Gradual Write-Barrier Insertion into a Ruby Interpreter

Koichi Sasada
Cookpad Inc.
Japan
ko1@cookpad.com

Abstract

Ruby is a popular object-oriented programming language, and the performance of the Ruby garbage collector (GC) directly affects the execution time of Ruby programs. Ruby 2.0 and earlier versions employed an inefficient non-generational conservative mark-and-sweep GC. To improve this and make it a generational collector, it is necessary to introduce write barriers (WBs), but this requires huge modification to existing source code, including third-party C-extensions. To avoid the need for adding WBs around legacy code, we invented a new concept called "WB-unprotected objects", which indicates to the GC to treat such objects more conservatively. By leveraging this design, we were able to improve the performance of Ruby 2.1 with a generational GC and of Ruby 2.2 with an incremental GC while preserving compatibility with existing C-extensions. Another significant advantage of this approach is that WBs can be added gradually, which reduces the difficulties associated with updating existing code.

CCS Concepts • **Software and its engineering → Garbage collection**.

Keywords Generational garbage collection, Write-barrier, Ruby

ACM Reference Format:
Koichi Sasada. 2019. Gradual Write-Barrier Insertion into a Ruby Interpreter. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3315573.3329986

1 Introduction

Generational garbage collectors require write barriers (WBs) or similar techniques to recognize reference creations from older generation objects (old objects) to younger generation objects (young objects) [2, 6]. If we forget to insert even one WB, the young objects may be wrongly collected. Of course, this would be a critical GC bug.

If you need to implement a generational GC on an interpreter which does not have WBs yet, how can you implement it? One straightforward approach is to completely introduce all WBs at once. However, if you cannot modify parts of the source code writing to objects (e.g., in C-extensions), it is not possible to add all required WBs.

This *was* the case for the Ruby interpreter in 2013 and before.

The Ruby object-oriented programming language [4] is used worldwide, especially for web application development with the Ruby on Rails framework [11]. Ruby runs programs by creating and mutating many objects, and thus the performance of garbage collection affects the performance of the interpreter. There are huge-scale web applications written in Ruby and therefore performance is important.

The latest Ruby version is Ruby 2.6, which was released in Dec. 2018. Ruby 2.6 has a generational and incremental GC and employs such well-known techniques to improve the throughput and reduce the pause time of the GC.

However, Ruby 2.0 and earlier versions of Ruby did not use generational and incremental techniques because of the lack of WBs. Previous Ruby interpreters employed a simple conservative mark-and-sweep GC, and its performance was poor. Thus, a generational GC was desired for a long time.

To introduce a generational GC, all reference write operations on objects must be detected by WBs. However, this is difficult for Ruby because of compatibility problems. It should be possible to introduce all necessary WBs into the interpreter core (using virtual machines, built-in methods, and so on written in the C language) with huge development efforts. Nonetheless, we cannot easily modify all *third-party* C-extension libraries which extend the Ruby interpreter and are also written in the C language (See 2.2 for details). C-extensions have been widely adopted and many Ruby libraries and applications rely on them. We needed to preserve compatibility with existing C-extensions to not upset the Ruby community. If we changed the C-extension API to use WBs correctly to improve garbage collection, many Ruby developers would not be able to upgrade to newer Ruby versions because their applications would not run on them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISMM '19, June 23, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6722-6/19/06...$15.00
https://doi.org/10.1145/3315573.3329986

In summary, the issue was that we were not able to introduce WBs without breaking compatibility with existing C-extensions.

To overcome this issue, we invented a new concept called "write-barrier unprotected objects". All objects are categorized as either write-barrier protected or unprotected. If we write a reference (a pointer) to a WB unprotected object, the write-operation is not detected by the GC. We extend GC algorithms which require WBs using the "WB unprotected object" concept by treating them carefully. We can mark uncertain objects as WB unprotected objects so that we can keep compatibility with existing C-extensions without modifying them.

The "WB unprotected objects" concept further helps to develop and improve an interpreter gradually. We can postpone inserting WBs for certain complex data structures, such as closure objects. This advantage is important because we only have a limited number of Ruby interpreter developers.

With this concept, we introduced generational garbage collection in Ruby 2.1 (2013) and incremental generational garbage collection in Ruby 2.2 (2014), and the overall performance was improved.

The contribution of this research is proposing the new "WB unprotected objects" concept to implement GC algorithms which require WBs for an interpreter in which it is not possible to insert WBs for all reference write operations. Improving the performance of the Ruby interpreter is a practical contribution because many people currently use the Ruby language. This technique is also applicable to other languages beyond Ruby which need to preserve compatibility with existing third-party code without modifications.

Our original idea for this work was conceived in 2013 and we made several presentations about it. However, there were no academic reports about it, so we therefore summarize it in this paper now with our experience of Ruby's GC development.

2 Introduction on the Ruby Interpreter

This section describes the Ruby 2.0 internals[10] related to garbage collection.

2.1 Object Representation

Basically, each object has a fixed-size memory block (size: $sizeof(void*) \times 5$, 40 bytes on a 64 bit CPU). This memory block consists of two parts, namely a header and a body. The header contains the class of the object, its data type and other information about the object. The usage of the body depends on the data type.

The Ruby interpreter handles an object with a pointer to this memory block. We call this pointer to the memory block as `VALUE`.

If an object requires more memory than the body can hold, we use `malloc()` to allocate external memory blocks. For example, if we allocate a array object with a length of 10, the Ruby interpreter allocates a sequential memory block (size: $10 \times sizeof(VALUE)$) and the body of the array object points to the allocated memory block.

2.2 GC Algorithm

Ruby traditionally used a simple conservative mark-and-sweep GC algorithm. The GC marks all objects traceable from the root set and sweeps unmarked objects. Pointer-like numbers in machine stacks, CPU registers and similar locations are conservatively assumed to be pointers, and pointed objects are marked.

This simple GC algorithm made writing C-extensions easy because writing additional code to communicate with the GC was not necessary in most cases.

The sweeping phase was incremental (lazy sweeping). However, the marking phase stopped all of the Ruby execution.

2.3 Compatibility Issue

Ruby supports C-extensions, which can be built from C language source code, to enhance the Ruby interpreter. C-extensions are dynamic-link libraries, and the Ruby interpreter loads them dynamically. The Ruby interpreter provides the Ruby C-API, which is used by C-extensions.

Using C-API, we can read and write to Ruby managed memory areas directly. For example, `RARRAY_PTR(ary)` API (macro) returns an array's memory block, and C-extensions can write a reference (`VALUE`) to this memory block directly as follows:

```
RARRAY_PTR(ary)[i] = obj;
```

We cannot detect this kind of memory access (writing) and it is thus difficult to insert WBs correctly, especially in third-party C-extensions that we can not modify them.

There is an alternative API `rb_ary_store(ary, i, obj)` for storing `obj` in the i_{th} index and it is easy to support WB because we only need to modify `rb_ary_store()`. However, we cannot force all C-extensions to use this kind of WB-friendly API.

If we forget to insert even a single write-barrier, it will cause critical issue (freeing of objects that are still in use). This is why we were not able to introduce a generational GC or other GC techniques which require WBs.

3 Proposal: WB-Unprotected Objects

To introduce a generational GC to Ruby 2.1, we invented a new concept called "write-barrier unprotected objects". In addition, an incremental GC was introduced using the same technique.

This section describes our ideas and shows how to implement these GCs.

3.1 WB-unprotected Objects

Facing the problem that we were unable to insert WBs on all write-locations, we gave up on complete WB insertion, and instead explicitly indicated which objects are guaranteed by WBs.

All objects have an attribute called "WB-protected". If an object is WB-unprotected, this attribute is false, which means that the Ruby interpreter does not support WBs for this object. Newly created references from this object to other objects cannot be detected by a GC. References written to WB-protected objects are detected completely.

The Ruby 2.0 interpreter does not have any WBs, and thus all objects are WB-unprotected. If we insert WBs for a data structure representing a class `K`, we can then set the WB-protected attribute for all instances of `K`. For example, `String`-class objects only have a few references (they usually only refer to an `Encoding` object), so it is easy to implement `String` objects as WB protected objects.

This means that we can increase the number of WB-protected objects gradually. We can prioritise WB-insertion development such that the more frequently used classes are assigned a higher priority than the others. `Array` and `Hash` objects are used frequently and have a strong impact on the performance of generational GC, so we supported them with top priority. For the classes that have a complex data structure and that present difficulties for introducing WBs, we can postpone making them WB-protected. As for classes with only a few instances or classes where most objects die young, we can also postpone making them WB-protected because the performance impact is minimal. In general, inserting WBs correctly is a difficult and time-consuming task because WB-related bugs cause critical issues and it is difficult to debug them. Thus, allowing for gradual development is beneficial.

3.2 WB Unprotect Operation

We can make `Array` objects WB protected because their implementation code is maintained by us. However, as described in the last section, C-extensions can obtain pointers to memory objects and can write references directly. This means that there is a possibility that some `Array` objects do not support WBs.

For this case, obtaining a pointer from an array object makes that array WB-unprotected. We call dropping the WB-protect attribute the "WB unprotect operation".

There are several classes other than `Array` that use the WB unprotect operation such as `Hash`. Fortunately, C-extensions acquire pointers from a `VALUE` using special macros (such as `RARRAY_PTR()`), so we can insert WB unprotect operations in these macros.

We can also use the WB unprotect operation in other cases. For example, it is difficult to determine where WBs are needed during a complex pointer manipulation involving multiple objects. After the manipulation, we can give up on keeping the WB protected attribute by performing the WB unprotect operation. Though this may slow down the interpreter, its health is maintained.

Objects can be changed from WB-protected objects into WB-unprotected objects. However, WB-unprotected objects can not become WB-protected objects.

3.3 Generational Garbage Collection with WB-Unprotected Objects

Our original GC was a conservative mark-and-sweep (incremental sweep) algorithm. We implemented generational marking with WB-unprotected objects to enhance the original GC. We named it *RGenGC*. The *R* prefix stands for *Restricted* (because of WB-unprotected objects) or *Ruby*.

We use two generations: young objects and old objects. There are two marking phases: minor and major marking. Minor marking marks young objects and sweeps unmarked young objects. Major marking marks all objects. A WB adds an old object into the remembered set if reference creations from the old object to young objects are detected.

Minor marking steps without WB-unprotected objects are as follows:

1. Put the root-set objects and objects referenced from the remembered set objects into the work queue.
2. Repeat the following until the work queue is empty:
   a. Dequeue an object p from the work queue.
   b. For each object c referenced from p do:
      i. If p is an old object:
         - If c is already marked, makes c an old object and add c to the remembered set.
         - If c is not marked and not an old object, makes c's age two (becomes an old object at the next step).
      ii. Increment the age of c by one, mark c, and then put c to work queue if c was not marked and is not an old object. Note that, in our implementation, if the age of an object becomes 3, the object becomes an old object.

After minor marking, unmarked young objects are swept. Sweeping phase is not generational (all objects are scanned).

We introduced the following new rule to support WB-unprotected objects:

Rule1 Prohibit the promotion of WB-unprotected objects. Their age is always 0, i.e. young objects.

Rule2 If the interpreter detects a reference from an old object to a WB-unprotected object A during the marking phase, add A to the remembered set.

Rule3 If an old object B becomes a WB-unprotected object via the WB unprotect operation, the interpreter should make B young (demote) and put it into the remembered set, as well.

Because we cannot manage references from WB-unprotected objects, we introduced the aforementioned additional rules. WB-unprotected objects added to the remembered set are marked by default during minor marking.

3.4 Incremental Garbage Collection with WB-Unprotected Objects

Based on RGenGC, we implemented an incremental mark-and-sweep algorithm with WB-unprotected objects. We named it *RIncGC*. We only introduced incremental marking for major garbage collection because minor garbage collection is already fast enough.

Combining incremental and generational garbage collection is not difficult, so we only show the incremental marking algorithm.

We will use the three colours (white, grey, black) to explain RIncGC. The incremental marking phase without WB-unprotected objects consists in the following steps:

1. Make all objects white.
2. Make all objects in the root set grey.
3. Repeat the following until there are no grey objects. These steps are interleaved with the Ruby program execution.
   a. Pickup an object (A) from the pool of grey objects.
   b. Make all unmarked white objects which are referred from A grey.
   c. Make A black.

If WBs detect a reference creation from a black object to a white object, then the white object is made grey.

After the marking phase, the white objects are swept. Our sweeping phase was already incremental (lazy sweeping).

We introduced a new rule to support WB-unprotected objects:

Rule4 At the end of the marking phase, the GC re-scan black and WB unprotected objects at once (not incremental).

WB-unprotected objects can refer to white living objects, and thus Rule4 is needed. The pause time caused by this step is proportional to the number of living WB-unprotected objects.

Listing WB-unprotected objects is a problem. We introduce a bitmap to represent the WB unprotected attribute for each object. Each object has a corresponding bit on the bitmap. Using this bitmap, it is easy to find all WB-unprotected objects.

For example, 1 M objects consume 1 M bits in the bitmap ($1M/8bit = 128KB$). A set of 1 M objects consumes at least $40B \times 1M = 40MB$, compared to which $128KB$ is very small.

4 Evaluation

In this section, we show the performance improvement measurements and our experience on how our proposal aids in the development.

```
def make_linked_list n
  list = []
  n.times{
    list = [list]
    if rand(100) < $prob
      list.wb_unprotect
    end
  }
  list
end

# Create a long linked list
huge_list = make_linked_list(10_000_000)

# Create 100 M empty arrays to invoke minor GC
100_000_000.times { [] }
```

Figure 1. Micro-benchmark program

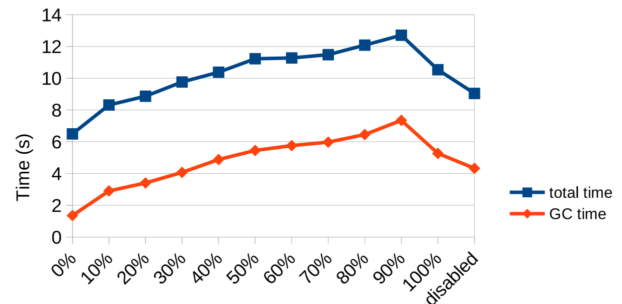

Figure 2. Micro-benchmark results

4.1 Performance

Several benchmark results are presented in this subsection. We used a Linux machine (CPU: Intel(R) Core(TM) i7-6700 CPU, 64GB of memory, Ubuntu 18.04.2, gcc 7.3.0) for this evaluation. We used the latest development Ruby version, namely "ruby 2.7.0dev (2019-03-08 trunk 67194) [x86_64-linux]". This version of Ruby contains RGenGC and RIncGC. To evaluate them, we prepare a "disabled" version by modifying this version of Ruby to force it to use full (major) marking and immediate (non-incremental) marking. We provided macros `USE_RGENGC` and `USE_RINCGC` to enable or disable these features including WBs. However, we found that disabling these features using these macros introduces a GC performance bug on this version (we usually perform tests on enabled versions). Therefore, in this evaluation, "disabled" means full and immediate marking and the WB overhead is included for the "disabled" version.

Table 1. RDoc benchmark results

	Total time (sec)	GC time (sec)
Disabled	30.46	10.20
Enabled	22.57	1.63

4.1.1 Micro-Benchmark

Figure 1 shows our micro-benchmark program, which makes an linked list with 10M array objects. To measure the effect of the WB-unprotected objects, we made some arrays WB unprotected via the `wb_unprotect` method (the `wb_unprotect` method performs the WB unprotect operation, and was prepared only for this evaluation). `$prob` specifies the percentage of objects that are WB-unprotected. With the linked list, we made 100 M empty arrays to invoke multiple minor marking operations.

Figure 2 shows the results of this program from 0% to 100%. The rightmost data-point shows the result on a disabled GC (no RGenGC, RIncGC, with 0% `$prob`).

We can see that increasing the percentage of WB-unprotected objects increases the overall garbage collection time. It is interesting to note that the "100%" case did not yield the worst result. There are no old objects (note that WB-unprotected objects can not become old objects), and thus putting arrays in the remembered set was not required.

The "disabled" result was slower than for 0% WB-unprotected objects. We can confirm that RGenGC resulted in an improvement. The "disabled" result was close to the results obtained for 30% to 40% WB-unprotected objects. This means that WB-unprotected objects introduce additional overhead compared to the interpreter without RGenGC. To achieve a better performance, the number of WB unprotected objects should be small.

4.1.2 RDoc Application

RDoc is Ruby's documentation system, which reads Ruby and C source code and generates documentation from comments embedded in the source code. We use RDoc workload as non-trivial application benchmark. The target source code in this experiment is the C and Ruby source code in the Ruby interpreter source repository.

Table 1 shows the evaluation results. With the RDoc benchmark, we can confirm an overall 35% performance improvement when using RGenGC. GC time was 6.25 times smaller.

In the RDoc benchmark, the Ruby interpreter process creates 32 M objects, of which 0.6 M are WB-unprotected objects. There were 657 WB-unprotect operations. In this case, there were only a few (approximately 2%) WB-unprotected objects.

Figure 3 shows the number of allocated objects, indexed by GC events (the x-axis shows the number of GCs that happened so far, sampled every 10 GCs). We can see that young objects die early by comparing the numbers of *living* and *old* objects, confirming the generational hypothesis in this case. Because the interpreter needs to mark only young living objects on each minor marking, the reduction in GC time is considerable compared to a full GC. Figure 3 also shows that the number of WB unprotected objects is really small.

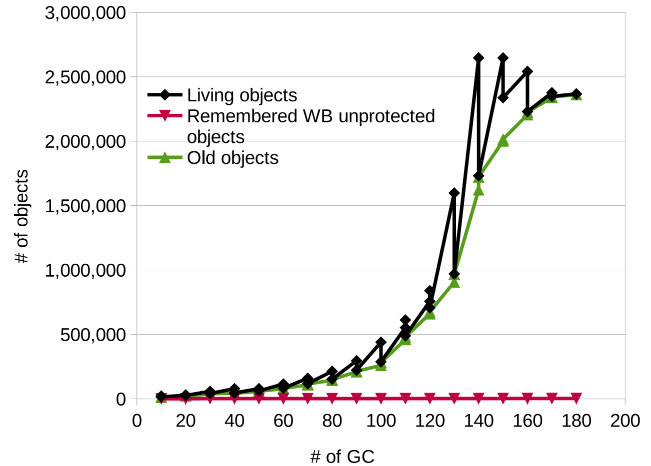

Figure 3. RDoc benchmark result (# of objects alive on each GC event)

Table 2. Maximum response times (ms) by percentile for the Discourse Web application benchmark

	50%	75%	90%	99%
categories	36	44	148	159
	35	36	52	87
	(×1.03)	(×1.22)	(×2.85)	(×1.83)
home	39	148	161	164
	39	42	56	96
	(×1.00)	(×3.52)	(×2.88)	(×1.71)
categories	64	179	186	194
_admin	63	70	82	147
	(×1.02)	(×2.56)	(×2.27)	(×1.32)
home_admin	71	180	186	194
	67	80	86	157
	(×1.06)	(×2.25)	(×2.16)	(×1.24)

Cell contents:
top: disabled version, middle: enabled version,
bottom: improvement ratio (enabled/disabled).

4.1.3 Web Application Benchmark

We evaluate the proposed approach on a practical web application using the Ruby on Rails web application framework: the Discourse benchmark[1]. The Discourse benchmark makes 500 requests to several pages and shows the results for different percentiles.

[1]https://github.com/discourse/discourse/. Our downloaded git commit hash is 41f09e. The benchmark can be run by running script/bench.rb.

Table 2 shows the results on RGenGC/RIncGC disabled and enabled Ruby interpreters. Percentile values in the table indicate that n% of all requests finished in less than x milliseconds. In each cell, the first and the second lines show the results for the disabled version and the enabled version, respectively, in milliseconds. The third line shows the ratio (disabled/enabled).

Half of the requests (the 50% column) were almost same for both the disabled and enabled versions. However, we can observe differences for 75% and higher percentiles. We believe that we can confirm the performance impact of RGenGC. In the results in the 99% column, we can see an improvement on the worst time because of RIncGC. Nevertheless, the worst time was clearly longer than the times shown in the 50% column, so we need to improve RIncGC in future.

In this benchmark test, a total of 128 M objects were created, of which 1.5 M were WB-unprotected objects. The interpreter invoked the WB unprotect operation 8,301 times. Compared with the total number of objects, the number of WB-unprotected objects was small enough.

4.2 Gradual Development

One advantage of the "WB-unprotected object" concept is that it allows for gradual development. We will now summarise our experience to shows how gradual development is helpful.

On Ruby 2.1 (released in 2013), we introduced WBs into 13 classes (data structures).

Container types `Array`, `Hash`, `Struct`, `Object` (User defined classes), `Class`

Scalar types `Bignum`, `Complex`, `Float`, `Rational`, `String`, `Range`, `Regexp`, `RubyVM::ISeq` (bytecode)

C-extensions can access (write) `Array` and `Hash` objects directly (C-extensions can obtain a raw pointer to their memory block). We apply the WB-unprotect operation if a raw pointer is acquired by C code.

The class `Class` represents Ruby's class objects. We were not able to eliminate a WB-related bug for some kinds of `Class` objects, so we used the WB unprotect operation if a class becomes such a kind of class to prevent this bug. This shows that we can give up on inserting WBs if it proves to be difficult.

Ruby 2.4 (released in 2016) introduced WB-protected `Proc` objects (closure objects) and internal environment objects (`Env` objects[2]). `Env` objects have complex pointer references and it was difficult to introduce efficient WBs (we had to introduce WBs for each local variable assignment made by the virtual machine). We invented a new efficient write barrier technique, and we managed to make `Env` and `Proc` objects WB-protected for Ruby 2.4. Our measurements showed a performance improvement of 57% on a special benchmark that created a huge number of `Proc` objects[3] with this change.

Ruby 2.5 (released in 2017) made five additional classes (`Dir`, `Binding`, `Thread::Queue`, `Thread::SizedQueue` and `Thread::ConditionVariable`) WB protected.

Additionally, we are maintaining compatibility with existing C-extensions and we have not received any reports on compatibility issues about it.

5 Related Work

TruffleRuby[9] is an alternative implementation of the Ruby language on the JVM, using the Graal dynamic compiler and the Truffle AST interpreter framework[12]. TruffleRuby supports MRI's C-extensions by passing special wrapper objects to native code and managing their lifetime beside normal objects, using a global table[7]. We can employ a similar approach, however using special wrapper objects introduces a measurable overhead. Also, that approach would require rewriting large parts of the interpreter code to obtain good performance. Our approach does not require large-scale rewriting.

Hanai et al. proposed an automatic WB insertion system to create a Scheme interpreter[5]. They created a special C preprocessor that detects "write" locations by analysing the C source code and suggests the introduction of WBs. We believe there are problems when using this approach: (1) It is difficult to maintain a preprocessor. (2) The proposed system cannot detect complex code patterns, such as using void pointers. (3) The preprocessor may suggests the introduction of unnecessary write barriers because it has to be conservative. This can introduce additional overhead.

Using a hardware memory protection mechanism is another approach[3] to provide write-detection without introducing WBs in the source code. However, the overhead of page faults cannot be ignored. Portability is also a problem, because we would need to use system-specific page-protection features. This issue makes maintenance difficult.

There is also an extreme approach: scanning all old spaces in the heap looking for references to young objects[1]. Of course, we cannot ignore the overhead of linear scanning, even if linear access has better locality than random access. Additionally we cannot trace all heap memory because C-extensions can allocate memory blocks which are not managed by the GC. C-extensions can write a pointer to such memory blocks[4].

[2] `Env` objects manage local variables on a method frame. A `Proc` object refers an `Env` object list.

[3] https://bugs.ruby-lang.org/issues/10212

[4] In such case, C-extensions must provide a special mark function to specify how to mark all objects referenced from a memory block which is allocated by C-extensions.

6 Conclusion

We proposed the "write barrier unprotected object" concept to introduce generational and incremental garbage collection techniques into the Ruby interpreter, which was non-write-barrier aware. With this concept, we succeeded in implementing a generational and incremental GC on the Ruby interpreter and improving GC performance. Using the "WB-unprotected object" concept, we can develop write-barrier-related code gradually on a flexible development schedule. Via measurements we demonstrated the performance impact of our GC improvements, mostly obtained via the generational GC.

We showed some poor GC performance results with a "disabled" version in Table 2. For web applications, there used to be a trick for disabling GC using the `GC.disable` method before the request, and then re-enabling GC after the request, in order to response the request quickly by avoiding GC during the request (also called "out-of-band GC"). In fact, our organization (Cookpad Inc.) used to apply this trick some years ago to improve response times on our service. Because RGenGC and RIncGC solved this issue significantly, our web applications do not use `GC.disable` anymore, and the management of our web applications became easier. Moreover, the overall CPU usages of our web applications were reduced. This is important because there is need to reduce the cost of computations. Github, one of the biggest Ruby users, also reported same results[8].

Finally, there are other GC techniques, and we should try to employ these advanced techniques with WB-unprotected objects in Ruby.

Acknowledgments

First of all, we would like to thank Yukihiro Matsumoto and other Ruby interpreter developers who helped us to implement (and debug) our new GCs. This work was started when I was a Heroku, Inc. employee. We would like to express our gratitude to Heroku, Inc. for their support. We also would like to thank Yusuke Endoh, Benoit Daloze, Martin Dürst and Samuel Williams for contributing to the editing of this paper. Finally, we would like to thank the anonymous reviewers and Christian Wimmer for reviewing and shepherding.

References

[1] Joel F. Bartlett. 1989. *Mostly-Copying Garbage Collection picks up Generations and C++*. Technical Note TN–12. DEC Western Research Laboratory, Palo Alto, CA. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-12.pdf

[2] Stephen M. Blackburn and Kathryn S. McKinley. 2002. In or Out? Putting Write Barriers in Their Place. In *3rd ACM SIGPLAN International Symposium on Memory Management (ACM SIGPLAN Notices 38(2 supplement))*, Hans-J. Boehm and David Detlefs (Eds.). ACM Press, Berlin, Germany, 175–184. https://doi.org/10.1145/512429.512452

[3] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly Parallel Garbage Collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (ACM SIGPLAN Notices 26(6))*. ACM Press, Toronto, Canada, 157–164. https://doi.org/10.1145/113445.113459

[4] David Flanagan and Yukihiro Matsumoto. 2008. *The Ruby Programming Language* (first ed.). O'Reilly.

[5] Ryo Hanai, Tsuneyasu Komiya, Masahiro Yasugi, and Taiichi Yuasa. 2003. Automatic Insertion of Write Barriers into C - based Extension Code for Scheme Systems (written in Japanese). *IPSJ Journal, Programming (PRO)* 44, SIG04(PRO17) (mar 2003), 17–24.

[6] Richard Jones, Antony Hosking, and Eliot Moss. 2012. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall.

[7] Duncan MacGregor. 2019. Better support for C extensions in TruffleRuby. (2019). https://aardvark179.github.io/blog/capi.html/

[8] Aaron Patterson. 2018. Performance Impact of Removing OOBGC. (May 2018). https://github.blog/2018-05-18-removing-oobgc/

[9] Chris Seaton, Benoit Daloze, Kevin Menard, Petr Chalupa, Brandon Fish, and Duncan MacGregor. 2019. TruffleRuby — A High Performance Implementation of the Ruby Programming Language. (2019). https://github.com/oracle/truffleruby

[10] P. Shaughnessy. 2013. *Ruby Under a Microscope: Learning Ruby Internals Through Experiment*. No Starch Press. https://books.google.co.jp/books?id=P7AdAgAAQBAJ

[11] Dave Thomas, David Hansson, Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland, and Andreas Schwarz. 2006. *Agile Web Development with Rails*. Pragmatic Bookshelf.

[12] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2647508.2647517