Buffered Garbage Collection for Self-Reflective Customization

Tetsuro Yamazaki
The University of Tokyo
yamazaki@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo
chiba@acm.org

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.

CCS CONCEPTS

• **Software and its engineering → Garbage collection**;

KEYWORDS

computational self-reflection, garbage collection

ACM Reference Format:
Tetsuro Yamazaki and Shigeru Chiba. 2018. Buffered Garbage Collection for Self-Reflective Customization. In *SAC 2018: SAC 2018: Symposium on Applied Computing , April 9–13, 2018, Pau, France.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3167132.3167416

1 INTRODUCTION

A garbage collector (GC) is a runtime component that is often customized for heap analysis [7, 9] or runtime object evolution [2, 12]. The customization is however not a simple task since 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 [11] 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 call such a program a *meta* program from this point on.

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167416

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 never collects the objects created by the collector itself. A different, uncustomized collector c_2 collects them. We could customize that different collector just like the original one. 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. If placing an object into the heap, the collector has to traverse the whole heap later to test whether the object is alive or dead. Thus, if a meta program allocates objects in the regular heap, these objects will not be collected during the current GC cycle. Since a meta program may create a large number of objects when a live object is found and copied, the meta program may create more objects in total than the existing ones in the heap. Such huge memory consumption is unacceptable.

This paper proposes a novel algorithm for reflective garbage collection, *buffered garbage collection*. 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. Buffered garbage collection is based on copying garbage collection [1, 5] 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 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

```
1 (define intern-table
2   (make-hashtable string-hash
3                   string=?))
4
5 (define intern (lambda (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 string-deduplication-callback
13   (lambda (obj)
14     (if (string? obj) (intern obj)
15         obj)))
16
17 (register-on-copy
18   string-deduplication-callback)
```

Figure 1: String deduplication implemented with reflection

the garbage collection moves into a stable state. We have implemented an interpreter for a subset of Scheme with the proposed garbage collector.

2 BUFFERED GARBAGE COLLECTION

In this section we propose an algorithm for garbage collection, *Buffered garbage collection*, which enables computational self-reflection on garbage collection while keeping extra memory consumption within a practical amount. Buffered garbage collection is based on Cheney's copying garbage collection [1] and it enables meta programs to create objects while avoiding both infinite regression and unacceptable memory consumption.

The reflection mechanism of Buffered garbage collection is simple; it only provides a copy-time callback function. Once a callback function is registered, the collector invokes it instead of the regular copy function to copy a live object into new space. The callback function can read/write to the given copied object and the global environment. It can also create new objects. The callback function can also replace the copied object by returning the replacement. The callback function is written in the same language that an application program is written in. For example, string deduplication [6] can be implemented as a copy-time callback function. Figure 1 shows an example implementation of string deduplication by using the copy-time callback. String deduplication is a technique to save memory by replacing duplicated string objects with a representative string object when they are immutable. Note that this callback function may create objects when registering a representative string object into a hash table `intern-table`.

In our algorithm, the objects created by the callback function are allocated in a dedicated small memory region, called *buffer*. 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. Since all the live objects created by the callback function

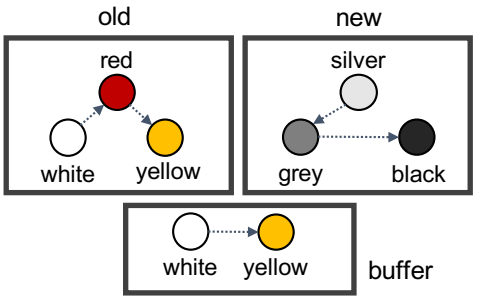

Figure 2: The colors of objects and the three memory spaces

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.

2.1 The colors of objects

Before presenting our algorithm, we introduce several states for objects. These states are based on the tri-color marking abstraction [4], 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 Figure 2). 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.

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.

2.2 The algorithm

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.

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 2.2.1.

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

2.2.1 Procedure callback-and-copy. *callback-and-copy* makes a copy of a live white object in the old space. 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 it creates a new object, the object is allocated in the buffer space and colored white. *callback-and-copy* finishes by returning 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 detailed are described in section 2.2.2). 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 makes a new copy of that white object. 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.

2.2.2 Procedure flush-buffer. *flush-buffer* performs copying collection from the buffer space to the new space. 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.

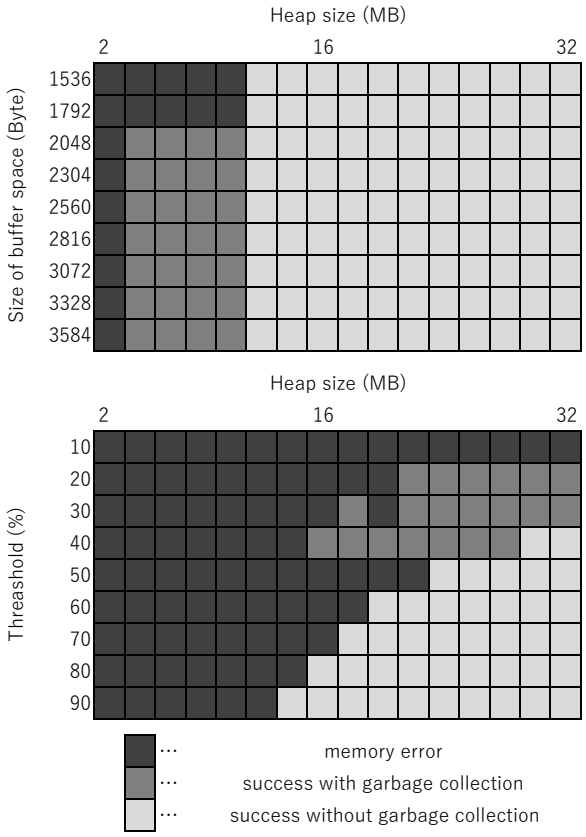

Figure 3: Bi-gram counting with the buffered garbage collector (top) and with the normal copying garbage collector (bottom)

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 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 EXPERIMENT

To evaluate Buffered garbage collection, we implemented an interpreter for a simple Scheme-like language in C++. The interpreter supports not only the Buffered garbage collector but also the normal copying collector, where the objects created by a copy-time callback function are allocated directly in the new space. The garbage collection is initiated when the memory consumption exceeds the preset threshold. In this Scheme-like language, a symbol value is not unique (the same-looking symbols may not be identical), numeric values are not unboxed, and every stack frame is allocated as an object in the heap memory.

We ran a micro benchmark program on this interpreter and examined whether the Buffered garbage collector could run with a smaller amount of memory than the normal copying collector when a copy-time callback function was registered. The benchmark program counted bi-gram frequencies in a long character string. This 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 Figure 1. The interpreter was run 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.

Figure 3 shows the results of the experiments with the buffered garbage collector and the normal copying collector, respectively. Each column represents the total heap size excluding the buffer space. The rows in the upper chart of Figure 3 represents the size of the buffer space in Bytes (not KBytes). The rows in the lower chart of Figure 3 represents the threshold when the garbage collection is initiated. Since the interpreter needs an extra margin in the heap memory to run a copy-time callback function with the normal collector, we examined different thresholds. For the buffered collector, we chose 100% for the threshold. The color of each cell shows the result of executing our micro-benchmark program. Black cells denote a heap memory shortage and the failure of execution. White cells denote that no heap memory shortage or garbage collection happened. Gray cells denote a successful execution with garbage collection and string deduplication.

The results of our experiments reveal that our collector could run the micro benchmark 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 with only 4 MB of heap memory (the total heap size including the buffer space was 8194 KB). The normal copying collector could not run the program with this amount of heap memory; it needed at least 12 MB. Note that the interpreter with this size of heap memory did not have to perform garbage collection. The garbage collection was initiated when the threshold was set from 20 to 40%. Under this configuration, however, the normal copying collector required more than 16 MB of heap memory.

4 RELATED WORK

Buffered garbage collection can be regarded as a variant of the generational collection [10]. 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. 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 [3, 8] 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.

5 CONCLUSION

This paper proposed 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] C. J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (Nov. 1970), 677–678.

[2] Tal Cohen and Joseph (Yossi) Gil. 2009. Three Approaches to Object Evolution. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ '09)*. ACM, 57–66.

[3] David Detlefs et al. 2004. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, 37–48.

[4] Edsger W. Dijkstra et al. 1978. On-the-fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM* 21, 11 (Nov. 1978), 966–975.

[5] Robert R. Fenichel and Jerome C. Yochelson. 1969. A LISP Garbage-collector for Virtual-memory Computer Systems. *Commun. ACM* 12, 11 (Nov. 1969), 611–612.

[6] Michihiro Horie et al. 2014. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '14)*. ACM, 177–188.

[7] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, 31–38.

[8] Felix S. Klock, II and William D. Clinger. 2011. Bounded-latency Regional Garbage Collection. In *Proceedings of the 7th Symposium on Dynamic Languages (DLS '11)*. ACM, 73–84.

[9] Du Li and Witawas Srisa-an. 2011. Quarantine: A Framework to Mitigate Memory Errors in JNI Applications. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. ACM, 1–10.

[10] Henry Lieberman and Carl Hewitt. 1983. A Real-time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM* 26, 6 (June 1983), 419–429.

[11] Brian Cantwell Smith. 1984. Reflection and Semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, 23–35.

[12] Thomas Würthinger et al. 2010. Dynamic Code Evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*. ACM, 10–19.