最近の Ruby のメモリ管理

Recent Ruby's memory management

Koichi Sasada

ko1@heroku.com

Summary

- Ruby’s new two GC implementation
  - RGenGC: Restricted Generational GC
  - RincGC: Restricted incremental GC

Who am I ?
Koichi Sasada from Heroku, Inc.

- CRuby/MRI committer
  - Virtual machine (YARV) from Ruby 1.9
  - YARV development since 2004/1/1
  - Recently, improving GC performance

- Matz team at Heroku, Inc.
  - Full-time CRuby developer
  - Working in Japan

- Director of Ruby Association

Ruby Association

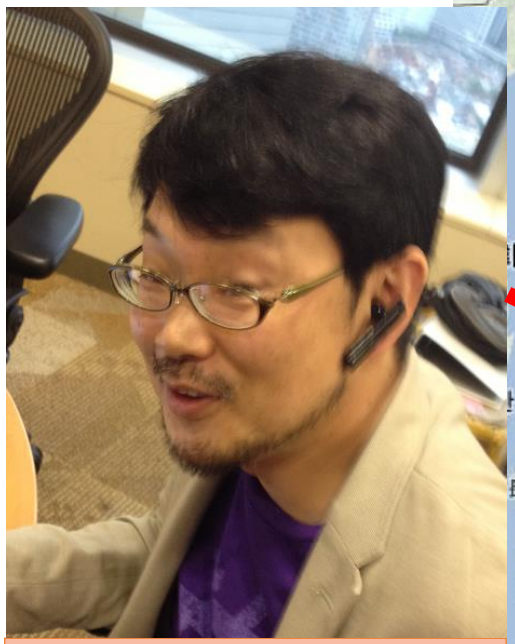
- Foundation to encourage Ruby developments and communities
  - Chairman is Matz
  - Located at Matsue-city, Shimane, Japan
- Activities
  - Maintenance of Ruby (Cruby) interpreter
    - Now, it is for Ruby 1.9.3
    - Ruby 2.0.0 in the future?
  - Events, especially RubyWorld Conference
  - Ruby Prize
  - Grant project. We have selected **3 proposals** in 2013
    - Win32Utils Support, Conductor, Smalruby - smalruby-editor
    - We will make this grant 2014!!
  - **Donation** for Ruby developments and communities

- Heroku, Inc. http://www.heroku.com
- Heroku supports Ruby development
  - Many talents for Ruby, and also other languages
  - Heroku employs 3 **Ruby interpreter core developers**
    - Matz
    - Nobu
    - Ko1 (me)
  - We name our group “Matz team”

“Matz team” in Heroku

Matz team in Heroku in Japan

Nobu @ Tochigi
Patch monster



Matz @ Shimane
Title collector

ko1 @ Tokyo
EDD developer

K.Sasada: 最近の Ruby のメモリ管理, 2014

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 2.2!!

Now, Ruby 2.1 is old version for us.

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

Ruby 2.1
Current stable

Ruby 2.1
a bit old Ruby

- **Ruby 2.1.0** was released at **2013/12/25**
  - New features
  - Performance improvements
- **Ruby 2.1.1** was released at 2014/02/24
  - Includes many bug fixes found after 2.1.0 release
  - Introduce a new GC tuning parameter to change generational GC behavior (introduce it later)
- **Ruby 2.1.2** was released at **2014/05/09**
  - Solves critical bugs (OpenSSL and so on)

Performance improvements

- Optimize “string literal”.freeze
- Sophisticated inline method cache
- Introducing Generational GC: RGenGC

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

Ruby 2.2
Next version

Ruby 2.2
Big features (planned)

- New syntax: not available now
- New method: no notable methods available now
- Libraries:
  - Minitest and test/unit will be removed (provided by bundled gem)

Ruby 2.2
Internal changes

- Internal
  - C APIs
    - Hide internal structures for Hash, Struct and so on
    - Remove obsolete APIs
  - GC
    - **Symbol GC (merged recently)**
    - **2age promotion strategy for RGenGC**
    - **Incremental GC** to reduce major GC pause time
  - VM
    - More sophisticated method cache

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

Break

Garbage collection
The automatic memory management

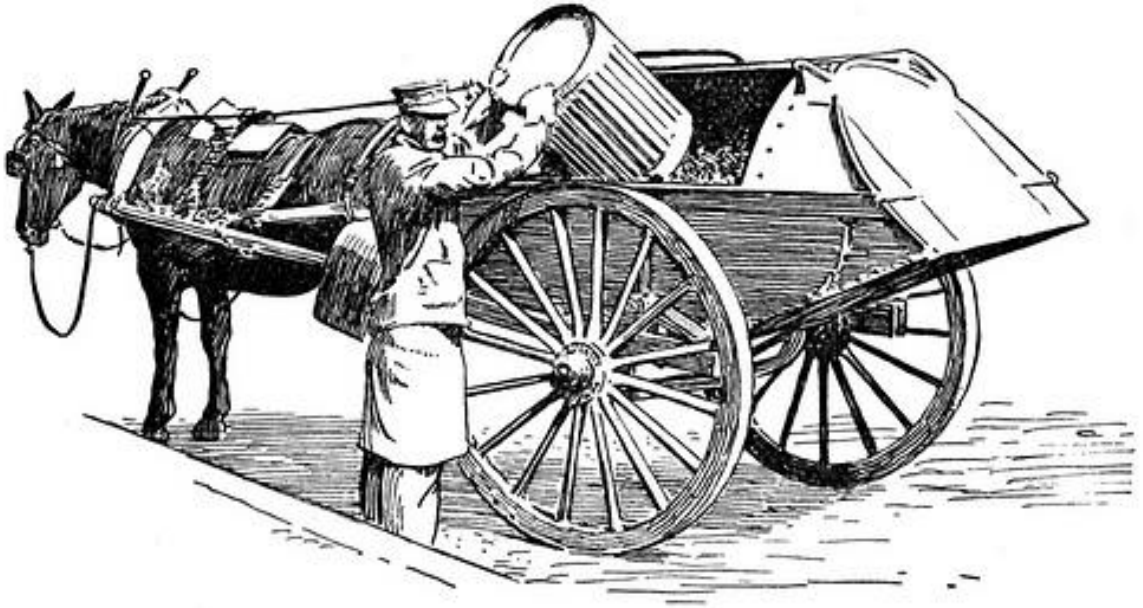


FIG. 109. — A GARBAGE COLLECTOR.

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

Automatic memory management Basic concept

- “Object.new” allocate a new object
  - “foo” (string literal) also allocate a new object
  - Everything are objects in Ruby!
- We don’t need to **“de-allocate”** objects manually

Automatic memory management Basic concept

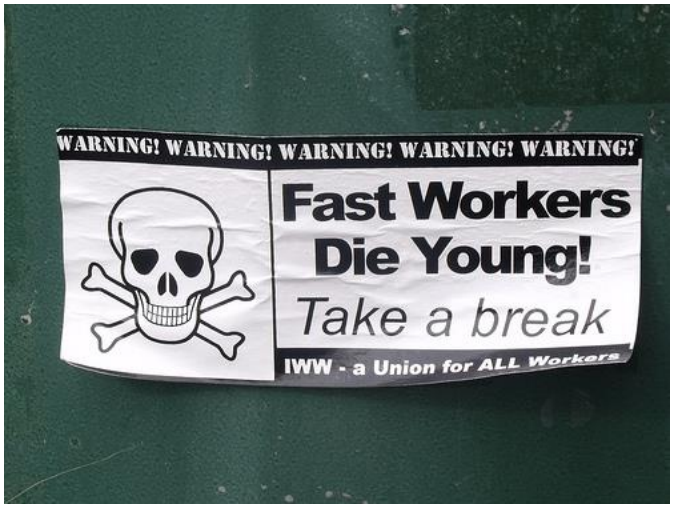
- **Garbage collector recycled “unused” objects automatically**

K.Sasada: 最近の Ruby のメモリ管理, 2014

Ruby's GC

- Mark & Sweep (from first release)
- Conservative marking (from first release)
- Lazy (incremental) sweep (from Ruby 1.9.3)
- Bitmap marking (from Ruby 2.0)
- **Generational marking (RGenGC, from Ruby 2.1)**
- **Incremental marking (PLANNED: from Ruby 2.2)**

RGenGC: Restricted Generational GC



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

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 WB-unprotected objects), so existing codes still works.

RGenGC: Background
Current CRuby's GC

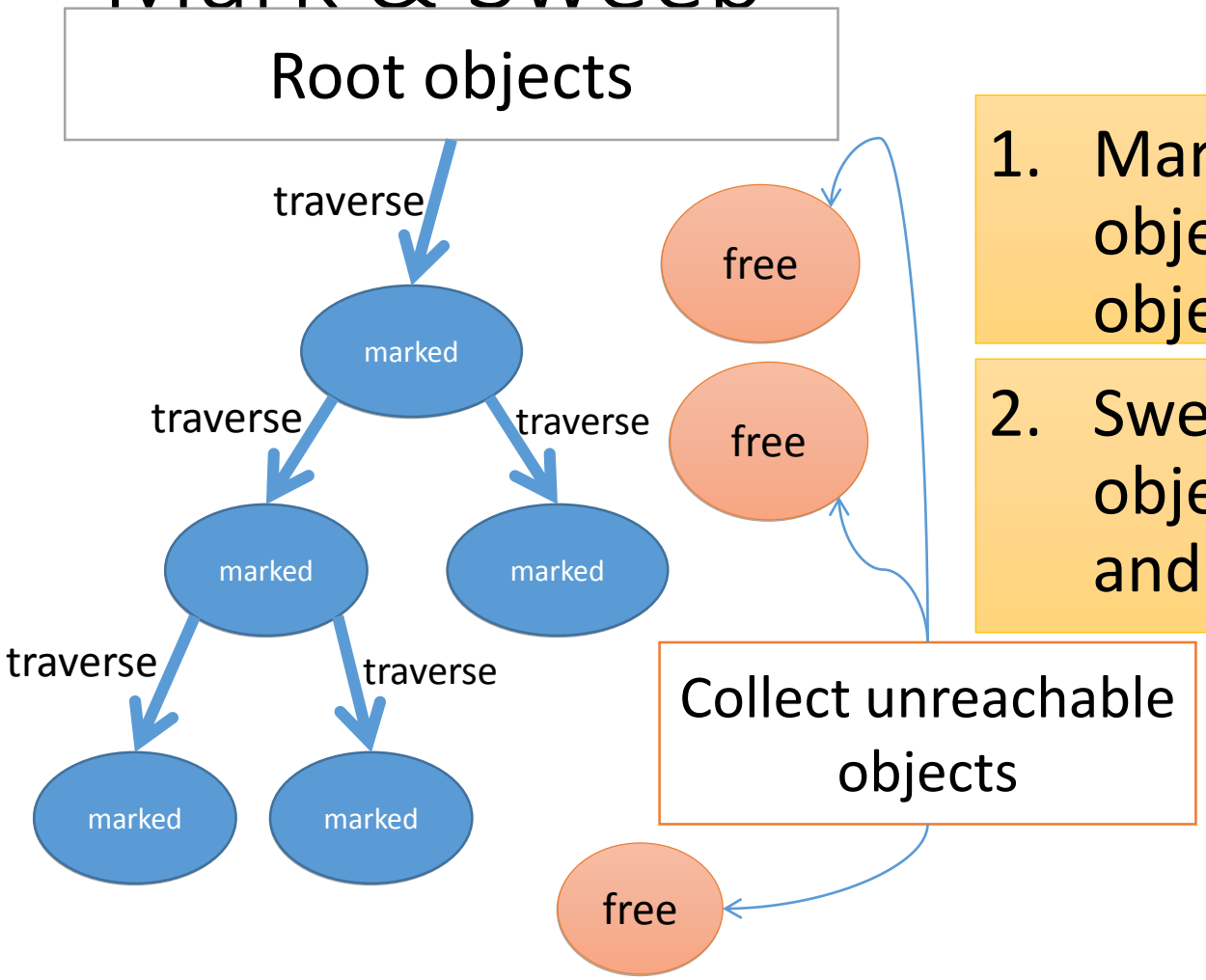
- Mark & Sweep
  - Conservative
  - Lazy sweep
  - Bitmap marking
  - Non-recursive marking
- C-friendly strategy
  - Don't need magical macros in C source codes
  - Many many C-extensions under this strategy

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



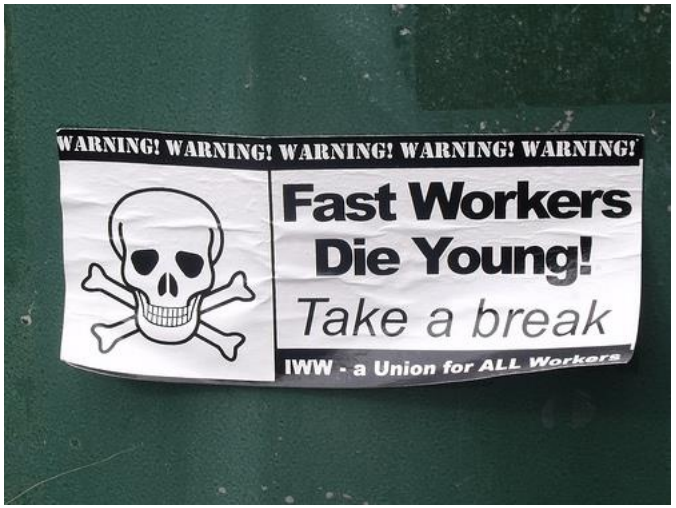
1. Mark reachable objects from root objects
2. Sweep ***unmarked*** objects (collection and de-allocation)

K.Sasada: 最近の Ruby のメモリ管理, 2014

RGenGC: Background Generational GC (GenGC)

- Weak generational hypothesis:

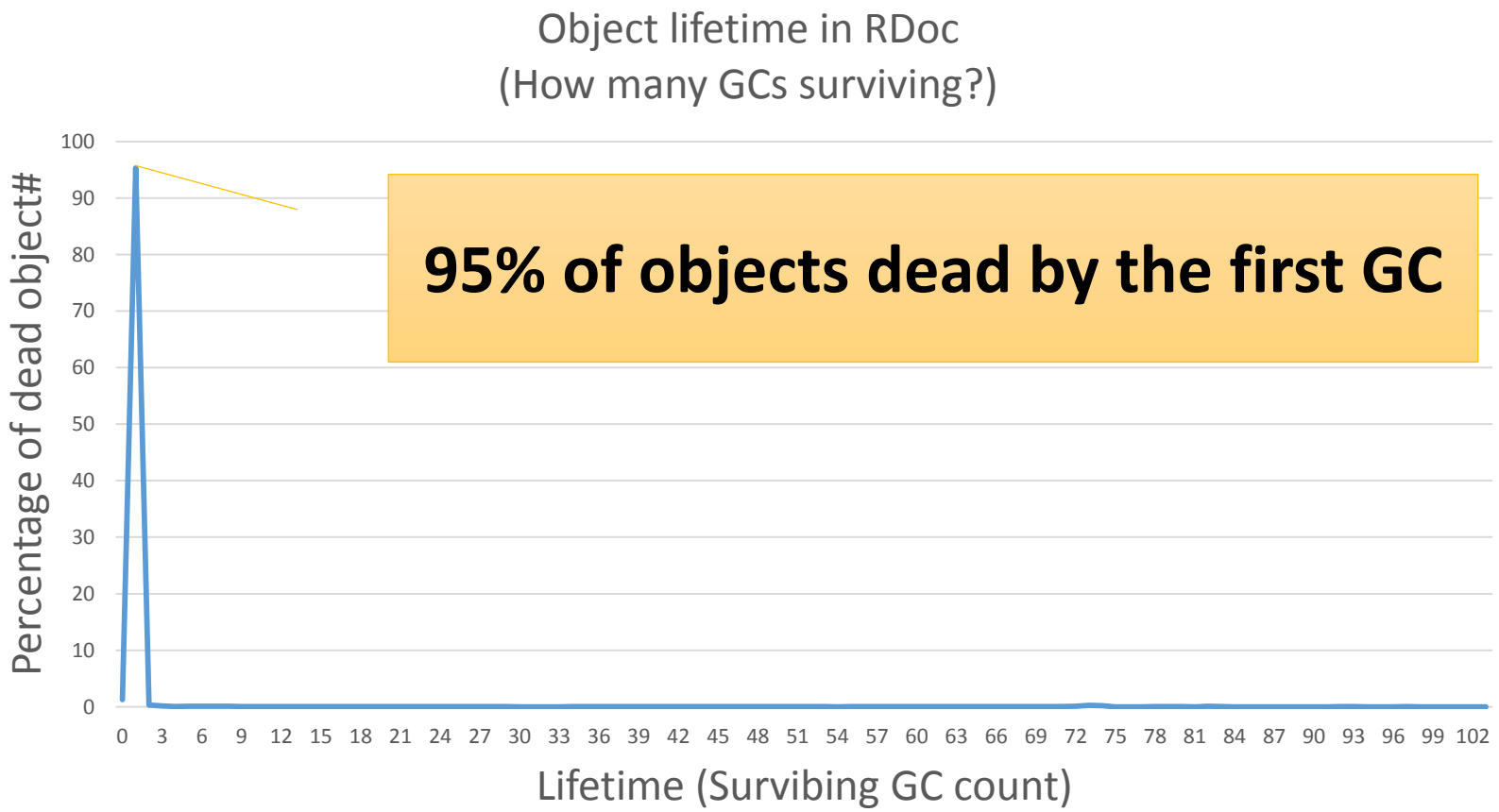
“Most objects die young”



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

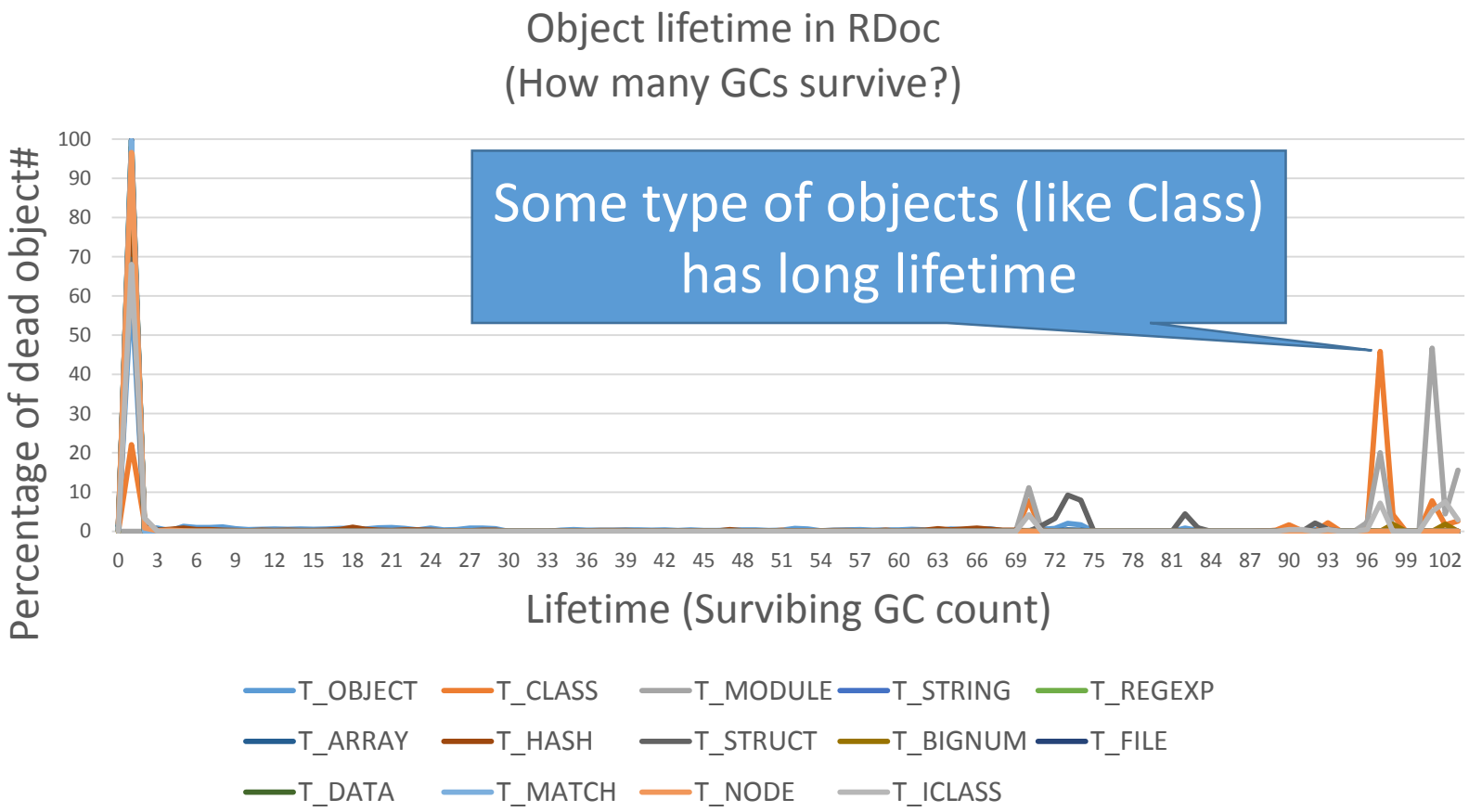
→ Concentrate reclamation effort only on the young objects

RGenGC: Background Generational hypothesis



K.Sasada: 最近の Ruby のメモリ管理, 2014

RGenGC: Background Generational hypothesis



K.Sasada: 最近の Ruby のメモリ管理, 2014

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)

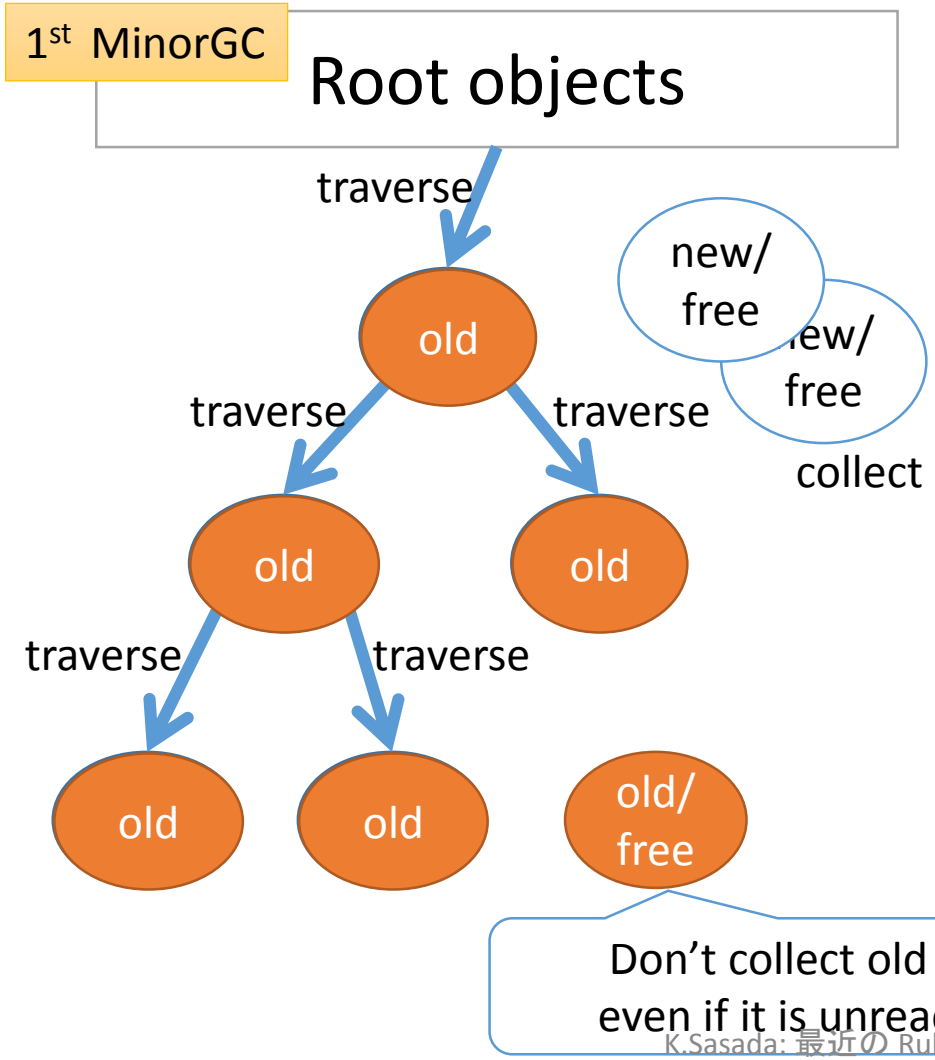
RGenGC: Background Generational GC (GenGC)

- 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]

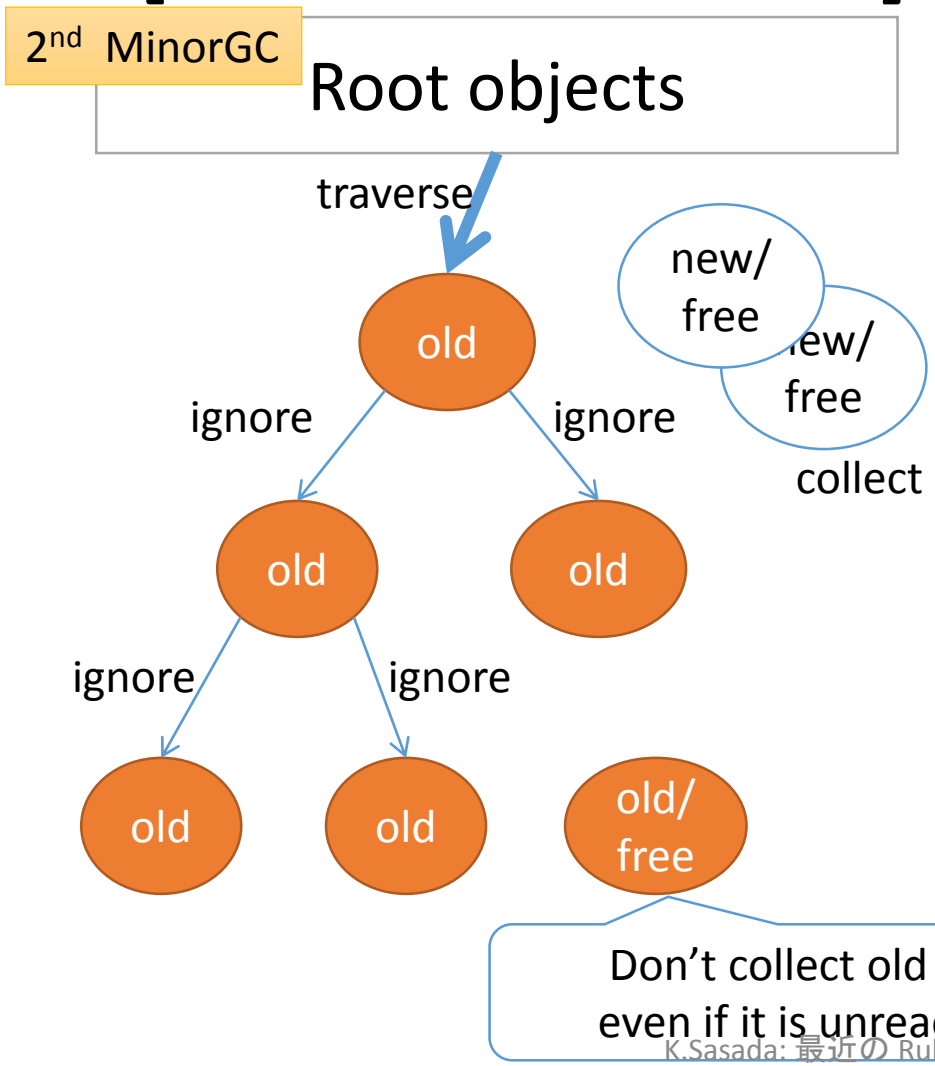


- Mark reachable objects from root objects.
  - Mark and **promote to old generation**
  - Stop traversing after old objects

→ Reduce mark overhead

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

RGenGC: Background: GenGC [Minor M&S GC]

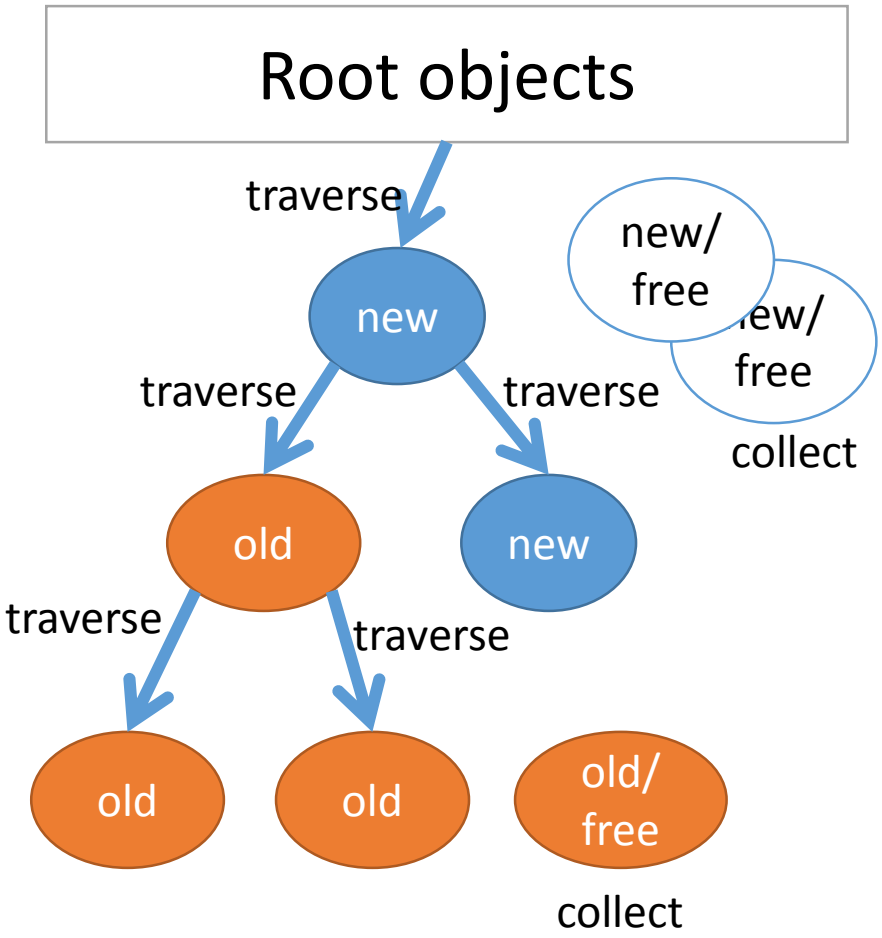


- Mark reachable objects from root objects.
  - Mark and **promote to old generation**
  - Stop traversing after old objects

→ Reduce mark overhead

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

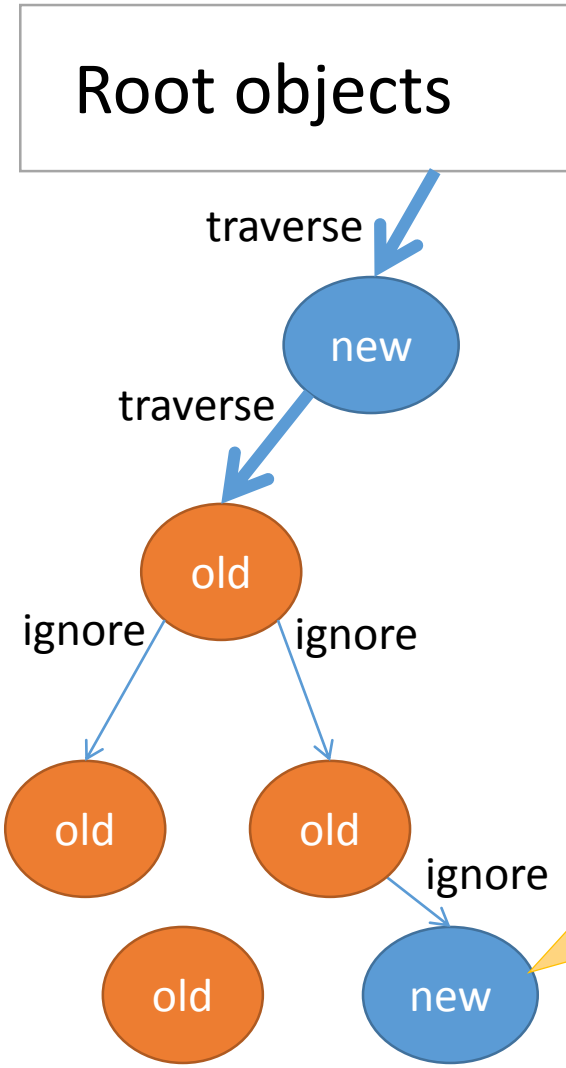
RGenGC: Background: GenGC [Major M&S GC]



- 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

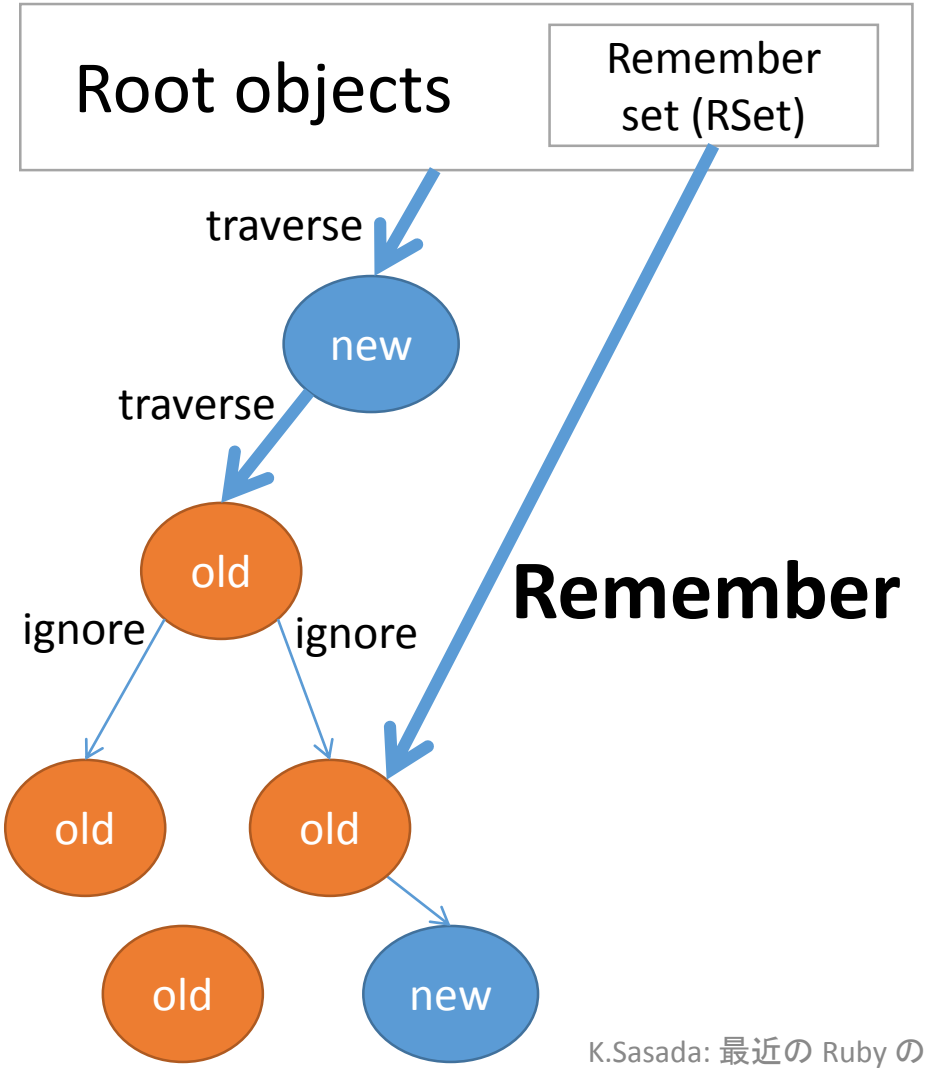
→ Minor GC causes marking leak!!

- Because minor GC ignores referenced objects by old objects

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

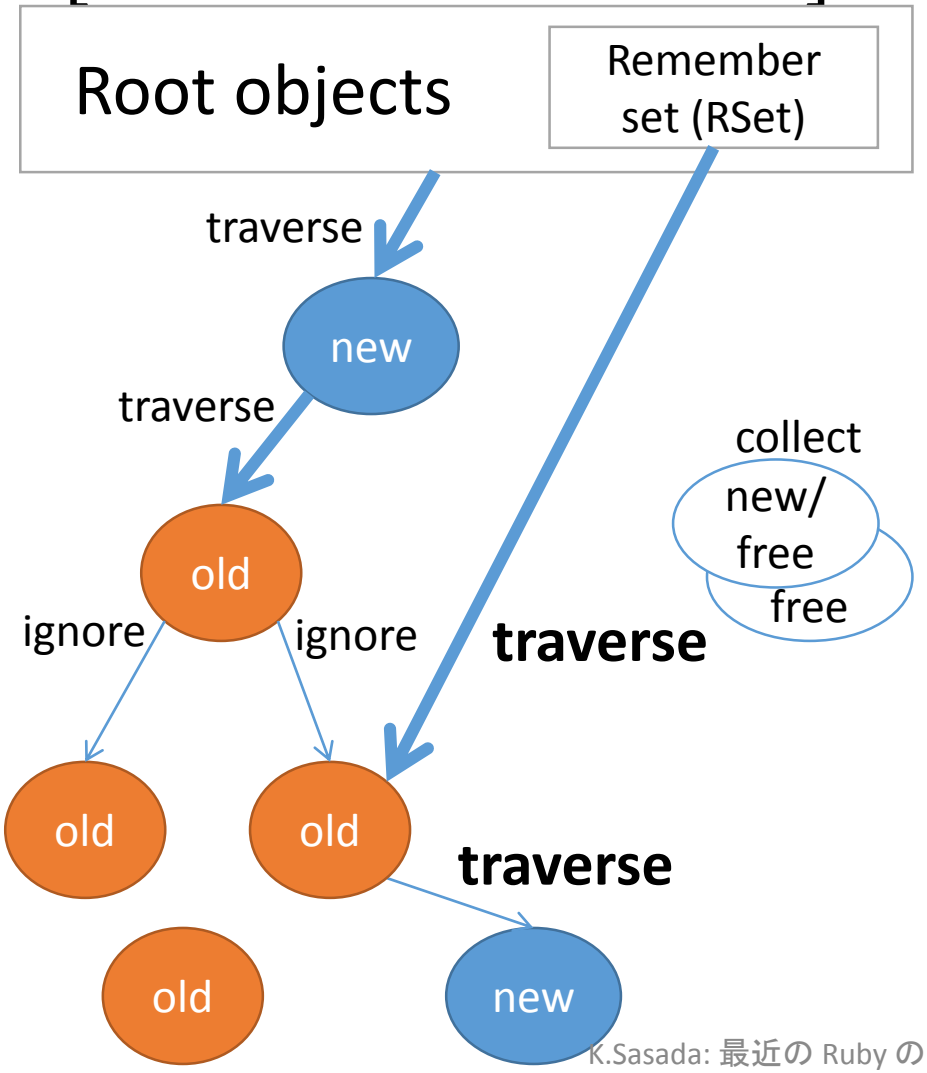
K.Sasada: 最近の Ruby のメモリ管理, 2014

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

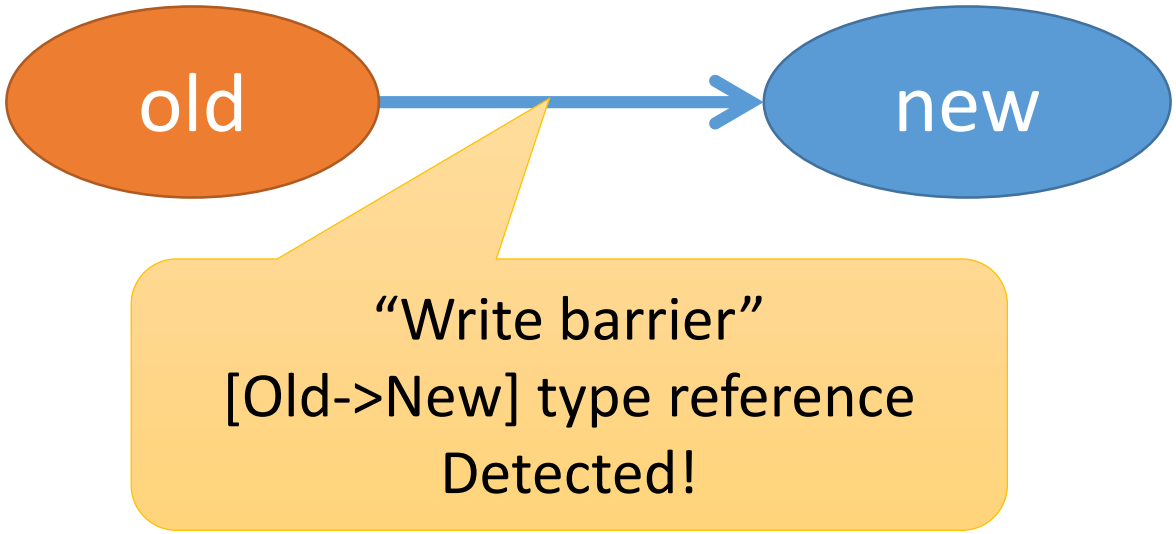


1. Mark reachable objects from root objects
   - **<u>Remembered objects are also root objects</u>**
2. Sweep not (marked or old) objects

K.Sasada: 最近の Ruby のメモリ管理, 2014

RGenGC: Background: GenGC Write barrier

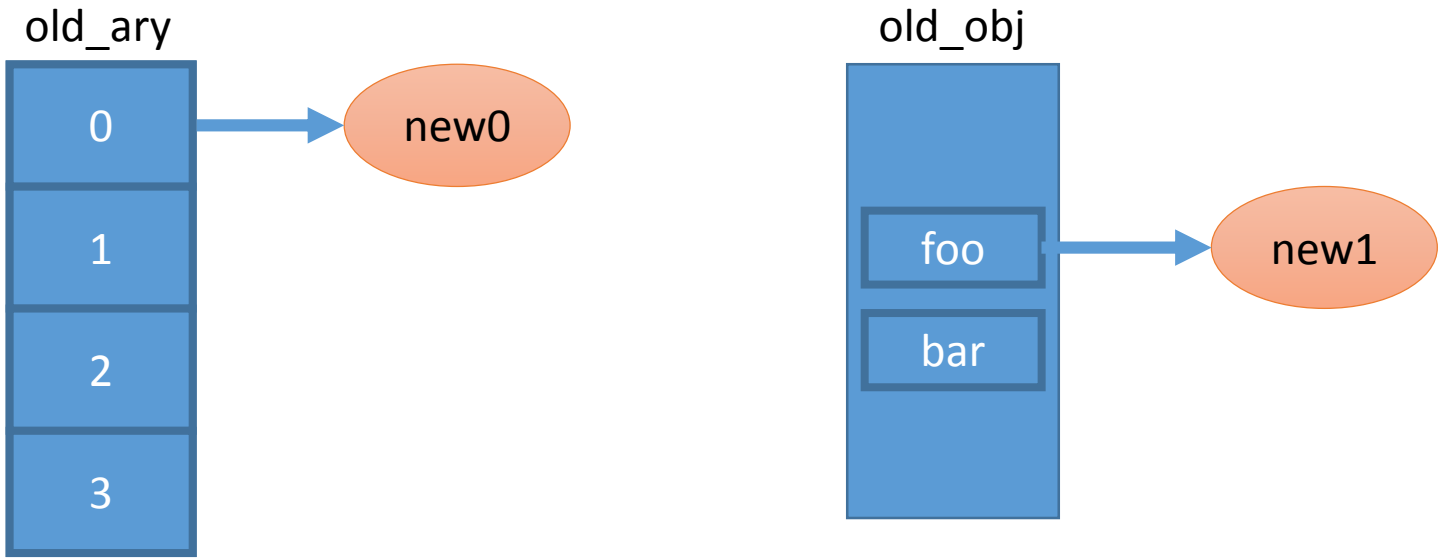
- To detect [old→new] type references, we need to insert **“Write-barrier”** into interpreter for all “Write” operation



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

		Performance	Compatibility
1	Give up GenGC	Poor	Good (No problem)
2	GenGC with re-writing all of C exts	Good	Most of C-exts doesn't work

2.0 and earlier conservative choice

Trade-off of Speed and Compatibility

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

		Performance	Compatibility
1	Give up GenGC	Poor	Good (No problem)
2	GenGC with re-writing all of C codes	Good	Most of C-exts doesn't work
3	Use new RGenGC	Good	Most of C-exts works!!

Ruby 2.1 choice

Breaking the trade off. You can praise us!!

RGenGC: Key idea

- Introduce **WB unprotected objects**

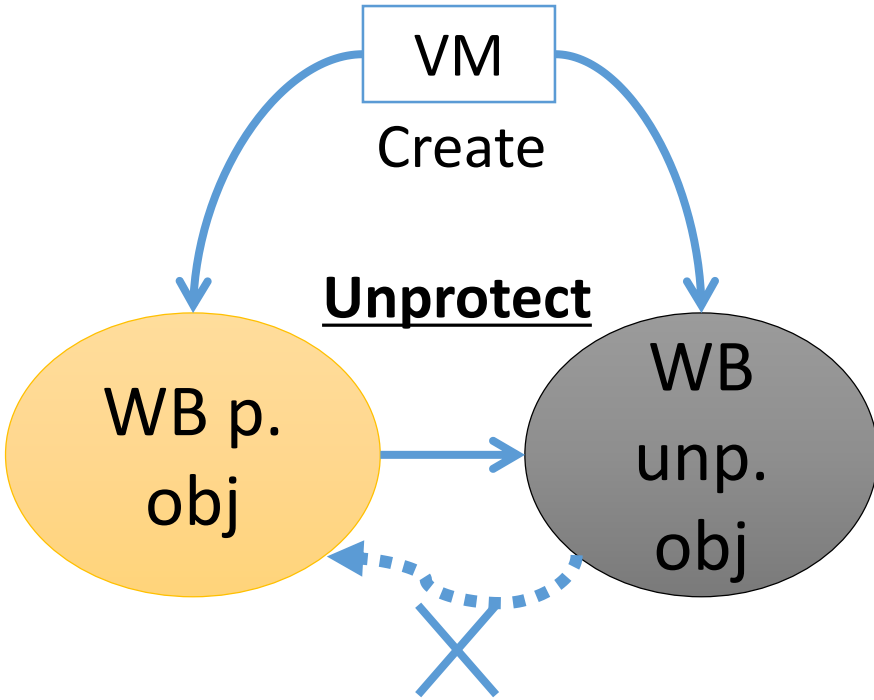
RGenGC:
Key Idea

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

- We are not sure that a WB unprotected objects point to new objects or not
- Decide this type at creation time
  - A class care about WB → WB protected object
  - A class don't care about WB → WB unprotected object

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”

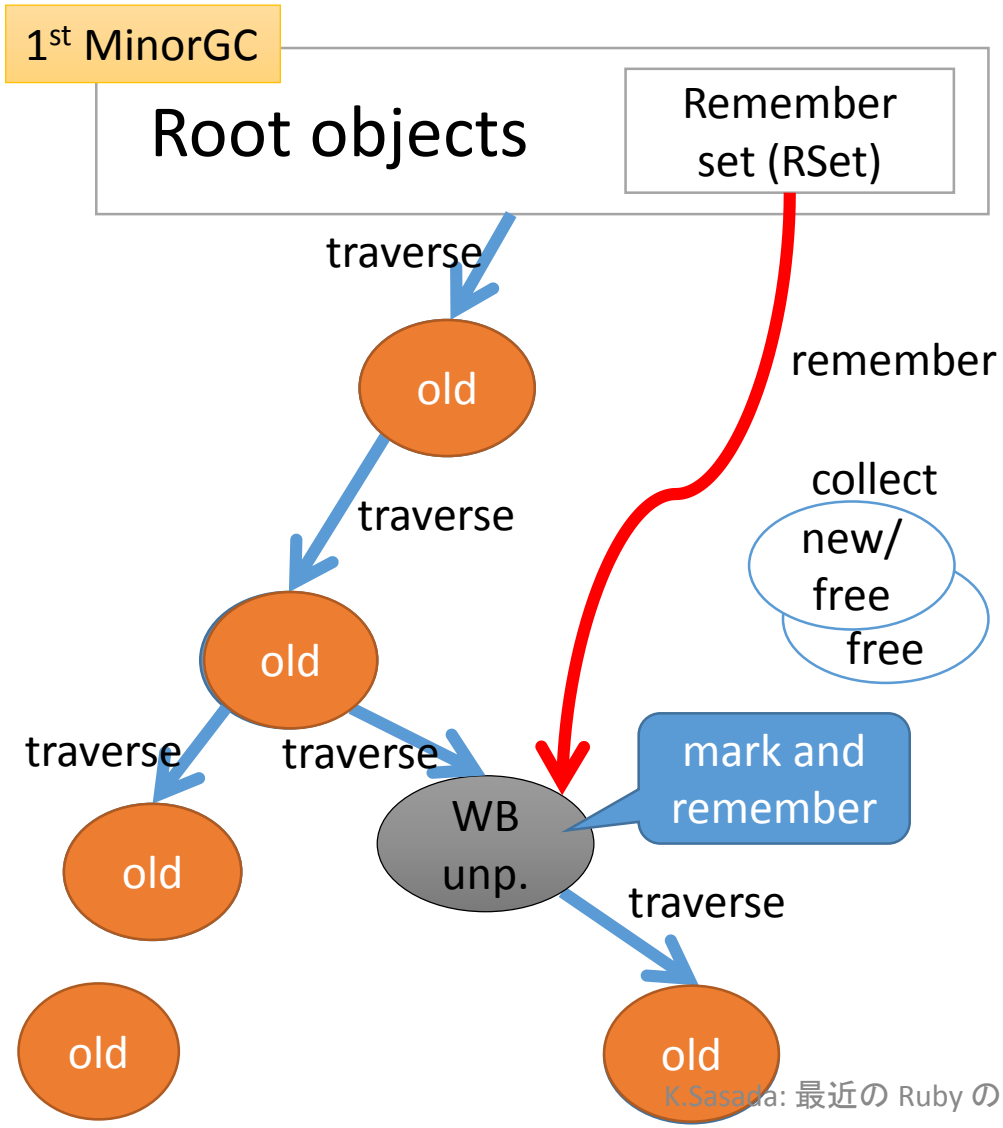


Now, WB unprotected object **can’t** change into WB p. object

RGenGC
Key Idea: Rule

- 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

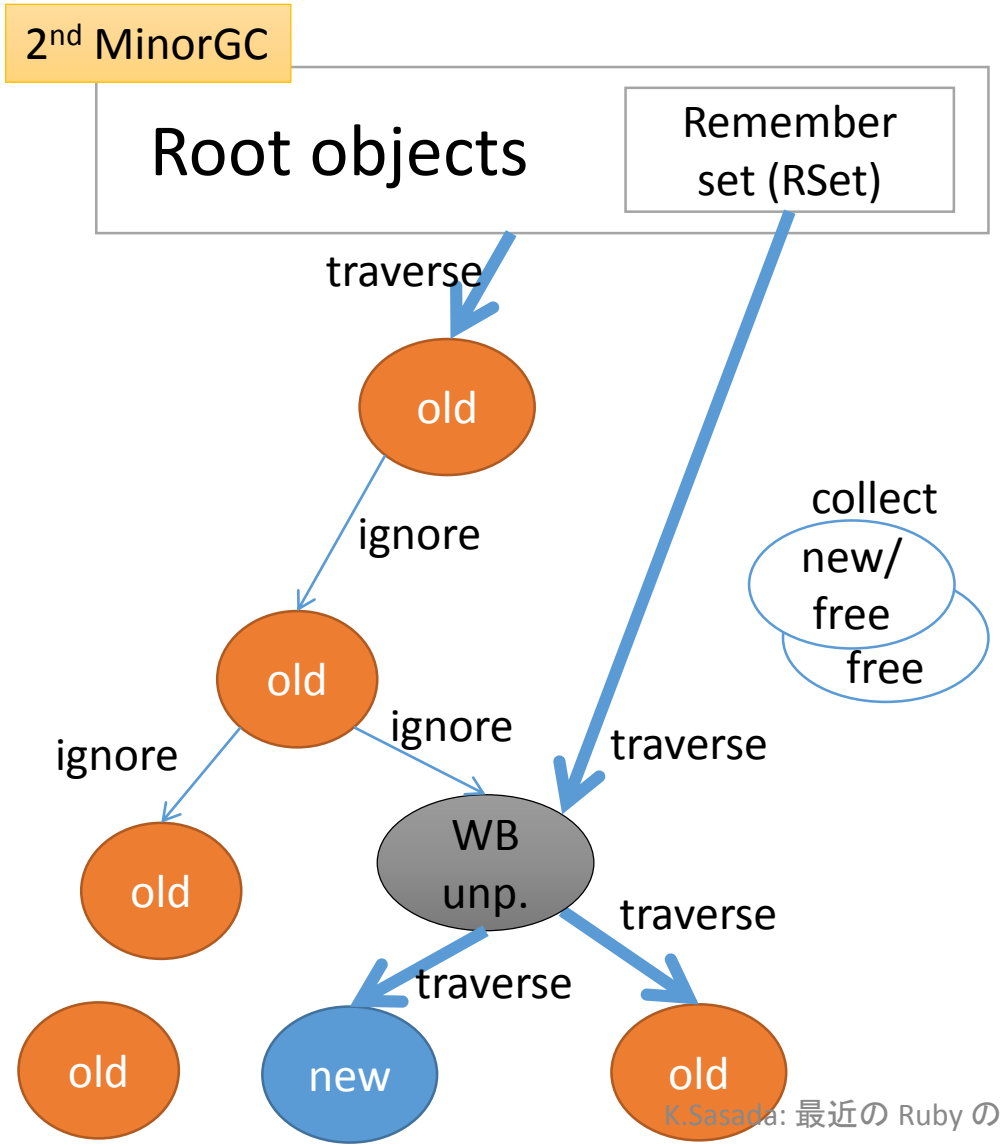
RGenGC
[Minor M&S GC w/WB unp. objects]



- 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!!

RGenGC
[Minor M&S GC w/WB unp. objects]

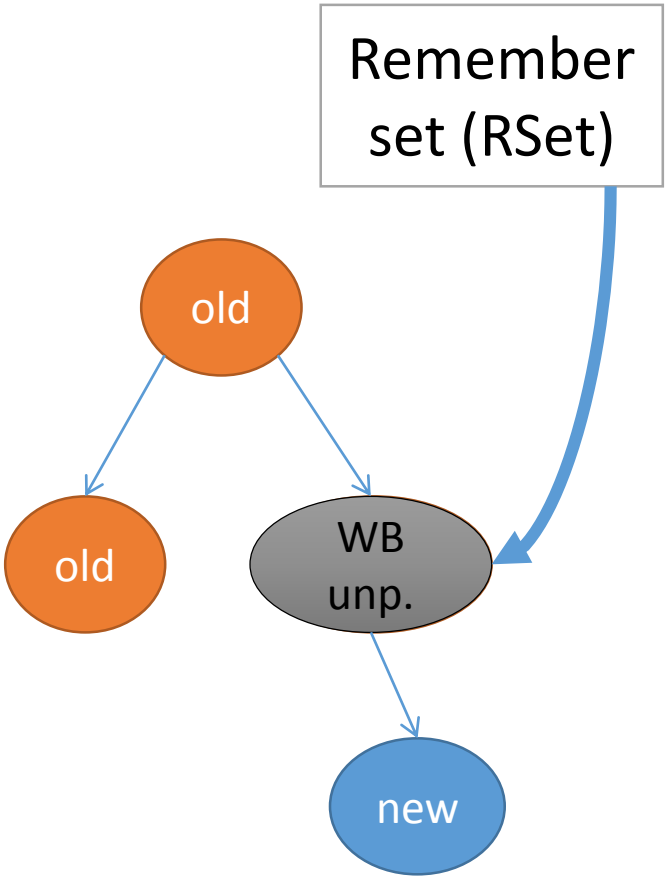


- Mark reachable objects from root objects
  - Mark WB unprotected objects, and ***don't promote*** them to old gen objects
  - If WB unprotected objects **pointed from old objects**, then **remember this WB unprotected objects** by RSet.

    → Mark WB unprotected objects every minor GC!!

K.Sasada: 最近の Ruby のメモリ管理, 2014

RGenGC [Unprotect operation]



- Anytime Object can give up to keep write barriers

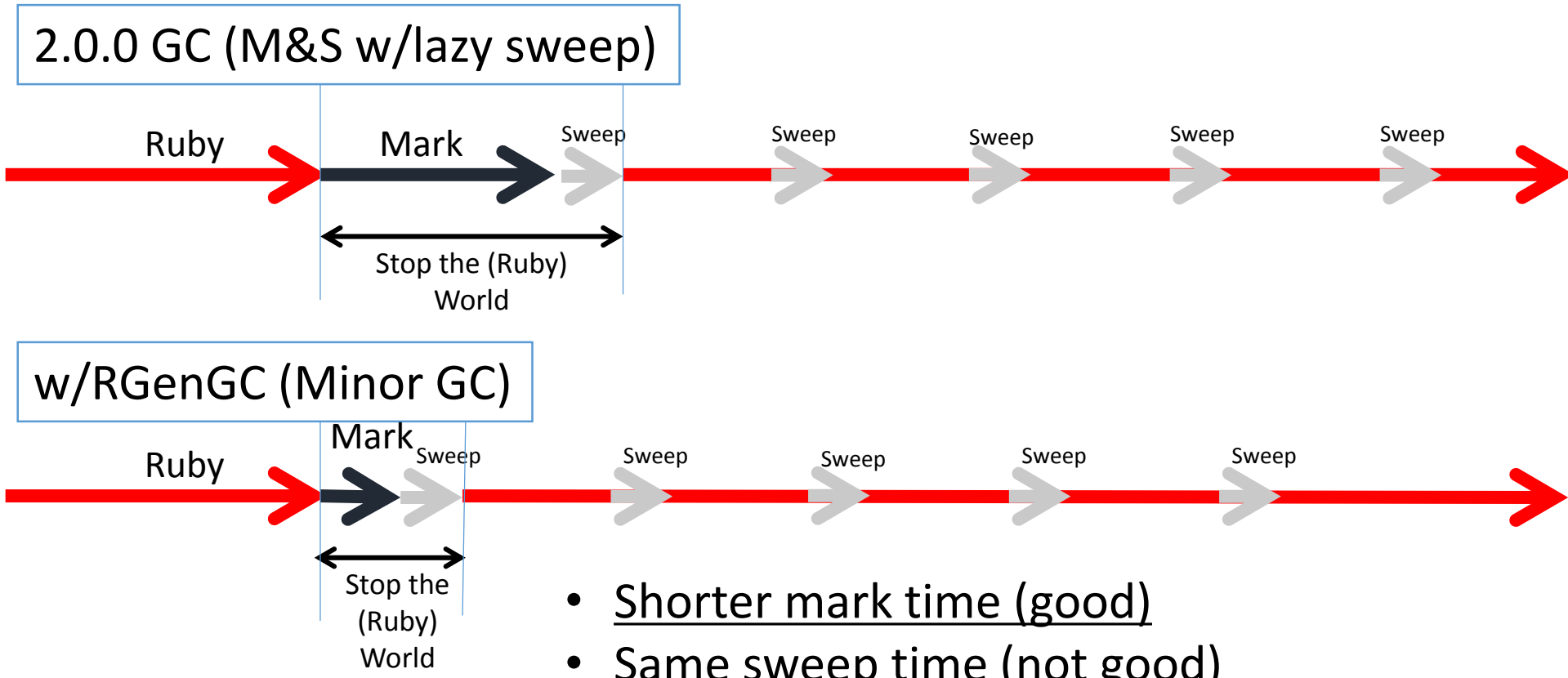
  → [Unprotect operation]

- Change old WB protected objects to WB unprotected objects
  - Example: RARRAY_PTR(ary)

  (1) Demote object (old → new)

  (2) Register it to Remember Set

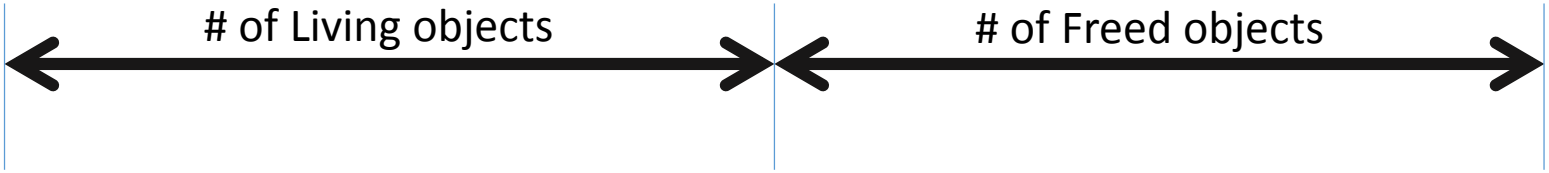
RGenGC Timing chart



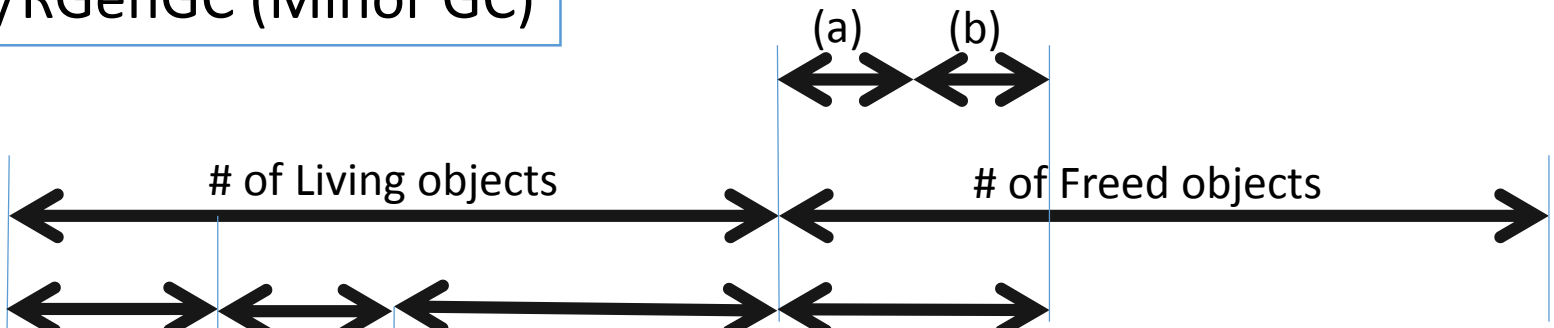
- Shorter mark time (good)
- Same sweep time (not good)
- (little) Longer execution time b/c WB (bad)

RGenGC
Number of objects

2.0.0 GC (M&S)



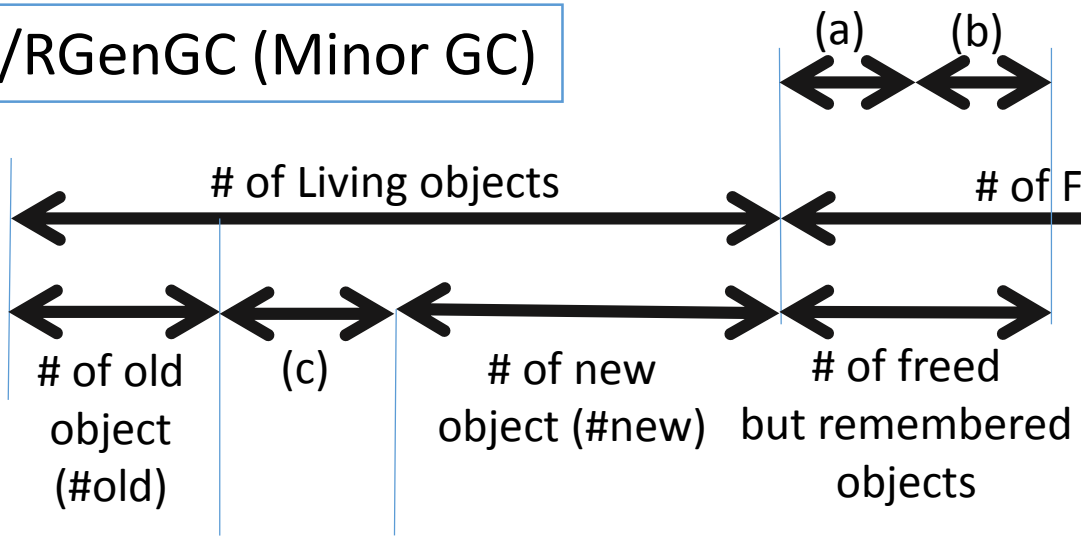
w/RGenGC (Minor GC)



(a) # of unused promoted objects
(b) # of unused WB unp. objects pointed by old
(c) # of WB unp. objects

RGenGC
Number of objects

w/RGenGC (Minor GC)



(a) # of unused promoted objects
(b) # of unused WB unp. objects pointed by old
(c) # of WB unp. objects

	Marking space	Number of unused, uncollected objs	Sweeping space
Mark&Swep GC	# of Living objects	0	Full heap
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
    - **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

RGenGC Discussion: Pros. and Cons.

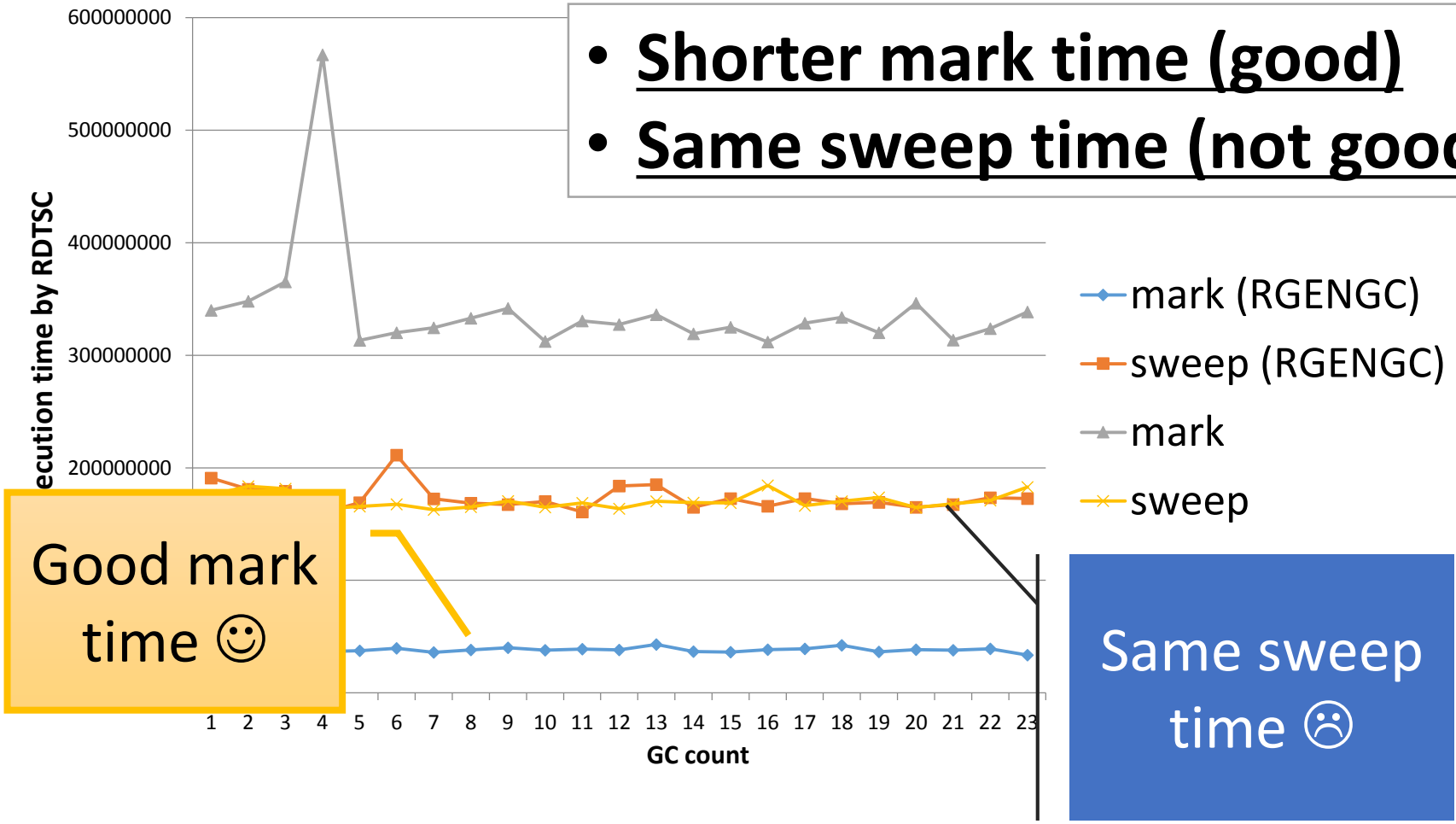
- Cons.
  - Increasing “unused, but not collected objects until full/major GC
    - Remembered normal objects (caused by traditional GenGC algorithm)
    - Remembered WB unprotected objects (caused by RGenGC algorithm)
  - WB insertion bugs (GC development issue)
    - WB protected objects need correct/perfect WBs. However, 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 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

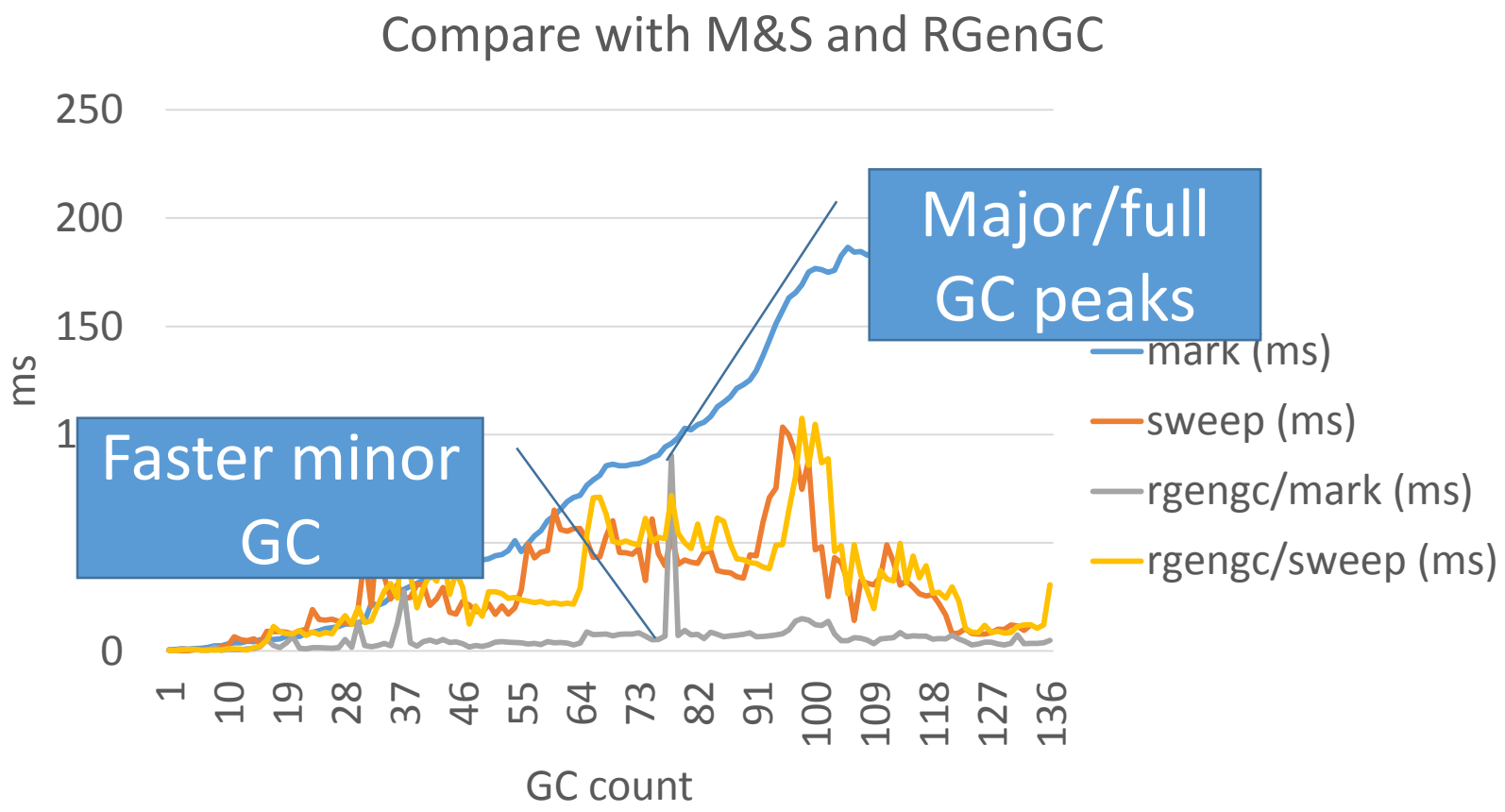
RGenGC
Performance evaluation (micro)

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



K.Sasada: 最近の Ruby のメモリ管理, 2014

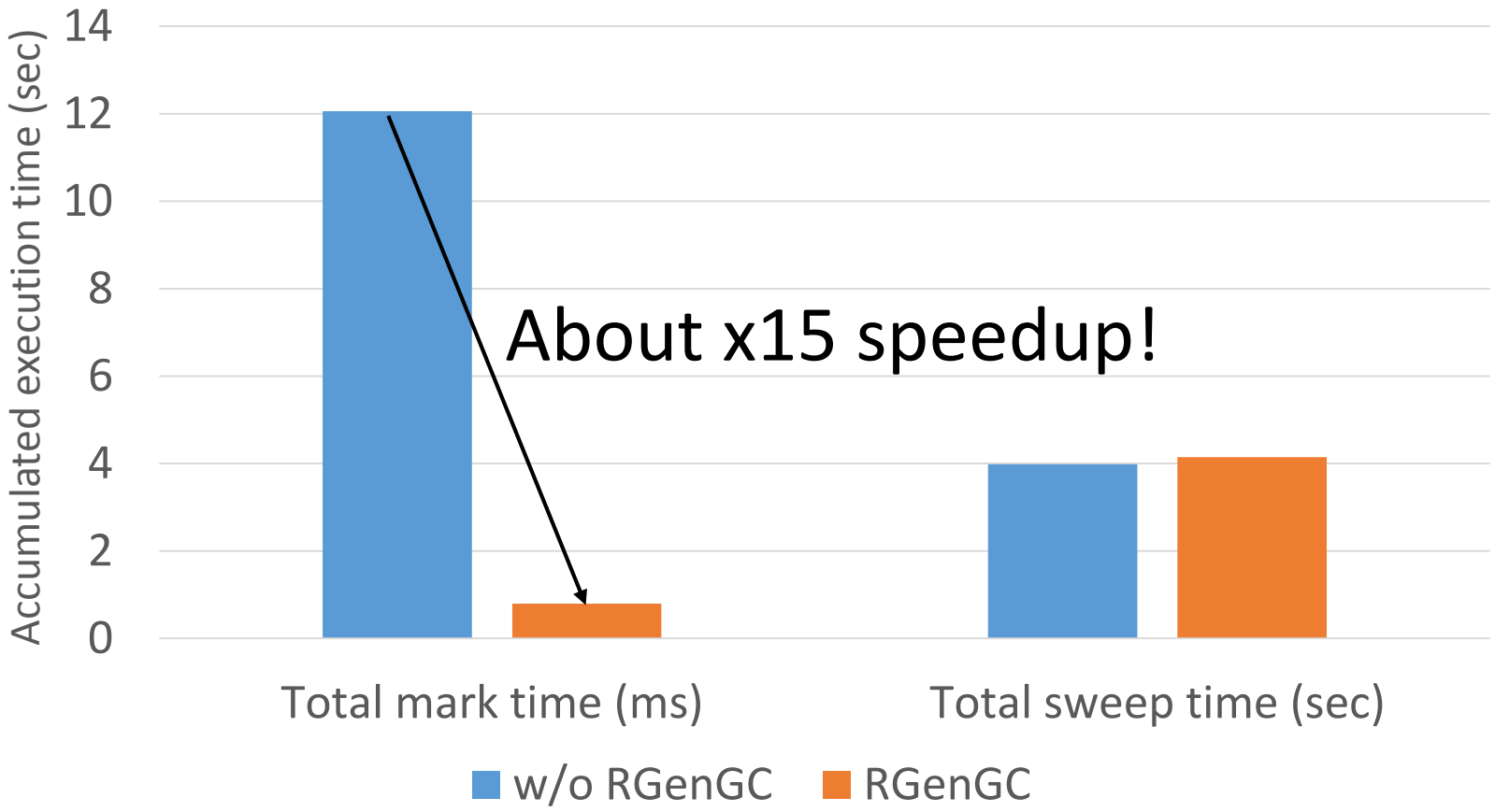
RGenGC
Performance evaluation (RDoc)



* Disabled lazy sweep to measure correctly.

K.Sasada: 最近の Ruby のメモリ管理, 2014

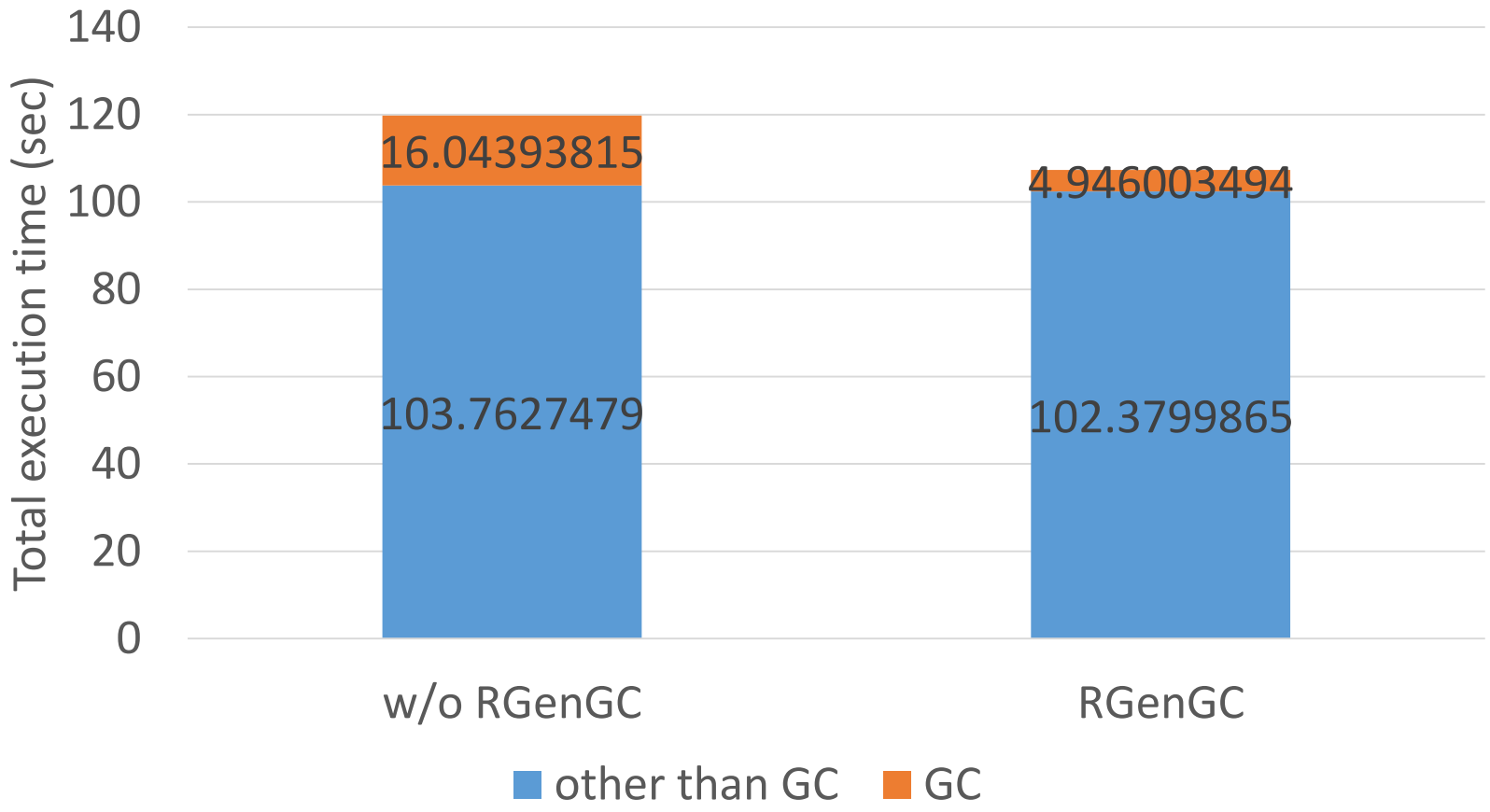
RGenGC
Performance evaluation (RDoc)



* Disabled lazy sweep to measure correctly.

K.Sasada: 最近の Ruby のメモリ管理, 2014

RGenGC
Performance evaluation (RDoc)



K.Sasada: 最近の Ruby のメモリ管理, 2014

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

RincGC: Restricted incremental GC

RincGC
Background and motivation

- Ruby 2.1 had introduced generational GC
  - Short marking time on minor GC
  - Improve application throughput
- Still long pause time on major GC
  - Long pause time affects user response time

Proposal:
RincGC: Incremental GC for major GC

- Introducing incremental GC to reduce pause time
- Can combine with Generational GC

	Generational GC	Incremental GC	Gen+Inc GC
Throughput	High	Low (a bit slow)	High
Pause time	Long	Short	Small

RincGC: Base idea
Incremental GC algorithm

- Move forward GC processes incrementally
  - Mark slots incrementally
  - Sweep slots incrementally
- Incremental marking in 3 phase
  - (1) Mark roots (pause)
  - (2) Mark objects reachable from roots (incremental)
  - (3) Mark roots again, and mark remembered objects (pause)
- Mark objects with three state (white/grey/black)
  - White: Untouched objects
  - Grey: Marked, and prepare to mark directly reachable objects
  - Black: Marked, and all directly reachable objects are marked
- Use write barriers to avoid marking miss from marked objects to live objects
  - Detect new reference from black objects to white objects
  - Remember such source black objects (marked at above (3))

RincGC: Incremental GC for CRuby/MRI

- Incremental marking
  - (1) mark roots (gc_mark_roots())
  - (2) Do incremental mark at rb_newobj_of()
  - (3) Make sure write barrier with WB-protected objects
  - (4) Take care of **WB-unprotected objects** (MRI specific)
- Incremental sweeping
  - Modify current lazy sweep implementation

RincGC: Incremental marking

- (1) mark roots (gc_mark_roots())
  - Push all root objects onto “mark_stack”
- (2) Do incremental mark at rb_newobj_of()
  - Fall back incremental marking process periodically
  - Consume (pop) some objects from “mark_stack” and make forward incremental marking
- (3) Make sure write barrier with WB-protected objects
  - Mark and push pointed object onto “mark_stack”
- (4) Take care of **WB-unprotected objects** (MRI specific)
  - After incremental marking (“mark_stack” is empty), re-scan all roots and all living non-WB-protected objects
  - WB-unprotected objects are represented by bitmap (WB_UNPROTECTED_BITS)

RincGC: Incremental marking

```
def mark(obj)
 return if obj.mark_bit
 obj.mark_bit = true
 obj.marking_bit = true
 $mark_stack.push(obj)
end

def start_marking
 GC.state = :mark
 $root_objects{|o| mark(o)}
end

def incremental_mark(n)
 n.times{
  return if $mark_stack.empty? && finish_marking
  obj = mark_stack.pop
  reachable_objects_from(obj){|o| mark(o)}
  obj.marking_bit = false
 }
end
```

```
def finish_marking
 root_objects{|o| mark(o)} # re-scan root objects
 return false unless mark_stack.empty?
 $marked_wb_unprotected_objects.each{|unprotected_obj|
  unprotected_obj.reachable_objects{|o| mark(o)}
 }
 mark(obj) while obj = $mark_stack.pop

 GC.state = :sweep
 return true
end

def write_barrier(a, b)
 if GC.state == :mark && a.mark_bit && !a.marking_bit && !b.mark_bit
  a.marking_bit = true
  mark(b) and $mark.stack.push(b)
 end
end
```

K.Sasada: 最近の Ruby のメモリ管理, 2014

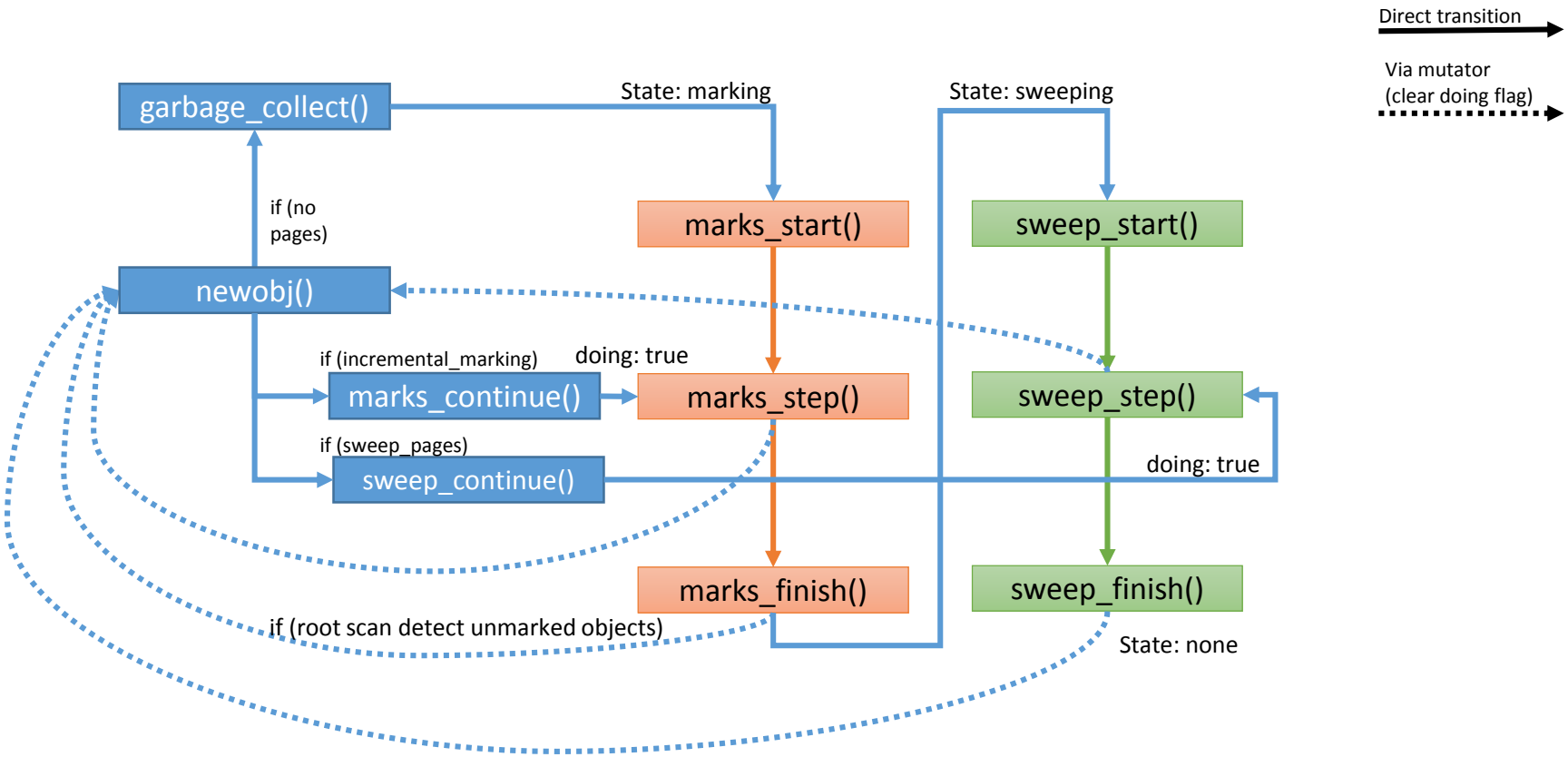
RincGC: Incremental marking

Traditional GC coloring terminology	RincGC	
	mark_bit	marking_bit
White	FALS	FALSE
Grey	TRUE	TRUE
Black	TRUE	FALSE

RincGC: Incremental sweeping

- Current implementation
  - Iterate until no pages
    - Sweep 1 page (a set of slots)
    - Consume 1 page
  - After that, no empty pages
- Modify implementation
  - Iterate
    - Sweep 2 page (a set of slots)
    - Consume ***1*** page (1 page remain)
  - After that, half of pages are left
  - We can use this half of pages for incremental marking

RincGC: Diagram



K.Sasada: 最近の Ruby のメモリ管理, 2014

Summary

- Ruby’s new two GC implementation
  - RGenGC: Restricted Generational GC
  - RincGC: Restricted incremental GC

Thank you for your attention
Q&A?

Koichi Sasada

<ko1@heroku.com>

heroku