GC takes long time

• Trade-off between time and memory usage
Estimate oldspace

Idea

• Estimate how much memory old-gen objects consumes

• Invoke full GC when estimation exceed the threshold
  • Good: More correct major GC timing
  • Bad: Some overhead to measure memory size
  • Bad: More tuning parameters

• We implemented this feature and evaluating it now
Future work
For smarter object management

• Need more tuning
  • Some program slower than Ruby 2.0.0
  • We need practical benchmarks (other than RDoc)

• Parallel marking/sweeping
  • Parallel sweeping is already implemented
    <https://github.com/ko1/ruby/tree/parallel_sweep>,
    however, it does not improve performance...

• Concurrent marking to reduce full marking stop
time
Summary of this talk

• Ruby 2.1.0 will be released soon!
  • 2013/12/25
  • Some new features and internal performance improvements

• Rewrite object management to improve performance
  • “gc.c” → 3414 insertions, 1121 deletions
  • Allocation/Deletion trace mechanism
  • Introduce generational mechanism
  • Tuning GC parameters
  • Optimize object allocation path
  • Refactoring (terminology, method path, etc)
Thank you

Koichi Sasada
Heroku, Inc.
<ko1@heroku.com>
Questions and answers
Questions and Answers

RGenGC and CoW friendly

• No problem because only touch flags for oldgen and shady
Questions and Answers

GC + Threads

• Parallel GC
  • Run GC process in parallel (simultaneously)
  • Parallel marking
  • Parallel sweeping (in today’s talk)

• Concurrent GC / Incremental GC
  • Run ruby threads (mutator threads) and GC threads concurrently
  • Major GC consumes huge time (same as current GC) → Need concurrent GC to reduce pause time
  • New WB API is also designed for concurrent GC