Electronic Preprint for Journal of Information Processing Vol.26

Research Paper

Promotion Condition Optimization based on Application Features in Generational GC of Android Application Runtime

Ryusuke Mori[1] Masato Oguchi[2] Saneyasu Yamaguchi[1,a)]

Received: September 30, 2017, Accepted: February 14, 2018

Abstract: Android Runtime (ART), which is the standard application runtime environment, has a garbage collection (GC) function. ART have an implementation of generational GC. The GC clusters objects into two groups, which are the young and old generations. An object in the young generation is promoted into an old object after passing several times of GC executions. In this paper, we propose to adjust the promoting condition based on the object feature, which is the size of each object. We then evaluate our proposed method and demonstrate that our method based on the feature can reduce the memory consumption of applications with smaller performance decline than the method without feature consideration.

Keywords: garbage collection, generational GC, Android, ART

1. Introduction

The Android operating system has become one of the most popular operating systems in mobile devices with a market share of 85.0% in Q1 2017 [1]. The Android operating system has a process terminating function, which is called Low Memory Killer. The function automatically terminates application processes when the size of available memory becomes small [2]. This function enables a user to invoke applications without manual termination of processes. However, re-use of an application that is terminated requires process re-creation. This takes longer time than that of non-terminated application. Thus, decreasing the number of process terminations by Low Memory Killer is desired. For relieving memory shortage, reducing the size of memory consumed by applications is effective.

In the Android operating system, application memory size is controlled by GC. In order to decrease the number of process termination, decreasing the sizes of the memories consumed by processes is important. Recent Android operating systems utilize Android Runtime (ART) as the standard application runtime environment. ART has a GC function. The function is invoked when the size of the available memory is small. It then searches the heap area for unused memories and releases them. ART has a generational GC [3] implementation, which is called Generational Semi space (GSS). Generational GCs, including GSS, separate objects into two groups, which are young and old objects. The GCs assume that young objects will probably die soon and old objects will not do. The GCs actively check young objects for achieving good throughput, which is the total size of the released objects per second. For covering the whole objects, these GCs sometimes check all the objects, including both of the young and old objects. In the Android GSS, every object is promoted into an old object after passing two times of GC executions independent to its feature, such as its size or class name. In addition, the existing works [4], [5] showed the possibility of that the performance of the GSS GC could be improved by optimizing the promoting condition.

In this paper, we focus on the GSS GC and propose new methods, which are a naïve method and a size-aware method, for improving its heap size and Stop The World (STW) time by optimizing its promotion condition. In this GC, the memory size and the STW time are in a trade-off relationship. The STW time has an effect on the application performance. STW will be explained in Section 2.1. Our method that considers an object feature, which is the size of each object, reduces the sizes of application memory and relieves the performance decline. We then evaluate the proposed methods with our implementations of the proposed GC. This paper is based on the previous works of Refs. [2], [4], [6].

2. GC Algorithms

In this section, we explain GCs, their algorithms, and their implementation. In this paper, we focus on the generational GC algorithm in Section 2.5 and its implementation in Section 2.6. The implementation and algorithm are based on the algorithms in Sections 2.2, 2.3, and 2.4.

2.1 Garbage Collection

GC is a function that finds garbage objects and releases their memories. It is invoked when the size of the available memory is small. Without GC, such as programming C language, programmers have to manage memory usage manually. They consider

[1] Kogakuin University, Shinjuku, Tokyo 163–8677, Japan
[2] Ochanomizu University, Bunkyo, Tokyo 112–8610, Japan
[a)] sane@cc.kogakuin.ac.jp

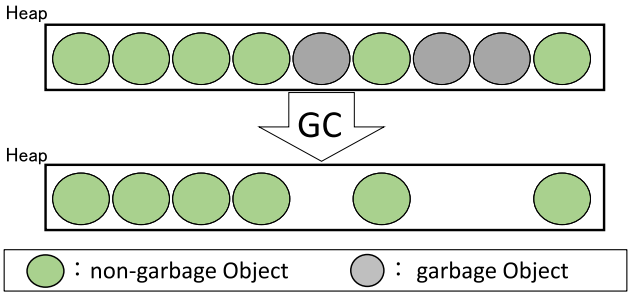

Fig. 1 Garbage collection.

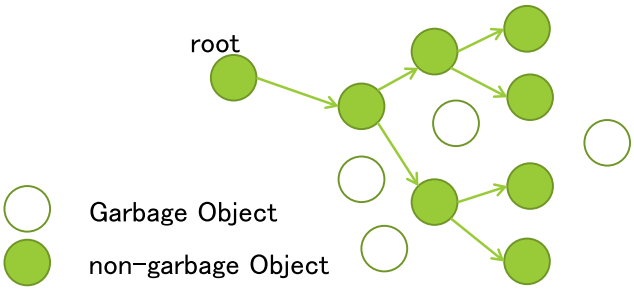

Fig. 2 Mark and Sweep.

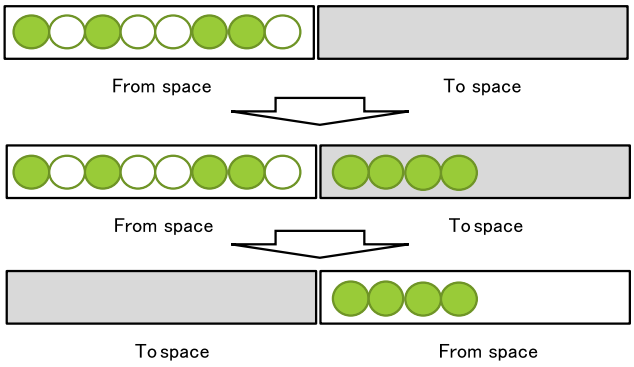

Fig. 3 Copying GC.

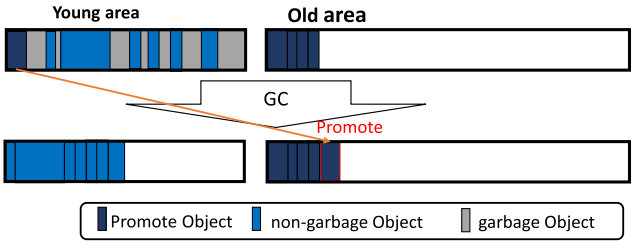

Fig. 4 Generational GC.

object lifetimes, and then explicitly allocate and release memory of used and unused objects. These are sometimes difficult for programmers and cause memory leaks. With GC, programmers are remarkably relieved from these bugs and hard management. In the following subsections, we introduce three basic GC algorithms and one advanced algorithm.

Figure 1 illustrates an overview of a GC behavior. There are objects in a memory space. Objects that are not referred directly or indirectly by the root objects of the application are called garbage objects. They cannot be accessed by the application. Application runtime environments cannot detect whether an object is a garbage object or not without GC execution. GC recursively checks the references from the root objects and finds garbage objects. In a case of Mark and Sweep GC, which is described in the next subsection, the GC marks objects that can be accessed from the root objects recursively. Objects that are not marked are garbage objects and they are released. During a GC execution, all the application threads stop. This is called Stop The World (STW).

Several GC algorithms have been proposed. We introduce them in the following subsections.

2.2 Mark and Seep

Mark and Sweep (MS) [8] is an algorithm that marks used object and releases unmarked objects. This GC algorithm is composed of the Mark phase and the Sweep phase. In the Mark phase, the GC recursively follows references from the root objects, which are directly referenced by the interpreter, and marks them. In the Sweep phase, unmarked objects are released. The unmarked objects are not referred directly or indirectly by the interpreter and no longer used. Its overview is shown in **Fig. 2**. The released memory can be used again. The advantages of this algorithm are as follows. Its overhead during non-GC time is small, unlike the reference counting algorithm. It can release objects with reference loops. The disadvantage is time to complete the GC process is relatively long because of following all the references.

2.3 Reference Counting

Reference counting [9] is an algorithm that counts the number of in-references of each object. It determines an object without incoming references as a garbage object and releases that. Its advantage is that this can release an object immediately at detection of no reference without following many references like MS GC. The disadvantages are that the GC requires updating the number of in-references at each change of references and this cannot release objects with reference loops.

2.4 Copying GC

The Copying GC [10] separates the heap area into two spaces, which are From and To spaces. Its overview is shown in **Fig. 3**. Memory for new objects is allocated in From space. The GC is invoked when the From space reaches full. This GC conducts the process same as MS GC for detecting garbage objects. In order to release the memories, this GC copies the used objects in From space to To space. After the copy, the GC releases the entire From space, and exchanges the From and To spaces. Its advantage is that the copy causes compaction and fragmentation is automatically solved. Its disadvantage is its ineffective memory usage. This GC divides the heap into two spaces. Thus, the size of usable spaces decreases. Moreover, this GC has to check the entire heap area.

2.5 Generational GC

Generational GC separates objects into two generations, the young and old generations, according to their ages like **Fig. 4**. The GC expect based on the heuristic hypothesis that young objects will probably die in the near future. Thus, the GC actively checks the young generation. Most GC algorithms, such as MS

GC, require a reference following process that covers all the alive objects. Time to complete this process is short in a case of many objects die and little objects are alive. Generational GC expects that a reference following process in young generation finishes shortly [11]. The GC can be tuned by adjusting the frequencies of checks for the young and old generations. Generational GC is not exclusive to and can coexist with other GC algorithms. This is utilized for improving other existing algorithms such as MS GC.

An object is stored in the young generation area at its creation. When the young generation area becomes full, a GC for the young generation is performed. An object that passes some GC executions is promoted to the old object. When the old generation area becomes full, a GC for the whole area, including both of the young and old area, is performed.

2.6 Generational GC of ART

ART has a generational GC implementation, called *GSS*. For collecting garbage objects among the young objects, a Copying GC is used. The young generation area is separated into the From and To spaces.

The GC has two types of GCs, the *bps* and *whole* GCs, inside. The former is only for the young generation. The latter is for both generations. Young objects are actively checked by the bps GC. An object that survived two times of GCs is promoted into an old object. When the total size of objects that promoted after the last whole GC execution exceeds the *PromotedThreshold*, the whole GC is performed. The default value of *PromotedThreshold* is 4 MB. The target of the whole GC is all the objects while that of the bps GC is only the young objects. Naturally, time to complete the whole GC is longer than that of the bps GC, and frequent execution of the whole GC should be avoided for achieving better performance. However, unused objects in the old area are released only by the whole GC. Thus, execution of the whole GC is important to decrease the heap sizes of applications.

3. Object Lifetime Trend in Smartphone Applications

In this section, we introduce the object lifetime trend of practical Android applications. The surveying results on the object lifetime trend, such as the relationship between object sizes and object lifetimes, of practical Android applications were reported in works [4], [5]. In these, the authors constructed their ART monitoring system and observed object creations and collections by GC in ART. They surveyed the behaviors of the top applications in all categories in Google Play Store on June 12, 2016.

On the relationship between lifetime length and the number of objects, they reported that the number of objects with the shortest lifetime was the largest. The number of objects monotonously decreased as lifetime length increases. They found that the number of objects with lifetime-zero, which died in the first GC, was the largest. These objects accounted for about 80%. In these papers, lifetime length was defined as the number of GC executions that the object passed.

On the relationship between size and the number of objects, the authors showed the number of objects whose size was about 32 bytes was the largest. Focusing on objects larger than 32 bytes, the number of objects generally decreased as the size of objects increases.

On the relationship between lifetime length and sizes, they found that the ratio of lifetime-zero decreased as the size increased. Similarly, the average lifetime length increased as size increased. Simply writing, small objects tended to die with short lifetimes.

We can expect that GC performance can be improved by considering these object lifetime trends.

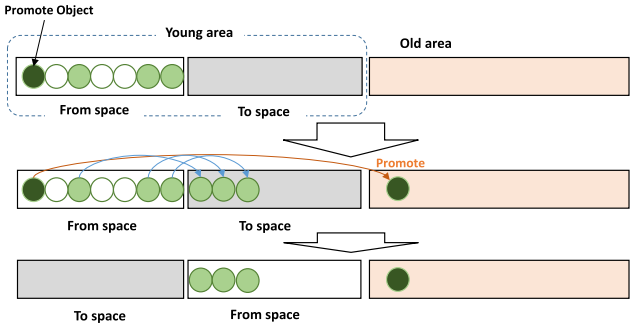

Fig. 5 Generational semi space (Default).

4. Proposed Methods

In this section, we propose two methods for reducing the heap size of applications by setting the promoting condition of GSS harder. One method does not consider the object lifetime trend. We call this naïve promotion restriction. The other considers the trend. We call this method size-aware promotion restriction. **Figures 5**, **6**, and **7** illustrate the original GSS, the naïve promotion restriction, and the size-aware promotion restriction. We expect that these two methods can avoid unsuitable promotion of objects that will die soon and that the size-aware method can avoid more

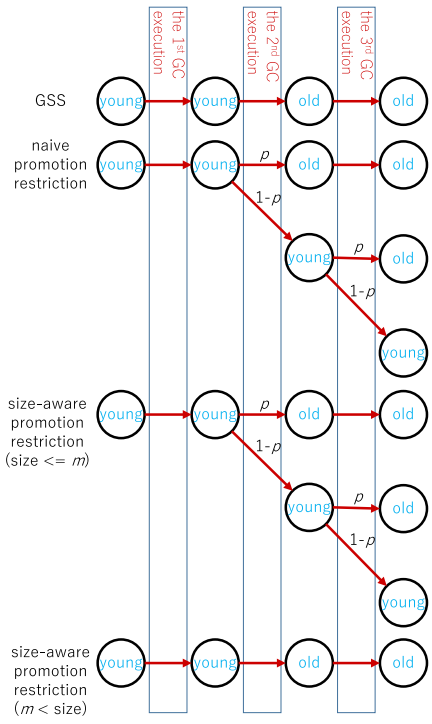

Fig. 6 Proposed methods.

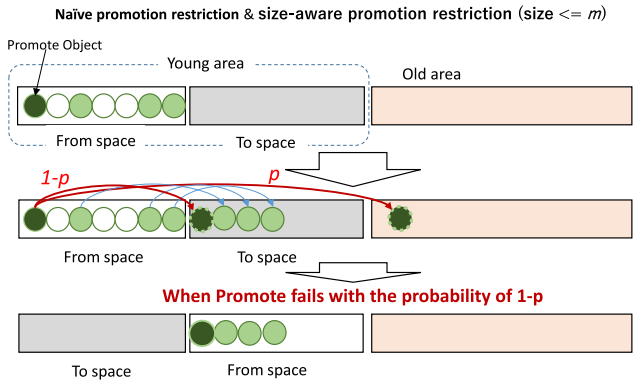

Fig. 7 Generational semi space (Proposed Methods).

```
mirror::Object* SemiSpace::MarkNonForwardedObject(mirror::Object* obj) {
  const size_t object_size = obj->SizeOf();
  size_t bytes_allocated, dummy;
  mirror::Object* forward_address = nullptr;
  bool sapr_promote = false;
  static int sapr_count = 0;
  if (generational_ && reinterpret_cast<uint8_t*>(obj) < last_gc_to_space_end_ ){
    sapr_count++;
  }
  if ((generational_ && reinterpret_cast<uint8_t*>(obj) <
last_gc_to_space_end_ )&&( object_size <= m )){
    sapr_count++;
    if(sapr_count % P_INVERSE == 0){
      sapr_promote = true;
    }
  } else if ( object_size > m ){
    sapr_promote = true;
  }
  if ((generational_ && reinterpret_cast<uint8_t*>(obj) < last_gc_to_space_end_ )
&& (sapr_promote == true)){
    // If it's allocated before the last GC (older), move
    // (pseudo-promote) it to the main free list space (as sortof an old generation.)
    forward_address = promo_dest_space_->AllocThreadUnsafe(self_, object_size,
&bytes_allocated, nullptr, &dummy);
    if (UNLIKELY(forward_address == nullptr)) {
      // If out of space, fall back to the to-space.
      forward_address = to_space_->AllocThreadUnsafe(self_, object_size,
&bytes_allocated, nullptr,&dummy);
                    .
                    .
                    .
```

Fig. 8 Implementation of size-aware promotion restriction.

effectively.

4.1 Naïve Promotion Restriction

In this subsection, we propose a method for decreasing the heap size of applications by restraining promotion without consideration of object size.

As described above, an object promotes to an old object at passing two times of GC executions in the original GSS in ART. In the proposed method, an object promotes with probability p and does not promote with $1 - p$ at passing two GC executions. If an object is decided to not to promote at a GC execution, the object can try to promote at the next GC execution.

4.2 Size-aware Promotion Restriction

In this subsection, we propose the size-aware promotion restriction. If the size of an object is less than or equals to m bytes, the method restrains promotion using p and $1 - p$ similar to the naïve method at passing two GC executions. Otherwise, the object invariably promotes like the original GSS. As described in Section 3, small objects are likely to die in the near future. The restriction of promotion of small objects will avoid a situation that an object that will die soon promotes to an old object and it will not be collected for a long time in the old space.

4.3 Implementation

Figure 8 shows a sample implementation of the size-aware promotion restriction. The method can be achieved by modifying the function `MarkNonForwardedObject()` in the source code file `art/runtime/gc/collector/semi_space.cc` in the Android operating system. This sample is based on the implementation of Android 6.0.1. The red bold italic letters were inserted into the original source code.

5. Evaluation

In this section, we evaluate our proposed methods.

5.1 Experimental Setup

We evaluated the proposed methods with actual Android applications and an actual Android device. The installed applications are Google Map and Youtube. m and p of the proposed methods are 16 bytes and 0.5, respectively. The *PromotedThreshold* was 1 MB. The used Android device is Nexus 7 (2013). Its operating system is Android 6.0.1 with our modified GC implementations. Its CPU is Snapdragon S4 Pro 1.5 GHz. Its memory size is 2 GB. In the first two experiments in Sections 5.2 and 5.3, we evaluated the proposed methods without parameter tuning. m was naively determined by the relationship between the number of objects and the object sizes in the existing work [4]. Namely, the number of objects whose sizes are between 17 and 32 bytes is the largest. In addition, the number of objects whose sizes are equal to or less than 16 bytes is larger than the number of objects whose sizes are larger than 32 bytes. Thus, setting m to 16 can separate all the objects into two groups whose sizes are similar. We chose 16 bytes as the simplest setting. Similar to m, we chose 0.5 as the simplest setting for p, which separated into the two same probabilities. Discussion on optimizations of p and m will be described in Sections 5.4 and 5.5. *PromotedThreshold* controls the positiveness to promotion. The smaller the *PromotedThreshold* is, the more aggressively the memory is saved. As the parameter is managed in MB, 1 MB is the smallest value of GSS implementation.

For evaluation, we implemented our proposed methods, the naïve and size-aware methods. In addition, we modified the GC implementation in order to output the heap size just after each GC execution and the STW time, which are managed inside ART.

5.2 Experimental Results: Google Map

We executed the Google Map application and monitored the GC behavior. We conducted searches in the application. The inputted words are the names of all the 29 stations of Yamanote line in Tokyo, Japan with the clockwise order from Tokyo. Namely, we inputted Tokyo, Yurakucho, Shinbashi, and the other stations to Kanda in Japanese. Each interval between two continuous searches is 5 seconds. The inputs of words are performed automatically by the macro function and our shell script. The measurements were performed five times.

Figure 9 shows the average heap sizes with the default GC,

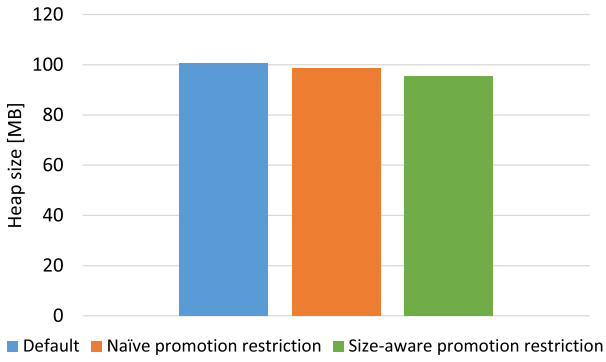

Fig. 9 Average of heap size (Google Map).

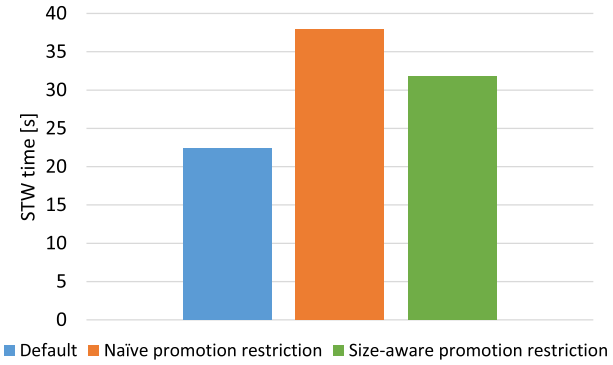

Fig. 10 Total STW time (Google Map).

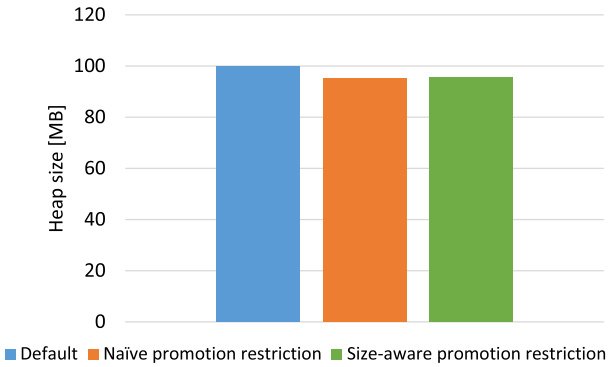

Fig. 11 Average of heap size (Youtube).

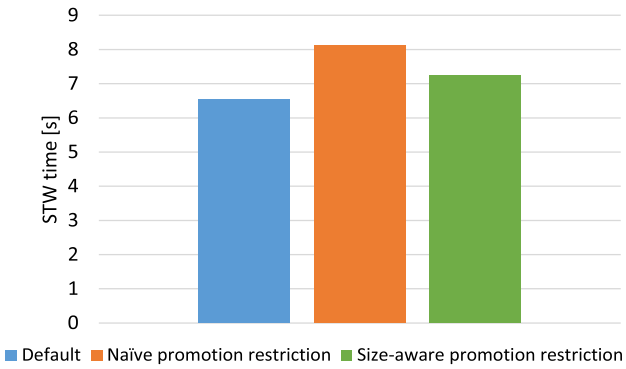

Fig. 12 Total of STW time (Youtube).

the naïve promotion restriction, and the size-aware promotion restriction. The numbers of GC executions with the default GC, the naïve promotion restriction, and the size-aware promotion restriction are 120, 120, and 121, respectively. We monitored the heap size of the application just after every GC execution and the figure depicts the average of all the monitored sizes. The results in the figure demonstrate that both of the proposed methods reduced the application heap sizes comparing the default method.

Figure 10 depicts the average of the total STW times of all the GC executions of five times measurements. The results in the figure show that both of the proposed methods increased the STW times. Comparing both methods, we can see that the increase of the size-aware method is smaller.

Figures 9 and 10 show that the application heap size can be reduced with an increase of STW time by optimizing GC. In addition, these show that consideration of object size achieved the same size of heap reduction with the smaller increase of the STW time.

5.3 Experimental Results: Youtube

In this subsection, we evaluate our methods with the Youtube application. We conducted video searches in the application and monitored the GC behaviors. For searching, we inputted single letters, which are a, b, c, and other letters to g, one by one to its search box. This experiment also was performed automatically. The measurements were performed five times.

Figure 11 shows the average heap sizes. The numbers of GC executions with the default GC, the naïve promotion restriction, and the size-aware promotion restriction are 66, 64, and 63, respectively. Similar to the results in the previous subsection, the results in this figure show that both of our proposed methods could reduce the heap size of the application comparing the default method. **Figure 12** shows the average of the total STW times of all the GC executions of these methods of five times measurements. The results in the figure indicate that the size-aware method could achieve the heap size reduction with less STW time increase.

5.4 Experimental Results: Device Dependency

In this subsection, we evaluate our proposed methods with another Android device for discussing device dependency. We performed the same experiment with Nexus 5. Its operating system is Android 6.0.1 with our modified GC implementations. Its CPU is Qualcomm Snapdragon 800 2.26 GHz. Its memory size is 2 GB. The other settings are the same as those of Section 5.1. **Figures 13** and **14** show the average heap size and total STW time with the Google Map application. **Figures 15** and **16** show those of the Youtube application. These figures indicate the results similar to those with Nexus 7. Namely, the size-aware method could save the same size of heap memories with the less increase of STW time. As a result, we can conclude that the size-aware method was effective independent of devices.

5.5 Sensitivity of Promotion Probability Parameter

This subsection presents a discussion on the sensitivity of the parameters p of the proposed methods. We measured the heap sizes and the STW times of the Google Map and Youtube applications with $p = 0.33$, 0.5, and 0.66. **Figures 17** and **18** show the heap sizes and the STW times of the Google Map application, respectively. **Figures 19** and **20** show those of the Youtube application.

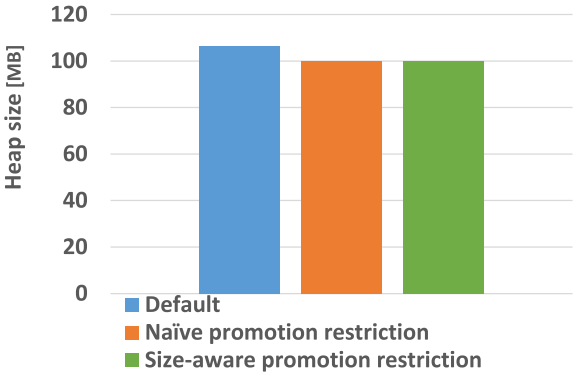

Fig. 13 Average of heap size (Nexus 5, Google Map).

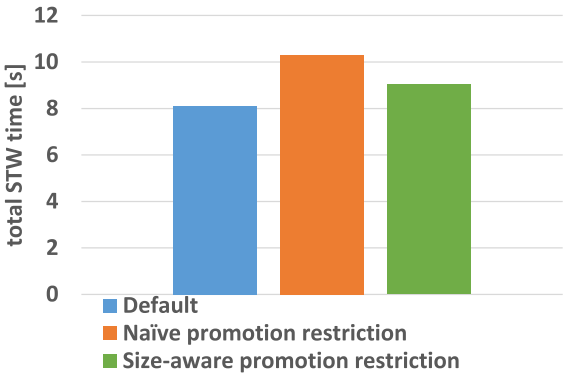

Fig. 14 Total of STW time (Nexus 5, Google Map).

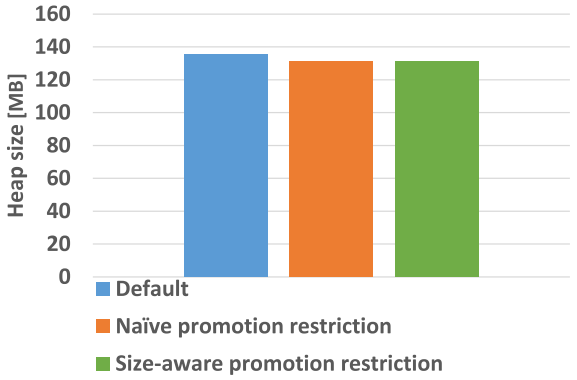

Fig. 15 Average of heap size (Nexus 5, Youtube).

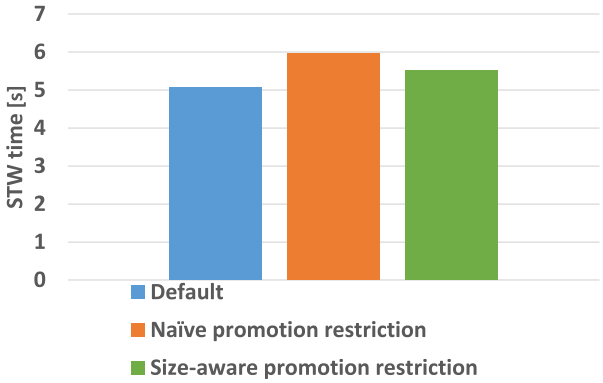

Fig. 16 Total of STW time (Nexus 5, Youtube).

Figures 17 and 18 imply that the size-aware method can provide the heap size and the STW time better than those of the default method independent on the parameter p. On the centrally, the naïve method has a slight dependency on the parameter.

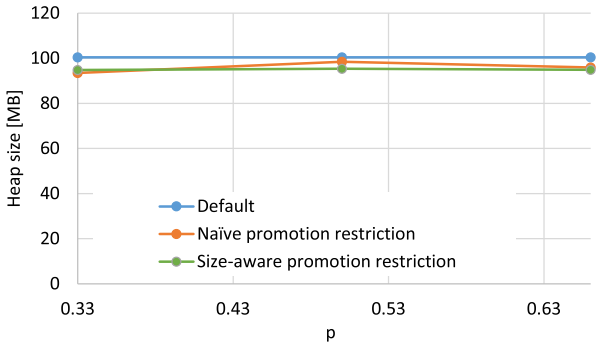

Fig. 17 Average of heap size (Google Map).

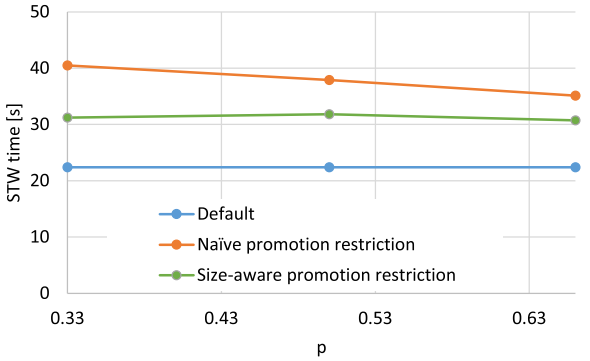

Fig. 18 Total STW time (Google Map).

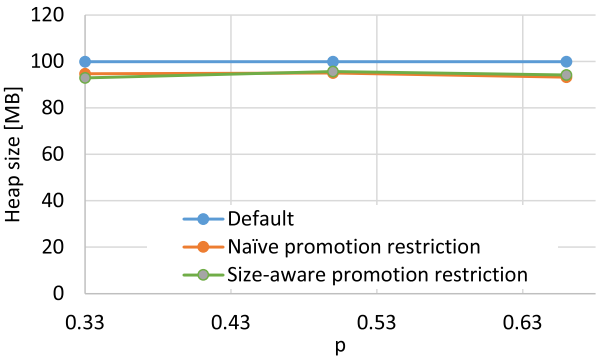

Fig. 19 Average of heap size (Youtube).

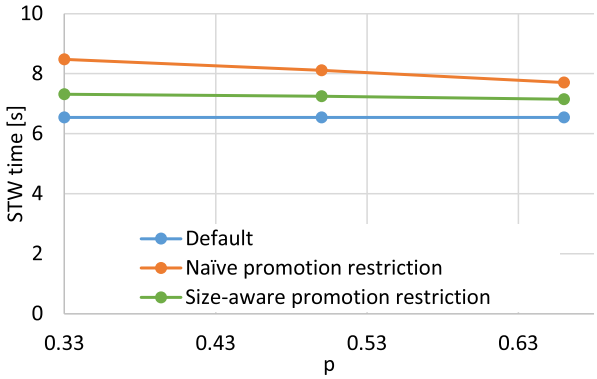

Fig. 20 Total STW time (Youtube).

Figure 19 shows that the heap size is not dependent on p. Figure 20 shows that the STW time of the size-aware method is not dependent on the parameter. As a result, we can say that the size-aware was not sensitive to the parameter p and could achieve the reduction of the heap size and small performance decline without tuning the parameter p. The naïve method was slightly dependent

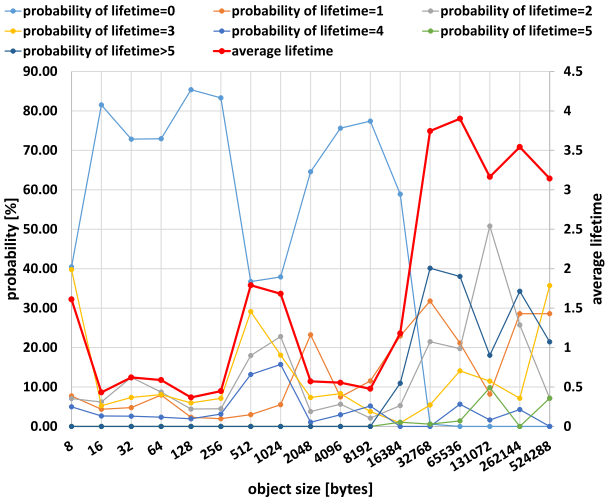

Fig. 21 Sizes of objects and their lifetimes.

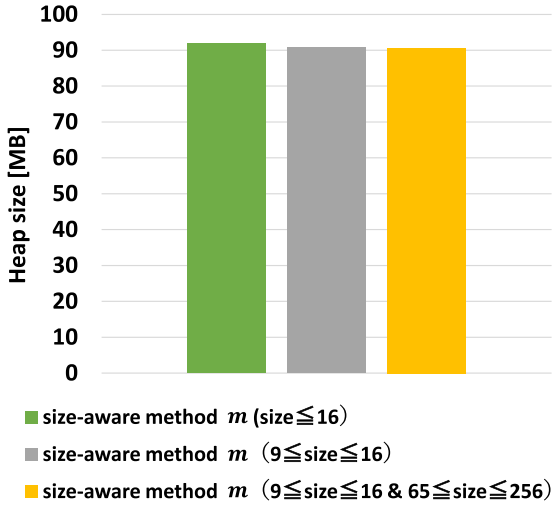

Fig. 22 Average of heap size (parameter m changing).

on the parameter.

From these results, we can conclude that the size-aware method can reduce the heap size with small performance decline independent of the parameter. In other words, a simple setting such as $p = 0.5$ is also suitable.

5.6 Tuning of Object Size Parameter

In this subsection, we discuss tuning of the parameter m. This parameter was determined based on the information of popular applications [5] in Section 5.1. However, this is desired to be determined according to the relationship between the sizes of objects and their lifetimes in the target application.

We observed object creations and GC behaviors using ART monitor [4] in the Google map application and revealed the relationship between the sizes of objects and their lifetimes. **Figure 21** depicts the relationship. The label "probability of lifetime=N" shows the probability that the object's lifetime is N. The results in the figure indicate that the lifetimes of objects whose sizes are in 9–16 or 65–256 are short. From these result, we can expect that objects whose sizes are in these ranges should not be promoted because they probably die soon.

We evaluated the performance with this condition. Namely, an object is promoted with the probability p if its size is in 9–16 or 65–256. The averages of heap sizes of five times measurements without and with the tuning based on the monitoring are depicted in **Fig. 22**. Their average STW times are depicted in **Fig. 23**. These figures show that increase of the STW time can be saved more with the similar decrease of the heap size by optimizing m based on monitoring.

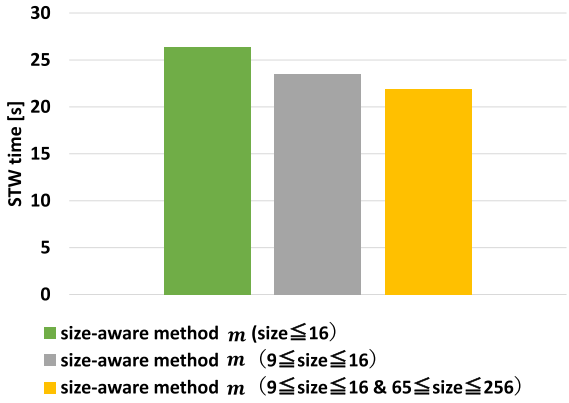

Fig. 23 Total of STW time (parameter m changing).

From the results without and with the tuning of m, we can conclude as follows. The size-aware method can save the increase of the STW time without tuning. The method can save the size more effectively by the tuning of m based on monitoring.

6. Discussion

6.1 Increase of STW

Our evaluations in the previous section have shown that the proposed methods have increased the STW time in some cases. The size-aware method restricts the promotion based on the trend that the probability of lifetime zero of a small object is high. However, not all of the small objects die in the next GC execution. We expect that the increase of STW time is small if many objects die until the next GC execution. If some objects survive, the objects cause the marks and copies of them in the next GC execution. Naturally, these increase the load of the GC and the STW time.

We argue that improvement of the accuracy of the estimation of object lifetime relieves increase of STW time. Figure 9 also supports this hypothesis. Namely, the naive method, whose accuracy is comparatively low, increased the STW more. The works of Refs. [4], [5] showed the trends between life and other features. The work of [21] showed the relationship between the time of creation and lifetime. We expect improving accuracy by taking account of these trends can reduce STW time.

6.2 Tuning GC Algorithm

In this subsection, we present a discussion on tuning GC algorithm. Some Android devices have a small size of memory. In addition, some applications consume large size of memory. In such cases, invoking many applications results in frequent process terminations by Low Memory Killer. This causes severe decline of user experience as described.

We think tuning GC based on the relationship between STW times and increases of memory sizes can improve user experi-

ences. In a case of an application that consumes large memory and is not interactive, a GC that actively saves memory size is suitable. In a case of an application that requires high interactivity, a GC that gives higher priority to STW time is suitable. For example, a web browser is an application of the former case. Entertainment application, such as games and video players, are applications of the latter case.

7. Related Work

In this section, we introduce works related to GC and memory management in the Android operating system.

For improving real-time processing in Java virtual machine, an incremental GC [12], a GC supporting snapshot [13], and several works [14], [15], [16] were published. For concurrent GC processing, works of Refs. [17], [18], [19], [20] were proposed.

The following works are on GC of Dalvik virtual machine of Android. The work of Ref. [21] investigated the relationship among the number of objects, the frequency of modification of references, and the application performance. That demonstrated that increases in the number of objects and the frequency of modifications of references increased the STW time. The work of [22] proposed to give the real-time priority to the target applications in CPU scheduling in order to apply the Android operating system in embedded systems. However, GC cannot avoid having an effect on performance even with this method. These works improved GC performances and real-time processing, but these do not consider the trend on object lifetime.

The following works are on performance improvement of Dalvik VM [23], [24]. Kawamura et al. showed that the recursively marking objects takes a long time and the reason is marking the already marked objects repeatedly. They then proposed to create a table that manages marked objects and check this table at following references to reduce redundant mark time. In the work of Ref. [24], a method for reducing STW time of GC in cases of an application with frequent reference updates was proposed. The concurrent mark and sweep (CMS) GC concurrently executes marks without stopping the threads of applications. After this concurrent mark, the GC checks the modified references during the mark and executes mark again, called re-mark, with stopping all the threads. This concurrent processing remarkably decreases the STW time because the time to mark only the modified objects is much less than that for all the objects. However, the STW time is still large in cases of the application heavily updates references. The method proposed in the work executes also the re-mark without stopping and adds a re-re-mark phase for addressing the inconsistent caused by the concurrent re-mark. The approaches of these works and this work are fundamentally different. In addition, these methods are not exclusive. Thus, we think that applying these methods to complete each other is desirable.

Hamanaka et al. proposed to choose a GC algorithm in ART based on the application state [25]. The authors evaluated the performances of CMS GC and SS GC in ART and explained their cons and pros. That is, CMS and SS are more suitable in aspects of application performance and memory saving, respectively. They then proposed to choose CMS and SS in cases of the application is in the foreground and background states, respectively. They argued that the proposed method could reduce application memory size without performance decline. In addition, they stated that the reduction resulted in a decrease in the number of process terminations by the Low Memory Killer.

In the work of Ref. [26], a method for choosing a process to terminate by Low Memory Killer was proposed. The method monitors the invocation of applications and avoids terminating processes that will be probably reused again in the near future based on LRU [26]. Their evaluation using the practical application invocation history of actual users demonstrated that their method could improve the user experience. However, unlike our method in this paper, their method does not reduce the memory consumption.

In works of Refs. [2], [6], we discussed methods for improving generational GC in ART and this paper is based on these works. However, the discussion in the work of Ref. [6] did not consider the relationship between objects sizes and lifetimes. The discussion in the work of Ref. [2] considered the relationship, but the method was not evaluated profoundly. That was evaluated only with an application. In addition, no discussion on the parameter was presented, unlike this paper.

8. Conclusion

In this paper, we have proposed methods for improving a generational GC algorithm, which is called GSS, in ART. The proposed method that is aware of the lifetime trend, takes the feature of smartphone applications into account. Our evaluation has demonstrated that the feature aware method has been able to reduce the memory consumption with less performance decline than the naïve method that does not take the feature account.

For future work, we plan to evaluate our methods with a variety of applications and discuss a method for reducing the STW time.

Acknowledgments This work was supported by JST CREST Grant Number JPMJCR1503, Japan. This work was supported by JSPS KAKENHI Grant Numbers 26730040, 15H02696, 17K00109.

References

[1] Smartphone OS Market Share, 2017 Q1, available from ⟨https://www.idc.com/promo/smartphone-market-share/os⟩.

[2] Mori, R., Oguchi, M. and Yamaguchi, S.: Memory Consumption Saving by Optimization of Promotion Condition of Generational GC in Android, *Proc. 2017 IEEE 6th Global Conference on Consumer Electronics* (*GCCE*) (2017).

[3] Appel, A.W.: Simple generational garbage collection and fast allocation, *Softw. Pract. Exper.*, Vol.19, No.2, pp.171–183, DOI: http://dx.doi.org/10.1002/spe.4380190206 (1989).

[4] Hamanaka, S., Kurihara, S., Fukuda, S., Oguchi, M. and Yamaguchi, S.: A Study on Object Lifetime in GC of Android Applications, *Proc. CANDAR 2016* (2016).

[5] Hamanaka, S., Kurihara, S., Fukuda, S., Mori, R., Oguchi, M. and Yamaguchi, S.: Object Lifetime Trend of Modern Android Applications for GC Performance Improvement, *Proc. 11th International Conference on Ubiquitous Information Management and Communication* (*IMCOM '17*), ACM, Article 85, DOI: https://doi.org/10.1145/3022227.3022311 (2017).

[6] Mori, R., Hamanaka, S., Oguchi, M. and Yamaguchi, S.: A study on promotion of generational GC in ART, *2017 IEEE International Conference on Consumer Electronics - Taiwan* (*ICCE-TW*), pp.377–378, DOI: 10.1109/ICCE-China.2017.7991153 (2017).

[7] Blackburn, S.M., Cheng, P. and McKinley, K.S.: Myth and realities: The performance impact of garbage collection, *SIGMETRICS '04/Performance '04 Proc. Joint International Conference on Measurement*

and Modeling of Computer Systems, pp.25–36 (2004).
[8] Wilson, P.R.: Uniprocessor Garbage Collection Techniques, *IWMM '92 Proc. International Workshop on Memory Management*, pp.1–42 (1992).
[9] McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM*, Vol.3, No.4, pp.184–195 (1960).
[10] Collins, G.E.: A method for overlapping and erasure of lists, *Comm. ACM*, Vol.3, No.12, pp.655–657 (1960).
[11] Appel, A.W.: Garbage collection can be faster than stack allocation, *Inf. Proc. Lett.*, Vol.25, No.4, pp.275–279 (1987).
[12] Yuasa, T.: Real-time garbage collection on general-purpose machines, *Journal of Systems and Software*, Vol.11, No.3, pp.181–198, DOI: http://dx.doi.org/10.1016/0164-1212(90)90084-Y (1990).
[13] Endo, T., Tanaka, Y., Maeda, A. and Yamaguchi, Y.: A Real-Time Garbage Collection for Java using Snapshot Algorithm, *IEICE Technical Report*, Vol.701, No.102, (CPSY2002 105-109), pp.7–12 (2002). (in Japanese)
[14] Henry, G. and Baker, Jr.: List processing in real time on a serial computer, *Comm. ACM*, Vol.21, No.4, pp.280–294, DOI: http://dx.doi.org/10.1145/359460.359470 (1978).
[15] Baker, H.G.: The Treadmill: Real-Time Garbage Collection Without Motion Sickness, ACM SIGPLAN Notices, Vol.27, No.3, pp.66–70 (1992).
[16] Lieberman, H. and Hewitt, C.E.: A Real-Time Garbage Collector based on the Lifetimes of Objects, *Comm. ACM*, Vol.26, No.6, pp.419–429 (1983).
[17] Boehm, H., Demers, A.J. and Shenker, S.: Mostly Parallel Garbage Collection, *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp.157–164, Toronto, Canada (1991).
[18] O'Toole, J., Nettles, S. and Gi_ord, D.: Concurrent compacting garbage collection of a persistent heap, *Proc. 14th ACM Symposium on Operating Systems Principles*, pp.161–174, Asheville, NC (USA) (1993).
[19] Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S. and Ste_ens, E.F.M.: On-the-fly garbage collection: An exercise in cooperation, *CACM*, Vol.21, No.11, pp.966–975 (1978).
[20] Doligez, D. and Leroy, X.: A concurrent, generational garbage collector for a multithreaded implementation of ML, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.113–123, ACM (1993).
[21] Mori, R., Hamanaka, S., Oguchi, M. and Yamaguchi, S.: A Study on Performance Improvement of Android Generational GC by Optimizing its Promote Condition, *The 79th National Convention of IPSJ*, 3G-01 (2017). (in Japanese)
[22] Higashiyama, T., Masuda, H. and Ochiai, S.: Evaluation of Android realtime performance, *The 75th National Convention of IPSJ*, pp.35–36 (2017). (in Japanese)
[23] Kawamura, S. and Tsumura, T.: Hardware Supported Marking for Common Garbage Collections, *Proc. 2016 4th International Symposium on Computing and Networking* (*CANDAR*), pp.381–387, DOI: 10.1109/CANDAR.2016.0073 (2016).
[24] Nagata, K., Nakamura, Y., Nomura, S. and Yamaguchi, S.: A Study on Shortening STW Time of Concurrent GC of Dalvik VM, *SIG Technical Reports*, 2014-CDS-9, Vol.19, pp.1–5 (2014). (in Japanese)
[25] Hamanaka, S., Kurihara, S., Fukuda, S., Oguchi, M. and Yamaguchi, S.: Application State Aware GC Selection Optimization in Android, *Proc. 2016 IEEE International Conference on Consumer Electronics-Taiwan* (*ICCE-TW*), Nantou, pp.1–2, DOI: 10.1109/ICCE-TW.2016.7520974 (2016).
[26] Nomura, S., Nakamura, Y., Sakamoto, H. and Yamaguchi, S.: LRU Based Memory Termination in Android Low Memory Killer, *IPSJ Transactions CDS*, Vol.1, No.5, pp.9–19, Information Processing Society of Japan (IPSJ), (2015) (in Japanese).

Ryusuke Mori received his B.E. degree from Kogakuin University in 2017. He is currently a master course student in Electrical Engineering and Electronics, Graduate School of Engineering, Kogakuin University.

Masato Oguchi received his B.E. from Keio University, M.E. and Ph.D. from The University of Tokyo in 1990, 1992, and 1995 respectively. In 1995, he was a researcher in the National Center for science Information System (NACSIS) – currently known as National Institute of Informatics (NII). From 1996 to 2000, he was research fellow at the Institute Science, University of Technology in Germany as a viviting reaarcher in 1998–2000, he became an associate professor at the Research and Development initiative in Chuo University. He joined Ochanomizu University in 2003 as an associate professor. Sciences, Ochanomizu University. His research filed is in network computing middleware, including high performance computing as well as mobile networking. He is a member of IEEE, ACM, IEICE, and IPSJ.

Saneyasu Yamaguchi received his Engineering Doctor's degree (Ph.D.) at The University of Tokyo in 2002. During 2002–2006, he stayed in Institute od Industrial Science, The University of Tokyo to study I/O processing. He now with Kogakuin University. Currently his researches focus on operating systems, virtualized systems, and storage system. He is a member of IPSJ, IEEE, and IEICE.