IPSJ Transactions on Programming Vol.5 No.1 1–8 (Mar. 2012)

Regular Paper

Improvements of Recovery from Marking Stack Overflow in Mark Sweep Garbage Collection

Tomoharu Ugawa[1,a)] Hideya Iwasaki[1] Taiichi Yuasa[2]

Received: June 29, 2011, Accepted: November 8, 2011

Abstract: Mark sweep garbage collection (GC) is usually implemented using a mark stack for a depth first search that marks all objects reachable from the root set. However, the required size of the mark stack depends on the application, and its upper bound is proportional to the size of the heap. It is not acceptable in most systems to reserve memory for such a large mark stack. To avoid unacceptable memory overhead, some systems limit the size of the mark stack. If the mark stack overflows, the system scans the entire heap to find objects that could not be pushed due to overflow and traverses their children. Since the scanning takes a long time, this technique is inefficient for applications that are likely to cause overflows. In this research, we propose a technique to record rough locations of objects that failed to be pushed so that they can be found without scanning the entire heap. We use a technique similar to the card table of mostly concurrent GC to record rough locations.

Keywords: garbage collection, embedded system, Android

1. Introduction

Mark sweep garbage collection (GC) follows pointer links from the root set and marks reachable objects. This traversal is usually implemented using a stack, which is called a mark stack. Traversing all the reachable objects may possibly require a very deep mark stack, depending on the data structures used by the application. Although GC should cope with any application, applications requiring deep mark stacks are rare. It is therefore unrealistic to reserve a large mark stack area to accommodate these extreme cases.

Our research goal is to improve GC on Dalvik virtual machine (VM), which is a Java VM used in the Android smart phone platform. Dalvik VM performs concurrent mark sweep GC to prevent the application from pausing for long periods while GC is in progress, thereby providing smart phone users with a comfortable operating experience. The current version of Dalvik VM [*1] depends on Linux demand paging. Every time GC is performed, it uses `mmap` to reserve a mark stack of a safe size calculated from the size of the heap in use, but only part of the reserved space that is actually needed for marking is used. However, Android has recently been put to use not only in smart phones but also in embedded devices that may not have much spare physical memory. It is therefore desirable to limit the size of the mark stack.

Mark sweep GC that uses a small fixed-size mark stack have been studied for a long time now [1]. Many systems deal with mark stack overflows as follows. When the mark stack is full, they mark traversed objects but leave them unpushed. Then, at a later stage, they follow pointer links again from all the objects that have been marked. However, it takes time to search the entire heap for marked objects, and an object whose referents have all been marked still has to be checked to see whether or not it has any pointers to unmarked objects, resulting in a large overhead when the mark stack has overflowed. Pointer reversal [1] is a technique that traverses objects without using a mark stack at all. Instead, it follows pointer links while reversing the direction of the pointers so that it can backtrack using the reversed pointers. However, in concurrent GC, mutators may read the reversed pointers while GC is still in progress, so the technique cannot be used. Furthermore, using this technique is slower than using a mark stack.

In this paper, we propose a method whereby marking can be performed correctly with little overhead when the mark stack has overflowed, in a processing system that performs mark sweep GC using a small fixed-length mark stack. In this method, we store the rough locations of objects that were left unpushed when the mark stack overflowed. In this way, we can find these unpushed objects in the mark stack without scanning the entire heap.

Dalvik VM uses *mostly concurrent GC* [2] to perform GC in parallel with mutators. In mostly concurrent GC, a write barrier acts when a mutator writes to an object so that the objects reachable from this object can be marked when GC scans this object again. In this case, instead of individually recording the objects that are written to, the heap is partitioned into small fixed-size chunks called "cards," and card marking is used to record whether or not a write has been performed in each card. When a card has been written to, a virtual dirty bit corresponding to the card is set. In a GC mechanism configured like this, we can efficiently im-

[1] Graduate School of Informatics and Engineering, The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[2] Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
[a)] ugawa@cs.uec.ac.jp

[*1] As of June 29, 2011, the source code of Android 2.3.4 is publicly available.

plement our proposed method by "piggy-backing" it on the dirty bits. That is, when the mark stack has overflowed, the dirty bits are set for cards in which unpushed objects reside. By doing so, it is then only necessary to retrace the links of marked objects in cards where the dirty bit is set, instead of having to work through the entire heap.

In Section 2 of this paper, we introduce the existing research that has been done to address mark stack overflows. In Section 3, we discuss our method for dealing with mark stack overflows, and in Section 4, we present a detailed discussion of how this method was implemented in Dalvik. The results of performance evaluations are shown in Section 5, and in Section 6 we conclude with a summary.

2. Background

2.1 Mark Stack Overflow

Mark sweep GC marks objects that can still be referenced by a mutator, and subsequently the memory occupied by unmarked objects is reclaimed by scanning the entire heap. These phases are called the "mark phase" and the "sweep phase," respectively.

In the mark phase, GC marks *live objects*, i.e., those of objects that are directly or indirectly referenced from the areas that mutators can reference directly, such as stacks and global variables (called the "root set"). To find the live objects, the pointer links are followed while applying marks, using the root set as a starting point. At this time, the mark stack is generally used to follow links recursively.

In **Fig. 1**, `markPhase` is the pseudocode of the mark phase. To simplify the discussion, the examples in this paper assume that all the objects consist of two pointer fields. First, `rootInsertion` marks objects that are directly referenced from the root set and pushes them onto the mark stack. This process is called root insertion. Next, object `P` is popped from the mark stack, and `scanObject` scans `P` to search its referent objects. If object `P`'s referents `P[0]` and `P[1]` are unmarked, they are marked and pushed onto the mark stack. This process is repeated until the mark stack becomes empty.

The size of the mark stack required for this process depends on the data structures used by the application. For example, if the application uses an array of objects, then all of these objects will be pushed onto the mark stack simultaneously. **Table 1** shows the results of an investigation into the actual sizes of mark stacks required by various applications. Here, **system_server** is an Android daemon process, **kXML** is a program that uses a lightweight XML parser library called kXML to read the DOM tree from an XML file that stores the Android settings, and **Hashtable** is an artificial program that uses `java.util.Hashtable` to record 100,000 Integer objects in a hash table. To give an indication of the scale of these applications, Table 1 also shows the total amount of memory occupied by live objects when GC has finished. From this table, it can be seen that the required size of the mark stack varies among applications, and in devices with strict memory constraints there may be cases where this memory requirement is too large to ignore.

```
rootInsertion() {
  foreach(P in RootSet) {
    P.markbit = MARKED;
    markStack.push(P); }}

scanObject(P) {
  for (i = 0; i < 2; i++) {
    if (P[i].markbit != MARKED) {
      P[i].markbit = MARKED;
      markStack.push(P[i]); }}}

markPhase() {
  rootInsertion();
  while (!markStack.isEmpty()) {
    P = markStack.pop();
    scanObject(P); }}
```

Fig. 1 Mark phase.

Although GC must be able to cope with whatever data structures are used in an application, applications that require deep mark stacks are rare in practice. Therefore, it is impractical to prepare a large mark stack based on a worst-case scenario. Thus, most systems use a small fixed-size mark stack to follow pointer links.

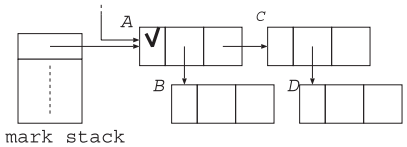
With a small fixed-size stack, some applications may cause the mark stack to overflow. **Figure 2** (a) shows a mark stack before and after an overflow. The mark stack area is completely used up, and a pointer to object *A* resides on top. When object *A* is popped off the stack and then an attempt is made to push objects *B* and *C* which are referents of object *A*, object *B* can be pushed but object *C* cannot, as shown in Fig. 2 (b). Here, the object that could not be pushed is indicated by a bold outline. Unless something is done, object *D* (which can only be reached through object *C*) is left unmarked, and is erroneously collected despite still being live.

A method commonly used to prevent this sort of erroneous collection involves marking traversed objects but leaving them unpushed when the mark stack is full, and after a simple traversal has been completed, performing a *recovery process* whereby all the marked objects are used as a root set to retrace the pointer links [3], [4]. Since the mark stack is also liable to overflow during the recovery process, the recovery process is repeated until it completes without mark stack overflows. In this method, since the recovery process takes time to perform, a large overhead penalty is incurred when the mark stack overflows.

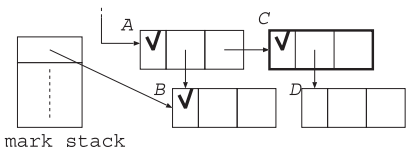
Figure 3 shows the pseudocode for the mark phase using a

Table 1 Size of mark stack used for GC.

	Application	Mark Stack (KB)	Live Objects (KB)	A/B (%)
1	**system_server**	40.4	4,030	1.0
2	**kXML**	3.0	268	1.1
3	**Hashtable**	399.1	5,905	6.8



(a) Before an Overflow.



(b) After an Overflow.

Fig. 2 Mark stack overflow.

small fixed-size mark stack. The occurrence of a mark stack overflow is stored in `overflowFlag`. Once the simple traversal has been completed, `recovery` is called to perform the recovery process if a mark stack overflow has occurred. The `recovery` procedure examines the heap in order of addresses to look for marked objects. In the example of this paper, since all the objects are assumed to consist of two pointer fields, the object size is also fixed (`OBJECT_SIZE`). It is also assumed that there are no mark bits set in unused regions. When a marked object is found, this object is scanned, and its unmarked referent objects are pushed onto the mark stack. After examining the entire heap, objects reachable from the objects pushed onto the mark stack are marked.

2.2 Related Work

Using a small fixed-size mark stack, many systems perform a recovery process according to the method discussed in Section 2.1 when the mark stack overflows [3], [4]. Knuth [5] mentions a method where, when using a small fixed-size mark stack, the mark stack is overwritten from the bottom up if the mark stack overflows. A recovery process is still required in this case.

The pointer reversal method [6] reverses the direction of pointers as they are followed, and uses these reversed pointers to backtrack. If the pointer reversal method is used, then there is no need for a mark stack, but the pointers within objects are temporarily rewritten. As a result, this method cannot be used for GC that runs concurrently with mutators. Also, since the pointer reversal method is slower than methods that use a mark stack, it is also not an option in cases where performance is required even in systems that do not perform concurrent GC.

There are also several other known methods that reduce the possibility of a mark stack overflow by reducing the number of objects pushed onto the mark stack. In one method for example, large arrays are handled virtually as lists of smaller arrays, thereby preventing a large number of objects from being pushed onto the mark stack simultaneously [4].

In environments where memory can be obtained from an underlying system such as an operating system, the mark stack can be split into chunks so that if the mark stack overflows then a new chunk can be topped up in its place [4], [7].

```
scanObject(P) {
  for (i = 0; i < 2; i++) {
    if (P[i].markbit != MARKED) {
      P[i].markbit = MARKED;
      if (!markStack.isFull()) {
        markStack.push(P[i]); }
      else {
        overflowFlag = true; }}}}

markPhase() {
  overflowFlag = false;
  rootInsertion();
  while (!markStack.isEmpty()) {
    P = markStack.pop();
    scanObject(P); }}
  while (overflowFlag) {
    recovery(); }}

recovery() {
  overflowFlag = false;
  for (P = heap.start; P < heap.end; P += OBJECT_SIZE) {
    if (P.markbit == MARKED) {
      scanObject(P); }}
  while (!markStack.isEmpty()) {
    P = markStack.pop();
    scanObject(); }}
```

Fig. 3 Mark phase using a small mark stack.

3. Recovery from Mark Stack Overflow by Using a Card Table

In this paper, we propose a method that is compatible with card marking used in mostly concurrent GC, and reduces the recovery processing overhead.

3.1 Mostly Concurrent GC

Mostly concurrent GC [2] is a concurrent GC based on incremental updates. In GC with incremental updates, a write barrier is used to detect objects that have been written to by a mutator during GC, and these objects are used as the starting points from which the pointer links are retraced. In mostly concurrent GC, instead of individually recording every object that is written, card marking [8] is used to keep records in units of a fixed-size area, and the objects that can be reached from written objects are all marked together after pausing all mutators.

Card marking is a method where the dirty bits of pages used in virtual memory are implemented in software. In card marking, the heap is virtually partitioned into fixed-size areas (called "cards"), and a dirty bit is provided for each card. The dirty bits are stored in a bit map called a "card table" that exists outside of the heap. When a pointer is written to an object, a write barrier sets the dirty bit of the card in which the object resides. This indicates that there is an object in this card from which pointer links need to be followed again.

`markPhase` in **Fig. 4** is pseudocode for the mark phase of mostly concurrent GC. In Fig. 4, the code for stopping and resuming the mutators is omitted. The definition of `scanObject` is the same as in Fig. 1. The GC cycle of mostly concurrent GC consists of the following phases. (Mark stack overflows are not considered, since our main purpose here is to provide a general outline of mostly concurrent GC.)

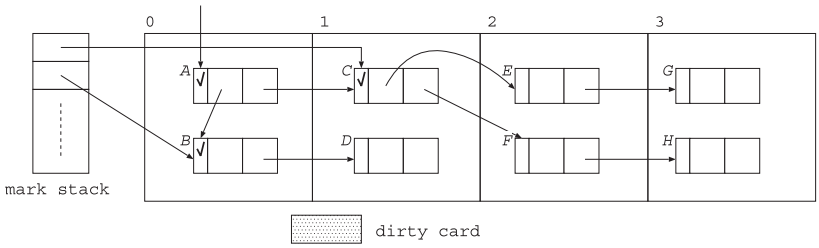
Root insertion phase: When a GC cycle starts, all the mutators are stopped and root insertion is performed.

Concurrent mark phase: GC allows the mutator threads to resume. The GC thread recursively follows pointer links from objects pushed onto the mark stack, and marks all the objects it reaches. When a mutator writes to a pointer during

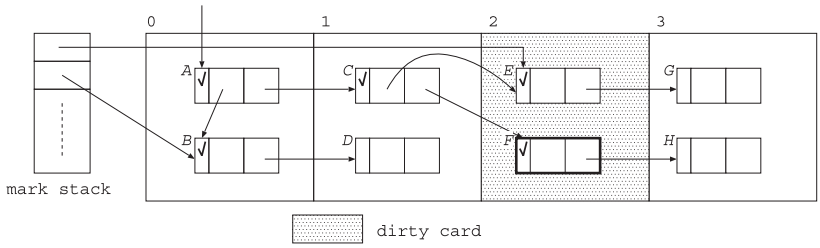
```
markPhase() {
  rootInsertion();
  // concurrent mark phase
  while (!markStack.isEmpty()) {
    P = markStack.pop();
    scanObject(P); }
  // stop-the-world mark phase
  cardCleaning(); }

cardCleaning() {
  rootInsertion();
  for (i = 0; i < cardTable.length; i++) {
    card = cardTable[i];
    if (card.dirtybit == DIRTY) {
      for (P = card.start; P < card.end; P += OBJECT_SIZE) {
        if (!isFree(P) && P.markbit == MARKED) {
          scanObject(P); }}}}
  while (!markStack.isEmpty()) {
    P = markStack.pop();
    scanObject(); }}
```

Fig. 4 Mostly concurrent GC.



(a) Before an Overflow.



(b) After an Overflow.

Fig. 5 Recovery from an mark stack overflow using card table.

this process, the dirty bit is set.

Stop-the-world mark phase: When the mark stack becomes empty, all the mutators are stopped. At this point, there may be live objects that have yet to be marked since mutators may have written pointers into objects and/or the root set. Consequently, the pointer links are retraced starting from the root set and all the objects that reside in cards whose dirty bit is set. This process is called *card cleaning*. Card cleaning is performed by `cardCleaning` in Fig. 4.

Concurrent sweep phase: GC allows the mutators to resume and reclaims garbage concurrently with the mutators.

3.2 Basic Idea

The recovery process indicated by `recovery` in Fig. 3 has poor efficiency because of the following points:

- The entire heap must be examined to find marked objects.
- It scans all the marked objects and searches for unmarked referents of these objects.

We propose a method that reduces the recovery process overheads by "piggy-backing" on card marking of mostly concurrent GC. The basic idea of this method is as follows.

Since it is possible that objects that could not be pushed onto the mark stack may have referent objects that have not yet been marked, it will subsequently be necessary to mark the objects that can be reached from these objects. The need to subsequently mark objects that can be reached from these objects is common to all objects that are written to by mutators during GC, and can thus be handled in the same way. That is, if the mark stack overflows during the concurrent mark phase, the dirty bits are set for cards containing objects that could not be pushed onto the mark stack. In this way, in card cleaning with the stop-the-world mark phase, objects that can be reached from objects that could not be pushed onto the mark stack are also marked.

Figure 5 (a) shows the mark stack just before an overflow occurs. Here, object *C* is popped from the stack and an attempt is made to push objects *E* and *F*, which are referents of object *C*, onto the mark stack. Since there is only one word of space on the mark stack, *E* can be pushed onto the stack but *F* cannot. Thus, as shown in Fig. 5 (b), the dirty bit of card No.2 in which *F* resides is set.

It is possible that the mark stack may overflow even in the stop-the-world mark phase since GC uses the mark stack to follow pointer links in this phase as well. The stop-the-world mark phase must therefore be modified as follows.

- To enable the recording of mark stack overflows during card cleaning, for each dirty card, the dirty bit is reset immediately before searching for marked objects in this card.
- Card cleaning is repeated until there are no more dirty cards.

3.3 Variations

Several variations can be considered in the method of Section 3.2, and it is not obvious which variation is better. In this section we show two variations, and in Section 5 we evaluate their performance.

3.3.1 Push Suppression

Marked objects that reside in dirty cards are scanned at the next card cleaning operation. In the method mentioned in Section 3.2, objects are pushed onto the mark stack regardless of which cards they reside in, and are subsequently scanned. Objects that reside in dirty cards are therefore scanned twice, since they are also scanned in the next card cleaning. To prevent this, the objects are first checked to see if they reside in dirty cards, and if so, they are not pushed onto the mark stack. This not only prevents the same object from being scanned twice, but can also reduce the frequency of mark stack overflows.

Barabash et al. [9] have proposed a method where, in order to prevent double scanning of objects in dirty cards in mostly concurrent GC, objects popped from the mark stack are checked to see if they reside in a dirty card before they are scanned. In contrast, the method proposed in this section checks whether or not an object resides in a dirty card before it is pushed onto the mark stack.

3.3.2 Clearing the Mark Stack

When the mark stack has overflowed, the card in which the ob-

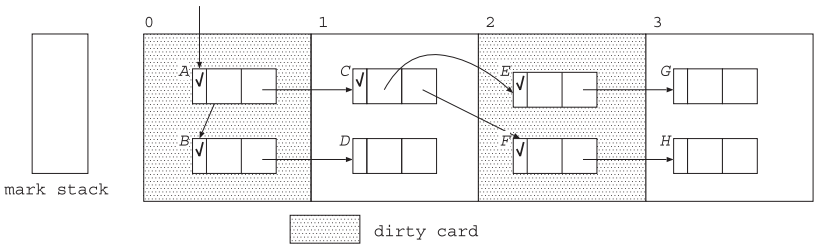


Fig. 6 Clearing mark stack.

ject that could not be pushed onto the mark stack resides becomes dirty, and is targeted by subsequent card cleaning operations. At this time, if other objects in the same card have already been pushed onto the mark stack, then these will be scanned twice.

This could be prevented by scanning the entire mark stack when a mark stack overflow has occurred, and removing from the mark stack all the objects that reside in the dirty card. However, when only a few objects are removed from the mark stack by this operation, the mark stack will most likely overflow again. Consequently, scanning the entire mark stack when the mark stack has overflowed has a large overhead.

One might therefore consider a method where all objects are removed from the mark stack, including objects placed in cards whose dirty bit is not set. This can reduce the frequency of mark stack overflows. In order to ensure that all reachable objects are eventually marked, it is necessary to set the dirty bit of all cards containing objects that have been on the mark stack. It is not obvious whether or not the benefit of avoiding double scanning outweighs the overhead penalty of making more cards dirty.

For example, in the situation shown in Fig. 5 (a), when an overflow is caused by attempting to push referent objects *E* and *F* of object *C* onto the mark stack, dirty bits are set not only in card 2, but also in card 0 which is the card of object *B* that has been pushed onto the mark stack, and *B* is removed from the mark stack, as shown in **Fig. 6**. Other objects pushed onto the mark stack are also removed in the same way, thereby clearing the mark stack.

4. Implementation

We implemented the method proposed in Section 3 in Dalvik VM. In Dalvik VM, the card size of mostly concurrent GC is 128 bytes. The original mostly concurrent GC proposed by Printezis et al. [2] is configured as a generational GC, but generational GC is not performed in Dalvik VM, which only incorporates concurrent marks using card marking.

Dalvik VM uses two bitmaps called *live bits* and *mark bits* to manage the positions and marks of objects. Since objects are aligned at 8-byte boundaries in Dalvik VM, every eight bytes on the heap correspond to 1 bit in these bitmaps. At the start address of each live object, the corresponding bit in live bits is set. Live bits enables GC to find the start address of an object in a card, which is a part of the heap, without scanning from the start address of the heap.

The *mark bits* bitmap consists of the marks for objects at the corresponding addresses. Because marks are collected together in this bitmap, GC can quickly search the entire heap for marked objects.

```
cardCleaning() {
  rootInsertion();
  for (i = 0; i < cardTable.length; i++) {
    card = cardTable[i];
    if (card.dirtybit == DIRTY) {
      for (P = card.start; P < card.end; P += OBJECT_SIZE) {
        if (!isFree(P) && P.markbit == MARKED) {
          scanObject(P);
          while (!markStack.isEmpty()) {
            P = markStack.pop();
            scanObject(); }}}}}}
```

Fig. 7 Modified card cleaning procedure.

We allocated fixed-size areas ranging from a few kilobytes to hundreds of kilobytes to the mark stack used by Dalvik VM, and we handled mark stack overflows by piggy-backing the card marking technique as shown in Section 3.2. We also implemented the two variations mentioned in Section 3.3.

Like `cardCleaning` in Fig. 4, the Dalvik VM card cleaning pushes the referents of all the marked objects onto the mark stack, and then uses the objects pushed onto the mark stack as a starting point for the recursive tracing of pointer links. Therefore, mark stack overflows can easily occur. It seems that this configuration was chosen because the mark stack was regarded as being sufficiently large. Since the size of the mark stack was limited in our implementation, we modified the procedure for card cleaning so that every time a marked object is scanned, objects reachable from the referents of this object are immediately traversed. Pseudocode for the modified card cleaning procedure is shown in **Fig. 7**. A similar change was made to the root insertion.

5. Performance Evaluation

We evaluated the performance of the implementation discussed in Section 4. For comparison, we also implemented GC based on a recovery process that searches for marked objects from the entire heap as shown in Fig. 3 without using a card table.

In the experiments, we used an Armadillo-500 FX ARM evaluation board. The experimental environment was as follows:

Android:	2.3.3 (Gingerbread)
GCC:	4.3.3 ARM EABI
Optimization:	Android SDK standard options (optimized for binary size)
CPU:	Freescale i.MX31 (ARM1136JF-S) 532 MHz
Main Memory:	128 MB DDR SDRAM

For the benchmark program, we used **SPECjbb2005**. Since some methods of class libraries of Dalvik VM have not been properly implemented, we modified **SPECjbb2005** so as not to use any problematic methods. In the experiments, the `warehouse`

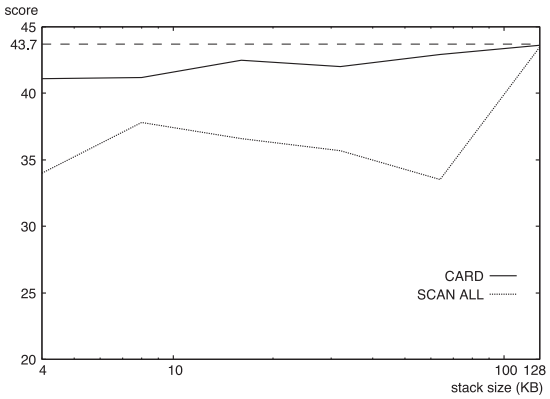


Fig. 8 Improvements by using card table.

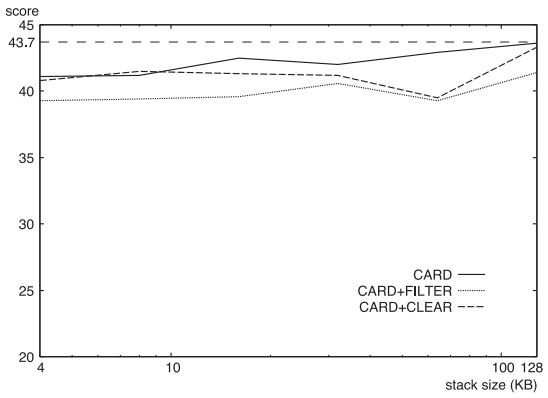


Fig. 9 Throughput of variants.

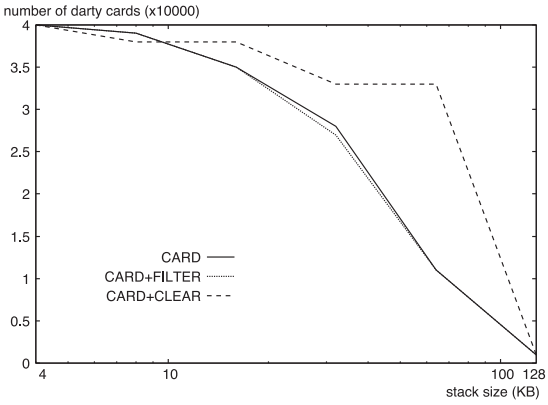


Fig. 10 Number of dirty cards in variants.

parameter of **SPECjbb2005** was fixed at 1.

The heap size was fixed at 64 MB [*2]. Since the card size of Dalvik VM is 128 bytes, the entire heap is partitioned into roughly 512,000 cards. Armadillo-500 FX is equipped with 128 MB of main memory, which is large enough for the heap of 64 MB. Also, since since the value of `warehouse` was fixed at 1, the heap usage was kept between 40% and 45%.

5.1 Throughput

Figure 8 shows the benchmark results for VM where the recovery process is performed by scanning only dirty cards (labeled "CARD"), and for VM where the recovery process scans the entire heap (labeled "SCAN ALL"). The vertical axis shows the **SPECjbb2005** score (the greater the better). The score for original VM, where a mark stack large enough not to overflow was secured with `mmap`, was 43.7. When using a 128 KB mark stack at the right-hand side of the graph, there were no mark stack overflows with either VM.

In Fig. 8, it can be seen that when using a card table so that the recovery process is only applied to dirty cards, there is hardly any reduction in throughput for smaller mark stacks. For example, when using a 4 KB mark stack, the SCAN ALL scheme suffered a 22% drop in throughput compared with the original VM with no mark stack overflows, while the throughput of the CARD scheme decreased only by 6%. Also, when using a 128 KB mark stack where no overflows occurred, the CARD scheme generated no overhead, just like the SCAN ALL scheme.

If we take a detailed look at the situation with a mark stack size of 4 KB, we can see that the mark stack overflowed twice per GC cycle in both VM schemes. In the CARD scheme, the number of cards marked as dirty was, on average, 39,644 per GC cycle. Since the SCAN ALL scheme entails searching for marked objects from all the cards every time the mark stack overflows, it is equivalent to checking roughly 1,024,000 cards in each GC cycle. This difference appears to be reflected in the throughput figures.

[*2] In Dalvik VM, only half the size specified by the "`-Xms`" option is allocated to the heap at start-up. Hence, we specified both the initial heap size and maximum heap size as 128 MB by using the options "`-Xms 128M -Xmx 128M`". In the execution log, we confirmed the heap size did not increased at runtime from the size of 64 MB allocated at start-up.

5.2 Variations

Figure 9 shows the performance of the variations discussed in Section 3.3. CARD+FILTER represents a VM where objects in dirty cards are not pushed onto the mark stack as discussed in Section 3.3.1, and CARD+CLEAR represents a VM where the mark stack is cleared when the mark stack overflows as discussed in Section 3.3.2. In the experimental results, the throughput of both variations is inferior to that of the CARD scheme.

A comparison of CARD and CARD+FILTER shows that the difference in throughput does not depend on the size of the mark stack. CARD+FILTER even had lower throughput with a 128 KB mark stack where no overflows occurred. This indicates that the source of the overhead is the time taken to check the dirty bit of the card in which an object resides when pushing the object onto the mark stack.

Next, we consider the reasons why lower throughput was observed with CARD+CLEAR. Except when using a 128 KB mark stack where no overflows occurred, the drop in throughput increased as the mark stack grew in size. **Figure 10** shows how many cards became dirty on average per GC cycle in each variation. As this figure shows, increasing the mark stack size causes hardly any reduction in the number of dirty cards in CARD+CLEAR. In CARD+CLEAR, as the mark stack becomes larger, more cards have their dirty bits set when the mark stack overflows. Consequently, even though the mark stack becomes larger, there is no reduction in the number of cards that become dirty, and the throughput observed with a large mark stack is worse than that of the CARD scheme.

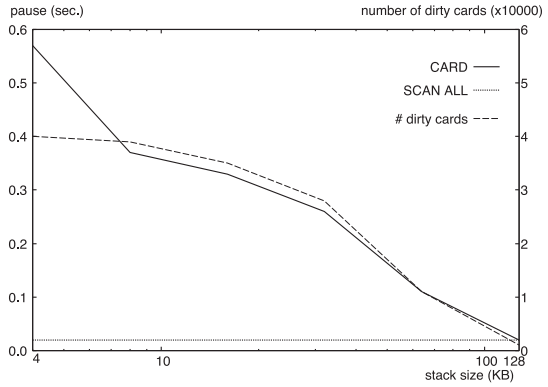


Fig. 11 Pause time and number of dirty cards.

5.3 Pause Time

In the proposed method, mark stack overflows are all dealt with in the stop-the-world mark phase. Therefore, compared with the original VM, where the proposed method is not used, it is possible that the stop-the-world mark phase may cause the pause time of mutators to increase. **Figure 11** shows the results of an investigation into pause times. In the SCAN ALL scheme, which does not use a card table for the recovery process, the recovery process is performed in the concurrent mark phase. Thus, the pause time is not different from that of the original VM. On the other hand, in the CARD scheme, card cleaning also involves the recovery process. Consequently, the pause time increases in proportion to the number of dirty cards. The pause time was very long with a mark stack size of 4 KB because two card cleaning cycles were required in this case.

Since concurrent GC is also sometimes used with the aim of reducing the stoppage of mutators by GC, this increased pause time is a problem. The method of Endo et al. [10] could be used to resolve this problem. They provided a time restriction to the stop-the-world mark phase to avoid stopping mutators for a long time in the stop-the-world mark phase in *mostly parallel GC* [11] (a method similar to mostly concurrent GC but with a hardware paging mechanism used for the write barrier). When a time-out occurs, mutators are allowed to resume even if pointers are still being followed. In this way, the process reverts to the concurrent mark phase. The concurrent mark phase and the stop-the-world mark phase are repeated until the stop-the-world mark phase finishes without timing out. Implementing this method and evaluation of its efficacy are issues for further study.

6. Conclusion

In this paper, we have proposed a low-overhead method for dealing with mark stack overflows that are liable to occur when the size of a mark stack is restricted. The proposed method is piggy-backed on top of the card marking performed by mostly concurrent GC, and sets the dirty bit of those cards in which objects that could not be pushed onto the mark stack reside, so that it is only necessary to scan the objects in the vicinity of objects that could not be pushed onto the stack.

We implemented this method in Dalvik VM and evaluated it. As a result, when using a 4 KB mark stack under conditions where a mark stack of at least 64 KB was needed to follow pointer links without causing a mark stack overflow, a 22% reduction in throughput is seen in the conventional method, but using the proposed method the drop in throughput was only 6%. Accordingly, the proposed method is effective for suppressing the drop in throughput when the mark stack overflows.

On the other hand, since the recovery process that follows a mark stack overflow is performed in the stop-the-world mark phase, the pause time was longer. It should be possible to resolve this issue by setting a time-out in the stop-the-world mark phase, but the implementation and evaluation of this technique are left as issues for further study.

Acknowledgments We would like to thank Carl Shapiro of Google Inc. and Ryo Fujii of Keio University for showing us how to modify the SPECjbb2005 benchmark software so it could run on Dalvik VM. This study was supported by a Grant-in-Aid for Scientific Research (22700026).

Reference

[1] Jones, R. and Lins, R.: *Garbage Collection*, John Wiley & Sons (1996).
[2] Printezis, T. and Detlefs, D.: A generational mostly-concurrent garbage collector, *Proc. ISMM '00*, pp.143–154 (2000).
[3] Sun mycrosystems, Inc: The K Virtual Machine – White Paper (2000), available from ⟨http://java.sun.com/products/kvm/wp⟩.
[4] Boehm, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software Practice & Experience*, Vol.18, No.9, pp.807–820 (1988).
[5] Knuth, D.E.: *The art of computer programming: Fundamental algorithms*, p. 416, Addison-Wesley (1997).
[6] Schorr, H. and Waite, W.M.: An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, *Comm. ACM*, Vol.10, No.8, pp.501–506 (1967).
[7] Alpern, B. et al.: The Jalapeño virtual machine, *IBM System Journal*, Vol.39, No.1, pp.211–238 (2000).
[8] Sobalvarro, P.G.: A Lifetime-based Garbage Collector for LISP Systems on General-Pourpose Computers, Technical report, MIT (1988).
[9] Barabash, K., Ossia, Y. and Petrank, E.: Mostly Concurrent Garbage Collection Revisited, *Proc. OOPSLA '03*, pp.255–268 (2003).
[10] Endo, T. and Taura, K.: Reducing Pause Time of Conservative Collectors, *Proc. ISMM '02*, pp.12–24 (2002).
[11] Boehm, H.-J.: Mostly Parallel Garbage Collection, *Proc. PLDI '91*, pp.157–164 (1991).

Tomoharu Ugawa received his B.Eng. degree in 2000, M.Inf. degree in 2002, and Dr.Inf. degree in 2005, all from Kyoto University. He worked for a research project on real-time Java at Kyoto University from 2005 to 2008. Since 2008, he is an assistant professor at the University of Electro-Communications, Tokyo, Japan. His research interests include program languages, language processing systems, and system software.

Hideya Iwasaki is a professor at the Graduate School of Informatics and Engineering, the University of Electro-Communications in Japan. He received degrees of B.Eng. in 1985 and Dr.Eng. in 1988, both from the University of Tokyo. After working as a research associate in the Department of Mathematical Engineering in the University of Tokyo, he joined the Educational Computer Centre in the University of Tokyo as an associate professor in 1993. He later served as an associate professor at Tokyo University of Agriculture and Technology and the University of Tokyo, he joined the University of Electro-Communications and was appointed a professor in 2004. His research interests include programming language and systems, and systems software.

Taiichi Yuasa received his B.Math. degree in 1977, M.Math.Sc. degree in 1979, and D.S. degree in 1987, all from Kyoto University, Kyoto, Japan. He joined the faculty of the Research Institute for Mathematical Sciences, Kyoto University, in 1982. He is currently a Professor at Graduate School of Informatics, Kyoto University, Kyoto, Japan. His current area of interest include symbolic computation and programming language systems. Dr. Yuasa is a member of ACM, IEEE, IEICE, and Japan Society for Software Science and Technology.