Regular Paper

Buffered Garbage Collection: An Approach to Collecting Self-Created Garbage Objects

Tetsuro Yamazaki[1,a)] Shigeru Chiba[1,b)]

Received: December 19, 2018, Accepted: March 16, 2019

Abstract: This paper proposes a new garbage-collection (GC) algorithm, named *buffered garbage collection*. It allows customizing a garbage collector through computational self-reflection. Although self-reflection seems a promising approach, self-reflection has not been well investigated for garbage collection as far as we know. Our buffered garbage collector collects garbage objects while avoiding both infinite regression and unacceptable memory consumption. We implemented a Scheme-subset interpreter supporting buffered garbage collection and evaluated its memory efficiency.

Keywords: computational self-reflection, garbage collection

1. Introduction

A garbage collector (GC) is a runtime component that is often customized for heap analysis [10], [12] or runtime object evolution [5], [17]. The customization is however not a simple task. A typical approach to customize a collector is to rewrite the program that implements the collector. Since the collector is a part of a programming language system, we cannot customize the collector as we customize applications. The developers have to modify a low-level implementation of the GC component, and most implementations do not provide clean programming interfaces for such customization.

GC customization via computational self-reflection [14] is a promising approach although self-reflection has not been well investigated for garbage collection as far as we know. Suppose that we run a program in a language L_{base} supporting garbage collection. A reflective programming interface for customizing the garbage collector allows developers to modify the behavior of that collector via a program written in the *base* language L_{base} instead of the language the collector is implemented in. That program can intercept the garbage collection during collection time and run as if it is part of the implementation of the garbage collector. For clarification, we below call such a program a *meta* program.

A design problem of such a reflection interface for garbage collectors is how to manage objects created by a meta program. Since a meta program is a normal program, it may create objects and also turn them into garbage during runtime. The garbage collector customized by the meta program should also collect these garbage objects, but naive implementation may cause infinite regression. Note that a meta program may create an object that will substitute another live (base-level) object when it implements object evolution. Hence an object created by a meta program should not be distinguished from normal (base-level) objects.

One possible approach is to allocate some special heap memory that only a meta program can use. Then we can separately collect garbage in that heap memory by a dedicated collector. However, this approach does not satisfy our motivating requirement. The customized garbage collector c_1 never collects the objects created by the collector itself. A different, uncustomized collector c_2 collects them. Hence, the meta program that customizes c_1 will never inspect these objects. We could customize that different collector c_2 just like the original one c_1. However, the customization would introduce the third heap space to place objects created by the second customized collector c_2 and the third space is managed by the third collector c_3 that uses the fourth heap space. Thus, this approach causes infinite regression.

Another approach would be to let a meta program allocate objects in the regular heap memory where the garbage collector is concurrently collecting garbage. It seems promising but its heap memory consumption would be a problem. Since the meta program may create several objects, it may create more objects in total than the existing live objects in the heap. Such huge memory consumption is unacceptable. Reclaiming the objects allocated by the meta program requires searching the whole heap space again. Since the heap space is under garbage collection when the meta program runs, reclaiming them is not feasible in the current GC cycle.

This paper proposes a novel algorithm for reflective garbage collection, *buffered garbage collection*. The paper is an extension to our previous paper [18]. This algorithm allows a meta program customizing a garbage collector to create objects that are also collected by that customized collector while avoiding infinite regression of garbage collection. The buffered garbage collection is based on copying garbage collection [4], [8] but it manages the third space named *buffer* to buffer objects created by a meta program. The buffer space is similar to the nursery space

[1] Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] yamazaki@csg.ci.i.u-tokyo.ac.jp
[b)] chiba@csg.ci.i.u-tokyo.ac.jp

of the generational algorithm. An advantage of this algorithm is that it will consume a smaller amount of heap memory than other approaches. Through a reflection interface, a program can register a callback function that is invoked whenever an object in the old heap is copied to the new heap during GC time. This callback function can customize the collector as it is a meta program in our algorithm. The objects created by the meta program are stored in the buffer space and effectively collected by the customized collector. Since our algorithm introduces staged collection, those objects are not collected until the garbage collection moves into a stable state. We have implemented an interpreter for a subset of Scheme with the proposed garbage collector.

The rest of this paper is as follows. In Section 2, we detail the problem that occurs when we apply computational self-reflection to garbage collectors; why infinite regression and unacceptable memory consumption occurs. In Section 3, we propose buffered garbage collection, our novel garbage collection algorithm that avoids both infinite regression and unacceptable memory consumption. In Section 4, we show the result of our experiment with our Scheme-subset interpreter. It confirms our algorithm avoids unacceptable memory consumption. In Section 5, we compare our algorithm to other possible approaches to clarify the advantages of our algorithm. In Section 6, we conclude this paper.

2. Self-reflective Customization of Garbage Collector

We often customize a garbage collector for adjusting the garbage collection algorithm to a specific application. The customization of garbage collectors is also used for inspecting and/or modifying live objects during the collection. An example is string deduplication [9]. It finds multiple string objects representing the same text and replaces them all with their single representative when they are immutable. Since a web application server tends to construct a large number of duplicated string objects for handling HTTP requests, the string deduplication can often improve the memory consumption and thereby the execution performance. However, searching a whole heap memory for duplicated string objects takes non-trivial time. To hide the cost of this search, string deduplication is often implemented by customizing a garbage collector so that the search for duplicated string objects will be overlapped with the search for live objects. Some garbage collectors perform compaction. Since they modify memory pointers contained in live objects, changing a pointer from a duplicated string object to its representative for string deduplication can be also overlapped with the compaction by the collector.

Since string deduplication does not change the garbage collection algorithm, the required customization is just to add extra work to the existing implementation of the collector. However, even such a small change is not easy to implement, in particular, for developers who did not develop the original implementation. A typical approach to customize a garbage collector is to implement the customization in the language that implements that collector. For example, we can customize the garbage collector of the Java virtual machine in C++ since it is implemented in C++, but such a customization will involve a number of low-level implementation issues of Java. First, the representation of Java objects in C++ is more complicated from the perspective of the GC implementation. The developers have to deal with C++ data structures implementing Java objects; they have to be aware of the objects' meta data, memory layout, and how references are implemented. Furthermore, various low-level invariants must be preserved in the virtual machine. Since C++ code can access hidden data such as meta data and accidentally destroy a memory layout, the developers have to carefully implement GC customization to satisfy the low-level invariants. Note that it is not a problem that the implementation language is C++. Even if the Java virtual machine is implemented in Java as the Jikes RVM [1] and the Maxine VM [16] are, those problems will be observed.

Computational self-reflection [14] is a promising approach to customizing garbage collectors. A program sometimes has to control a non-first-class data structure such as program text, an inheritance mechanism, a JIT compiler, and a garbage collector. Computational self-reflection can be regarded as a design pattern for programming interfaces to access such non-first-class data. Suppose that we run a program in a language L_{base} supporting garbage collection and the collector is not a first-class in L_{base}. A reflective programming interface for garbage collector allows developers to modify the collector as if it has been implemented in L_{base} instead of the language that the collector is implemented in. It often provides a proxy object as a first-class object and it reflects the operations on the proxy object to the corresponding non-first class data. The proxy object can hide low-level implementation details of the non-first-class data. It therefore releases developers from being bothered about those details, for example, when they are customizing the garbage collector for string deduplication, heap analysis, and dynamic object evolution, which do not need to access implementation details such as objects' meta data.

For example, **Fig. 1** presents a pseudo implementation of string deduplication in Scheme. It is a normal Scheme program but uses computational self-reflection. `def-refl!` in line 13 is a special-form for computational self-reflection, which overwrites a (meta-level) function constituting the Scheme interpreter. The syntax of `def-refl!` is similar to `define`. It redefines the `gc:copy-object` function, which is part of the program of the garbage collector. We here assume that the interpreter uses a copying collector. The new function body is the expression in lines 14 to 27. It moves a live object to a new space for compaction. `gc:copy-object` takes three parameters `obj`, `old`, and `new`. `obj` specifies the object being copied, and `old` and `new` specify the memory regions where `obj` is moved from and `obj` is moved to, respectively. Besides copying an object, the new `gc:copy-object` function first finds the representative of the given object by `intern` if it is a string object. If the found object `actual-obj` is allocated in the memory region specified by `old` (that is, the object has not been moved yet), `gc:copy-object` allocates a copy of `actual-obj` in the memory region specified by `new` and calls `set-color!` in line 21 to change the meta data of the object allocated in `dst`. It also calls `set-forward!` in lines 22 and 23 to leave a forward pointer which specifies where `actual-obj` was copied to. `gc:copy-object` leaves the forward pointer not only in `actual-obj` but also in `obj` so that

```
 1 (define intern-table
 2   (make-hashtable string-hash
 3                   string=?))
 4
 5 (define (intern str)
 6   (if (hashtable-contains?
 7        intern-table str)
 8     (hashtable-ref intern-table str)
 9     (begin (hashtable-set!
10             intern-table str str)
11            str)))
12
13 (def-refl! (gc:copy-object obj old new)
14   (let ((actual-obj (if (string? obj)
15                       (intern obj)
16                       obj)))
17     (if (region-contains? actual-obj old)
18       (let ((size (sizeof actual-obj))
19             (new-obj (malloc new size)))
20         (memcpy new-obj actual-obj)
21         (set-color! new-obj GREY)
22         (set-forward! obj new-obj)
23         (set-forward! actual-obj new-obj)
24         new-obj)
25       (begin
26         (set-forward! obj actual-obj)
27         actual-obj))))
```

Fig. 1 String deduplication implemented with reflection.

the collector will update pointers to obj to point to the copy of actual-obj later. It finally returns the allocated object new-obj in line 24. If actual-obj was not allocated in the memory region specified by old, gc:copy-object leaves a forward pointer to actual-obj in new in line 26 and returns actual-obj in line 27. The intern function maintains a hash table from string texts to their unique representative. It is a normal Scheme function but runs at the meta level. It is part of the implementation of the garbage collection.

The gc:copy-object function is invoked when the garbage collector of the Scheme interpreter moves a Scheme object for compaction. Thus, when a string object is moved and another object representing the same character string has been already moved, that object is deleted and all the references to that object are modified to point to the equivalent object that was already moved. Note that the meta program invoked instead of the normal copy function may put a new element into the hash table intern-table. This implies the creation of a new key-value pair. It may cause rehashing and create a number of key-value pairs.

To enable the customization by reflection shown in Fig. 1, we have to address the garbage collection for the objects created by the meta program. As mentioned above, the gc:copy-object function creates an object representing a key-value pair for each string object. It will be garbage when rehashing occurs. Some implementation may use an object to represent a stack frame. Then every call to gc:copy-object will create an object and it will be garbage soon after the call. A Scheme program including a meta program like Fig. 1 tends to create objects with short lifespans. For example, string concatenation usually creates a new string object. Evaluating a list literal allocates a new list instance. Most list operations in Scheme create cons cells.

One possible approach is to allocate a special heap memory that only the meta program can use. Then we can separately collect garbage in that heap memory by a dedicated collector. However, this approach does not satisfy our motivating requirement. The garbage collector customized by the meta program never collects the objects created in that heap memory by the meta program itself. A different, uncustomized collector will collect them. For example, when the meta program for string deduplication creates an object, this object will be never inspected by the garbage collector c_1 customized by that meta program. They are not passed to redefined gc:copy-object when a garbage collector c_1 does collection later. To avoid this, we might have to write another meta program for customizing the garbage collector c_2 for that heap memory where c_1 creates objects. The meta program for c_2, however, will need another heap memory where c_2 creates objects. Who collects garbage in the heap memory used by c_2? The third collector c_3 will collect dead objects, and so forth. Hence, this approach causes infinite regression.

Another approach would be to let a meta program allocate objects in the regular heap memory where the garbage collector is concurrently collecting the garbage. It seems promising but its heap memory consumption would be a problem. For example, the gc:copy-object is invoked for every live object during garbage collection. When it creates objects, therefore, it may rapidly consume the remaining heap memory under the garbage collection and then the heap memory may run out. This might be avoided by concurrently running another collector to reclaim dead objects in that heap. This approach, however, would not work since this second collector would also invoke gc:copy-object for each object.

3. Buffered Garbage Collection

In this section we propose an algorithm for garbage collection, *buffered garbage collection*, which enables computational self-reflection while keeping extra memory consumption within a practical amount. The buffered garbage collection is based on Cheney's copying garbage collection [4] and it enables meta programs to create objects while avoiding both infinite regression and unacceptable memory consumption.

The buffered garbage collection only supports a restricted reflection API we call *copy-time callbacks*. Since customizing a garbage collector is delicate and possibly crashes the interpreter, we restrict the reflection API to have customizations safe and easy. With this style of API, a program can register a callback function that is invoked whenever the garbage collector copies a live object from the old space to the new space. For every copying of a live object, the callback function is called with the object being copied. It can inspect it and returns an object. The garbage collector copies the returned object to the new space for heap compaction if the returned object is still in the old space. Otherwise, the collector does not perform copying. The callback function may return a different object from the object received as the argument. For example, it may return not a duplicated string object but its representative, which may already be copied into the new space. All the pointers to the duplicated objects are changed to the pointers to their representatives returned by the callback function during garbage collection, when pointers to the

```
 1 (define intern-table
 2   (make-hashtable string-hash
 3                   string=?))
 4
 5 (define (intern str)
 6   (if (hashtable-contains?
 7          intern-table str)
 8     (hashtable-ref intern-table str)
 9     (begin (hashtable-set! str str)
10            str)))
11
12 (define (callback-function obj)
13   (if (string? obj) (intern obj) obj))
14
15 (register-on-copy callback-function)
```

Fig. 2 String deduplication implemented with *copy-time callback*.

old space are replaced with the corresponding pointers to the new space. We designed this style of reflection API for extensions to the garbage collection such as string deduplication. Note that this API does not enable to change a garbage collection algorithm.

Despite the simplicity of our copy-time callback API for reflection, we can easily implement string deduplication with this API. **Fig. 2** shows its implementation in a Scheme-like language. `register-on-copy` in line 15 is a primitive for reflection. It is the special form that registers a function as a copy-time callback function. The registered function `callback-function` is a normal Scheme function defined in lines 12–13. It passes the given argument to `intern` if the argument is a string object. Otherwise, it returns the given argument as it is. `intern` is also a normal Scheme function and it is the same as the `intern` function in Fig. 1. It maintains a hash table so that it can quickly look up the representative of the given string object. It may allocate a new object to add a hash table entry or rehash the hash table.

Our buffered garbage collection efficiently manages the objects created by the callback function. Those objects are allocated in a dedicated small memory region, called the *buffer* space, when the callback function creates them. The garbage in the buffer space is frequently collected by minor copying collection between the buffer space and the new space. Those objects in the buffer space are copied into the new space if they are alive when the garbage collector reaches a certain safe point, for example, when each invocation of the callback function finishes. This minor collection does not invokes the callback function. Since all the live objects created by the callback function are moved into the new space, they are processed by the callback function at the next major collection between the new and old spaces. The next major collection will copy them as it copies other normal objects if they are alive.

This buffered garbage collection is based on a simple idea but its algorithm is not such simple. Since two copying collections between old and new and between buffer and new are concurrently performed, the algorithm has to consider more intermediate states of objects. We will next mention details of the algorithm.

3.1 The Colors of Objects

First we introduce several states for objects. These states are based on the tri-color marking abstraction [7], but we use more colors. The tri-color marking abstraction categorizes objects into three groups: white, black, and grey, but for buffered garbage collection we add yellow, silver, and red (see **Fig. 3**). Yellow denotes that the object is already copied into the new region, it contains a forward pointer to the copy, and it may be destroyed. Silver denotes that the object may contain pointers to the buffer region. Red is a special color to detect cyclic object replacements and prevent infinite recursion during the GC.

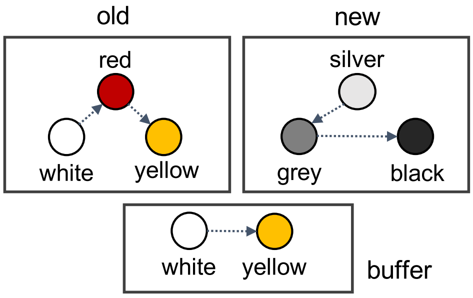


Fig. 3 The colors of objects and the three memory spaces. The arrows represent state transitions (not pointers).

Since our algorithm is an extension to copying collection, the memory space is divided into three regions: old, new, and buffer. At the beginning, objects are in the old space. They are white and their liveness is uncertain. When the garbage collector finds a live white object, it makes a copy of that object in the new space. The copy is colored grey. The grey objects become black if they do not contain any references to white or yellow objects. After making the copy, the collector modifies the copied white object to contain a forward pointer to the copy. We assign the color yellow to the object containing a forward pointer.

Some objects are created by a copy-time callback function and allocated in the buffer space. At first, these objects are white. Our algorithm performs copying collection from the buffer space to the new space as well as one from old to new. Hence a copy made in the new space may contain a reference to an object in the buffer space. Such an object is colored silver. A silver object may contain a reference to the old space.

Finally, we assign the color red to an object in the old space while it is processed by a callback function. This is for avoiding an infinite loop in our algorithm.

3.2 The Algorithm

The buffered garbage collection splits memory space into three regions: old, new, and buffer. An object newly created is allocated in the old space. When the garbage collection starts running, the objects in the old space are moved to the new space if they are alive. The collection continues until all live objects have been moved.

Figure 4 shows an outline of our algorithm. First, the collector colors all objects white in the old space. Then, it makes a copy of every root object in the new space. The copy is colored grey. The references in the root set are updated to point to the copies. The original object in the old space is changed into yellow to contain a forward pointer to its copy. Before making a copy, the collector invokes a callback function. Details of this procedure named *callback-and-copy* are described in Section 3.2.1.

The collector examines all grey objects to find a reference to a

```
def collect():
  for obj in old-space:
    color(obj) = WHITE
  grey-list = {}
  for root in roots:
    if color(*root) == WHITE:
      *root = callback-and-copy(*root)
    else if color(*root) == YELLOW:
      *root = forward(*root)
  until IsEmpty(grey-list):
    grey <- Pop(grey-list)
    for child in Children(grey):
      if color(*child) == WHITE:
        *child = callback-and-copy(*child)
      else if color(*child) == YELLOW:
        *child = forward(*child)
    color(grey) = BLACK
  Clear(old-space)
  Flip(old-space, new-space)
```

Fig. 4 Procedure `collect`.

```
def callback-and-copy(obj):
  if color(obj) == YELLOW:
    return forward(obj)
  color(obj) = RED
  actual-obj = callback(obj)
  Add(remembered-set, &actual-obj)
  flush-buffer()
  if actual-obj == obj:
    forward(obj) = copy(obj)
  else if color(actual-obj) == WHITE:
    callback-and-copy(actual-obj)
    forward(obj) = forward(actual-obj)
  else if color(actual-obj) == RED:
    error
  else:
    # since the buffer space is empty
    # after flash-buffer(), otherwise
    # actual-obj is in the new space
    forward(obj) = actual-obj
  color(obj) = YELLOW
  return forward(obj)
```

Fig. 5 Procedure `callback-and-copy`.

```
def copy(obj):
  if color(obj) == YELLOW:
    return forward(obj)
  result = Allocate(new-space, Size(obj))
  Memcpy(result, obj, Size(obj))
  forward(obj) = result
  color(obj) = YELLOW
  if flushing:
    color(result) = SILVER
    Add(silver-list, result)
  else:
    color(result) = GREY
    Add(grey-list, result)
  return result
```

Fig. 6 Procedure `copy`.

white or yellow object in the old space. If the object is white, a copy of the object is created in the new space by *callback-and-copy*. The resulting copy is (usually) a grey object. The reference is modified to refer to this grey object. If the reference points to a yellow object, it is modified to refer to the object that the forwarding pointer in the yellow object points to. After this examination, the object is turned into black since it does not contain a reference to a white or yellow object.

The garbage collection finishes when all the objects in the new space become black. Then the old space is cleared and the roles of the old and new spaces are swapped. The program execution is resumed.

3.2.1 Procedure Callback-and-copy

callback-and-copy makes a copy of a live white object in the old space. The details of *callback-and-copy* are shown in **Fig. 5**. The copy is stored in the new space. Then, the garbage collector invokes a callback function if it is registered. The white object being copied is passed to the function as an argument. The callback function can access any objects except yellow since yellow objects are already copied and may be destroyed. To avoid accesses to yellow, the algorithm introduces read barriers. When the callback function creates a new object, the object is allocated in the buffer space and colored white. When finishing, the callback function returns any object except yellow ones. The returned object may be in either the old, new, or buffer space.

When the callback function finishes, the procedure named *flush-buffer* is executed (its details are described in Section 3.2.2). It performs copying collection from the buffer space to the new space. After the execution of *flush-buffer*, there exist no references to an object in the buffer space. If the callback function returns the given white object *as is*, the collector passes it to *copy* function shown in **Fig. 6** to make a new copy of that white object. Since it is not flushing, the copy is created in the new space and colored grey. If the callback function returns a white object different from the given white one, the collector recursively invokes *callback-and-copy* to make a copy of that different white object. Otherwise, if the object returned by the callback function is grey or black, the returned object is regarded as a new copy that this invocation of *callback-and-copy* is supposed to make. In either case, the collector finally gives the yellow color to the white object passed to the callback function. It modifies the white object to contain a forward pointer to the copy of that object created by *callback-and-copy*.

As shown above, *callback-and-copy* may recursively invoke itself. To avoid infinite regression, the white object being copied is changed into red before being passed to the callback function. If *callback-and-copy* is invoked later to make a copy of a red object, the collector throws an error.

3.2.2 Procedure Flush-buffer

flush-buffer performs copying collection from the buffer space to the new space as shown in **Fig. 7**. We call this *minor collection*. The buffer space contains the objects created by the callback function. *flush-buffer* does not invoke the callback function when it copies a live object from the buffer space to the new space.

Since *flush-buffer* is invoked after the callback function finishes, *flush-buffer* does not consider stack frames as the root set. The root set for this copying collection is only the remembered set constructed by the write barriers. The reference value returned by the callback function is included in the root set. Hence, if it points to an object in the buffer space, it will be updated to a reference to an object in the new space after the minor collection.

During this minor collection, a copy of an object in the buffer

```
def flush-buffer():
  flushing = true
  silver-list = {}
  for rem in remembered-set:
    if *rem in buffer-space:
      *rem = copy(*rem)
  remembered-set = {}
  until IsEmpty(silver-list):
    silver = Pop(silver-list)
    for child in Children():
      if *child in buffer-space:
        *child = copy(*child)
    color(silver) = GREY
    Add(grey-list, silver)
  Clear(buffer-space)
  flushing = false
```

Fig. 7 Procedure `flush-buffer`.

```
def read(obj):
  if collecting && color(obj) == YELLOW:
    return forward(obj)
  return obj
```

Fig. 8 Procedure `read`.

```
def write(obj, i, value):
  if collecting:
    if color(obj) == BLACK
       && value in old-space:
      color(obj) = GREY
      Add(grey-list, obj)
    else if obj not in buffer-space
            && value in buffer-space:
      Add(remembered-set, &obj[i])
  obj[i] = value
```

Fig. 9 Procedure `write`.

space is made in the new space. The copy is at first colored silver. The minor collector modifies a reference in the silver object only if the reference points to an object in the buffer space. A copy of this object is made in the new space and the reference is updated to point to that copy. This modification is repeated until all the references into the buffer space are updated. A silver object containing no reference into the buffer space is changed into a grey object, which may contain a reference into the old space.

3.2.3 Read and Write Barriers

In our algorithm, a user program runs as a callback function while garbage collector is running. This is similar to concurrent garbage collection, hence our algorithm requires read and write barriers. Their details are shown in **Fig. 8** and **Fig. 9**.

When a callback function writes a value to a black object and the value is a reference pointing to a white object, the black object is changed into grey. When such a reference is written to a grey object, the grey object is marked to be revisited for examining the references in that object. As mentioned in Ref. [15], if this write barriers are used, a live object might be accidentally reclaimed since the references contained in the black object are never examined.

When a callback function writes a value to an object in the old or new space and the value is a reference pointing to an object in the buffer space, the written reference is added to the remember set, which is used by *flush-buffer* as the root set for the minor collection. Like in the generational collection [13] and the regional collection [6], cross-region references have to be remembered. Note that our algorithm requires remembering only cross-region references from the old or new space to the buffer space.

Our algorithm also needs a read barrier. When a callback function attempts to read a value from an object and the value is a reference to a yellow object, the function does not obtain this reference to the yellow object but a forward pointer contained in the yellow object. An object is yellow after its copy is created in the new space. The forward pointer points to this copy. The read barrier guarantees that a callback function never accesses a yellow object and a reference to a yellow object is never written to other objects.

4. Experiment

We compared a buffered garbage collector with a normal copying collector by examining their memory consumption when running micro benchmarks. Since a different configuration of the collectors causes different amount of memory consumption, we ran benchmarks with various restrictions on the heap size and examined whether it could run without a memory error or not. The normal copying collector also supported a copy-time callback. The objects created by the callback function were allocated directly in the new space as discussed in Section 2. It performed garbage collection when the memory consumption exceeds the preset threshold to keep available memory space for the callback function. We examined various thresholds.

For the experiment, we implemented an interpreter for simple Scheme-like language in C++. The interpreter supports not only the buffered garbage collector but also the normal copying collector. In this Scheme-like language, a symbol value is not unique (the same-looking symbols may not be identical), numeric values are not unboxed, every stack frame is allocated as an object in the heap memory, and key/value pairs in hash tables are allocated as objects in the heap memory. The experiment was taken on a machine with the Intel® Core i7-4770S processor (eight 3.10 GHz cores) and 16 GB of memory. Its operating system was Ubuntu16.04 LTS. We used GNU gcc 5.4.0 for compiling the interpreter.

4.1 Benchmarks

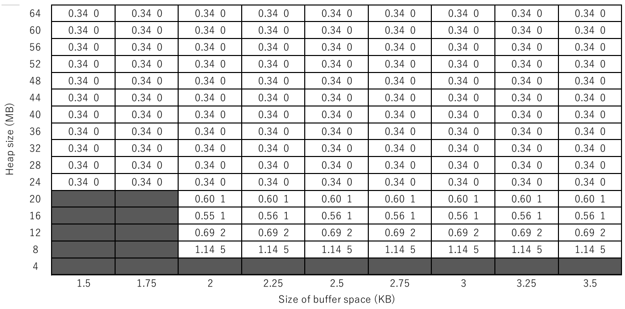
We prepared two micro benchmark programs. The first one counts bi-gram frequencies in a long character string while string deduplication is performed. The character string contained 10240 letters randomly selected among four letters: `a`, `b`, `c`, and `d`. The copy-time callback function implemented string deduplication and its code was shown in Fig. 2. The callback function just returns the given object if it is not a string, or otherwise it returns a string object taken from `intern-table`. The returned object is in the old space or in the new space, depending on the order of copying. Most objects created by the callback function turn into garbage after the callback function finishes, but key/value pairs in `intern-table` will not. Note that key/value pairs created by the callback function has eternal lifetime since no key/value pair is deleted from `intern-table`.

Another micro benchmark is the n-body simulation in **Fig. 10**.

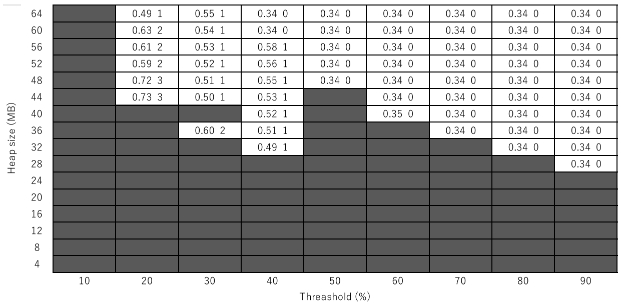
```
(define (v2 x y)
  (cons 'v2-tag
        (lambda (msg)
          (cond [(eq? msg 'x) x]
                [(eq? msg 'y) y]))))
(define (v2? obj)
  (and (cons? obj)
       (eq? (car obj) 'v2-tag)
       (function? (cdr obj))))
(define (x v)
  ((cdr v) 'x))
;; ... define other methods of v2 ...
(define (v3 x y z)
  (cons 'v3-tag
        (lambda (msg)
          (cond [(eq? msg 'x) x]
                [(eq? msg 'y) y]
                [(eq? msg 'z) z]))))
;; ... define methods of v3 ...
(define (mass id w pos vel)
  (cons 'mass-tag
        (lambda (msg)
          (cond [(eq? msg 'id) id]
                [(eq? msg 'w) weight]
                [(eq? msg 'pos) pos]
                [(eq? msg 'vel) vel]))))
;; ... define methods of mass ...
(define mass-points
  (list (mass 1 20.90 (v2 -32.17 43.64)
                      (v2 0.0 0.0)))
  ;; ... more 29 mass points ...
  )
(define (simulate n)
  (times n
    (set! mass-points
          (map update mass-points))))
(define (update m)
  (let* ([id (mass-id m)]
         [weight (mass-weight m)]
         [f (calc-force m)]
         [vel (v2-add (vel m) f)]
         [pos (v2-add (pos m) vel)])
    (mass id weight pos vel)))
(define (v2->v3 v)
  (v3 (x v) (y v) (random)))
(define (evolve-v2->v3 obj)
  (if (v2? obj) (v2->v3 obj) obj))
(simulate 20)
(disable-gc)
(register-on-copy evolve-v2->v3)
(force-gc)
(register-on-copy '())
;; replace methods of v2 to methods of v3
(set! v2-add v3-add)
;; ... and so on
(enable-gc)
(simulate 20)
(print (to-string mass-points))
```

Fig. 10 The benchmark program of the n-body simulation.

When running this benchmark program, the garbage collector was enhanced to perform simple object evolution. It replaced two-dimensional vector objects with three-dimensional vector objects dynamically. The n-body simulation simulates gravitation between mass points. It does not consider collisions. The mass points and vectors are implemented as immutable message-passing style objects; they are functions and receive a symbol as the first argument. It represents the name of the method to invoke. The simulation is non-destructive and all the mass points and vectors are reconstructed at every step of the simulation. The simulation simulates thirty mass points for forty steps. After twenty steps of the simulation, all the vector objects are evolved to three-dimensional ones. The functions on the vectors are also replaced at the same time so that the rest of the simulation does not throw type errors.



Heap size (MB)	1.5	1.75	2	2.25	2.5	2.75	3	3.25	3.5
64	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
60	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
56	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
52	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
48	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
44	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
40	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
36	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
32	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
28	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
24	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
20	■	■	0.60 1	0.60 1	0.60 1	0.60 1	0.60 1	0.60 1	0.60 1
16	■	■	0.55 1	0.56 1	0.56 1	0.56 1	0.56 1	0.56 1	0.56 1
12	■	■	0.69 2	0.69 2	0.69 2	0.69 2	0.69 2	0.69 2	0.69 2
8	■	■	1.14 5	1.14 5	1.14 5	1.14 5	1.14 5	1.14 5	1.14 5
4	■	■	■	■	■	■	■	■	■



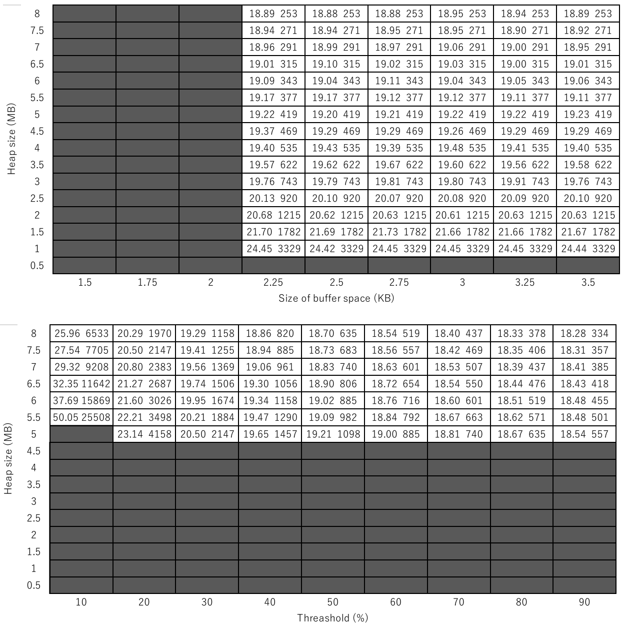
Heap size (MB)	10	20	30	40	50	60	70	80	90
64	■	0.49 1	0.55 1	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
60	■	0.63 2	0.54 1	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
56	■	0.61 2	0.53 1	0.58 1	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
52	■	0.59 2	0.52 1	0.56 1	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
48	■	0.72 3	0.51 1	0.55 1	0.34 0	0.34 0	0.34 0	0.34 0	0.34 0
44	■	0.73 3	0.50 1	0.53 1	■	0.34 0	0.34 0	0.34 0	0.34 0
40	■	■	■	0.52 1	■	0.35 0	0.34 0	0.34 0	0.34 0
36	■	■	0.60 2	0.51 1	■	■	0.34 0	0.34 0	0.34 0
32	■	■	■	0.49 1	■	■	■	0.34 0	0.34 0
28	■	■	■	■	■	■	■	■	0.34 0
24	■	■	■	■	■	■	■	■	■
20	■	■	■	■	■	■	■	■	■
16	■	■	■	■	■	■	■	■	■
12	■	■	■	■	■	■	■	■	■
8	■	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■	■

Fig. 11 Bi-gram counting with the buffered garbage collector (top) and with the normal copying collector (bottom).

4.2 Experimental Results

Figure 11 and **Fig. 12** show the results of the experiments with the benchmarks for bi-gram counting and n-body simulation, respectively. Each result contains two charts: the results of the experiment with buffered garbage collector and the normal copying collector. Each row represents the total heap size excluding the buffer space. The columns in the upper charts in both Fig. 11 and Fig. 12 represent the size of the buffer space. The columns in the lower charts in both Fig. 11 and Fig. 12 represent the threshold when the garbage collection is invoked. The black cells denote heap memory exhaustion and the failures of the execution. The left numbers in the white cells denote the average execution time in seconds over 100 executions. The right numbers in the white cells denote how many times major garbage collection was performed during each execution.

The results of our experiments revealed that our collector could run the micro benchmarks with a smaller amount of heap memory than the normal copying collector. When more than 2 KB was given to the buffer space, the buffered garbage collector could run the program of bi-gram counting with only 8 MB of heap memory (the total heap size including the buffer space was 8,104 KB) and when more than 2.25 KB was given to the buffer space, the buffered garbage collector could run the program of n-body simulation with only 1 MB of heap memory (the total heap size was 1,026.25 KB). The normal copying collector could not run the programs with those amounts of heap memory; it needed at least 24 MB for bi-gram counting and needed 5 MB for n-body simulation. Note that the interpreter with 24 MB of heap memory did



Heap size (MB)	1.5	1.75	2	2.25	2.5	2.75	3	3.25	3.5
8	■	■	■	18.89 253	18.88 253	18.88 253	18.95 253	18.94 253	18.89 253
7.5	■	■	■	18.94 271	18.94 271	18.95 271	18.95 271	18.90 271	18.92 271
7	■	■	■	18.96 291	18.99 291	18.97 291	19.06 291	19.00 291	18.95 291
6.5	■	■	■	19.01 315	19.10 315	19.02 315	19.03 315	19.00 315	19.01 315
6	■	■	■	19.09 343	19.04 343	19.11 343	19.04 343	19.05 343	19.06 343
5.5	■	■	■	19.17 377	19.17 377	19.12 377	19.12 377	19.11 377	19.11 377
5	■	■	■	19.22 419	19.20 419	19.21 419	19.22 419	19.22 419	19.23 419
4.5	■	■	■	19.37 469	19.29 469	19.29 469	19.26 469	19.29 469	19.29 469
4	■	■	■	19.40 535	19.43 535	19.39 535	19.48 535	19.41 535	19.40 535
3.5	■	■	■	19.57 622	19.62 622	19.67 622	19.60 622	19.56 622	19.58 622
3	■	■	■	19.76 743	19.79 743	19.81 743	19.80 743	19.91 743	19.76 743
2.5	■	■	■	20.13 920	20.10 920	20.07 920	20.08 920	20.09 920	20.10 920
2	■	■	■	20.68 1215	20.62 1215	20.63 1215	20.61 1215	20.63 1215	20.63 1215
1.5	■	■	■	21.70 1782	21.69 1782	21.73 1782	21.66 1782	21.66 1782	21.67 1782
1	■	■	■	24.45 3329	24.42 3329	24.45 3329	24.45 3329	24.45 3329	24.44 3329
0.5	■	■	■	■	■	■	■	■	■

Heap size (MB)	10	20	30	40	50	60	70	80	90
8	25.96 6533	20.29 1970	19.29 1158	18.86 820	18.70 635	18.54 519	18.40 437	18.33 378	18.28 334
7.5	27.54 7705	20.50 2147	19.41 1255	18.94 885	18.73 683	18.56 557	18.42 469	18.35 406	18.31 357
7	29.32 9208	20.80 2383	19.56 1369	19.06 961	18.83 740	18.63 601	18.53 507	18.39 437	18.41 385
6.5	32.35 11642	21.27 2687	19.74 1506	19.30 1056	18.90 806	18.72 654	18.54 550	18.44 476	18.43 418
6	37.69 15869	21.60 3026	19.95 1674	19.34 1158	19.02 885	18.76 716	18.60 601	18.51 519	18.48 455
5.5	50.05 25508	22.21 3498	20.21 1884	19.47 1290	19.09 982	18.84 792	18.67 663	18.62 571	18.48 501
5	■	23.14 4158	20.50 2147	19.65 1457	19.21 1098	19.00 885	18.81 740	18.67 635	18.54 557
4.5	■	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■	■
3.5	■	■	■	■	■	■	■	■	■
3	■	■	■	■	■	■	■	■	■
2.5	■	■	■	■	■	■	■	■	■
2	■	■	■	■	■	■	■	■	■
1.5	■	■	■	■	■	■	■	■	■
1	■	■	■	■	■	■	■	■	■
0.5	■	■	■	■	■	■	■	■	■

Fig. 12 N-body simulation with the buffered garbage collector (top) and with the normal copying collector (bottom).

not have to perform garbage collection to run bi-gram counting.

The normal copying collector mostly ran the benchmark programs faster. In the experiment with the benchmark of n-body simulation, the normal copying collector involved a huge overhead caused by a huge number of collections when the threshold was 10%. However, the normal copying collector runs faster than the buffered collector at the same heap size if the threshold was greater than 10%. For example, the normal copying collector runs in 18.39 sec. with 80% threshold but the buffered collector runs in 18.95 sec. with 3.5 KB buffer space when the heap size was 7 MB. Since the variance was 1e-2, the difference is not thought as an error of measurement.

5. Comparison to Related Works

Although there are several approaches to customizable garbage collectors, we designed a copy-time callback function and buffered garbage collection to satisfy our requirements presented in Section 2. The first requirement is the use of reflection because of its ease of customization. The other is to avoid large memory overheads due to the customizability. In this section, we compare the buffered garbage collection to other approaches, which include ones we have briefly presented in Section 2.

5.1 Dynamically Linked Library

One of the simplest approaches to implementing a garbage collector with a copy-time callback function is to have the interpreter dynamically link a callback function written in C++ (for the clarity of the presentation, we assume that we are implementing a Scheme interpreter in C++).

Most C++ execution environments support dynamically linked libraries. We can build a library module containing a callback function and load it on demand to be linked with the garbage collector of the interpreter. The callback function can flexibly customize the garbage collector since both the callback function and the collector are written in C++.

```
 1 #include <unordered_map>
 2 #include <functional>
 3 #include "SchemeObject.hpp"
 4 #include "SchemeHash.hpp"
 5 using namespace std;
 6
 7 typedef SchemeObject    SO;
 8 typedef SchemeObjectRef SORef;
 9
10 unordered_map<
11   SORef,
12   SORef,
13   function<size_t(SORef)>,
14   function<bool(SORef, SORef)>
15 > intern_table(0,
16                schemeHash,
17                schemeEquals);
18
19 SORef intern(SORef str){
20   if(!map_contains(intern_table, str)){
21     return intern_table[str] = str;
22   }else{
23     return intern_table[str];
24   }
25 }
26
27 SORef onCopy(SORef obj){
28   if(obj.ref_type == SORef::OBJECT &&
29      obj->type    == SO::STRING){
30     return intern(obj);
31   }else{
32     return obj;
33   }
34 }
```

Fig. 13 String deduplication by a dynamically-linked callback function.

A problem of this approach is that a callback function has to be written at the level of abstraction of the C++ implementation of the garbage collector. **Figure 13** shows an example of the callback function implementing string deduplication in this approach. It is equivalent to Fig. 2, which our approach enables. SchemeObject (or SO in short) and SchemeObjectRef (or SORef in short) are utility data types provided by the interpreter implementation for the developers of callback functions. SO is a C++ object implementing a Scheme object. SORef is also a C++ object but implementing a reference value in Scheme. It is a smart pointer that points to an SO and encapsulates the maintenance of the root set for garbage collection. Without these utility data types, implementing a callback function would be extremely error-prone for developers who do not know the detailed implementation of the interpreter. For example, developers need careful attention for appropriately removing reference values from the root set when an exception is thrown.

The main part of the callback function is onCopy in lines 27–34. Note that it inspects the meta data of obj to determine whether it is a string object or not. Although line 28 inspects the meta data of a given reference value, line 29 inspects the meta data of the Scheme object that the reference value points to. The developers have to be aware of these details.

onCopy calls intern, which returns a canonical representation for the given string object. It looks into the hash table intern_table. This hash table is declared in lines 10–17. schemeHash and schemeEqual are provided by the interpreter

```
 1 (#%module-begin
 2   (library gc-customization
 3     (export on-copy)
 4     (import scheme-interpreter))
 5
 6 (define intern-table
 7   (make-hashtable scheme-hash
 8                   scheme-equal?))
 9
10 (define intern
11   (lambda (str)
12     (if (hashtable-contains?
13           intern-table str)
14       (hashtable-ref intern-table str)
15       (begin (hashtable-set!
16                intern-table str str)
17              str))))
18
19 (define on-copy
20   (lambda (obj)
21     (if (and (eq? (ref-type obj)
22                   'OBJECT)
23              (eq? (type-of  obj)
24                   'STRING))
25       (inetrn obj)
26       obj))))
```

Fig. 14 String deduplication by a callback function written in Scheme.

implementation. The former computes a hash value of the string object and the latter compares two string objects. These string objects are not ones in C++ but the C++ objects implementing Scheme's string objects. This *meta* perception often causes errors.

5.2 Single Language

Another approach is to implement an interpreter in the same language that the interpreter interprets. If we are implementing a Scheme interpreter, the interpreter is implemented in Scheme. Implementing such an interpreter is feasible; we can run a Scheme interpreter written in Scheme on top of the Scheme interpreter written in C++.

Developers can now write a copy-time callback function in Scheme since the interpreter is written in Scheme. Although Scheme provides a much higher-level programming abstraction than C++, they still encounter the problem of the low-level abstraction used by the interpreter implementation. If the interpreter directly exposes the implementations of the garbage collector to a callback function, the developers have to write a Scheme program similar to the C++ program in the previous approach. A callback function would have to process a given object from the perspective of the interpreter implementation. The programming might be confusing and worse than the previous approach since the developers have to distinguish Scheme objects processed by the garbage collector from Scheme objects used in the callback function.

Figure 14 shows a callback function written in this approach for string deduplication. We can observe a one-to-one similarity between Fig. 13 and Fig. 14 while Fig. 14 has to deal with lower-level abstractions than Fig. 2 that our approach enables. The callback function `on-copy` in Fig. 14 uses `ref-type` and `type-of` in lines 21 and 23 for accessing the meta data of `obj`. The developers cannot use the standard function `string?` to determine whether `obj` is a string object or not. If we call `string?` with an object implementing a string object in the interpreted Scheme, `string?` will return `false`. A similar problem is seen for the hash table.

This single-language approach was adopted by a more practical system, Jikes Research Virtual Machine (Jikes RVM) [1]. It is an implementation of the Java virtual machine written in Java. Jikes RVM provides the Memory Management Toolkit (MMTk) [3] for implementing a new garbage collector as easily as our copy-time callback function. However, implementing a new garbage collector with MMTk for Jikes RVM causes the problem mentioned above.

5.3 Other Garbage Collectors

As we have shown in Section 2, a copy-time callback function will need a large amount of extra memory space if the objects created by the callback function are allocated directly in the new space, which live objects are being moved into. According to our requirement, the callback function is invoked for every live object in the old space and thus the total amount of objects created by the callback function is proportional to the number of the live objects. In some interpreter implementations, a callback function may implicitly create an object for its stack frame and so on. If so, collecting every small live object consumes a bigger memory block than the object's.

Our buffered garbage collection first allocates those objects in the buffer space and then moves only live objects to the new space. The move from the buffer space to the new space is frequently performed every time when the callback function finishes. We can expect that our collector consumes a smaller amount of memory.

A key idea of our approach is to reclaim garbage objects created by a callback function while the garbage collector is still running. Hence, a concurrent garbage collector [2] might seem to be able to reclaim the garbage objects created by the callback function if they are allocated in the old space. An issue with this approach however is whether garbage is reclaimed faster than it is produced. Recall that the callback function may create several larger objects when it collects one live object. We will also need to study the execution of the callback function during the second pause time (or the remark phase) of the concurrent collection.

The buffered garbage collection can be regarded as a variant of the generational collection [13]. Like the generational collection, the minor collection from the buffer space to the new space exploits the fact that most objects created by the callback function are garbage when the function finishes. The buffered garbage collection uses the buffer space to identify such short-lived objects. On the other hand, the generational collection exploits the fact that most of the recently created objects are short-lived. It does not provide multiple regions where objects are initially allocated. All objects are initially allocated in the young space and they are equally treated.

We see a similarity to the regional garbage collection [6], [11] in the fact that objects are initially allocated in two regions, the old space or the buffer space, and that they are separately scavenged. Although our aim is not to reduce the pause time related to garbage collection, it would be possible to emulate our algorithm by customizing a regional collector. The customized collector would use one region as the buffer space and, at the first time, scavenge that region without invoking a copy-time callback function. Then that region would be changed into part of the normal space where the callback function does not create objects. When that region is scavenged next time, the collector invokes the callback function. The collector uses another fresh region as the buffer space where the invoked callback creates objects.

6. Conclusion

This paper proposed the buffered garbage collection, which allows us to customize a garbage collector via computational self-reflection. The experiment showed that the buffered garbage collector could run our benchmark program without consuming unacceptable huge memory. A limitation is that our garbage collection is based on copying algorithm. Therefore, the collector stops the world during garbage collection and only the half of an available memory space is used. To avoid these problems, applying our idea to regional collectors is a future work.

References

[1] Alpern, B., Attanasio, C.R., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J.J., Hummel, S.F., Sheperd, J.C. and Mergen, M.: Implementing JalapeÑO in Java, *Proc. 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp.314–324, ACM (1999).

[2] Baker, Jr., H.G.: List Processing in Real Time on a Serial Computer, *Comm. ACM*, Vol.21, No.4, pp.280–294 (online), DOI: 10.1145/359460.359470 (1978).

[3] Blackburn, S.M., Cheng, P. and McKinley, K.S.: Oil and Water? High Performance Garbage Collection in Java with MMTk, *Proc. 26th International Conference on Software Engineering, ICSE '04*, pp.137–146, IEEE Computer Society (online), available from ⟨http://dl.acm.org/citation.cfm?id=998675.999420⟩ (2004).

[4] Cheney, C.J.: A Nonrecursive List Compacting Algorithm, *Comm. ACM*, Vol.13, No.11, pp.677–678 (online), DOI: 10.1145/362790.362798 (1970).

[5] Cohen, T. and Gil, J.Y.: Three Approaches to Object Evolution, *Proc. 7th International Conference on Principles and Practice of Programming in Java, PPPJ '09*, pp.57–66, ACM (online), DOI: 10.1145/1596655.1596665 (2009).

[6] Detlefs, D., Flood, C., Heller, S. and Printezis, T.: Garbage-first Garbage Collection, *Proc. 4th International Symposium on Memory Management, ISMM '04*, pp.37–48 ACM (online), DOI: 10.1145/1029873.1029879 (2004).

[7] Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S. and Steffens, E.F.M.: On-the-fly Garbage Collection: An Exercise in Cooperation, *Comm. ACM*, Vol.21, No.11, pp.966–975 (online), DOI: 10.1145/359642.359655 (1978).

[8] Fenichel, R.R. and Yochelson, J.C.: A LISP Garbage-collector for Virtual-memory Computer Systems, *Comm. ACM*, Vol.12, No.11, pp.611–612 (online), DOI: 10.1145/363269.363280 (1969).

[9] Horie, M., Ogata, K., Kawachiya, K. and Onodera, T.: String Deduplication for Java-based Middleware in Virtualized Environments, *Proc. 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14*, pp.177–188, ACM (online), DOI: 10.1145/2576195.2576210 (2014).

[10] Jump, M. and McKinley, K.S.: Cork: Dynamic Memory Leak Detection for Garbage-collected Languages, *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '07*, pp.31–38, ACM (online), DOI: 10.1145/1190216.1190224 (2007).

[11] Klock, II, F.S. and Clinger, W.D.: Bounded-latency Regional Garbage Collection, *Proc. 7th Symposium on Dynamic Languages, DLS '11*, pp.73–84, ACM (online), DOI: 10.1145/2047849.2047859 (2011).

[12] Li, D. and Srisa-an, W.: Quarantine: A Framework to Mitigate Memory Errors in JNI Applications, *Proc. 9th International Conference on Principles and Practice of Programming in Java, PPPJ '11*, pp.1–10, ACM (online), DOI: 10.1145/2093157.2093159 (2011).

[13] Lieberman, H. and Hewitt, C.: A Real-time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM*, Vol.26, No.6, pp.419–429 (online), DOI: 10.1145/358141.358147 (1983).

[14] Smith, B.C.: Reflection and Semantics in LISP, *Proc. 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '84*, pp.23–35, ACM (online), DOI: 10.1145/800017.800513 (1984).

[15] Wilson, P.R.: Uniprocessor Garbage Collection Techniques, *Proc. International Workshop on Memory Management, IWMM '92*, pp.1–42, Springer-Verlag (online), available from ⟨http://dl.acm.org/citation.cfm?id=645648.664824⟩ (1992).

[16] Wimmer, C., Haupt, M., Van De Vanter, M.L., Jordan, M., Daynès, L. and Simon, D.: Maxine: An Approachable Virtual Machine for, and in, Java, *ACM Trans. Archit. Code Optim.*, Vol.9, No.4, pp.30:1–30:24 (2013).

[17] Würthinger, T., Wimmer, C. and Stadler, L.: Dynamic Code Evolution for Java, *Proc. 8th International Conference on the Principles and Practice of Programming in Java, PPPJ '10*, pp.10–19, ACM (online), DOI: 10.1145/1852761.1852764 (2010).

[18] Yamazaki, T. and Chiba, S.: Buffered Garbage Collection for Self-reflective Customization, *Proc. 33rd Annual ACM Symposium on Applied Computing, SAC '18*, pp.1256–1259, ACM (online), DOI: 10.1145/3167132.3167416 (2018).

Tetsuro Yamazaki received his Master degree from Graduate School of Information Science and Technology of the University of Tokyo in 2017. His reserch interest is programming languages.

Shigeru Chiba received his Ph.D. degree from the University of Tokyo 1996. He became an assistant professor at University of Tsukuba in 1997 and at Tokyo Institute of Technology in 2001, and a professor at Tokyo Institute of Technology in 2008. He is a professor at the University of Tokyo since 2011. His research interests include programming languages, software engineering, and system software.