Combining Card Marking with Remembered Sets: How to Save Scanning Time

Alain Azagury* Elliot K. Kolodner† Erez Petrank‡ Zvi Yehudai§

Abstract

We consider the combination of card marking with remembered sets for generational garbage collection as suggested by Hosking and Hudson [3]. When more than two generations are used, a naive implementation may cause excessive and wasteful scanning of the cards and thus increase the collection time. We offer a simple data structure and a corresponding algorithm to keep track of which cards need be scanned for which generation. We then extend these ideas for the Train Algorithm of [4]. Here, the solution is more involved, and allows tracking of which card should be scanned for which car-collection in the train.

Keywords: Garbage collection, Generational garbage collection, The train algorithm.

*E-mail: azagury@haifa.vnet.ibm.com.
†E-mail: kolodner@haifa.vnet.ibm.com.
‡E-mail: erezp@haifa.vnet.ibm.com.
§E-mail: yehudai@haifa.vnet.ibm.com.
Postal address (of all authors): Department of System Technology, IBM Haifa Research Lab, MATAM, Haifa 31905, Israel.

1 Introduction

Generational garbage collection was introduced by Lieberman and Hewitt [5] and has been accepted as an excellent solution for reducing pause times induced by garbage collection. Generational garbage collectors rely on the assumption that most objects die young. Under this assumption, it is useful to collect the garbage in the young area more frequently. The same line of reasoning can be refined to motivate using more than two generations.

Generational collectors divide the heap into (at least) two parts: the old and young generation. New objects are allocated in the young generation which is collected frequently. Young objects that survive several collections are "promoted" to the older generation. Since the young generation is kept small, most collections are fast and do not stall the application for too long. We go into more detail in Section 2.1.

1.1 Card marking and remembered sets

Since we want to be able to collect a part of the heap, e.g., the young generation, we must keep track of pointers that reference objects in the young generation from the outside (i.e., from older generations). These pointers must be treated as roots, since referenced objects may

be live objects. Scanning the full heap to discover the pointers that go into the young generation is an expensive operation which would foil the quick collection of the young generation.

Two common solutions are remembered sets and card marking [5, 8, 7, 10]. We explain these methods in Section 2.2 below. Hosking and Hudson [3] suggested a hybrid method enjoying the advantages of both methods. In this work, we study this method for a generational scheme that uses more than two generations. Here, the naive approach turns out wasteful: cards are being scanned over and over again since we don't have the accurate information on when each of them must be scanned. We offer a simple space-efficient solution to hold the needed information so that we can scan cards only when the scan is really required for the correctness of the collection. This increases the efficiency of scavenging the younger generations.

1.2 The Train Algorithm

In generational schemes the scavenges of the young generation are fast and the pauses of the application are very short. However, when collecting the old generation (or the full heap) the application suffers long delays. The train algorithm [4] deals with this problem by partitioning the old generation to small parts (that can be thought of as cars in trains) and collecting one part at a time. Thus, each collection does not cause a long delay to the application. We go into more details in Section 3.1.

When collecting a car (a part of the old generation), we must scan all pointers that reference objects in the car from the outside. Here, again, one may use the combination of card marking and remembered sets as suggested in [3]. In this case, the problem of which cards to scan for which car collection is more involved. We provide a data structure and a corresponding algorithm that deals efficiently with choosing the cards to be scanned for the current car collection. Our solution is space-efficient, it reduces collection time, and incurs no additional overhead on the application.

1.3 Organization

In Section 2 we present our simple solution for the generational scheme. We begin by reviewing generational collection (Section 2.1) and how card marking interacts with remembered sets (Section 2.2). Then, we present our idea for an efficient combination of the two methods for a scheme with several (more than two) generations (Section 2.3). In Section 3 we study the more involved case of an efficient solution for the train algorithm. We start in Section 3.1 by reviewing the Train algorithm, including the details that are relevant to our solution. Finally, in Section 3.2, we explain the data structure and corresponding algorithm to maintain the relevant information.

2 Generational Collection

We begin with a brief review of generational collection.

2.1 Generational Collectors

As explained in the introduction, generational garbage collectors rely on the assumption that many objects die young. The heap is partitioned into two or more generations and the younger the generation the more frequently it is collected. New objects are allocated in the youngest generation, which is collected when it fills. Some of the surviving objects (the older objects) are promoted to the next generation. When a generation becomes low on free space (after promotion from younger generation), it

is collected. Some of the surviving objects in the generation are promoted to an older generation.

The scheme presented in this work is independent of the specific algorithm used to collect each generation and of the promotion policy. We concentrate on a scheme to maintain the inter-generational pointers.

Since we collect only a part of the heap, we must find all pointers that reference objects in this part of the heap from the outside. These pointers are called inter-generational pointers. Usually, when we collect a generation, we also collect all the generations that are younger than that generation. This reduces the bookkeeping for inter-generational pointers, so that only pointers from older to younger generations need to be kept. Typically, the number of such pointers is relatively small and thus, generational collections can use a data structure to maintain an (almost) updated list of these inter-generational pointers. Two possible data structures were suggested [5, 8, 7]: card marking and remembered sets. A combination of the two was suggested in [3].

2.2 Card Marking and Remembered Sets

One way to keep inter-generational pointers to a given generation is to keep a remembered set for the generation [5, 8]. In the remembered set of generation g, all locations of the inter-generational pointers that reference objects in generation g are kept. Maintenance of this set is done by the application whenever a pointer is stored, and by the collector when objects are promoted. The reader is referred to [1, 9] for a detailed description of various variants on this method.

Maintaining the remembered set implies a costly overhead on the application during normal operation. Card marking reduces this cost [7]. Here, the heap is partitioned into cards of equal size, and whenever the application modifies an object in a card, it marks the card as dirty. Marking a card is a very short operation [7, 10, 2]. Depending on the specific processor, it may be implemented in 3-6 instructions. However, the collector performs more work in a card marking system. It must scan all the dirty cards to find the inter-generational pointers, instead of just getting the pointer from the remembered set. Dirty cards here are both cards that were recently modified by the application and also cards that contain inter-generational pointers. The latter are being scanned repeatedly.

The advantage of combining these two methods was pointed out by Hosking and Moss [3]. After scanning a card once to find all modifications, the relevant inter-generational pointers can be kept in a remembered set and the card need not be scanned again unless it is modified. This keeps the advantage of low overhead on the application, but also increases the collector efficiency, since cards are scanned once and not repeatedly: their dirty flag is cleared, and only dirty (modified) cards are scanned.

2.3 Our Solution

Let us now describe our enhancement to the combination of card marking and remembered sets. In this section, we provide a simple solution for the generational scheme. We enhance the data structure and algorithm for the train algorithm in Section 3 below. In both cases, the key to the solution is to understand the relations between the cards and the generations or cars. This understanding leads to the compaction of the information that has to be kept per card, and to the efficient use of this information.

Wilson and Moher [10] suggest to keep a byte per card in the card table. This idea was adopted in subsequent collectors [2, 3]. It is

this "waste" of space that allows the very low overhead on the application. When the mutator has to mark a card dirty, it is much cheaper to clear a byte in the card table, than to modify a single bit. Longer units, such as words, can also be used on platforms that don't support byte operations. In the remainder of this paper we assume that card table entries are one byte, but the discussion applies equally to card tables with longer entry sizes. In our solution, we keep the low overhead on the application for card marking, and we use the byte in the entry of the card table to store additional useful information.

Our solution is useful for the case of several (more than two) generations. The problem is simple. Suppose that a few young generations are collected, all dirty cards are scanned, and the remembered sets of the collected generations are updated. Should we update all the remembered sets including remembered sets of generations that we do not collect? If we do, we get longer delays while collecting the younger generations. Recall that updating the remembered set means removing all entries that have become irrelevant plus adding entries for new inter-generational pointers.

On the other hand, if we don't update all remembered sets, then we cannot clear the mark of the card since we have to scan it again for older generations. However, if we do not clear the mark, then we are going to wastefully scan this card again and again during future collections of the young generation.

The solution is simple. For each card, we keep the number of the youngest generation for which this card has not been scanned. For example, if a card was scanned for the collection of generations 0, 1, 2, and 3, then we write the number 4 into the corresponding entry in the card table. Whenever we collect a generation which is younger than 4, the card table tells us that this card does not need to be scanned. But when we collect a generation which is number 4 or older, then we scan this card, update the relevant remembered set and modify the entry in the card table to indicate for which generations this card has already been scanned.

It remains to specify what the application has to do when modifying a card. But this operation is not changed: The application has to set the value of the entry in the card table to zero when a card gets "dirty". This means that this card must be scanned for all generations 0 or up, which translates to all generations.

3 Combining Card Marking and Remembered Sets for the Train Algorithm

In this section we explain the more involved solution for the train algorithm. Here, again, we must determine for each car collection which cards should be scanned in order to update its remembered set, but we don't want to scan a card when the scan is not really needed. In addition, we do not want to keep detailed space-consuming information in order to select the "right" set of cards to scan for the current car collection. We present a data structure with a corresponding algorithm, which solves this problem accurately and with efficient space consumption. Let us begin by reviewing the train algorithm and specifying the relevant details for our scheme.

3.1 The train algorithm

The advantage of generational collection is the fast collection of the young generation, which implies a short disruption to the application during these collections. However, when the old generation is collected, the delay is still long. Hudson and Moss [4] suggested the train algorithm to deal with this problem. The train

algorithm partitions the old generation into parts called cars, and collects one car at a time.

The problem with a naive implementation of this idea is that cycles of garbage that spread among several cars cannot be collected. Thus, the cars are grouped into trains, and the algorithm aims at relocating clusters of garbage into a single train and at the same time relocating living objects (that are not part of the cluster) out of the train. In this paper, we do not discuss the correctness of the algorithm, nor its efficiency. We refer the reader to [4, 6] for motivations, details, and an implementation. Here, we specify some details of the algorithm that are relevant for our solution.

First, when a car is collected, the young generation is collected as well. Actually, the young generation may sometimes be collected without any car collection. Second, trains are numbered, cars are numbered, and the lowest car in the lowest train is chosen for collection. During this collection, objects are moved out of the car into other trains or the same train. Sometimes the objects are moved into existing cars and sometimes new cars are created to store them. These new cars may reside in an existing train or a new train may be created for them. After all live objects are moved from the car being collected, the car is reclaimed. The objects that are promoted from the young generation may also go into either old or new cars in various trains.

Each car has a remembered set so that when a car is collected, incoming pointers may be located efficiently. This remembered set is used to keep track of pointers that go from other cars into this car. It is not necessary to keep track of pointers that reference this car from the young generation, since the young generation is fully scanned during each collection, and thus all pointers from the young generation into the collected car will be discovered during this scan.

In order to keep the overhead during normal operation low, we also propose to combine card marking with the remembered sets method as suggested in [3, 4]. The old heap is partitioned into cards and when the application modifies a card, it marks the card dirty. When a car has to be scanned, all "dirty" cards are traversed and the remembered set is updated and later used to find roots for the collection. How do we decide which card is dirty for the current scan?

Here again we have a similar problem about updating the remembered sets. We have many remembered sets (as many as the number of cars) and we wouldn't like to update them all when we update the remembered sets that are relevant for the current collection. Again, we must keep track of which cards should be scanned for which car collection. Otherwise, we will wastefully rescan a card again and again.

Note that this case is different from the generational scheme. There, the young generations are repeatedly scavenged, and we only had to keep the simple indication of which generation is updated with respect to which card (see Section 2.3). Here, the scenario is more dynamic: after a car is scavenged it is reclaimed, and not scavenged again. Nevertheless, we want to be able to keep track of which cards should be scanned for each car in an efficient manner.

3.2 Our Solution

In the following, we discuss the relations of cards and cars. The reader should envision the old generation as containing cars (plus free space) and each car is partitioned into cards. Namely, cards are smaller than cars and there is an integral number of cards in each car. We now explain our solution and we start with two simple observations.

Observation 3.1 *When a new car is created (during a collection) and objects are moved into the car, it's remembered set can be computed exactly by the collector.*

It is always the case that when we move an object we know all references to the object, since we must update these references. This is true no matter if the objects arrive through promotion or through relocation of objects in the old generation. Thus, while setting up a new car during a garbage collection cycle and moving objects into the car, we can compute the initial remembered set for the car.

Observation 3.2 *When a card is modified by the application, it should be rescanned once for updating the remembered sets of the following areas only: the young generation and each car that currently exists in the old generation.*

Clearly, the modified card has to be rescanned in order to update the remembered sets of the areas mentioned. If a new car is later created then this card does not need to be rescanned when the new car will later be collected since the new car will have a completely updated remembered set (also with respect to the previously modified cars).

From Observations 3.1 and 3.2 it follows that we should keep for each car a time-stamp indicating when it was created and for each card a time-stamp indicating when it was last modified. The *time-stamp* is the number of the garbage collection cycle. The cycle number is incremented in the beginning of each collection. But a third point indicates that creation time of the car is not sufficient information. We will want to keep the level of update for each remembered set. Here is a third point relevant to this scheme.

Fact 3.3 *The remembered set of a car may be fully updated also after the car is created.*

For example, when checking whether a train contains only garbage, the remembered sets of its cars may be updated. After such an update, the remembered set of the car is updated with respect to all previous modifications of pointers in the heap, and we would like to record this fact for future collections.

We suggest keeping the following information. For each card, we keep in the card entry table the last time (i.e., garbage collection cycle) this card was modified. For each car, we keep the last time its remembered set was updated. Initially, this is the creation time of the car, but it may be later updated during a collection of another car in the same train. It is the application's responsibility to update the card table and it is the collector responsibility to update the time-stamp of the remembered set for each car. We will talk further on the overhead of card marking by the application in Section 3.3 below. It is important that this overhead is kept minimal.

During a collection of a car, the collector notes the time-stamp of the car, and then scans all cards that have greater (newer) or equal modification time-stamp. When collecting the new generation, the collector scan only cards that have been modified in the last collection cycle.

One important issue still requires consideration. How do we set the time stamp of a card when we copy objects into the card? Note that during car collection, we move objects from the collected car to other cars. This has the additional implication of moving objects from one card to another. We do not need to update the time-stamp of the original card, since it is going to be reclaimed with the car to which it belongs. However, the time-stamp of the new card, i.e., the card to which we copy the object, should be updated appropriately.

The card-table entry of the new card (onto which the object is copied) should contain the latest modification time of objects in the new card. This means a maximum over the modification times of all objects in the card. This property is maintained in the card so far, and we must keep this property also with respect to the new object copied into the card. Namely, if the modification time of the new object is higher than the maximum in the card, then we need to put this modification time in the card entry. If its modification time is lower than the maximum (as indicated in the card-table entry), then no change in the card-table is needed. It remains to explain how to obtain the modification time of the object that is copied. In fact, we don't have this information. However, we do know the time-stamp of the old card (the one from which the object is copied). This time stamp is an upper bound on the modification time of the copied object and can be used as a (conservative) estimate on the last time this object was modified. To summarize the discussion in the last two paragraphs, when an object is copied from Card C_1 into Card C_2, the time-stamp of Card C_2 is set to the maximum over the time-stamps of C_1 and C_2.

3.3 How to mark the cards

The advantage of card marking is in lowering the overhead during normal operation. Very efficient card marking implementations have been suggested for several systems [3, 10, 2]. In our above approach, there is a change in the application overhead. Instead of writing the constant zero into the entry in the card table, we now move a value, i.e., the time, kept in some variable, into this entry. This consumes more machine cycles (even if we assume that the counter of garbage collection cycles remains in cache).

Therefore we refine our algorithm as follows. We reserve the value 0 so that it cannot be a valid time (a number of a collection cycle). Valid times are 1–255. Using this reservation, we let the application use the original procedure of writing a zero into the entry of a modified card. This way, the overhead on the application is exactly the same as the overhead for card marking without using our algorithm. But we must fix this value to the right time stamp. Recall that at collection time, the collector must traverse all entries in the card table, and determine which of the cards must be scanned. During this traversal, the collector changes all zeros to the current time stamp and we are done. Note that for the collector this is a negligible increase in time, since it has to scan the card-table in any case.

3.4 Overflow Handling

Finally, what do we do about overflow? Recall that we are using the collection number for the time-stamp, and this may grow beyond the time-stamp range. We remark that in this work we discuss a time-stamp of one byte. Various considerations may dictate using more than one byte for a card-table entry. For example, if an atomic assignment in a multithreaded environment is defined on a word (of 4 bytes) then one might prefer an entry of 4 bytes in the card-table. But no matter what the range of the time-stamp is, an overflow may eventually occur and must be dealt with. For simplicity, we discuss overflow for a time-stamp of one byte. Our ideas can be easily extended to any range of time-stamps.

So suppose we would like to keep a single byte to hold a time stamp, and the 0 value is reserved (see Section 3.3 above). After the 255th collection, we might encounter the following worst case scenario: All 255 values are used by either cards or cars, and we do not

have any more values for future use. In this case, we are going to give up the accuracy of the information we have kept in order to clear space for new information. In the two solutions that are presented here, we choose to clear 3/4 of the possible values. Namely, to compact the current information into 64 values, and clear the other 192 values for the next collection cycles. This can be easily augmented to any other fraction of the possible values.

The first solution (which we prefer) is to divide all existing time stamps by 4 and take the (ceiling) integral value of the division. The division is applied to the time-stamps of the cards, the time-stamps of the cars and the collection counter. This transformation is fine since it preserves the relations between the time stamps in the system. A card should be scanned for a car if the time-stamp on the card is bigger or equal to the time-stamp on the car. Any card that would have been scanned for a car before the division will be scanned for the car after the division. However, we will scan also some more cards. This loss of accuracy in the information is inevitable, but note that the loss is proportional to the number of possible values (256 in case one byte is used for the time stamp). We believe that this loss of information should have little effect when the time stamp is kept in a full byte. Note the tradeoff that we get between space and time efficiency: using more bytes per time-stamp reduces the loss of information and improves the performance of the collection. Note also that the zero-value preservation required in our scheme (see Section 3.3 above) is preserved since any non-zero time-stamp is translated to a non-zero time stamp.

A second possible solution is good in case the distribution of times is not uniform at overflow time. Namely, if typically, most of the times recorded are recent times and only few cards/cars or even none at all have old time stamps. In this case, it is better to keep better records for recent time and lose accuracy for older times. Thus, we use the following trick. At overflow time, we go over all time stamps. All times lower than 193 are set to the value 1, whereas we subtract 192 from all times higher than 192. (We choose 192 since we want to evacuate 3/4 of the 256 possible values.) Thus, relations between cars and cards with recorded times over 192 are fully kept. All cars with lower update time must engage in scanning all cards for collection. This is only good if the number of such cars is small, and infrequent long collections are allowed.

The reader may envision other transformations to fit a specific behavior of a specific system. For example, it may be desirable to group immutable old objects on cards of their own and reserve the time-stamp 1 for these cards. The important property of any such a transformation is that it should be monotone, so that the relations between the cars and the cards are kept.

3.5 Summarizing the algorithm

Let us summarize the discussion in this section and shortly present the data structure and the algorithm to use and modify it.

The data structure: We keep a card-table for the cards. For each card there is an entry (a byte) indicating the last time this card was modified. Also, with each car in the old generation, we keep a byte with a time-stamp indicating the last time in which the remembered set of this car was updated.

Card marking during normal operation: When a pointer is modified in the old generation, the corresponding card number (to which the pointer belongs) is computed, and a 0 is

written into the entry of this card in the card-table.

Update and use of data structure by the collector: The collector has a static parameter *counter* indicating the number of collections performed so far. The parameter *counter* is initialized with 1.

Suppose that the collector wants to update the remembered set of a car C. The collector checks the time-stamp of the car C. Denote this time-stamp by t. The collector goes over all of the card-table. For each entry in the card table the collector starts by fixing zeros: if the entry contains a zero, then the collector writes the value of *counter* into the entry (this is the number of the previous collection since *counter* has not yet been incremented in this collection). Next, the collector compares the entry to the number t. If the number in the entry is greater or equal to t, then the collector scans the corresponding card and updates the remembered set of the car with the pointers in the card. After going over all of the card-table, the remembered set of Car C is updated. The collector records this fact by writing $counter + 1$ into the time-stamp of Car C. To update the remembered set of the young generation, the collector must scan all cards for which the time-stamp is *counter* (these are the cards for which the zero fix was implemented.

We skip the details of the collection, which are not relevant.

Finally, when the collection is done, the counter of collections is incremented. If the collection number is 255, than the collector handles overflow in one of the methods discussed in Section 3.4 above.

3.6 Keeping more than one young generation

In this section we have described a scheme that deals with one young generation and one old generation (which is split into cars). It is also possible to adapt our scheme for a scheme with several young generations and one old generation that is split into cars (such a scheme was suggested in [4, 6].) We only have to add for each generation a time-stamp indicating when the remembered set of this generation was last updated. This time-stamp should be updated by the collector whenever it scans the corresponding generation. When scanning a generation g with time stamp t, it is easy to decide if a card has to be scanned: if the time-stamp of the card is greater or equal to t, then it means that the card was modified after the last collection of Generation g, and therefore this card has to be scanned. All cards with time-stamps lower than t need not be scanned.

4 Acknowledgment

We thank the anonymous referees of the *International Symposium on Memory Management* for their useful comments.

References

[1] R. E. Jones and R. D. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, July 1996.

[2] U. Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors. *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993

[3] A. L. Hosking and R. L. Hudson. Remembered Sets Can Also Play Cards. In *OOPSLA'93 Workshop on Garbage Collection and Memory Management.* Washington, DC, September 1993.

[4] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors. Proceedings of International Workshop on Memory Management, volume 637 of Lecture Notes in Computer Science, 1992. Springer-Verlag.

[5] H. Lieberman and C. E. Hewitt. A Real Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6), pages 419-429, 1983.

[6] J. Seligmann and S. Grarup. Incremental mature garbage collection using the train algorithm. In O. Nierstras, editor. Proceedings of 1995 *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag, August 1995.

[7] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT, AI Lab, February 1988.

[8] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. Proceedings of the *ACM Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices Vol. 19, No. 5, May 1984, pp. 157-167.

[9] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors. *Proceedings of International Workshop on Memory Management*, volume 637 of Lecture Notes in Computer Science, 1992. Springer-Verlag.

[10] P. R. Wilson and T. G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87-92, 1989.