Introducing *Restricted Generational Garbage Collection* into CRuby/MRI

Generational Garbage Collection under the Sunshine

2013/April

Koichi Sasada

Heroku, Inc.

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

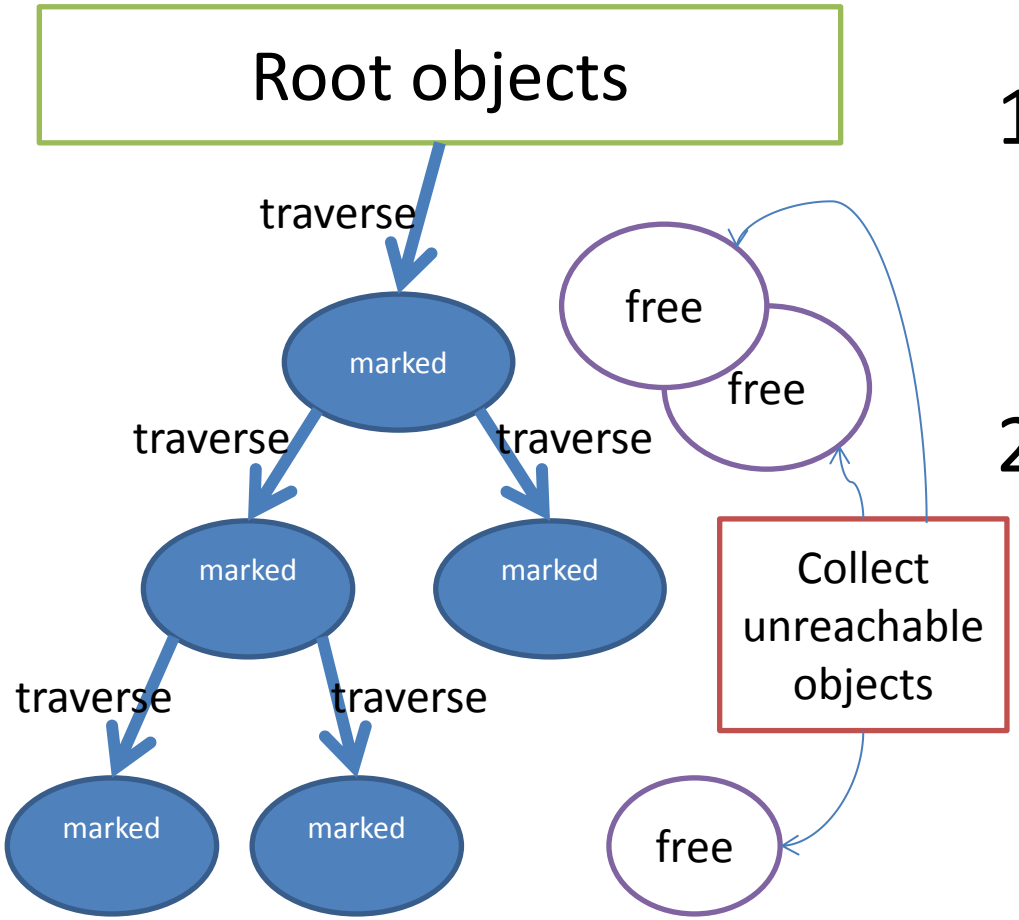
Agenda

- Background
  - Generational GC
  - Ruby’s GC strategy
- Proposal: RGenGC
  - Separating into sunny and shady objects
  - Shady objects at marking
  - Shade operation
- Implementation

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

Background
Mark & Sweep



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

Background
Generational GC (GenGC)

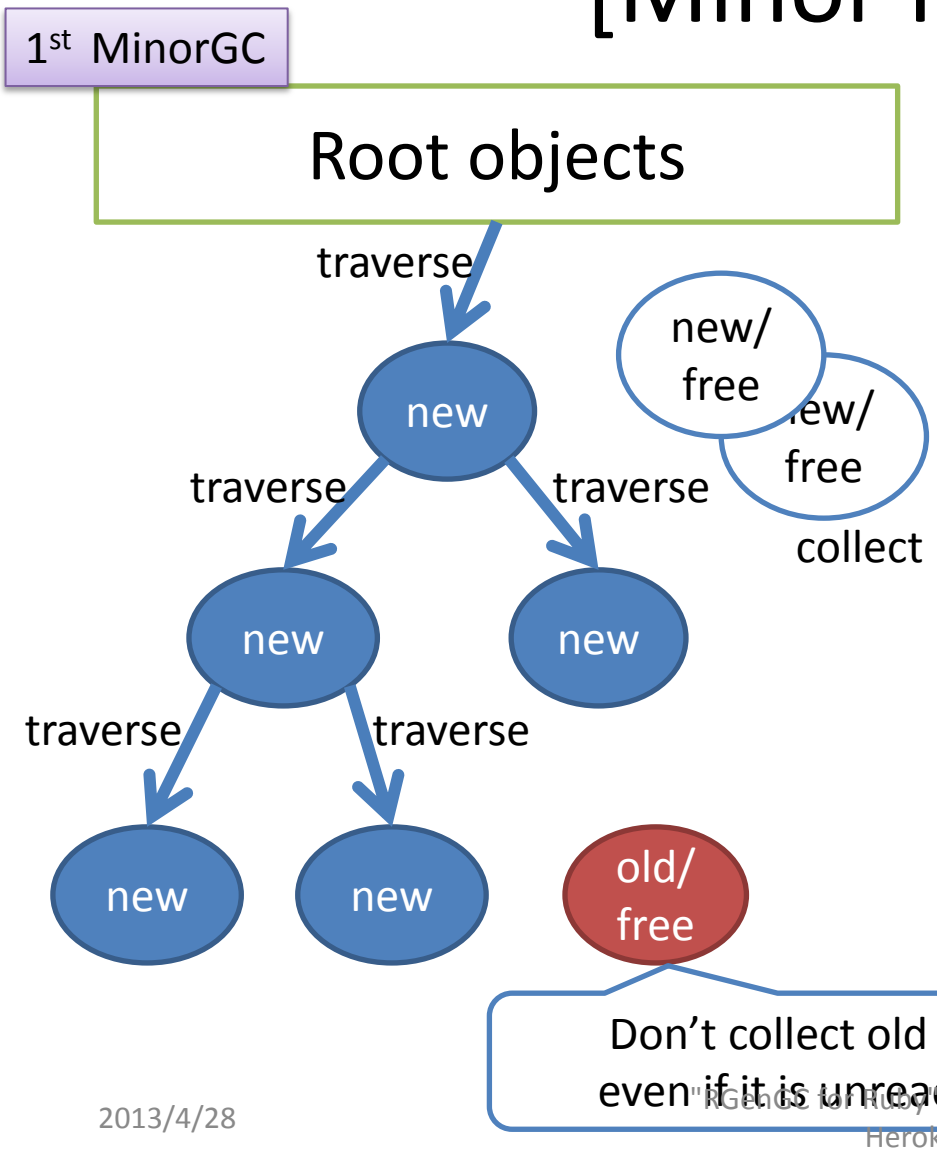
- Weak generational hypothesis: Most objects die young → Concentrating reclamation effort on the youngest objects
- Separate young generation and old generation
  - Create objects as young generation
  - Promote to old generation after surviving *nth* GC
  - In CRuby, *n == 1* (after 1 GC, objects become old)
- Usually, GC on young space (minor GC)
- GC on both spaces if no memory (major/full GC)

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 **<u>Minor&Major GCs should be M&S</u>** because CRuby's GC (and existing codes) based on conservative M&S algorithm

Background: GenGC [Minor M&S GC]



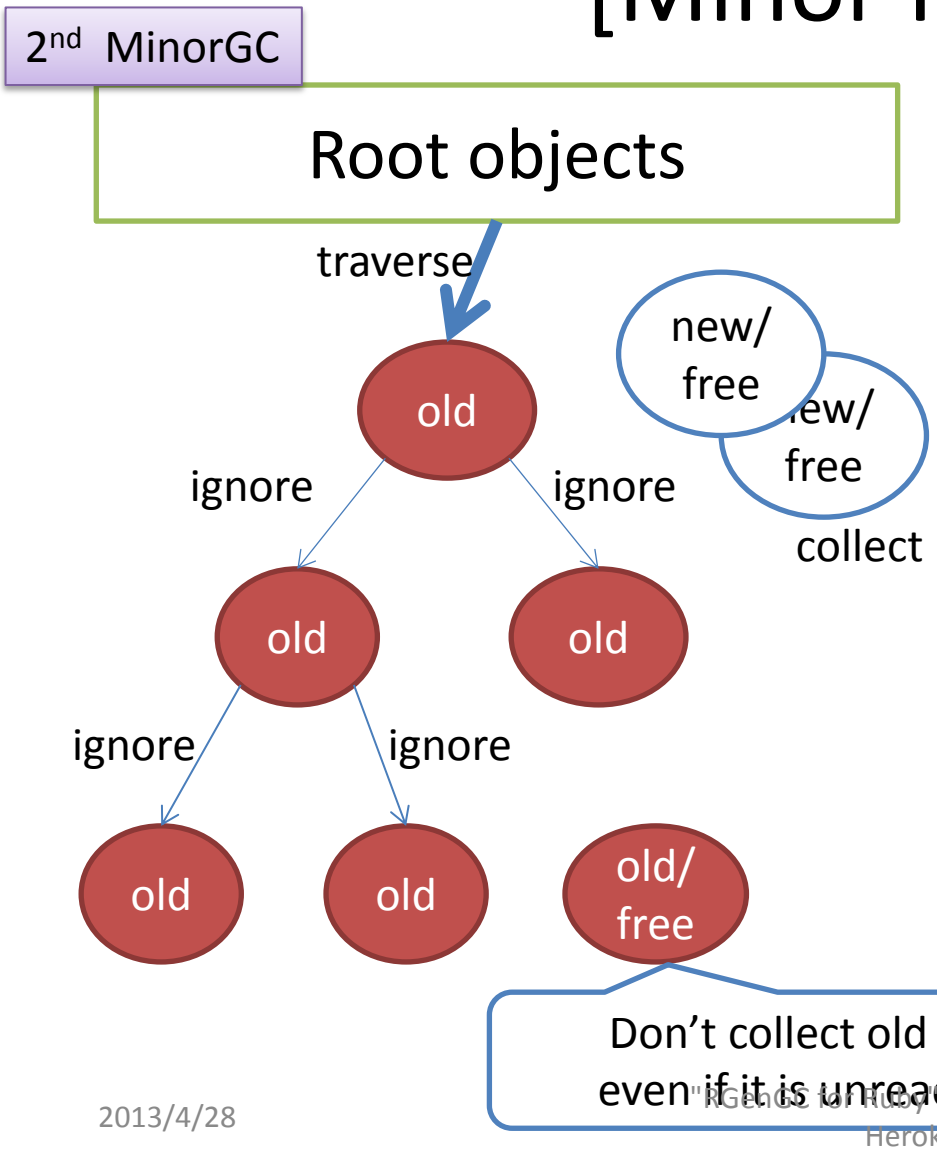
- Mark reachable objects from root objects.
  - Mark and **promote to old gen**
  - Stop traversing after old objects

→ Reduce mark overhead

- Sweep not (marked or old) objects

- Can't collect Some unreachable objects

Background: GenGC [Minor M&S GC]



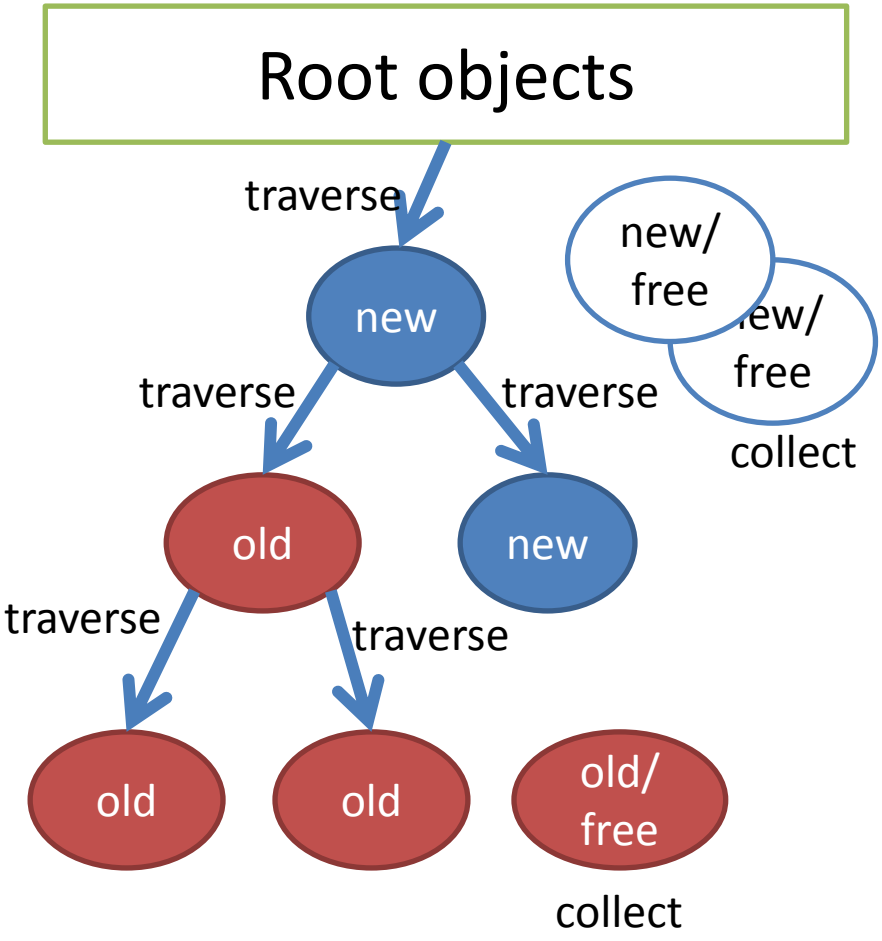
- Mark reachable objects from root objects.
  - Mark and **promote to old gen**
  - Stop traversing after old objects

→ Reduce mark overhead

- Sweep not (marked or old) objects

- Can't collect Some unreachable objects

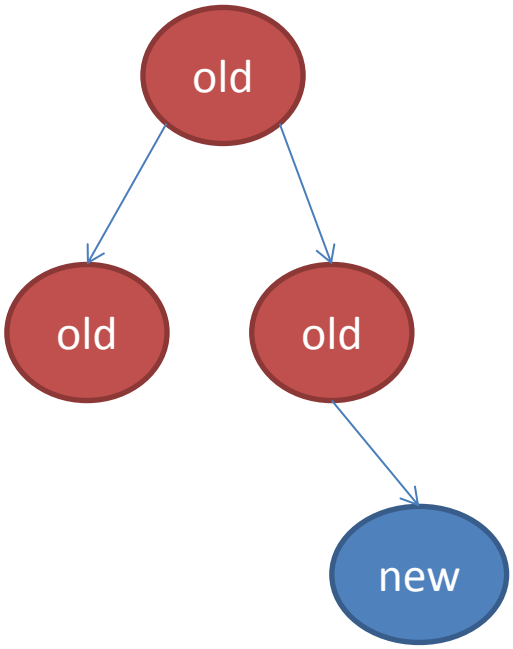
Background: GenGC [Major M&S GC]



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

- *<u>Sweep all unreachable (unused) objects</u>*

Background: GenGC Remember Set (RSet)



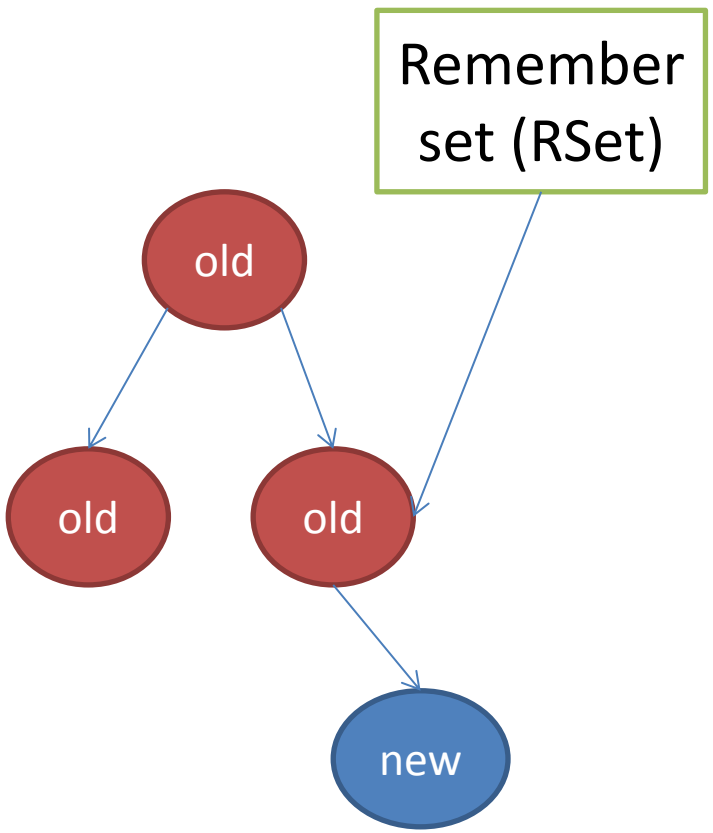
Can't mark new object!
→ Sweeping living object! (BUG)

- Old objects refer young objects

→ Minor GC causes marking leak!!

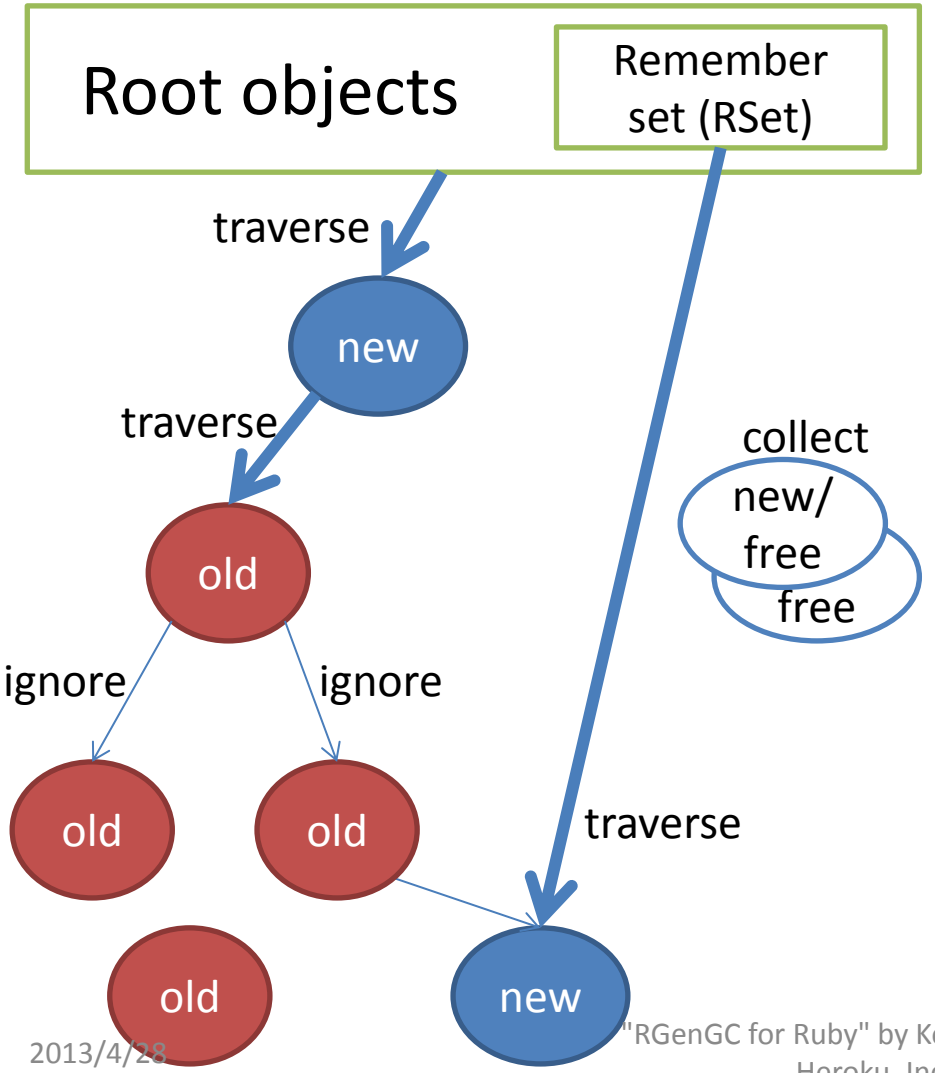
  – Because minor GC ignores referenced objects by old objects

Background: GenGC 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→new] type references, insert **"Write-barrier"**
  - "Generating references" == "Write"

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



- Mark reachable objects from root objects
  - **<u>Remembered objects are also root objects</u>**
- Stop traversing after old objects
- Sweep not (marked or old) objects

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

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 codes 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

Problem
Inserting WBs into C-extensions (C-ext)

“CHOSE!!”

Current Choice

[Give up GenGC]

or

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

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)

Related work on Ruby's GenGC

- Make interpreter with other language infrastructures which have GC
  - JRuby, IronRuby
  - Can't keep compatibility
- Separate core heap and CRuby C-ext heap
  - Rubinius
  - High development cost

RGenGC: Challenge

- How to treat Write-barriers?
  - In Ruby-core, we can chnage w/ huge effort
  - We can't touch existing C-exts ← Problem
- Several approaches
  - Separate heaps into the WB world and non-WB world
    - Rubinius way
    - Need huge development effort
  - WB auto-insertion
    - Modify C-compiler
    - Need huge development effort

Challenge to introduce GenGC
Goal

- **Create GC algorithm permits WB protected objects AND WB unsafe object in the same heap**

RGenGC: Restricted Generational Garbage Collection

Generational Garbage Collection under the Sunshine

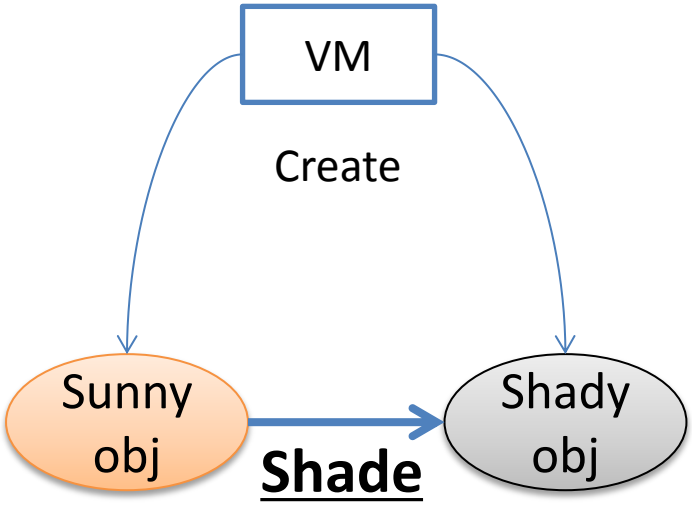
RGenGC

Key Idea

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

Shady: doubtful, questionable, ...

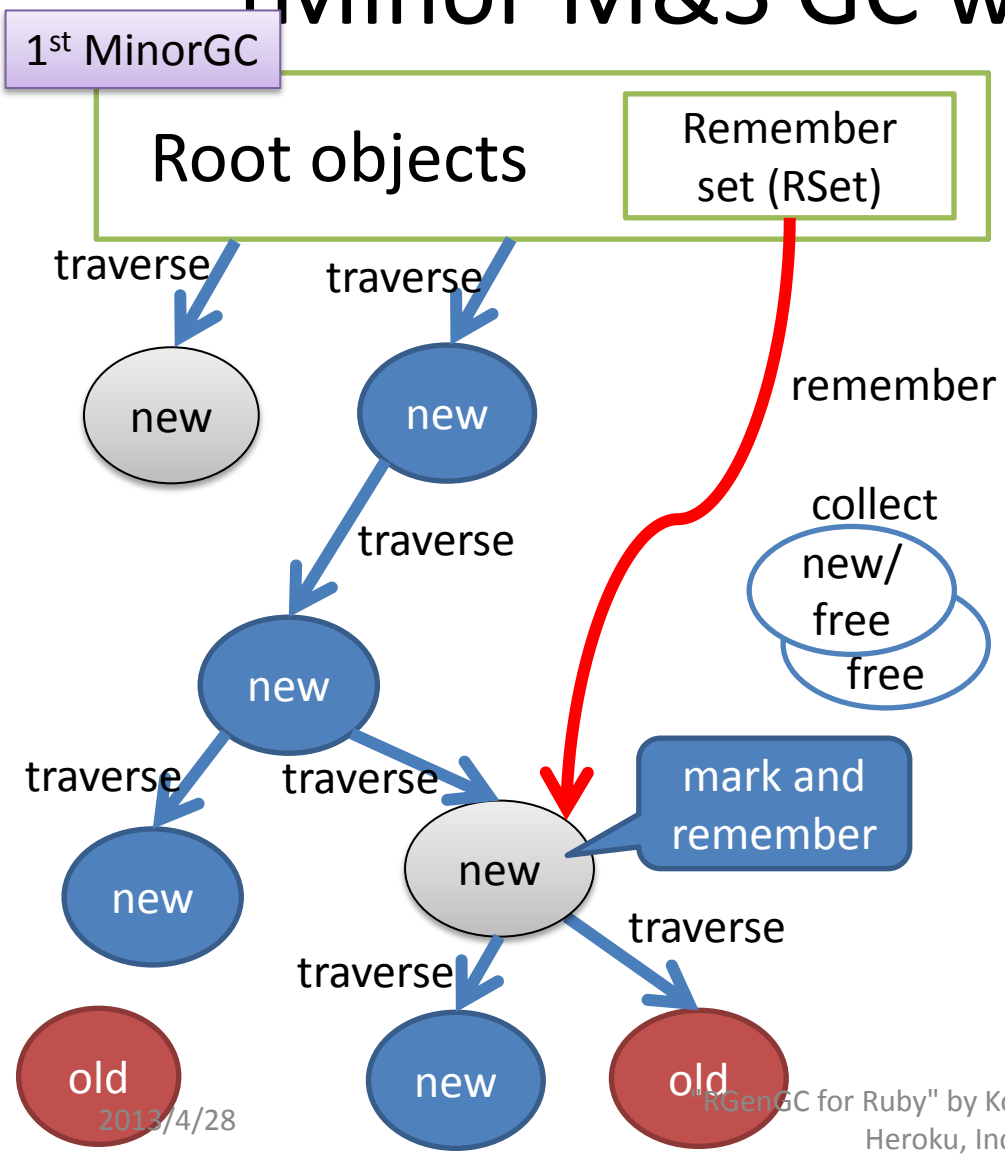
- Decide type at creation time
  - A class don't care about WB → Shady obj
  - A class care about WB → Sunny obj
  - Currently, most of classes **DOESN'T** care about WB, so **most of objects are created as Shady objects.**
- Sunny objects can change to Shady objects
  - "Shade" operation
  - Example
    - ptr = RARRAY_PTR(ary)
    - In this case, we can't insert WB for ptr operation, so VM shade "ary"



RGenGC
Key Idea

- 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

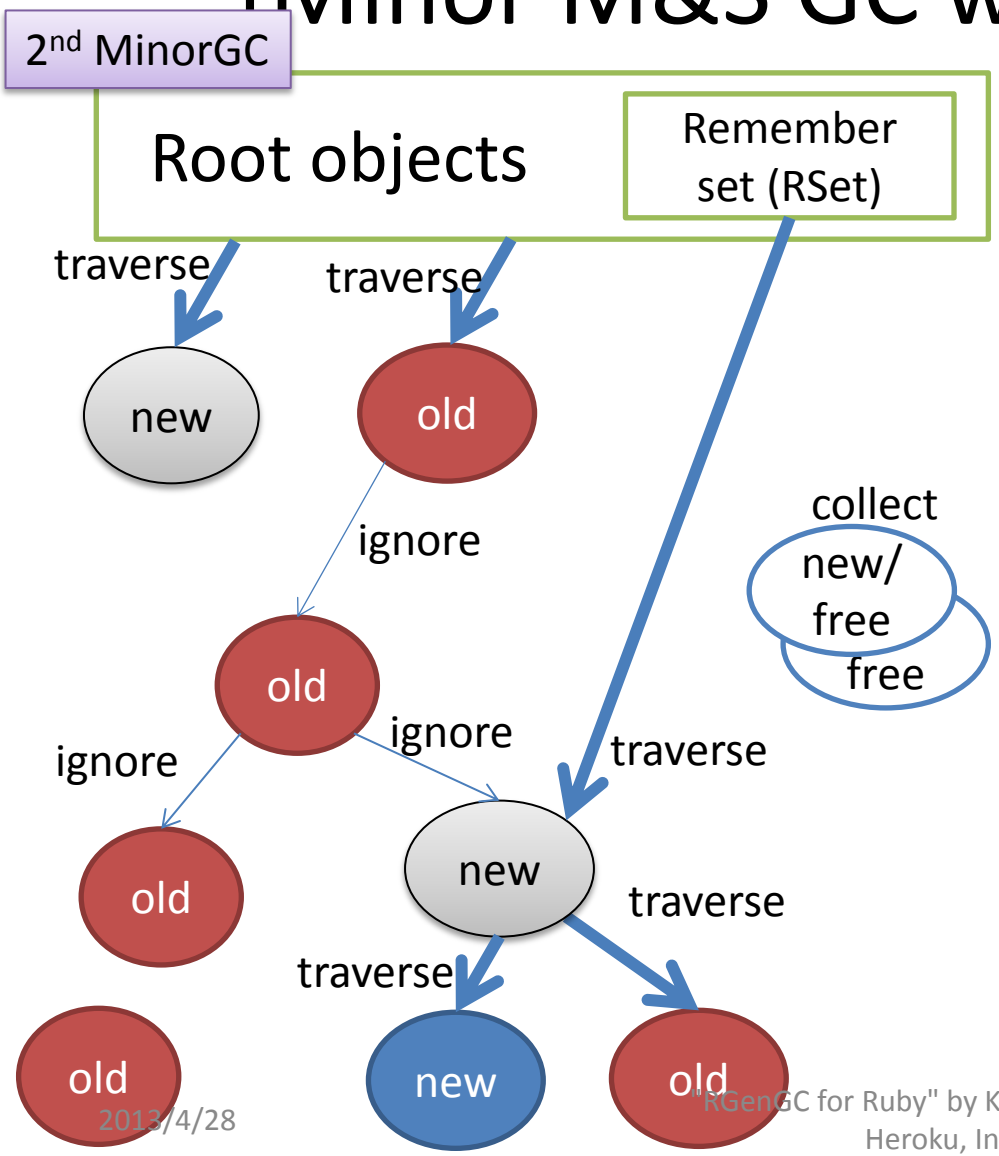
RGenGC
[Minor M&S GC w/Shady object]



- Mark reachable objects from root objects
  - Mark shady objects, and **<u>*don't promote*</u>** to old gen objects
  - If shady objects **pointed from old objects**, then **<u>remember shady objects</u>** by RSet.

    → Mark shady objects every minor GC!!

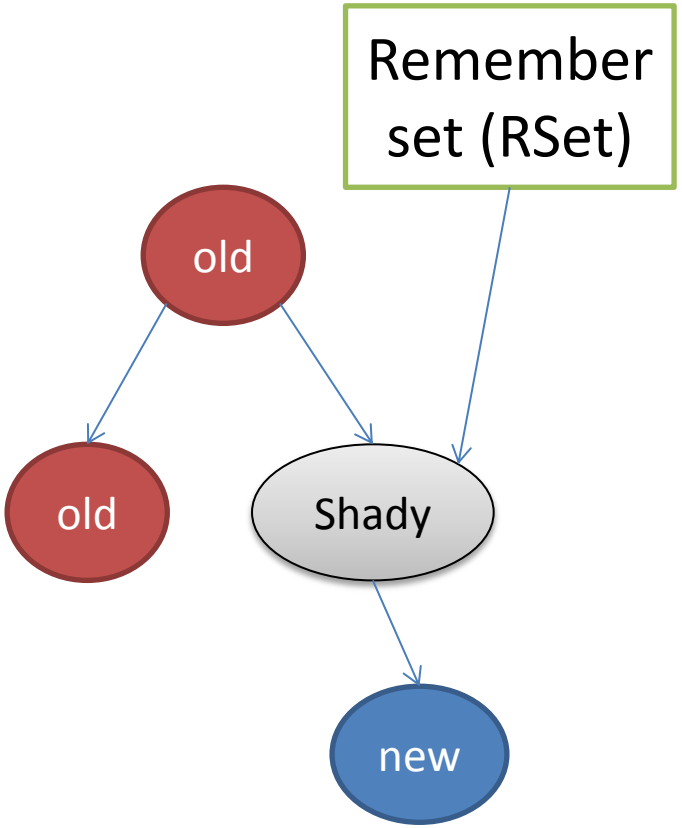
RGenGC
[Minor M&S GC w/Shady object]



- Mark reachable objects from root objects
  - Mark shady objects, and **<u>*don't promote*</u>** to old gen objects
  - If shady objects pointed from old objects, then remember shady objects by RSet.

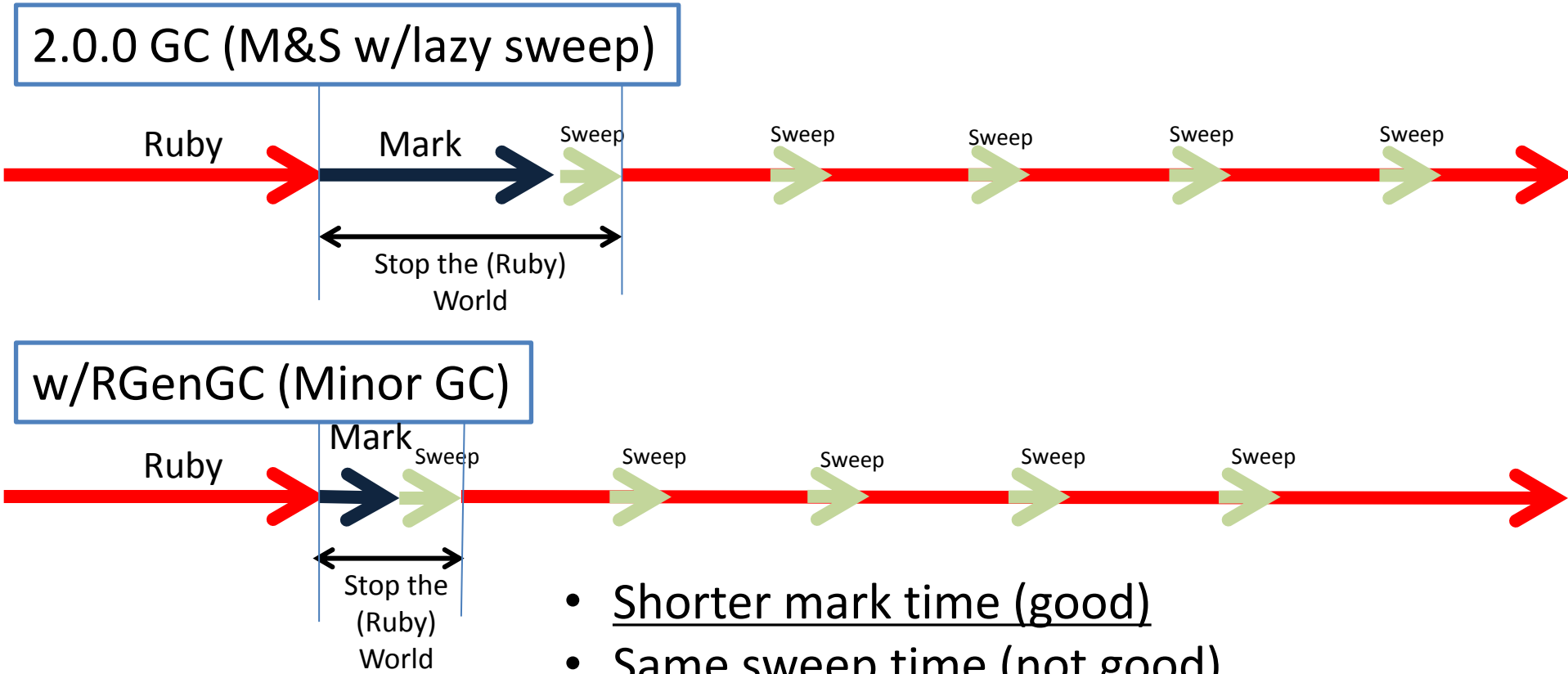
    → Mark shady objects every minor GC!!

RGenGC
[Shade operation]



- Old sunny objects → Shade objects
  - Example: RARRAY_PTR(ary)
  - (1) Demote object (old → new)
  - (2) Register it to Remember Set

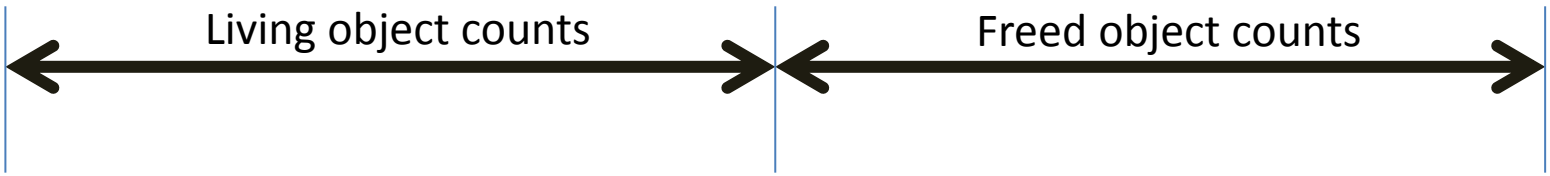
RGenGC
Timing chart



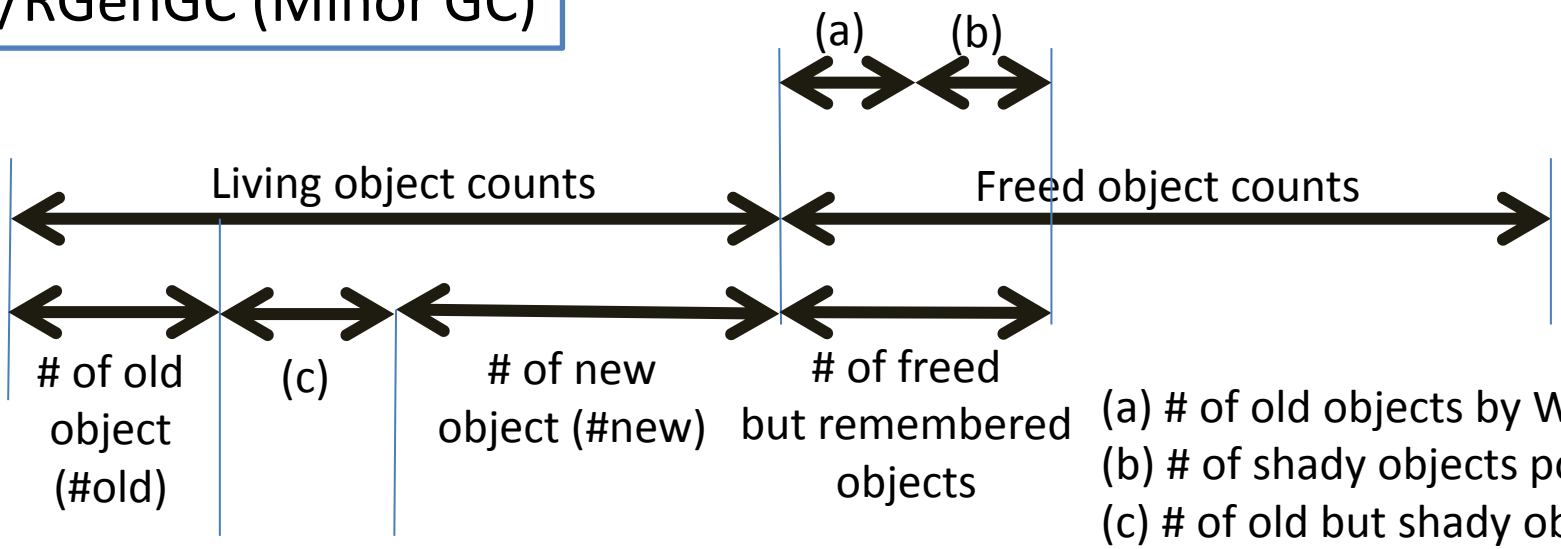
- <u>Shorter mark time (good)</u>
- <u>Same sweep time (not good)</u>
- (little) Longer execution time b/c WB (bad)

RGenGC
Number of marking objects

2.0.0 GC (M&S w/lazy sweep)

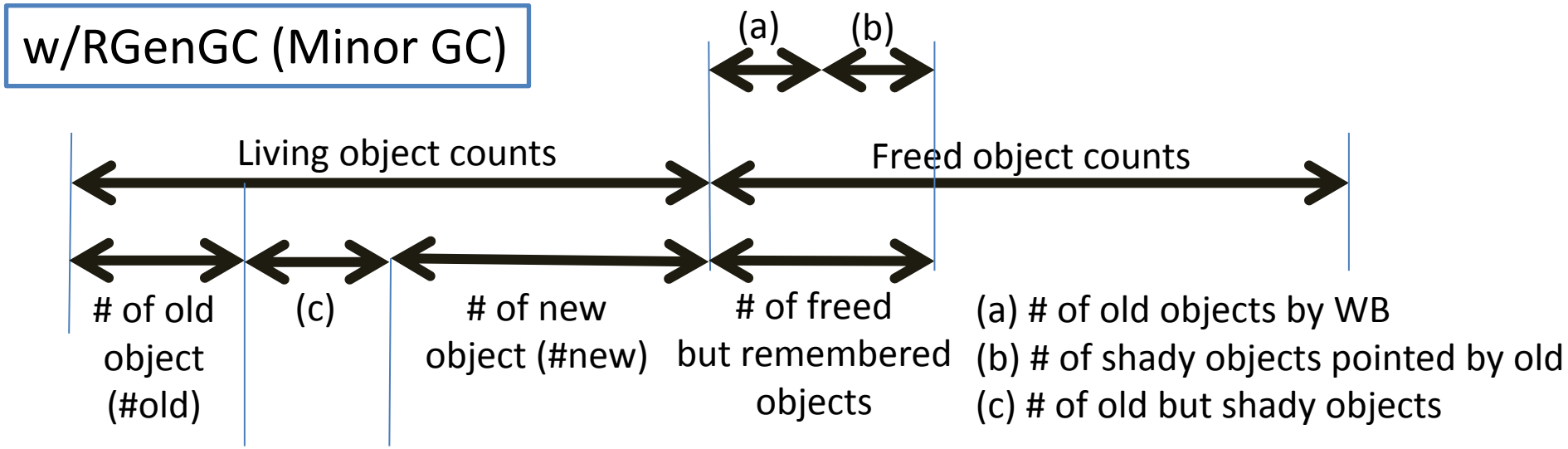


w/RGenGC (Minor GC)



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

RGenGC
Number of marking objects



	Marking space	Number of unused, uncollected objs	Sweeping space
Traditional GenGC	#new + (a)	(a)	#new
RGenGC	#new + (a) + (b) + (c)	(a) + (b)	Full heap

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
      - At first, we can try from Array and String, the most popular classes.
    - We can concentrate into popular (frequent) classes/methods.
    - We can ignore minor classes/methods.
  - Simple algorithm, easy to develop (done!)
- 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).

RGenGC Implementation

- Introduce two flasgs 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_WB(a, b)
  - Declare “a” refers “b”
  - OBJ_WB(a, b) returns “a”

RGenGC Implementation: WB operation API

- T_ARRAY
  - **<u>RARRAY_PTR(ary) causes shade operation</u>**
    - 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*).
    - **<u>Programmers need to insert collect WBs (miss causes BUG)</u>**.

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_??? style API should be provided.

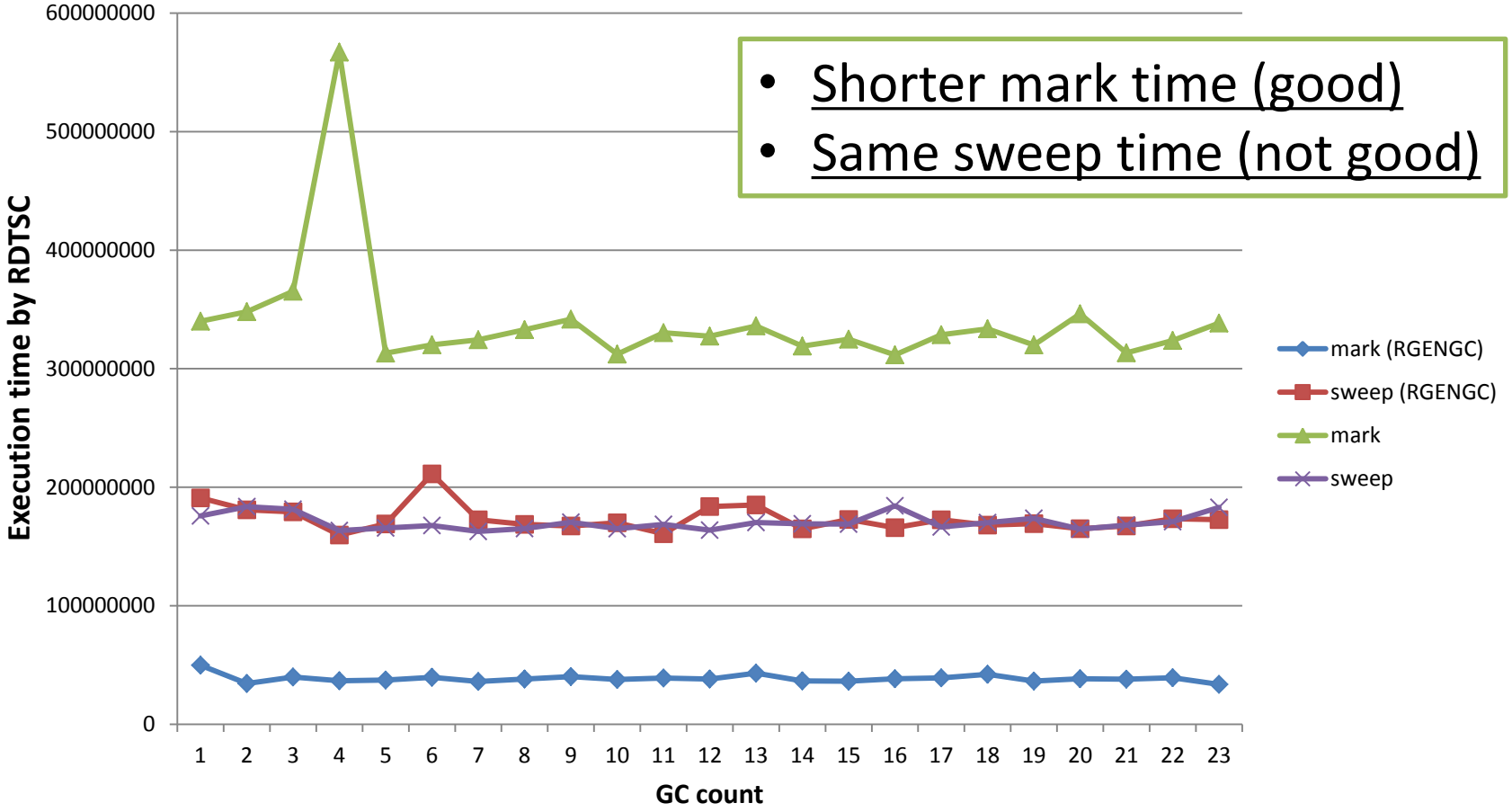
RGenGC
Future work

- Minor GC / Major GC timing
- 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
Performance evaluation

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

RGenGC
Performance evaluation



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

RGenGC
Performance evaluation

- Not yet for other application data
- Please wait ☺

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

Ask and Question from Ko1

- Please check my proposed algorithm
- Do not touch any program for RGenGC (WBs, etc)
  - APIs can be changed
- Please tell me any related works you know
  - I have surveyed about this GC algorithm, but I can't find that (I guess most of interpreters have perfect WBs)
  - I want to write a paper for DLS 2013 (Dynamic Language Symposium) ☺

Thank you

[Contact information]

Koichi Sasada

Heroku, Inc.

<ko1@heroku.com>