

Rubyにおける
ライトバリアのないオブジェクトを考慮した
世代別インクリメンタルGCの実装

笹田 耕一, 松本 行弘
(Heroku, Inc.)

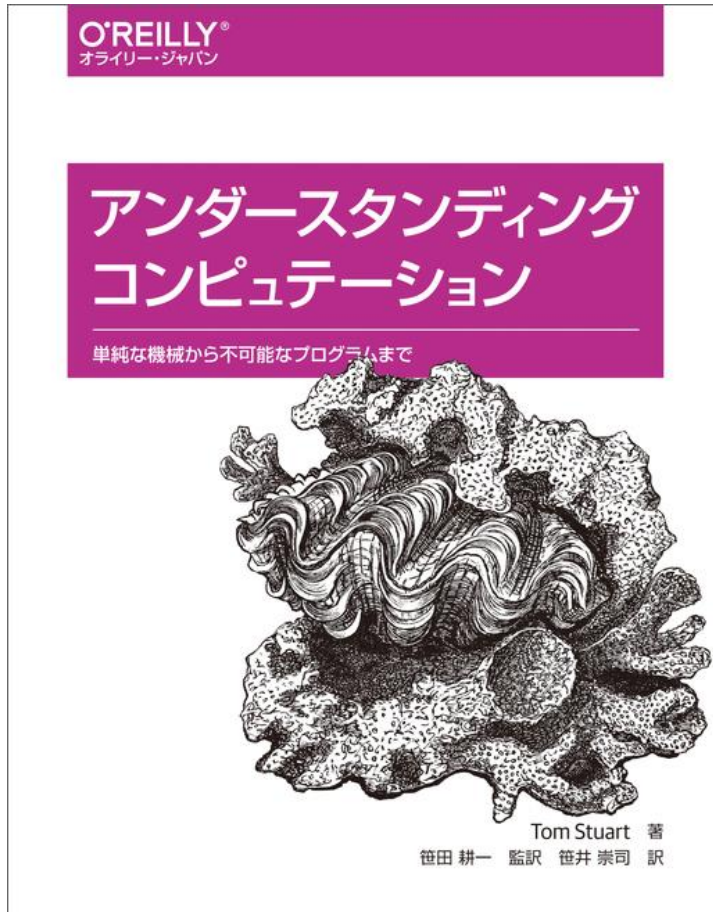
この発表を3行でまとめると

- Ruby処理系にはライトバリア(WB)が入っていないし、実装大変
- 世代別GC、インクリメンタルGCはWBが無いと実装できない
- 完璧にWBが入って無くても出来るアルゴリズムを考えました

Programming language Ruby

- Object oriented programming language Ruby
- Ruby is widely used in world-wide.

Rubyが使われている例



- アンダースタンディング コンピューテーション(オライリー)
- Tom Stuart著、笹田 耕一監訳、笹井 崇司訳
- Ruby で意味論、オートマトン、チューリング機械、ラムダ計算、その他の万能機械、停止性問題、型推論を解説

Programming language Ruby

- Object oriented programming language Ruby
- Ruby is widely used in world-wide.
 - Web application development, such as Twitter (early days), github, and so on.
- Performance of Ruby interpreter is a matter.

One of performance bottleneck Ruby's GC

- Conservative mark and sweep GC (from 1993)
- Issues:
 1. Low throughput
 2. Long pause time

Traditional solution: well-known GC algorithms

- Generational GC (GenGC) to improve throughput by separating heaps per generations.
- Incremental GC (IncGC) to reduce pause time by doing GC steps incremental
- They needs “Write Barriers” (WB) to work correctly

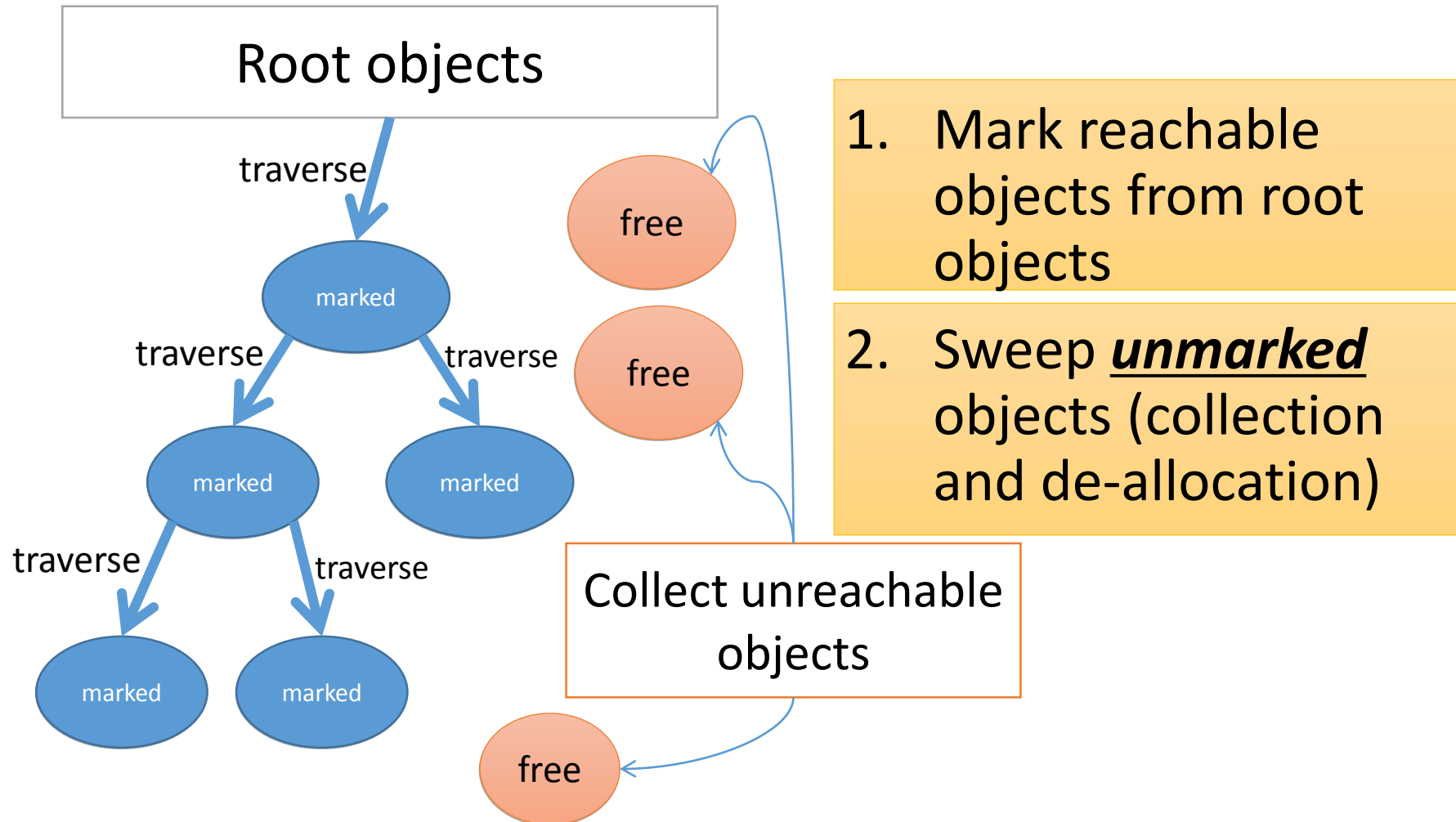
Challenge

- Inserting perfect WBs is very difficult
 - Unfortunately, Ruby interpreter does not support WBs
 - High implementation cost for perfect WBs
 - API change becomes compatibility issue for existing extension libraries
- ***Allowing GenGC and IncGC with “Incomplete” WB insertions***
 - ***RGenGC: Restricted Generational GC algorithm***
 - ***RincGC: Restricted Incremental GC algorithm***

RGenGC: Restricted Generational GC algorithms

RGenGC algorithm is implemented at Ruby 2.1
which is already shipped.

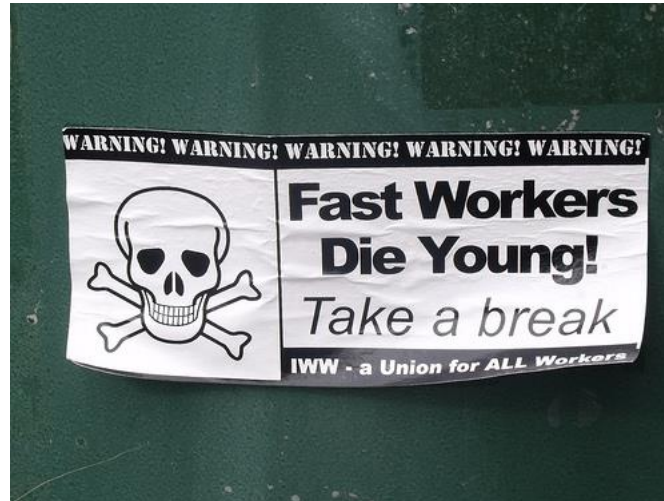
Simple Mark & Sweep



Generational GC (GenGC)

- Weak generational hypothesis:

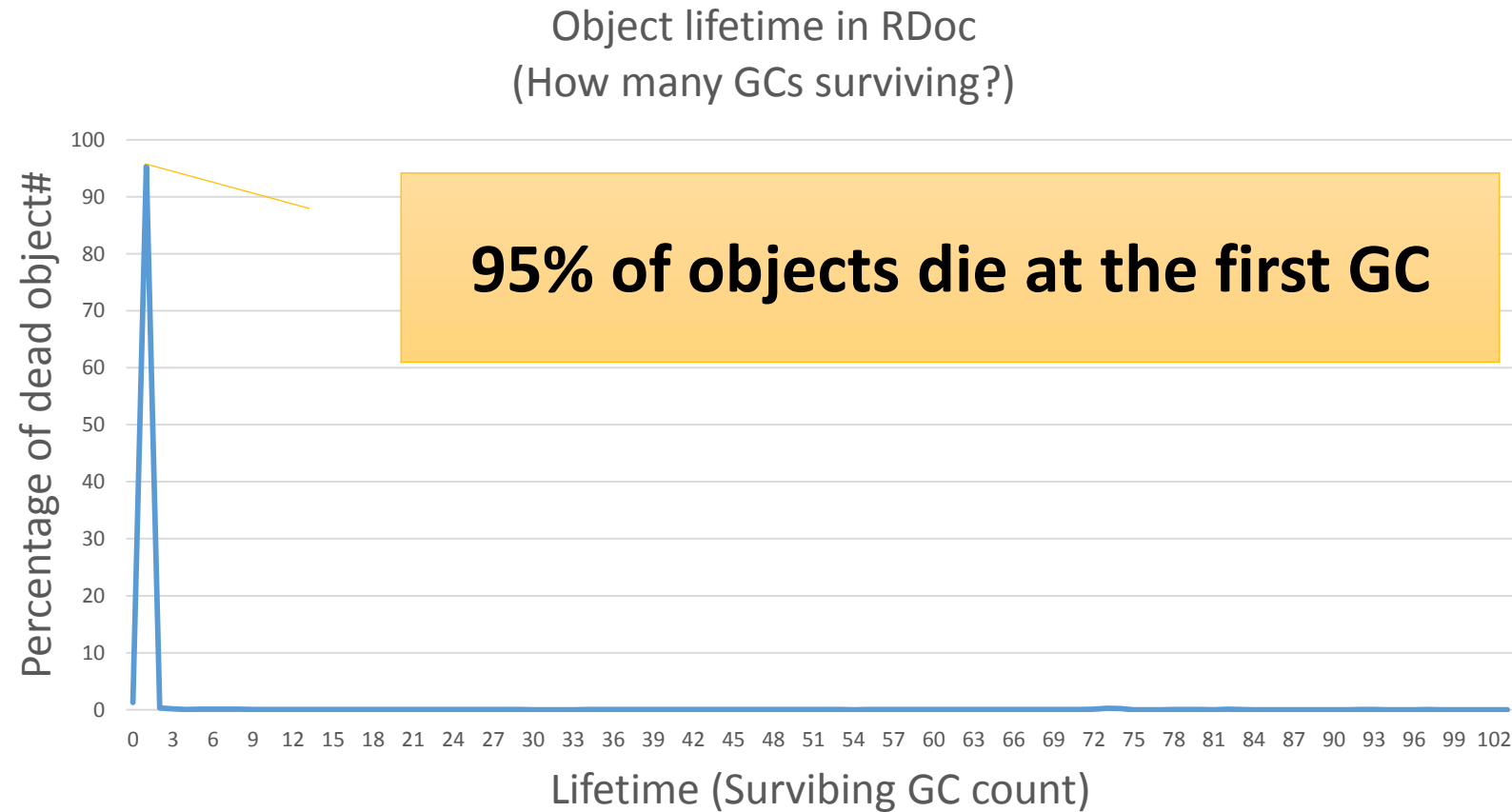
“Most objects die young”



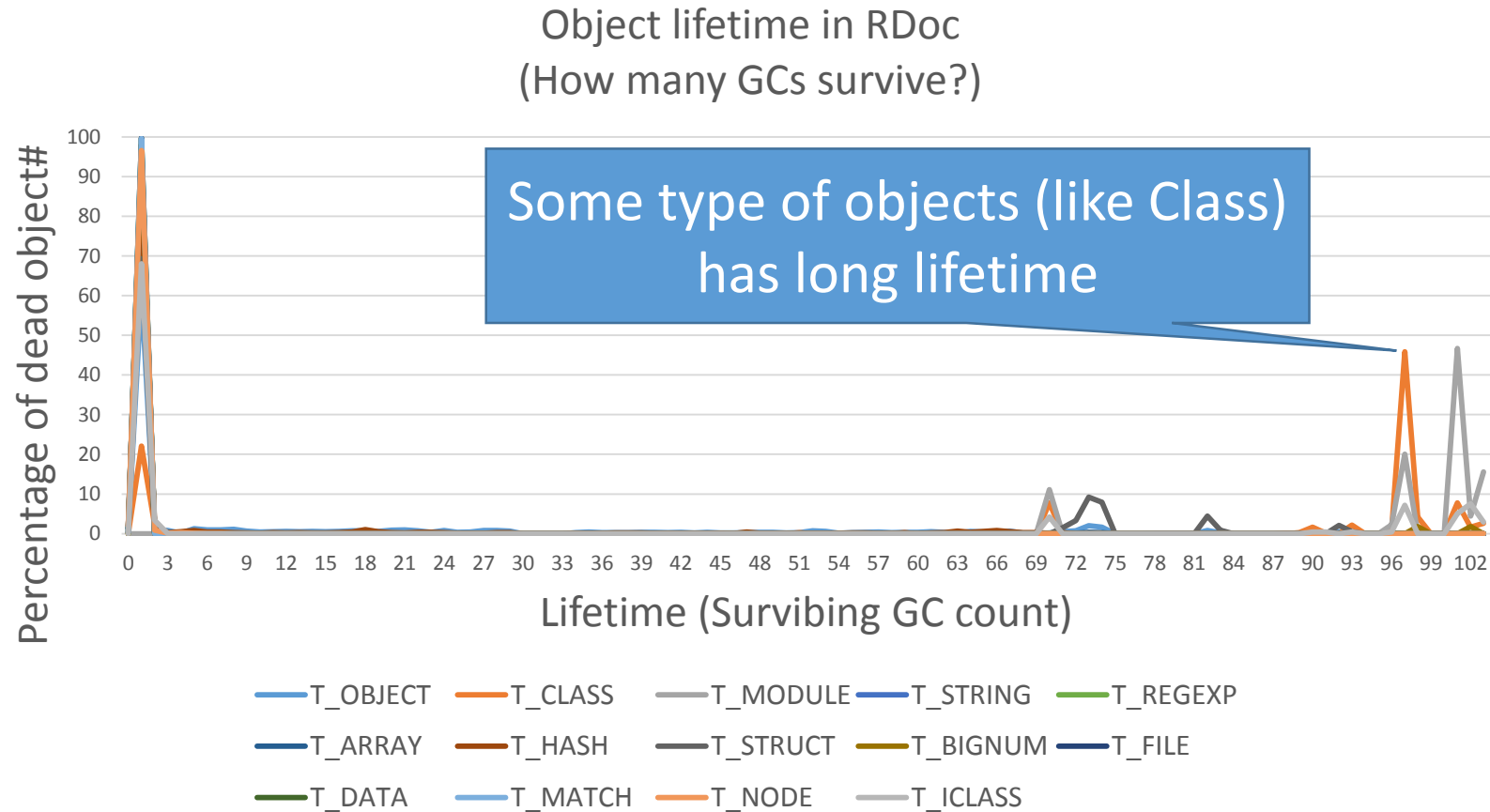
<http://www.flickr.com/photos/ell-r-brown/5026593710>

**→ Concentrate reclamation effort
only on the young objects**

Generational hypothesis



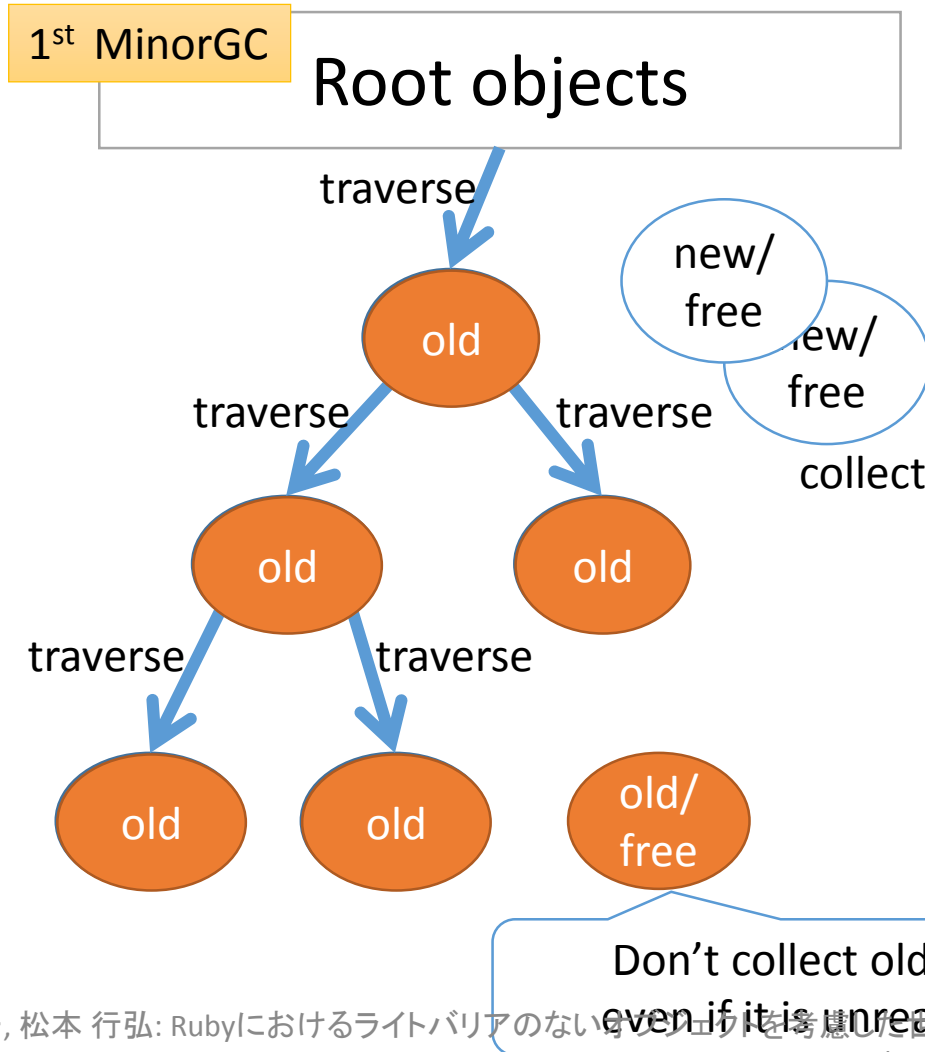
Generational hypothesis



Generational GC (GenGC)

- Separate young generations and old generations
 - Create objects as youngest generation
 - Promote to old generations after surviving GCs
- Usually, GC on young space (minor GC)
- GC on both spaces if no memory (major/full GC)
- We can choose different GC algorithms on Minor GC and Major GC
 - In this presentation, we choose M&S for both GCs (because we choose M&S)

[Minor M&S GC]

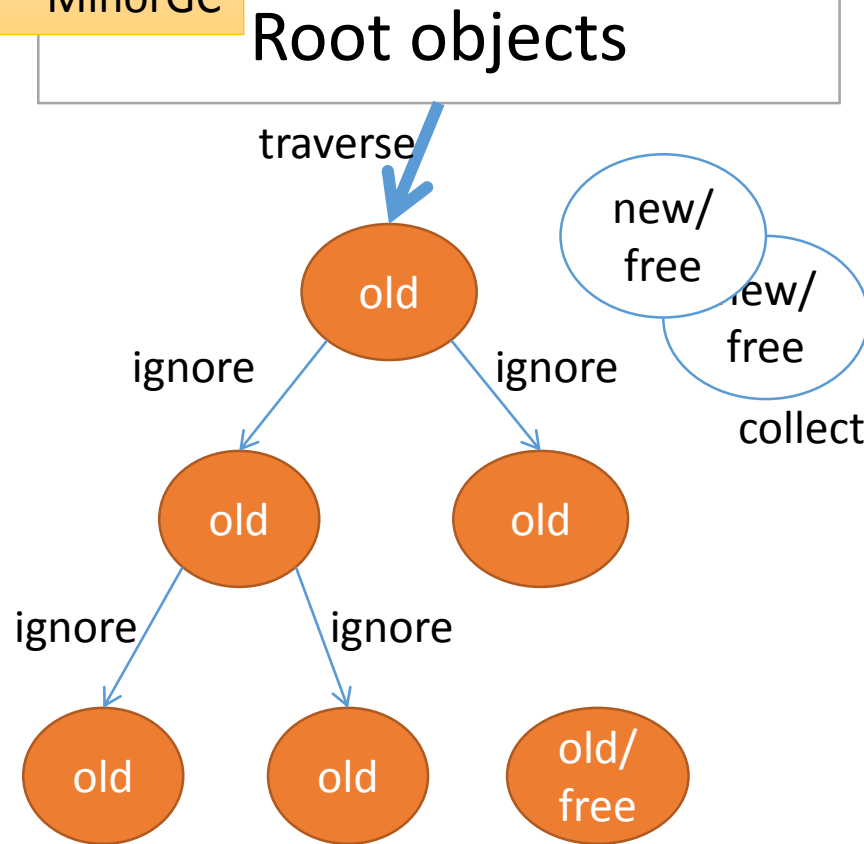


- Mark reachable objects from root objects.
 - Mark and **promote to old generation**
 - Stop traversing after old objects
- **Reduce mark overhead**
- Sweep not (marked or old) objects
- Can't collect Some unreachable objects

RGenGC: Background: GenGC

[Minor M&S GC]

2nd MinorGC

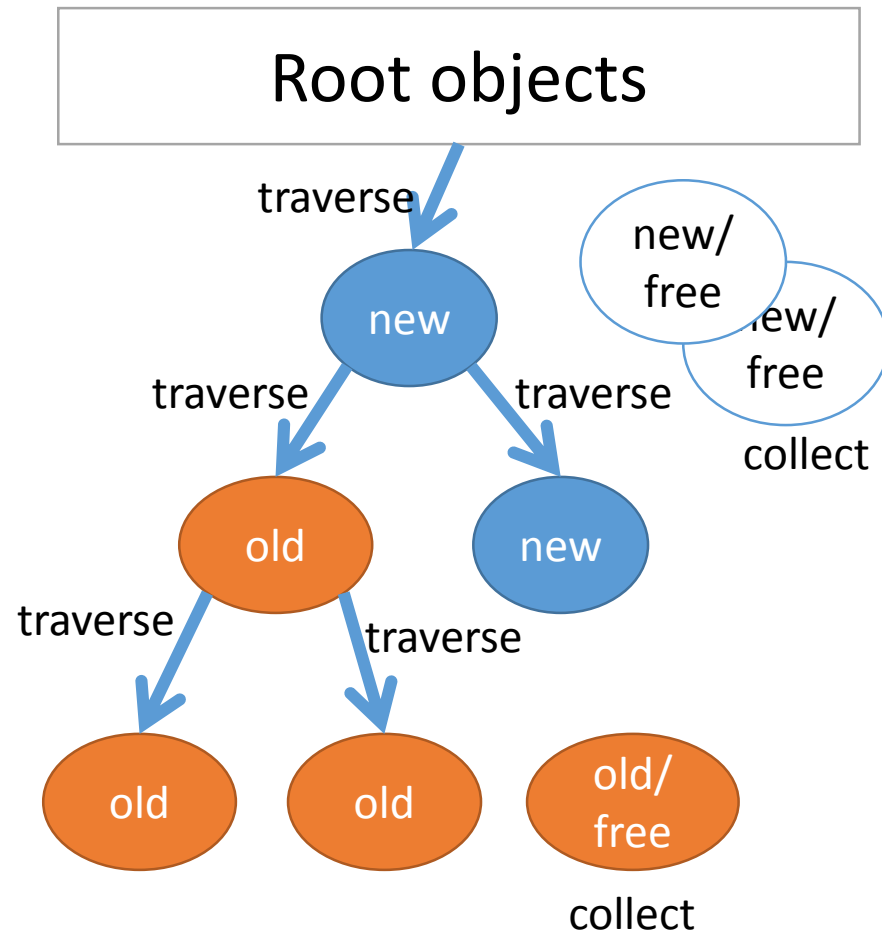


- Mark reachable objects from root objects.
 - Mark and **promote to old generation**
 - Stop traversing after old objects
- **→ Reduce mark overhead**
- Sweep not (marked or old) objects
- Can't collect Some unreachable objects

Don't collect old object even if it is unreachable

RGenGC: Background: GenGC

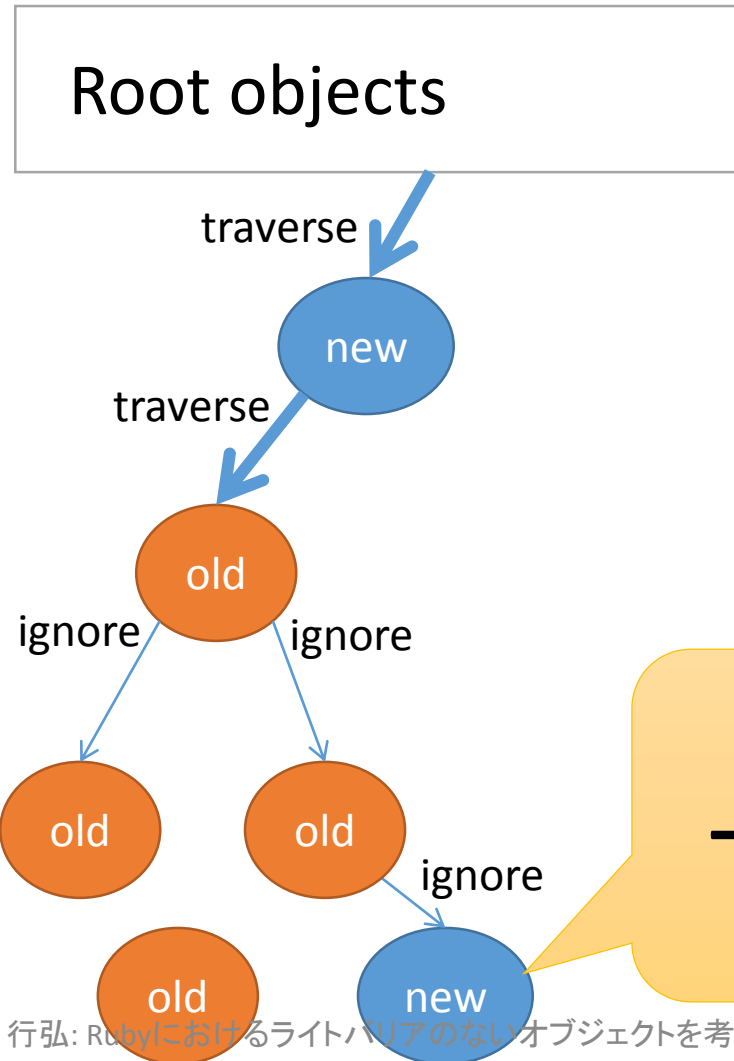
[Major M&S GC]



- 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**

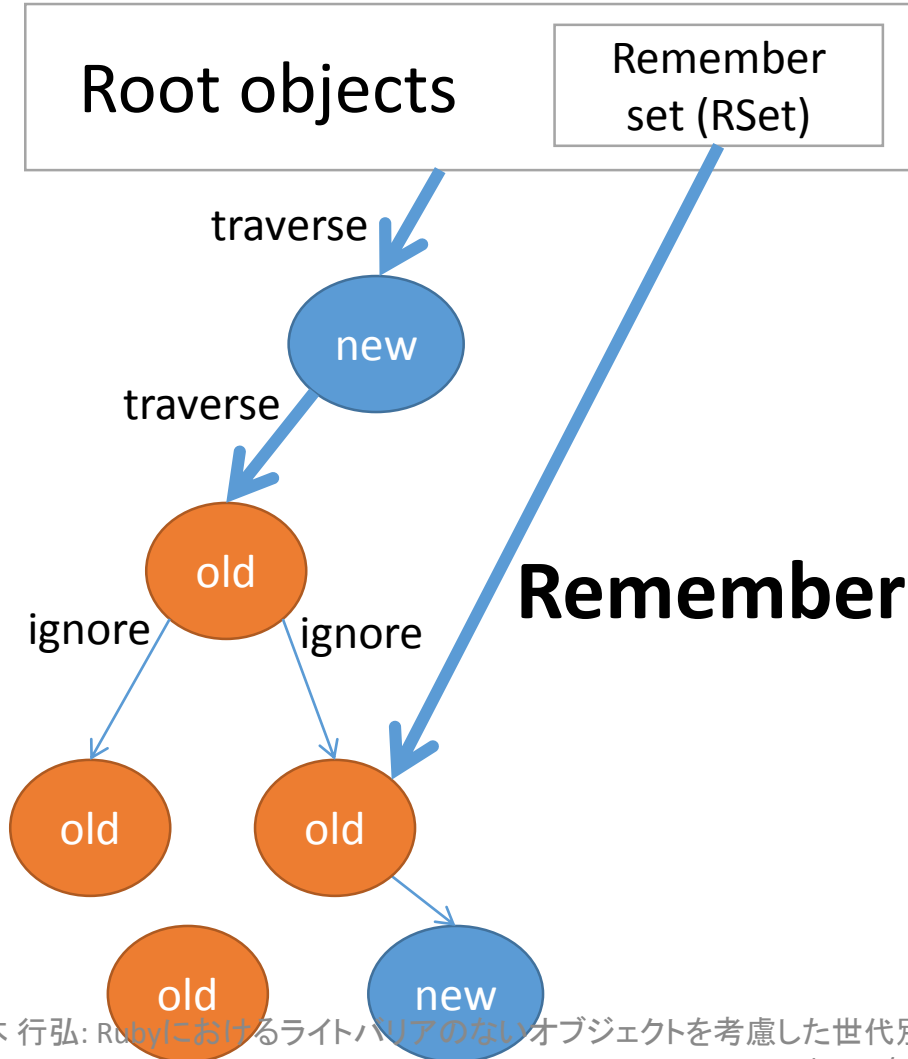
marking leak!!

- Because minor GC ignores referenced objects by old objects

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

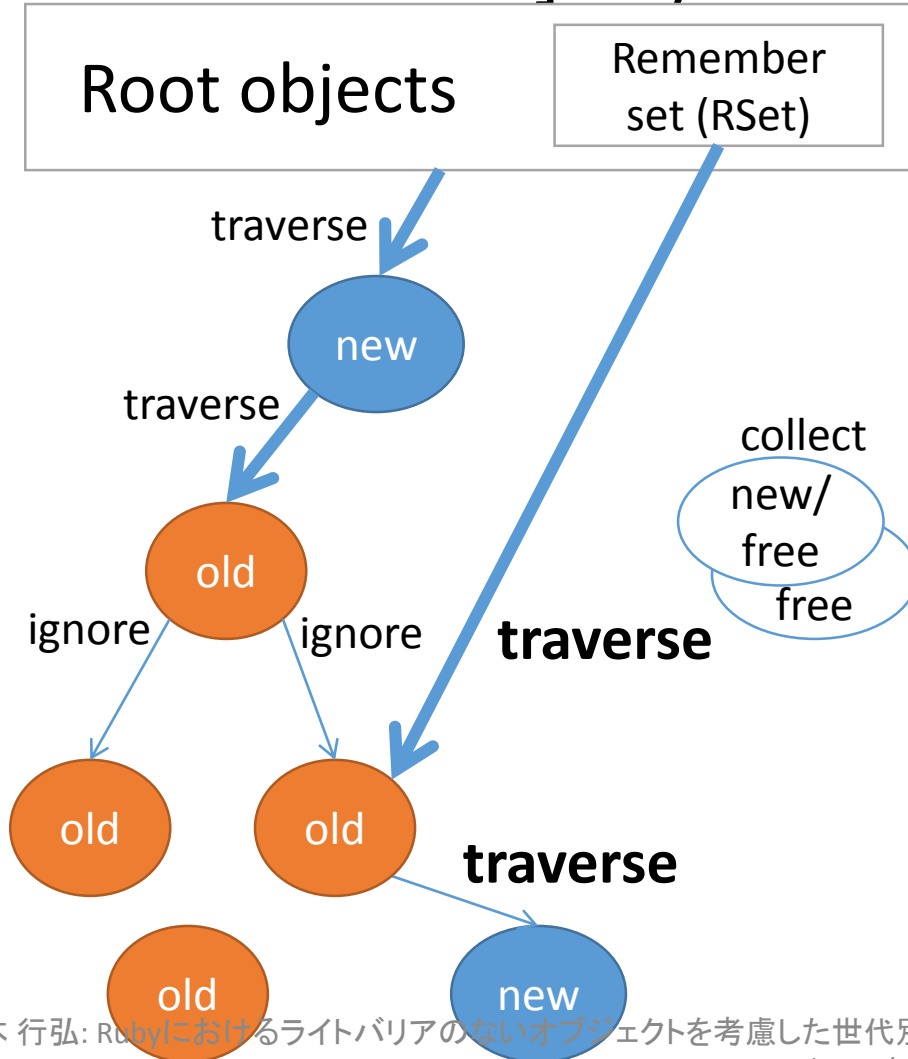
RGenGC: Background: GenGC

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

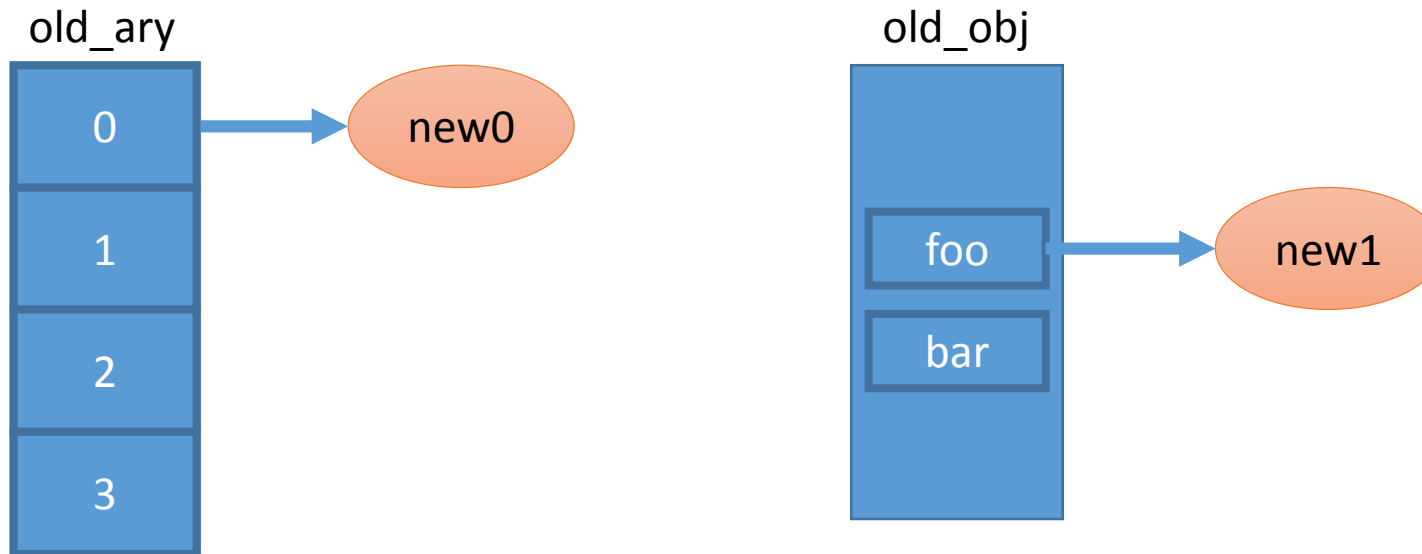
RGenGC: Background: GenGC [Minor M&S GC] w/ RSet



1. Mark reachable objects from root objects
 - Remembered objects are also root objects
2. Sweep not (marked or old) objects

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`]



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!!**

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

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

Ruby 2.0 and earlier conservative choice

Trade-off of Speed and Compatibility

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!!

RGenGC:

Key idea

- Introduce **WB unprotected objects**

RGenGC:

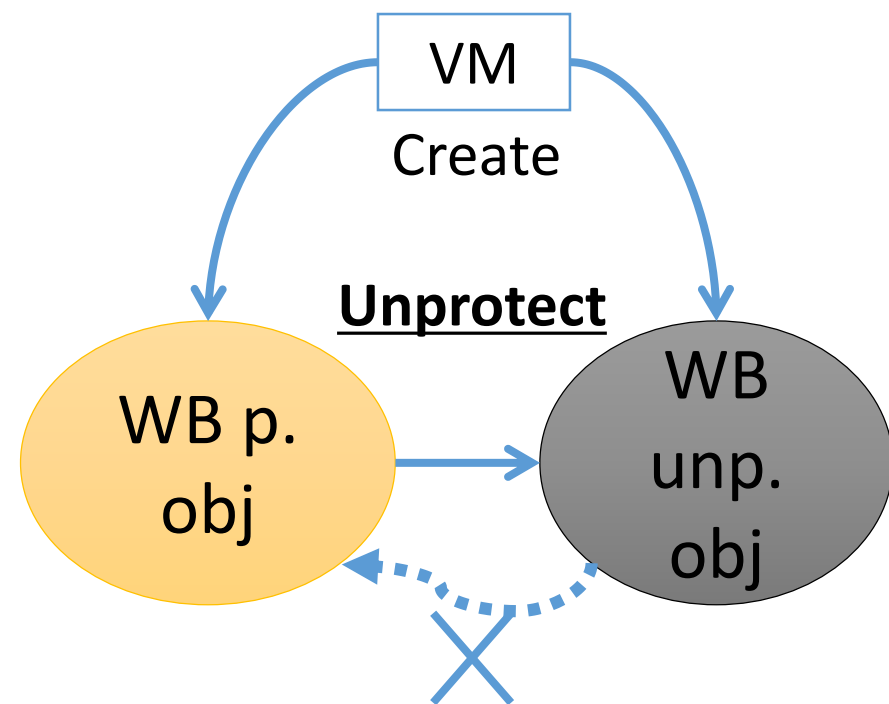
Key idea

- **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

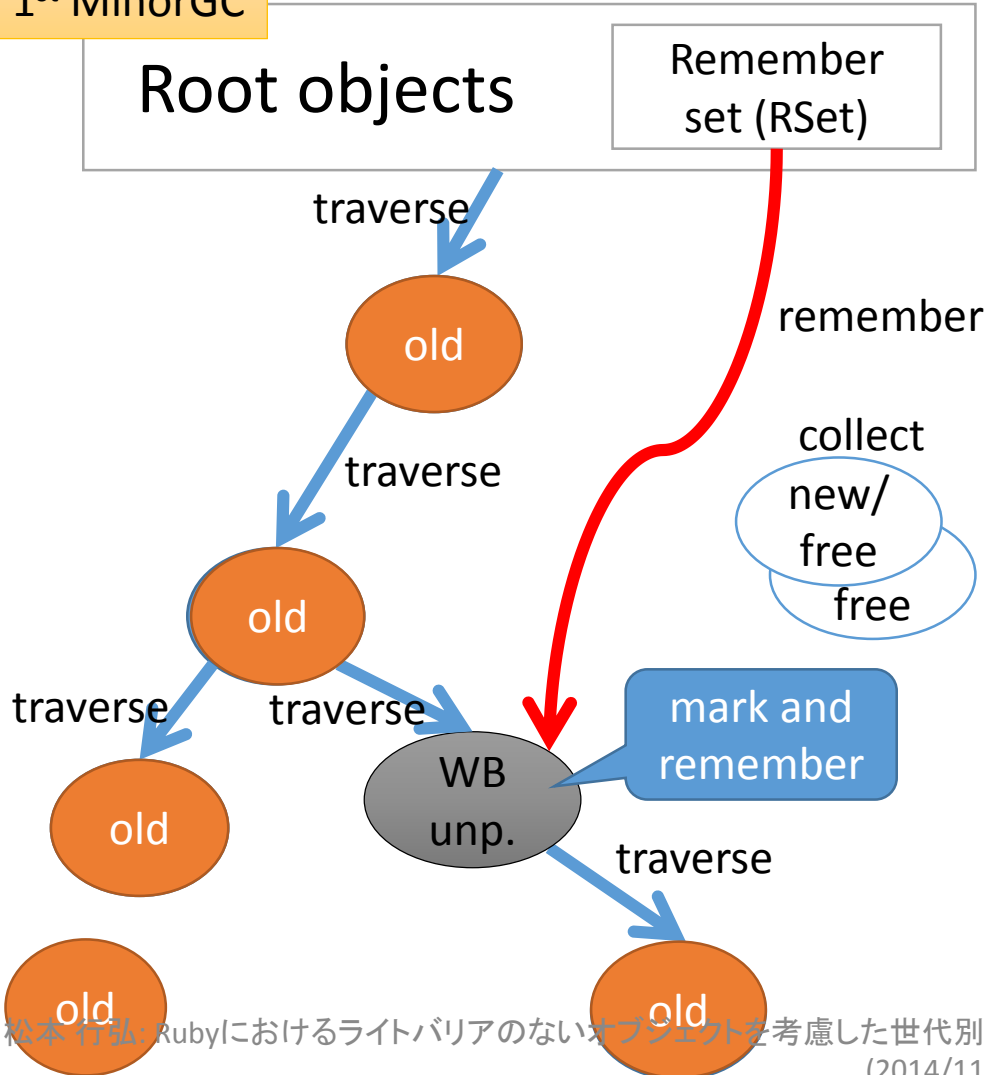
Rule

- 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 unpr. objects]

1st MinorGC



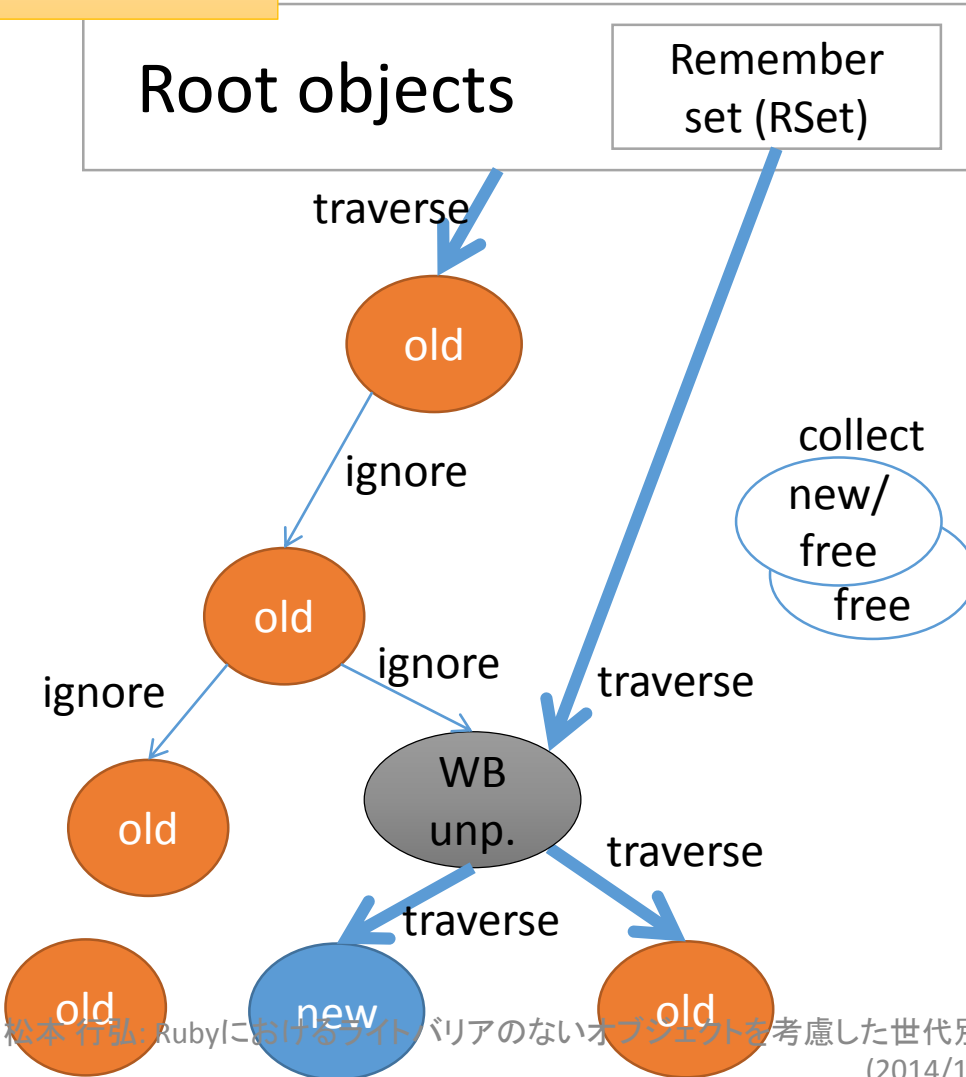
- Mark reachable objects from root objects
 - Mark WB unprotected objects, and ***don't promote*** them to old gen objects
 - If WB unprotected objects **pointed from old objects**, then **remember this WB unprotected objects** by RSet.

→ Mark WB unprotected objects every minor GC!!

RGenGC

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

2nd MinorGC

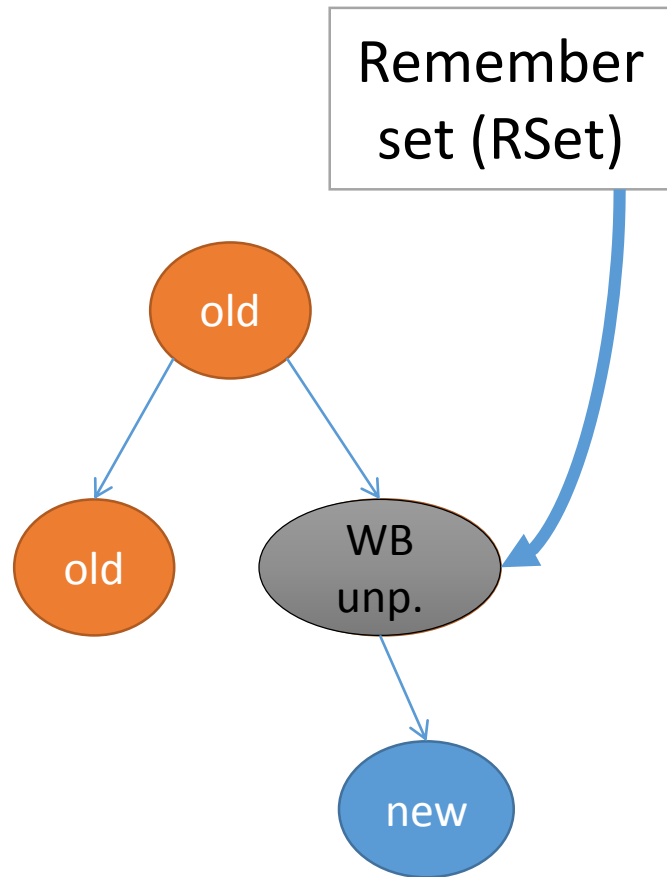


- Mark reachable objects from root objects
 - Mark WB unprotected objects, and ***don't promote*** them to old gen objects
 - If WB unprotected objects **pointed from old objects**, then **remember this WB unprotected objects** by RSet.

→ Mark WB unprotected objects every minor GC!!

RGenGC

[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

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

Issue of RGenGC: Long pause time

- Good: RGenGC achieves **high throughput**
 - Good: Minor GC stops only **short pause time**
 - Bad: Major GC still stops **long pause time**
- **Introducing Incremental GC for major GC**

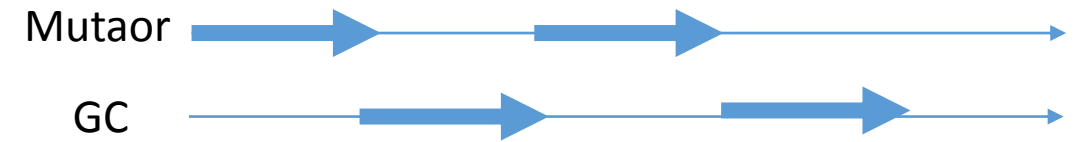
	Generational GC	Incremental GC	Gen+Inc GC
Throughput	High	Low (a bit slow)	High
Pause time	Long	Short	Small

RincGC: Restricted Incremental GC algorithms

RincGC algorithm is begin implemented for Ruby 2.2.

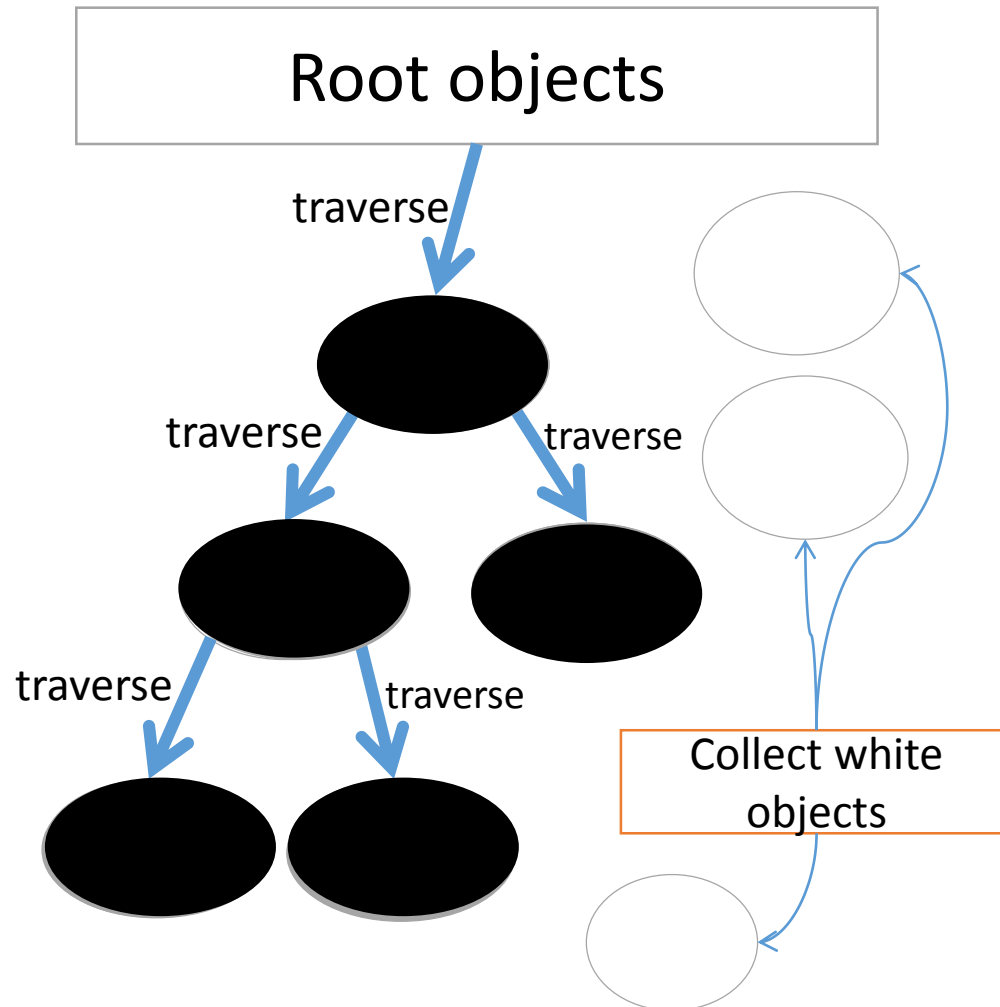
Incremental GC

- GC steps incrementally
 - Interleaving with mutators and GC process.



- Using 3 colors
 - White objects is not traversed objects
 - Grey objects can point white objects
 - Black objects only point grey or black objects

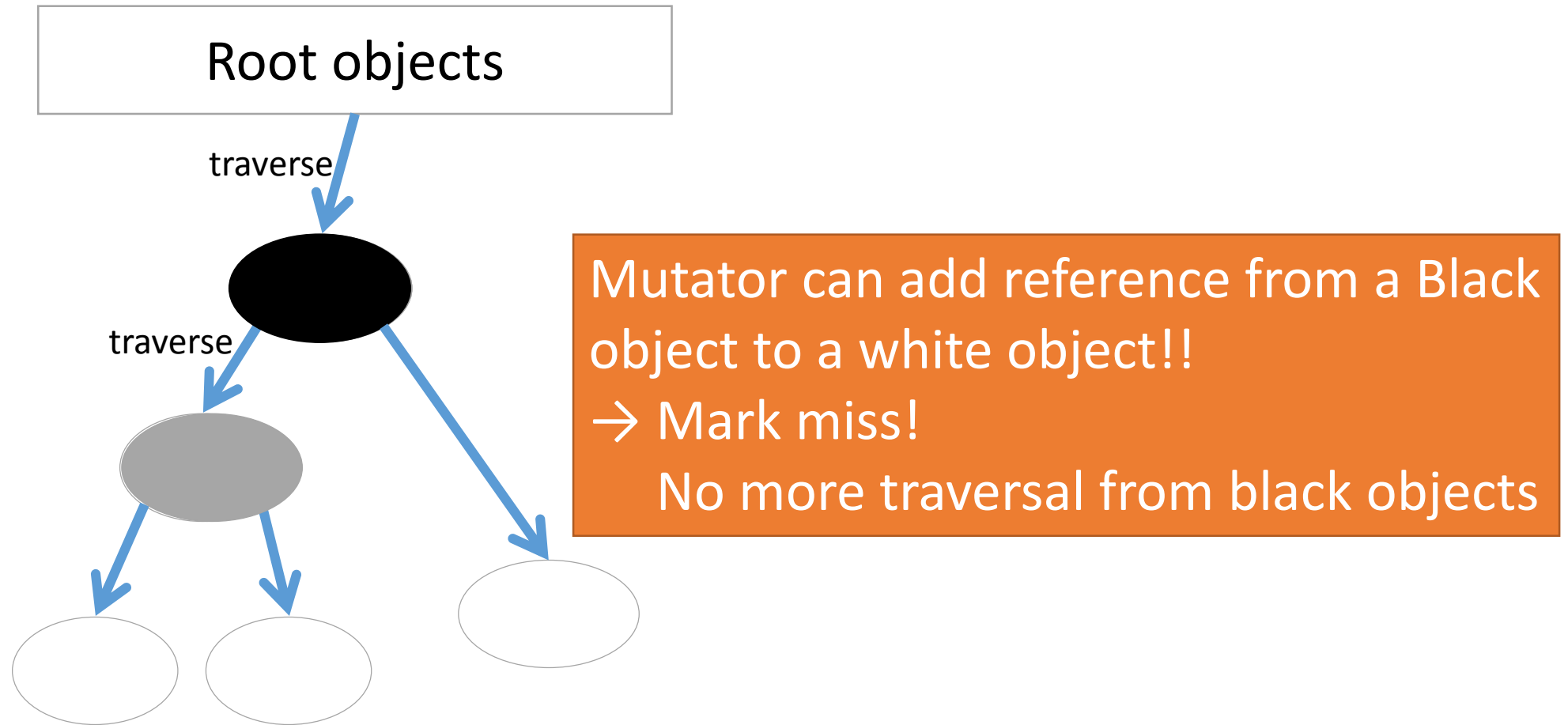
Incremental GC



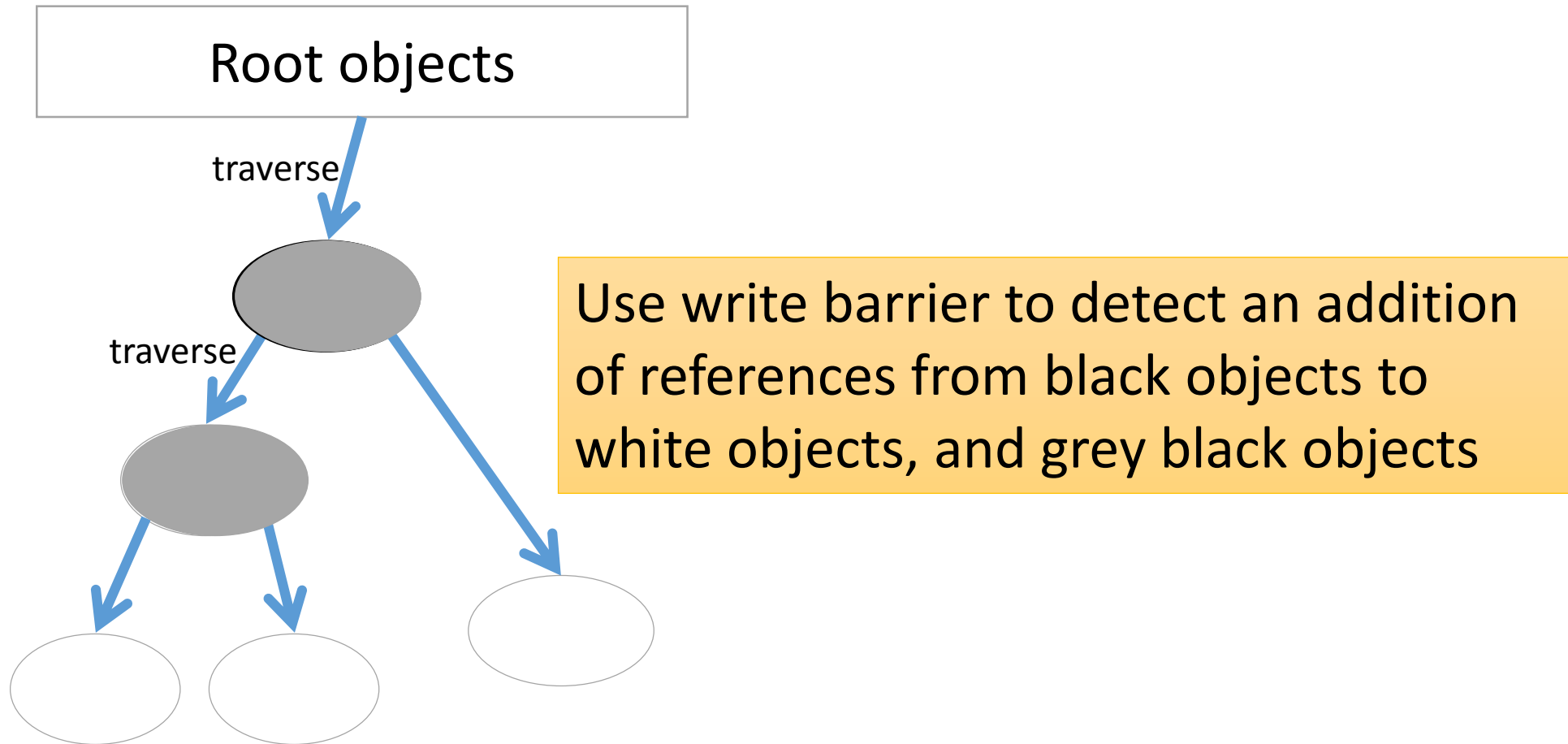
1. Color all objects “white”
2. Grey root objects
3. Choose a grey object and grey reachable white objects, and black the chosen object
4. Finish marking when no grey objects
5. Sweep white objects as **unmarked** objects

**Do step 3
incrementally**

Incremental GC requires WBs



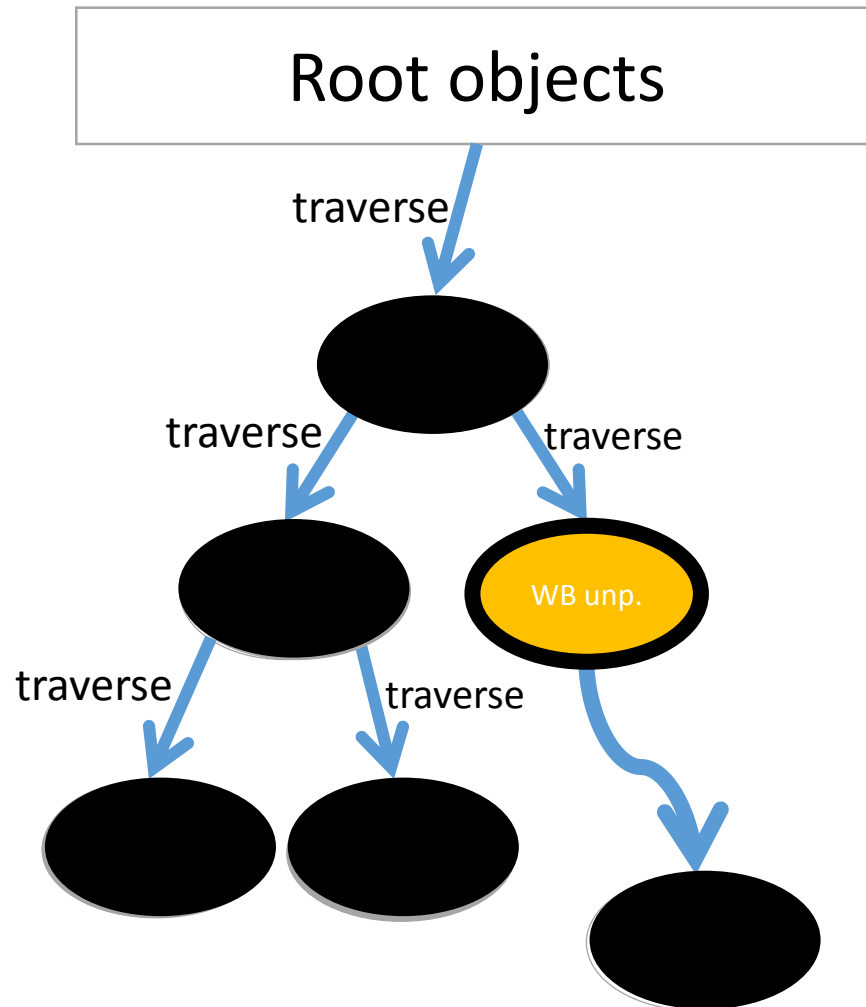
Incremental GC requires WBs



RincGC: Restricted Incremental GC using WB-unprotected objects

- Use WB unprotected objects like RGenGC
- Introducing a new rule: “Scan all black WB unprotected objects at the end of incremental GC at once”
 - WB unprotected objects can point white objects, so black such white objects without interleaving any mutators

RincGC



1. Color all objects “white”
2. Grey root objects
3. Choose a grey object and grey reachable white objects, and black the chosen object
4. Finish marking when no grey objects
5. **Scan all black WB unprotected objects at once**
6. Sweep white objects as *unmarked* objects

RincGC: Discussion

- Long pause time
 - This technique can introduce long pause time, relative to the number of WB unprotected objects at last. This is why this algorithm is named “Restricted”
 - Shorter pause time than “Minor GC” of RGenGC.

Implementation

Ruby's implementation WB protected/unprotected

- Make popular class instances WB protected
 - String, Array, Hash, and so on
- Implement “unprotect operation” for Array and so on
- Remain WB unprotected objects
 - Difficult to insert WBs: a part of Module, Proc (local variables) and so on.
 - Minor features

Ruby's implementation

Data structure

- Introduce 2 bits age information to represent young and old objects
 - Age 0 to 2 is young object
 - Age 3 is old object
 - Surviving 1 GC increase one age
- Add 3 bits information for each objects (we already have mark bit)
 - WB unprotected bit
 - Long lived bit (old objects or remembered WB unprotected objects)
 - Remembered old object bit / Marking (Grey) bit
 - They can share 1 bit field because the former is used only at minor GC and the latter is used only at major GC (incremental GC)

Ruby's implementation Bitmap technique

- Each bits are managed by bitmap technique
 - Easy to manage remember set
 - Fast traversing
 - Easy to get a set
 - Remember set: (Remembered old object bitmap) | (Long lived bitmap & WB unpr. Bitmap)
 - Living unprotected objects: (mark bitmap & WB unprotected bitmap)

Evaluation

- RGenGC
 - Micro-benchmark
 - Application benchmark
- RincGC
 - Micro-benchmark
 - Application benchmark
- Environment
 - Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz (Thanks Prof. Sugaya, Shibaura-IT)
 - Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-37-generic x86_64)
 - ruby 2.2.0dev (2014-11-09 trunk 48334) [x86_64-linux]

RGenGC: Micro-benchmark

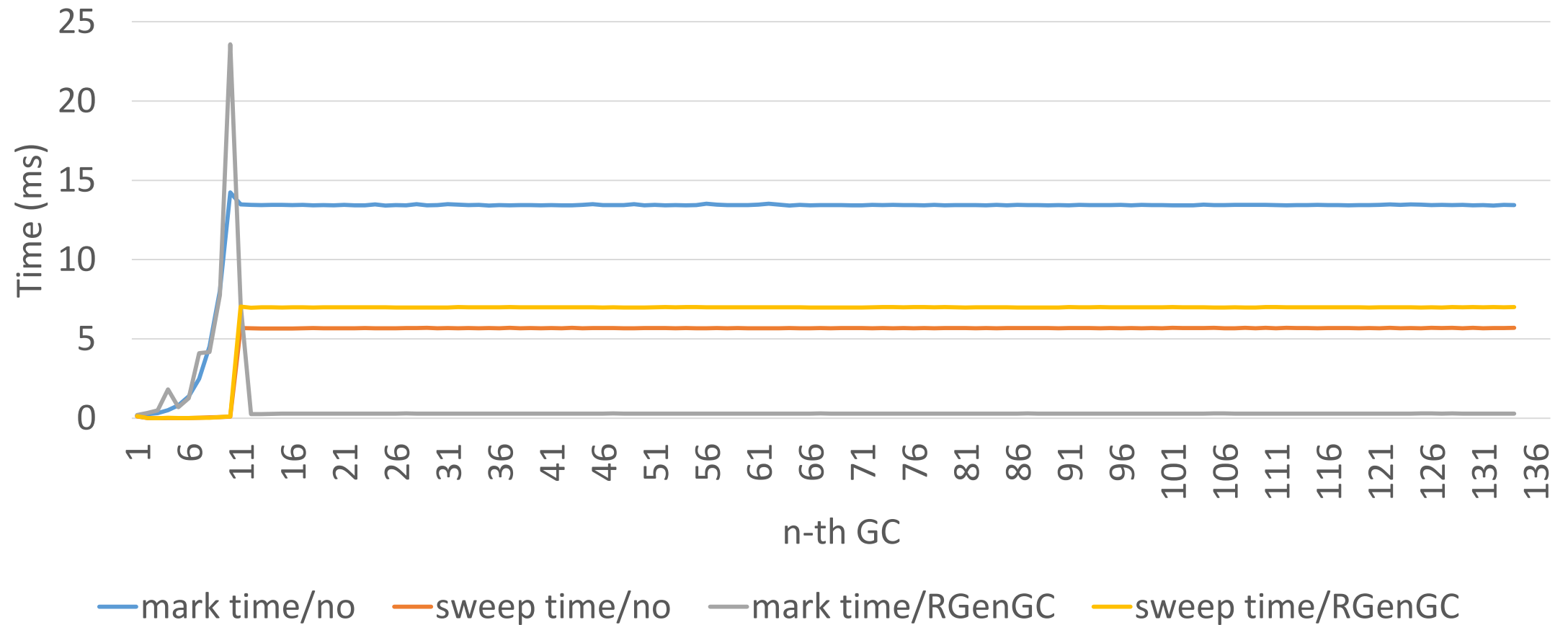
```
# Make 1M objects (long lived)
```

```
ary = (1..1_000_000).map{|e| Object.new}
```

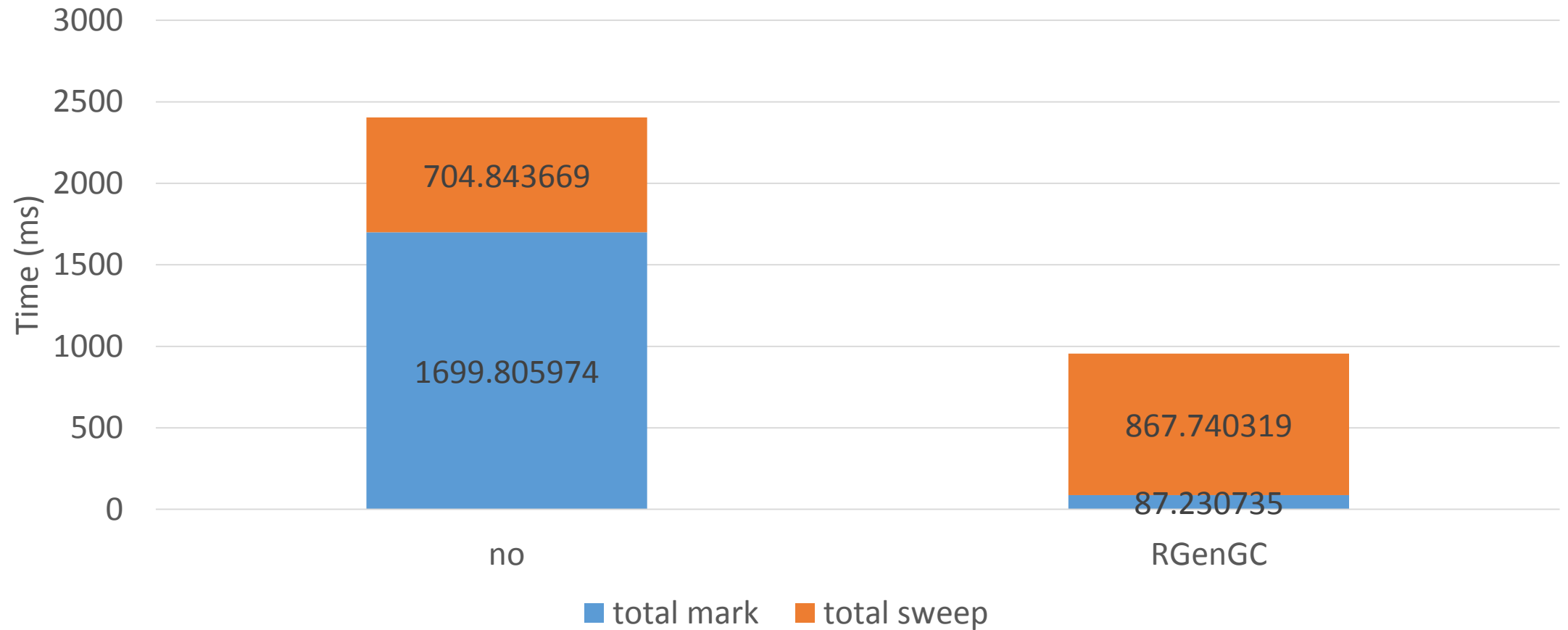
```
# Make short-lived 100M objects
```

```
100_000_000.times{Object.new}
```

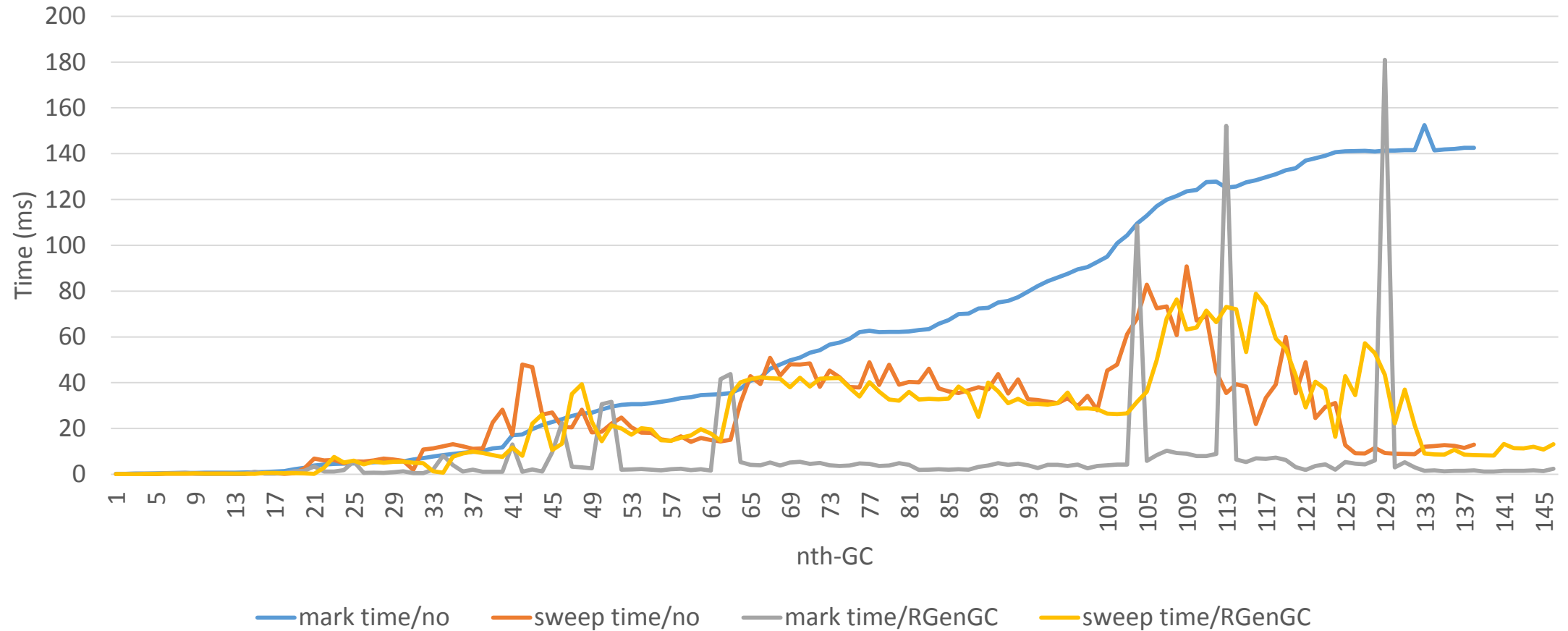
RGenGC: Micro-benchmark



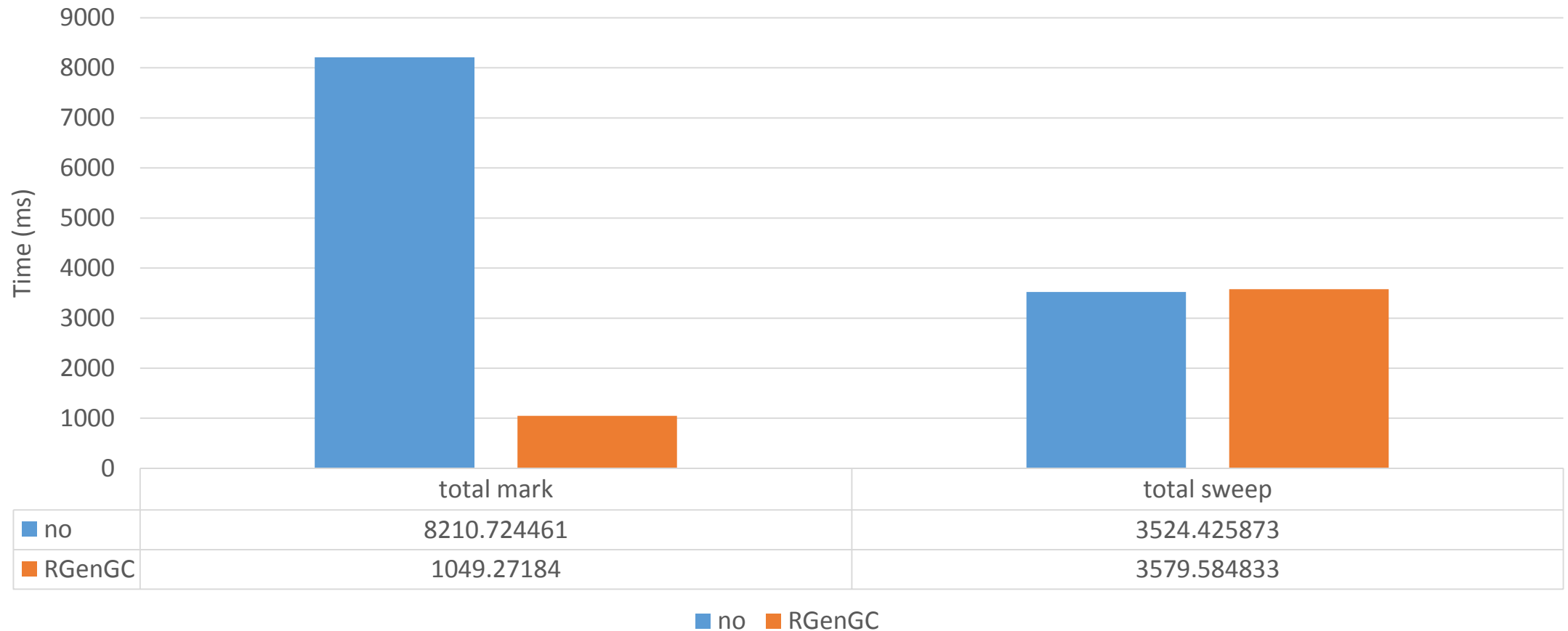
RGenGC: Micro-benchmark



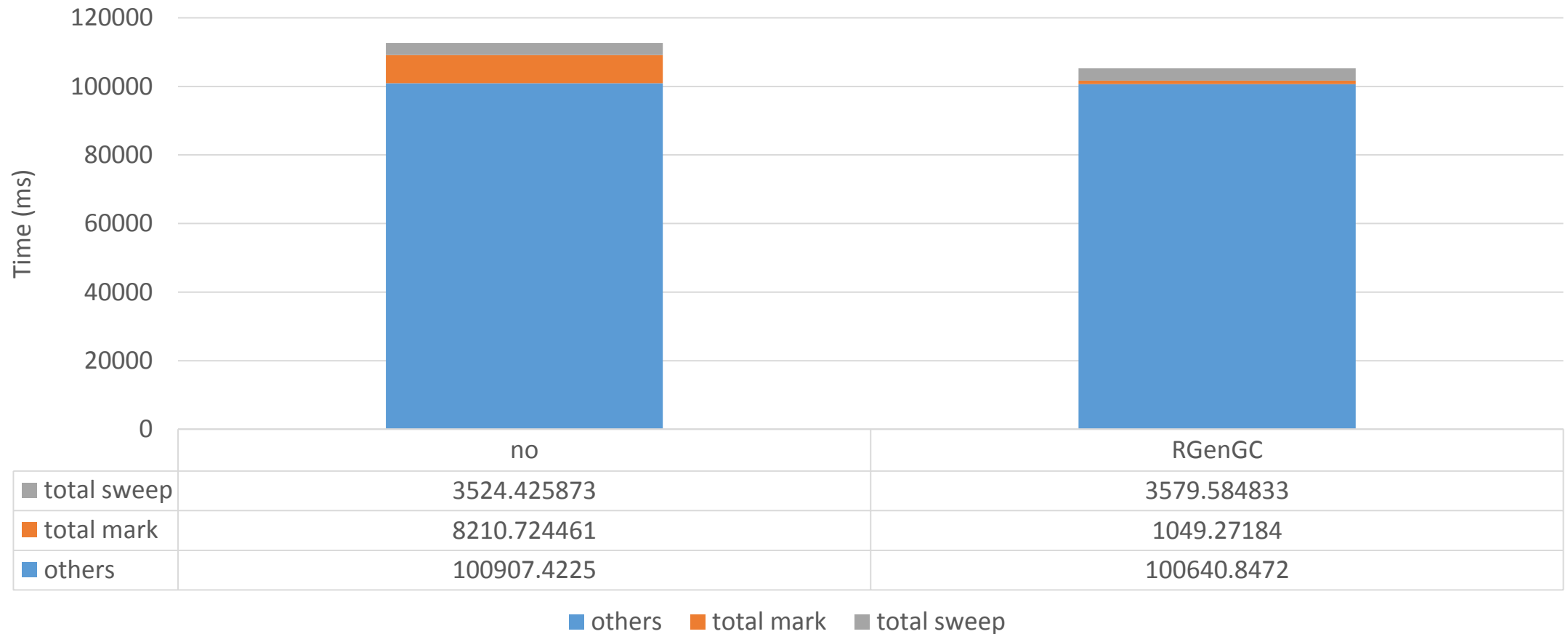
RGenGC: Rdoc application



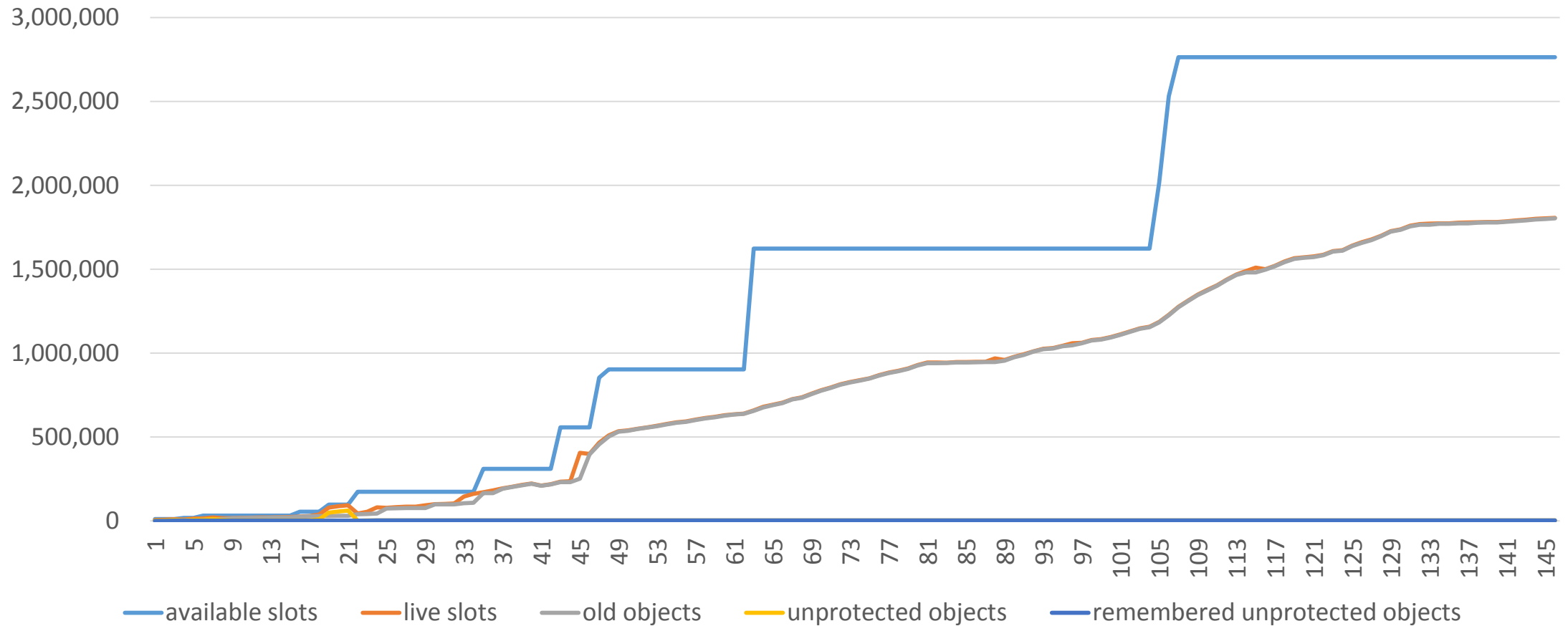
RGenGC: Rdoc application



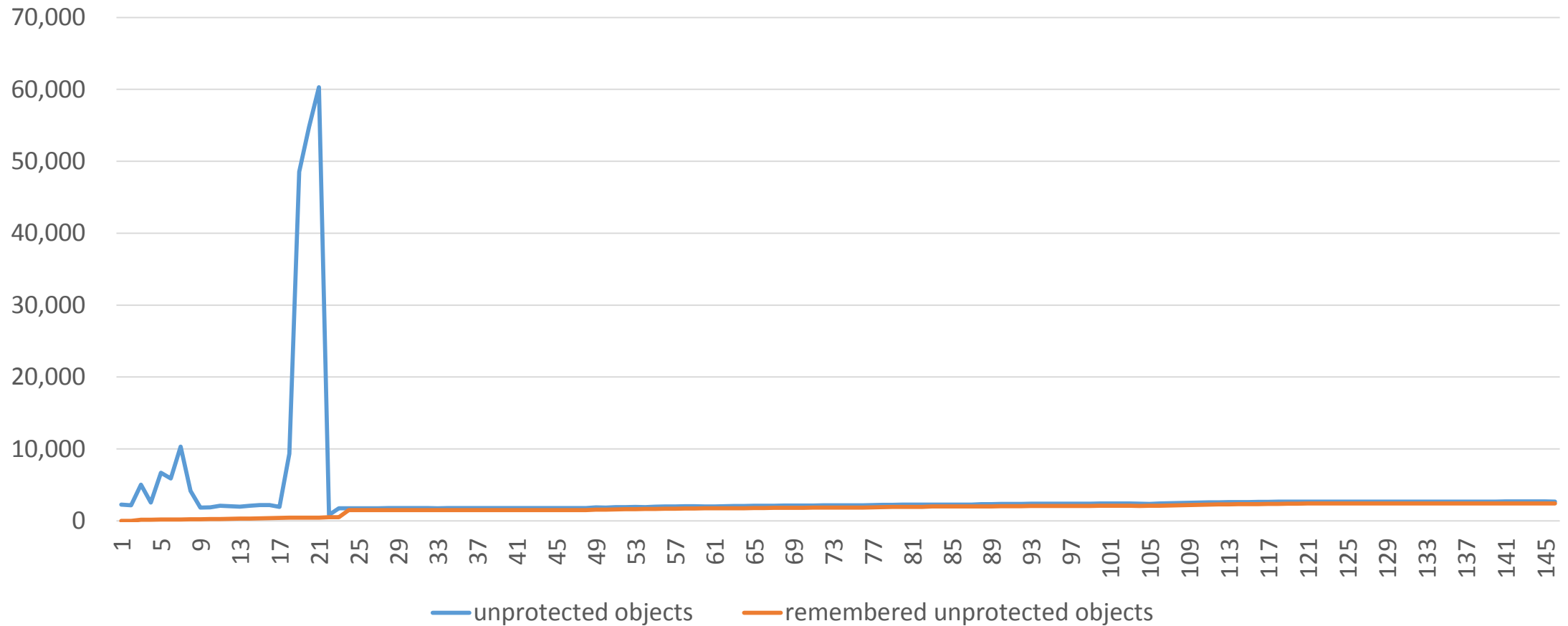
RGenGC: Rdoc application



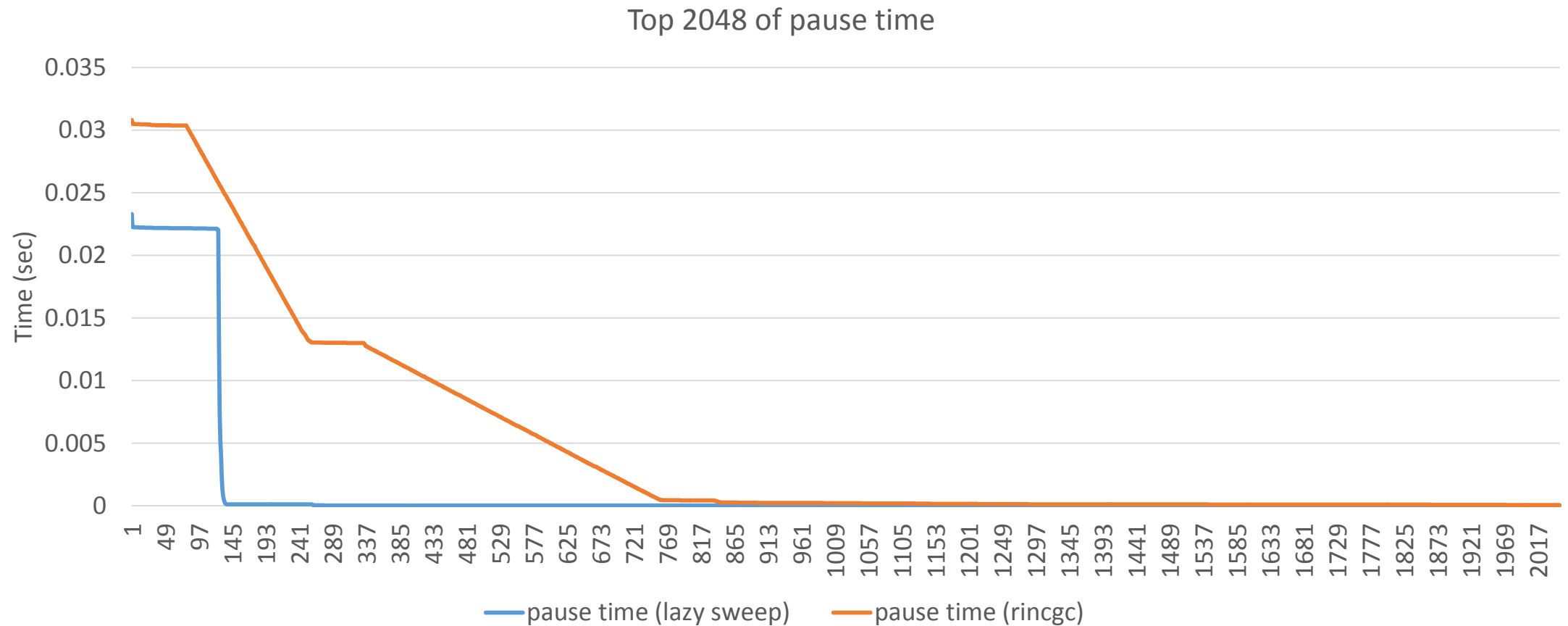
RGenGC: Rdoc application



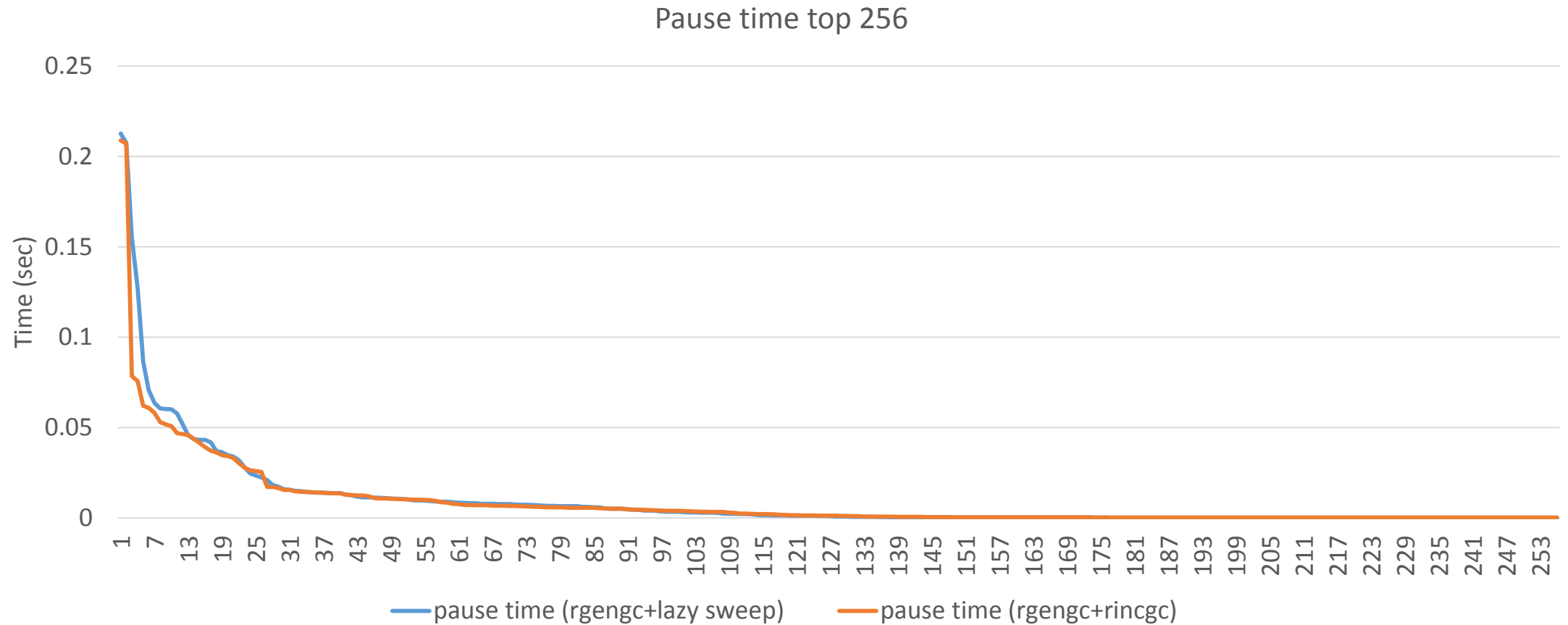
RGenGC: Rdoc application



RincGC: Micro-benchmark



RincGC: Rdoc application



Related work

- Unexpectedly, only a few work. Maybe most of interpreters are initially developed using write barriers.
- Memory protection is widely used for card marking instead of fine-grain write barriers.
- Hanai, et.al. proposed auto WB insertion technique by analyzing interpreter source code.
- Aikawa, et.al. proposed WB location detection using memory protection technique.

Future work

- Improve performance of sweep-phase
- Tuning RincGC to reduce pause time

Summary

- 背景
 - Ruby処理系にはライトバリア(WB)が入っていないし、実装大変
 - 世代別GC、インクリメンタルGCはWBが無いと実装できない
- 本研究の貢献
 - 完璧にWBが入って無くても出来るアルゴリズムを考えました
- 実世界への貢献
 - この世代別GCが入ったRuby 2.1は2013年12月にリリース済み
 - このインクリメンタルGCを追加したRuby 2.2を2014年12月にリリース予定