Toward efficient Ruby 2.1

Koichi Sasada
<ko1@heroku.com>

Heroku, Inc.
Agenda

• Ruby 2.1 Schedule
• Ruby 2.1 new “internal” features
  • Internal object management hooks
    • Object allocation tracing
    • GC hooks
  • RGenGC: Restricted Generational Garbage Collection ← Today’s main topic
• Ruby 2.1 expected “internal” features
  • Sophisticated inline cache invalidation mechanism
  • Memory efficient string management
  • Useful debugger
Summary

• We are implementing new features and improving Ruby’s quality for Ruby 2.1
• Especially introducing “Generational garbage collector” which I’m working on will improve huge performance
• Ruby 2.1 is currently scheduled on Dec 25, 2013
“2:1 And there went a man of the house of Levi, and took to wife a daughter of Levi.”

- Book of Exodus

“2:1 さて、レビの家のひとりの人が行ってレビの娘をめとった。”

- 出エジプト記
Quoted “2.1”

In this presentation, there are some quoted “2.1” sentence.

Idea of “Quoting” is from “Things a Computer Scientist Rarely Talks About” “コンピュータ科学者がめったに語らないこと” by Donald E. Knuth

But no consideration in this presentation about them.
Who am I?

• 笹田耕一 (Koichi Sasada)
  • Matz team at Heroku, Inc.
    • Full-time CRuby development
• CRuby/MRI committer
  • Virtual machine (YARV) from Ruby 1.9
  • YARV development since 2004/1/1
Matz team at Heroku, Inc.
Hierarchy

Matz @ Shimane
Title collector

ko1 @ Tokyo
EDD developer

Communication
with Skype

Nobu @ Tochigi
Drunker
Recent status

• 5/2 I got sprain...
• 5/27 I got cold...

• All: Please care about yourself
  • Especially, do not walk with book reading
Object-oriented scripting language Ruby is a programming language designed by Matsumoto.

- Efficient Implementation of Ruby Virtual Machine
  Doctoral thesis by Koichi Sasada

“オブジェクト指向スクリプト言語Rubyは, 松本によって設計されたプログラミング言語である．”

- 高速なRuby用仮想マシンの開発
  笹田耕一, 博士論文
Ruby’s rough history

1993 2/24
Birth of Ruby
(in Matz’ computer)

1995/12
Ruby 0.95
1st release

1996/12
Ruby 1.0

1998/12
Ruby 1.2

1999/12
Ruby 1.4

2000/6
Ruby 1.6

2003/8
Ruby 1.8

2009/1
Ruby 1.9.0

2013/02
Ruby 2.0.0

2004
Ruby on Rails

2000 Book:
Programming Ruby

2004/
Start YARV proj.

2012/4
ISO Ruby

2004～
Ruby on Rails
Quoted “2.1”

“2.1 Changes from Ruby 1.9
Added and modified libraries from Ruby 1.9 are follows”

- Programming Ruby 1.9 Library edition
  by Dave Thomas, with Chad Fowler and Andy Hunt

“2.1 Ruby 1.9 のライブラリの変更点
Ruby 1.9 で追加または変更されたライブラリは次のとおりです。”

- プログラミングRuby 1.9 ライブラリ編
Ruby 2.0

• New features (see Rubyist Magazine)
  • Keyword arguments
  • Refinements
  • Module#prepend

• Ruby 2.0.0-p195 was already released
NEWS file of Ruby 2.0

Many new features!!
“Ruby is almost matured as a programming language with 2.0”
http://itpro.nikkeibp.co.jp/article/NEWS/20130214/456322/
Ruby 2.1 release announcement

“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.”

“ちなみに、Ruby 2.1.0 は2013 年12 月25 日のリリースを予定しています。そのうち 2.0.0 の時のように機能提案募集をするつもりなので、われこそをという方はそろそろネタの仕込みを始めてくださいませ。”

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

by NARUSE, Yui
Ruby 2.1 schedule

2013/02
Ruby 2.0.0

2013/12
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
Ruby 2.1

• New features
**NEWS for Ruby 2.1.0**

This document is a list of user visible feature changes made between releases except for bug fixes. It does not contain internal changes or changes where the difference is too subtle to explain clearly.

## Changes since the 2.0.0 release

### Language changes

### Core classes updates (outstanding ones only)

- **GC**
  - Added environment variable: `RUBY_HEAP_SLOTS_GROWTH_FACTOR`: growth rate of the heap.

- **IO**
  - Extended methods:
    - `IO#seek` accepts symbols (`:CUR, :END, :SET`) for 2nd argument.

- **Kernel**
  - New methods:
    - `Kernel#singleton_method`

- **Mutex**
  - Misc:
    - `Mutex#owned?` is no longer experimental.

- **String**
  - New methods:
    - `String#scrub` and `String#scrub!` verify and fix invalid byte sequence.
  - Extended methods:
    - If invalid: `:replace` is specified for `String#encode`, replace invalid byte sequence even if the destination encoding equals to the source encoding.

### Core classes compatibility issues (excluding feature bug fixes)

- **IO**
  - Incompatible changes:
    - `open` ignore internal encoding if external encoding is ASCII-8BIT.

### Stdlib updates (outstanding ones only)

- **Digest**
  - Extended methods:
    - `Digest::Class.file` takes optional arguments for its constructor.

- **Matrix**
  - Added `Vector#cross_product`.

- **Net::SMTP**
  - Added `Net::SMTP#rset` to implement the RSET command.

- **Pathname**
  - New methods:
    - `Pathname#write`
    - `Pathname#binwrite`

- **OpenSSL::BN**
  - Extended methods:
    - `OpenSSL::BN.new` allows `Fixnum/Bignum` argument.

- **open-uri**
  - Support multiple fields with same field name (like `Set-Cookie`).

- **StringScanner**
  - Extended methods:
    - `StringScanner#[]` supports named captures.

- **Tempfile**
  - New methods:
    - `Tempfile.create`

### Stdlib compatibility issues (excluding feature bug fixes)

- **URI**
  - Incompatible changes:
    - `URI.decode_www_form` follows current WHATWG URL Standard. It gets encoding argument to specify the character encoding. It now allows loose percent encoded strings, but denies `;` separator.
    - `URI.encode_www_form` follows current WHATWG URL Standard. It gets encoding argument to convert before percent encode. UTF-16 strings aren't converted to UTF-8 before percent encoding by default.

### C API updates

See NEWS file

Now, much smaller than Ruby 2.0
“Character set and CES which application should support is different by users. However, it is not high priority to support one application supports multi-CES.”

- Implementation of Practical Multilingual Text Manipulation for Ruby (academic paper) by Yukihiro Matsumoto (translated by Koichi Sasada)

“アプリケーションが対応すべき文字集合およびCESはユーザごとに異なるが、1つのアプリケーションが同時に複数のCESに対応する必要性はさほど高くない。”

- Rubyにおける実用的な多言語処理の実装（論文） 松本 行弘
Ruby 2.1 features

• Refine m17n introduced from Ruby 1.9
  • String#scrub, String#scrub!
    • Verify and fix invalid byte sequence.
  • More efforts? I heard Matz has some ideas.

• Refine features introduced from Ruby 2.0
  • Keyword arguments
  • Refinements
  • Module#prepend
Matz, the creator of Ruby, spoke at Waza for the 20th anniversary of the language and the release of Ruby 2.0. If you weren't in the sold out crowd, not to worry. Information should flow free and experiences should be shared; in line with those concepts you can watch Matz’s talk right here, then read about what’s new in this version of Ruby and how to run it on Heroku.

With slides available on speakerdeck
Running 2.0 on Heroku

If you’re interested in taking advantage of these new features give it a try on Heroku today. To run Ruby 2.0 on Heroku you’ll need this line in your Gemfile:

```
ruby "2.0.0"
```

Then commit to git:

```
$ git add .
$ git commit -m "Using Ruby 2.0 in production"
```

We recommend that you test your app using 2.0 locally and deploy to a staging app before pushing to production. Now when you `$ git push heroku master` our Ruby buildpack will see that you’ve declared your Ruby version and make sure you get the right one.

Of course, Ruby 2.0.0 is ready on Heroku!
20 years of simplicity, elegance, and programmer happiness

Heroku, since its founding, has been aligned with the key values of Ruby – simplicity, elegance, and programmer happiness. Heroku still believes in the power and flexibility of Ruby, and we've invested in the language by hiring Yukihiro "Matz" Matsumoto, Koichi Sasada, and Nobuyoshi Nakada. We would like to thank them and the whole Ruby core team for making this release happen. Join us in celebrating Ruby's successes and in looking forward to the next twenty years by trying Ruby 2.0 on Heroku today.
Ruby apps are running using 1.8.7, you should upgrade. Ruby 1.8.7 is approaching End of Life (EOL) in three months on June 2013. EOL for Ruby 1.8.7 means no security or bug patches will be provided by the maintainers. Not upgrading means you're potentially opening up your application and your users to vulnerabilities. Don't wait till the final hour, upgrade now to be confident and secure.

**Speed**

Ruby 2.0 has a faster garbage collector and is *Copy on Write* friendly. Copy on Write or COW is an optimization that can reduce the memory footprint of a Ruby process when it is copied. Instead of allocating duplicate memory when a process is forked, COW allows multiple processes to share the same memory until one of the processes needs to modify a piece of information. Depending on the program, this optimization can dramatically reduce the amount of memory used to run multiple processes. Most Ruby programs are memory bound, so reducing your memory footprint with Ruby 2.0 may allow you to run more processes in fewer dynos.

*If you're not already running a concurrent backend consider trying the *Unicorn* web server.*

**Features**

In addition to running faster than 1.9.3, and having a smaller footprint, Ruby 2.0 has a number of new features added to the language including:
Mention about “Speed”

Ruby 2.0 has a faster garbage collector and is Copy on Write friendly. This means that Ruby 2.0 can reduce the amount of memory used when it is copied from one process to another. Copy on Write (CoW) allows multiple processes to share the same memory until one of the processes needs to modify a piece of information. Depending on the program, this optimization can dramatically reduce the amount of memory used to run multiple processes. Most Ruby programs are memory bound, so reducing your memory footprint with Ruby 2.0 may allow you to run more processes in fewer dynos.

If you’re not already running a concurrent backend consider trying the Unicorn web server.

Short summary: GC uses bitmap marking and CoW friendly

Short summary: Let’s try Unicorn!
Only mention about GC!!??
(I don’t work on GC)
Let’s consider about GC/memory management!
Ruby 2.1 internal features

- Internal hooks for memory management
- RGenGC: Restricted generational garbage collection
Internal hooks for memory 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 memory management

• 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
Internal hooks for memory 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
Internal hooks for memory management

Sample features

• 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
Internal hooks for memory management

Postponed job

• You may want to write hooks in Ruby
  → Use ‘Postponed job’
    • ‘Postponed jobs’ run at same timing as finalizers
    • Usage: rb_postponed_job_register(func, data)
    • `func(data)` will be called at a safe-point

• See an sample code in “ext/objspace/gc_hooks.c”
  • ObjectSpace.after_gc_(start | end) = proc{GC.start}
  • Proc is called after GC
“2.1 Structure of VALUE and objects

In ruby, the contents of an object is expressed by a C structure, always handled via a pointer. A different kind of structure is used for each class, but the pointer type will always be VALUE.”

- Ruby Hacking Guide
  by Minero Aoki

“2.1 VALUEとオブジェクト構造体

rubyではオブジェクトの実体を構造体で表現し、扱うときは常にポインタ経由で扱う。構造体のほうはクラスごとに違う型を使うが、ポインタのほうはどのクラスの構造体でも常にVALUE型だ。“

- Rubyソースコード完全解説
  青木峰郎
RGenGC: Summary

• RGenGC: Restricted Generational GC
  • New 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: Agenda

• Background
  • Generational GC
  • Ruby’s GC strategy
• Proposal: RGenGC
  • Separating into sunny 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
“2.1 About Mark&Sweep GC
Mark&Sweep GC consists of mark and sweep phase.”
- Garbage Collection-Algorithms and Implementations
  By Narihiro Nakamura, Hikaru Aikawa
  (translated by Koichi Sasada)

“2.1
マークスイープGCはその名の通り、マークフェーズとスイープフェーズから成ります。”
- ガベージコレクションのアルゴリズムと実装
  By 中村成洋, 相川光
1. Mark reachable objects from root objects

2. Sweep unmarked objects (collection and de-allocation)
RGenGC: Background
Generational GC (GenGC)

• Weak generational hypothesis: Most objects die young → Concentrating reclamation effort on the youngest objects

• Separate young generation and old generation
  • Create objects as young generation
  • Promote to old generation after surviving $nth$ 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
    → Minor GC: Copy GC, Major GC: M&S
  • On the CRuby’s: both **Minor&Major GCs should be M&S** because CRuby’s GC (and existing codes) based on conservative M&S algorithm
RGenGC: Background: GenGC
[Minor M&S GC]

1st MinorGC

Root objects

- Mark reachable objects from root objects.
  - Mark and promote to old gen
  - 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.

RubyKaigi 2013 Toward efficient Ruby 2.1 by Koichi Sasada
RGenGC: Background: GenGC

[MInor M&S GC]

• Mark reachable objects from root objects.
  • Mark and promote to old gen
  • 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
“2.1 The mark-sweep algorithm

From the viewpoint of the garbage collector, mutator threads perform just three operations of interest, New, Read and Write, which each collection algorithm must redefine appropriately.”

- The Garbage Collection Handbook by Richard Jones, Antony Hosking, Eliot Moss
RGenGC: Background: GenGC
WB & Remember Set (RSet)

- Old objects refer young objects

→ **Minor GC causes marking leak!!**

- Because minor GC ignores referenced objects by old objects

Can’t mark new object!
→ Sweeping living object! (BUG)
RGenGC: Background: GenGC
WB & Remember Set (RSet)

• Add an old object into **Remember set (RSet)** if an old object refer new objects
  • At minor GC, mark all remembered objects
• To detect [old$\rightarrow$new] type references, insert **"Write-barrier"**
  • “Generating references” == “Write”
RGenGC: Background: GenGC [Minor M&S GC] w/ RSet

- Mark reachable objects from root objects
  - Remembered objects are also root objects
- Stop traversing after old objects
- Sweep not (marked or old) objects
RGenGC: Problem
Write-barrier (WB) and CRuby

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

• Write-barrier (WB) example in Ruby world
  • (Ruby) old0[0] = new0 # [old0 → new0]
  • (Ruby) old1.foo = new0 # [old1 → new1]

• Write-barriers miss causes terrible failure
  • WB miss
    → Remember-set registration miss
    → (minor GC) marking-miss → Terrible GC BUG!!

• All of C-extensions need perfect Write-barriers
  • Manipulate Ruby objects in C language (in C-ext)
  • C-level WBs are needed
RGenGC: Problem
Inserting WBs into C-extensions (C-ext)

- **Problem: Compatibility**
  - Example (C) `RARRAY_PTR(old0)[0] = new1`
  - There are Many Many C-exts’ sources like that

- CRuby core code uses C-APIs, but we can rewrite all of source code (with terrible debugging!!)

- We can’t rewrite all of C-exts which are written by 3rd party
RGenGC: Problem
Inserting WBs into C-extensions (C-ext)

“Two options”

[Give up on GenGC]
or

[GenGC with re-writing all of C-extensions without C-exts compatibility]
RGenGC:
Related work on Ruby’s GenGC
• Kiyama, et. al. GenGC for CRuby
  • Straightforward implementation for Ruby 1.6
  • Need WBs in correct places
  • High development cost
  • Can’t keep compatibility → Drop all C-exts
• Nari, et.al longlife GC for CRuby
  • Introduce GenGC only for Node object
  • No compatibility issues because C-exts don’t use node
  • Now CRuby doesn’t use many number of node objects
  • High development cost (to guarantee WBs)
RGenGC: Related work on Ruby’s GenGC

• Make interpreter with other language infrastructures which have GC
  • JRuby, IronRuby
    • Can’t keep compatibility with current C-exts
• Separate core heap and CRuby C-ext heap
  • High development cost
RGenGC: Challenge

• How to insert Write-barriers?
  • In Ruby-core, we can change with huge effort
  • However, we can’t touch existing C-exts ← Problem

• Several 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: Challenge to introduce GenGC

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

RGenGC: Restricted Generational Garbage Collection
RGenGC: Goal
Inserting WBs into C-extensions (C-ext)

“2 → 3 options”

[Give up on GenGC]

or

[GenGC with re-writing all of C-extensions without C-exts compatibility]

or

[Use RGenGC]
RGenGC:
Key idea

• Introduce **Shady object**
  • In this context, “Shady” means questionable, doubtful, etc
  • Something feeling dark
  • 日陰者, in Japanese
Google image search: “日陰者”
RGenGC: Key Idea

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

• Decide this type at creation time
  • A class don’t care about WB → Shady obj
  • A class care about WB → Sunny obj
  • Currently, most of classes DON’t care about WB, so most of objects are created as Shady objects.
RGenGC: Key Idea

• Sunny objects can change to Shady objects
  • “Shade” operation
  • In the C program doesn’t care about RGenGC
• Example
  • `ptr = RARRAY_PTR(ary)`
  • In this case, we can’t insert WB for `ptr` operation, so VM shade “ary”

Shady object can’t change into sunny object
RGenGC
Key Idea: Rule

• Mark “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 sunny objects
    1. Demote objects
    2. Remember shaded shady objects
RGenGC

[Minor M&S GC w/Shady object]

1st MinorGC

- Root objects
- Remember set (RSet)

- 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!!
• 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]

- Old sunny objects → 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)

Ruby → Mark → Sweep → Sweep → Sweep → Sweep → Sweep

Stop the (Ruby) World

w/RGenGC (Minor GC)

Ruby → Mark → Sweep → Sweep → Sweep → Sweep → Sweep

Stop the (Ruby) World

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

2.0.0 GC (M&S w/lazy sweep)

w/RGenGC (Minor GC)

<table>
<thead>
<tr>
<th># of old object (#old)</th>
<th># of new object (#new)</th>
<th># of freed but remembered objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) # of old objects by WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) # of shady objects pointed by old</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) # of old but shady objects</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RGenGC

Number of marking objects

w/RGenGC (Minor GC)

<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>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>
RGenGC
Discussion: Pros. and Cons.

• Pros.
  • Allow WB unprotected objects (shady objects)
    • 100% compatible w/ existing extensions (and standard classes/methods)
  • 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 (frequent) classes/methods.
    • We can ignore minor classes/methods.
  • Simple algorithm, easy to develop (done!)
RGenGC
Discussion: Pros. and Cons.

• Cons.
  • Increasing “unused, but not corrected objects until full/major GC
    • Remembered objects (caused by well known GenGC algorithm)
    • Remembered shady objects (caused by RGenGC algorithm)
  • WB insertion (potential) bugs
    • 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).
Quoted “2.1”

“2.1 Character set
...

- C Reference manual
By Samuel P. Harbison III, Guy L. Steele Jr.

“2.1 文字集合
一つのCソースファイルは、一つの文字集合に含まれる文字の並びである。”

-C リファレンスマニュアル
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_OLDGEN: Young gen or Old gen?
    • 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
  • RARRAY_PTR(ary) causes shade operation
    • Can’t get RGenGC performance improvement
    • But works well 😊

• Instead of RARRAY_PTR(ary), use alternatives
  • RARRAY_AREF(ary, n) → RARRAY_PTR(ary)[n]
  • RARRAY_ASET(ary, n, obj) → RARRAY_PTR(ary)[n] = obj w/ Write-barrier
  • RARRAY_PTR_USE(ary, ptrname, {...block...})
    • Only in block, pointers can be accessed by `ptrname` variable (VALUE*).
    • Programmers need to insert collect WBs (miss causes BUG).
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
Implementation

• RGENGC_CHECK_MODE in gc.c
  • 1: Enable assertions
  • 2: Enable “WB checking” mode

• WB checking mode
  • (1) do minor GC
  • (2) do major/full GC
  • (3) compare result with (1) and (2)
    • If living objects in (2) but not living in (1) it should be BUG!!

• Not a perfect (implementation limitation), but a good method to detect bugs
RGenGC
Implementation

• Macros in ruby/ruby.h
  • USE_RGENGC
    • You can enable/disable RGenGC with this macro.
  • RGENGC_WB_PROTECTED_???
    • RGENGC_WB_PROTECTED_ARRAY, RGENGC_WB_PROTECTED_HASH, RGENGC_WB_PROTECTED_STRING, RGENGC_WB_PROTECTED_OBJECT, RGENGC_WB_PROTECTED_FLOAT, RGENGC_WB_PROTECTED_COMPLEX, RGENGC_WB_PROTECTED_RATIONAL, RGENGC_WB_PROTECTED_BIGNUM
    • Now, only supports above types (T_???).
      • T_CLASS, T_MODULE and T_DATA is needed to support with high priority.
    • You can enable/disable RGenGC for each types.

• If you have trouble with RGenGC, try to disable them.
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 RDoc generation as Ruby’s trunk
RGenGC
Performance evaluation (micro)

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

RubyKaigi 2013 Toward efficient Ruby 2.1 by Koichi Sasada
RGenGC
Performance evaluation (RDoc)

Several major/full GC peaks

Total GC count is different

Faster minor GC
RGenGC
Performance evaluation (RDoc)

![Bar chart comparing Normal and RGenGC](chart.png)

- **Normal**
- **RGenGC**

**Y-axis**: ms

**Legend**
- Blue: Total mark
- Orange: Total sweep

RubyKaigi 2013 Toward efficient Ruby 2.1 by Koichi Sasada
RGenGC
Performance evaluation (RDoc)

Impressive!!

Of course, this is “Graph magic”. If a student submits this graph, his score is fail.
RGenGC
Performance evaluation (RDoc)

About 15% speedup!

Execution time

Normal  RGenGC
RGenGC: Summary

- **RGenGC: Restricted Generational GC**
  - New GC algorithm allow 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** and **String** objects are WB protected
  - Array and String objects are very popular in Ruby
  - Array objects using `RARRAY_PTR()` change to WB unprotected objects (called as Shady objects), so existing codes work well
RGenGC
Future work
• Minor GC / Major GC timing
  • Too many major GC → slow down
  • Too few major GC → memory consumption issue, etc
• Make more sunny objects (especially T_CLASS)
• Optimize remember set representation
• Inserting WBs w/ application profiling
  • Profiling system
  • Benchmark programs
• Detection system for WBs insertion miss
  • RGENGC_CHECK_MODE (2, in gc.c) is not enough
RGenGC
Issues: Terminology

• **Matz rejected the word “Sunny”**

• “Shady” has a meaning of “questionable, doubtful, …”, but “Sunny” has no meaning of against “questionable, doubtful, etc”.

\[\text{Doubtful, questionable, etc} \]

\[\text{???)} \]
RGenGC

Issues: Terminology

• This is a last presentation to use “Shady” and “Sunny”
• We will replace codes and documents with:
  • “Shady” → “WB unprotected”
  • “Sunny” → “WB protected”
• Or
  • “Shady” → “Shady” (remain)
  • “Sunny” → “Normal” (not shady)

If you have any idea of the words, please let us know.
“2:1 Now when Jesus was born in Bethlehem of Judaea in the days of Herod the king, behold, there came wise men from the east to Jerusalem,”

- Gospel of Matthew

“2:1 イエスがヘロデ王の代に、ユダヤのベツレヘムでお生れになったとき、見よ、東からきた博士たちがエルサレムに着いて言った、”

- マタイによる福音書
Ruby 2.1 expected “internal” features

• Sophisticated inline cache invalidation mechanism
• Memory efficient string management & Symbol GC
• Fine-grain memory protection to detect WB insertion miss
• Signal thread
• More efficient inter-process migration technique
• JIT compilation for small part of Ruby code
• Introduce fastpath C-methods type
• Inlined Proc.call invocation
• AOT Compiler and extending “require” behavior
• Useful debugger
Sophisticated inline cache invalidation mechanism

• From Ruby 1.9 (YARV), inline cache technique is used in several codes
  • Inline method caching ← Huge opportunity
  • Constant lookup
  • ...

• Cache invalidation with only one variable “global_state_version”

• Invalidate inline cache, other non-related inline caches are also invalidated
Sophisticated inline cache invalidation mechanism

• Invalidate all classes’ method cache

Redefine X, invalidate all of classes

RubyKaigi 2013 Toward efficient Ruby 2.1 by Koichi Sasada
Sophisticated inline cache invalidation mechanism

“This patch adds class hierarchy method caching to CRuby. This is the algorithm used by JRuby and Rubinius.”

[ruby-core:55053] [ruby-trunk - Feature #8426][Open] Implement class hierarchy method caching by Charlie Somerville
Sophisticated inline cache invalidation mechanism

• Invalid only sub-classes under effective class

Redefine X, invalidate X and X’s subclasses
Memory efficient string management

• Each string has their string body (space acquired by malloc())
Memory efficient string management

• For some strings have same “string body”, they has own string body each other.
Memory efficient string management

• It can be shared by strings w/ dirty bit.

→ Reduce memory consumption!!

† Sharing string body is implemented now if a string object is duped. This technique is more aggressive approach.
Memory efficient string management

• This mechanism can work with Symbol management

→ GC-able Symbol

“String body”  
(shared by 5 places)
Quoted “2.1”

“2:1 And the heavens and the earth were finished, and all the host of them.”

- Genesis

“2:1 こうして天と地と、その万象とが完成した。”

- 創世記
Agenda

• Ruby 2.1 Schedule
• Ruby 2.1 new “internal” features
  • Internal object management hooks
    • Object allocation tracing
    • GC hooks
  • RGenGC: Restricted Generational Garbage Collection ← Today’s main topic
• Ruby 2.1 expected “internal” features
  • Sophisticated inline cache invalidation mechanism
  • Memory efficient string management
  • Useful debugger
Summary

• We are implementing new features and improving Ruby’s quality for Ruby 2.1
• Especially introducing “Generational garbage collector” which I’m working on will improve huge performance
• Ruby 2.1 is currently scheduled on Dec 25, 2013
Thank you
Any questions?

Koichi Sasada
Heroku, Inc.
<ko1@heroku.com>