An Improvement to Sliding Garbage Collection

Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
robert.strandh@u-bordeaux1.fr

ABSTRACT

Garbage collection algorithms are divided into three main categories, namely *mark-and-sweep*, *mark-and-compact*, and *copying* collectors. The collectors in the *mark-and-compact* category are frequently overlooked, perhaps because they have traditionally been associated with greater cost than collectors in the other categories. Among the compacting collectors, the *sliding* collector has some advantages in that it preserves the *relative age* of objects. The main problem with the traditional sliding collector by Haddon and Waite [4] is that building address-forwarding tables is costly. We suggest an improvement to the existing algorithm that reverses the order between building the forwarding table and moving the objects. Our method improves performance of building the table, making the sliding collector a better contestant for young generations of objects (nurseries).

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Algorithms, Performance

Keywords

Compaction, Sliding garbage collection

1. INTRODUCTION

As is evident from the books by Jones et al [5] [6], garbage collection is a rich and much researched field. With the evolution of processor technology, advantages and inconveniences of different techniques change. Current technology requires techniques that take into account the big difference in performance between the processor and the memory, as well as the existence of several *cores* and *threads*.

In this paper, we consider an improvement to the table-based sliding collector proposed by Haddon and Waite [4] in 1967.

The sliding collector is a member of the family of *mark-and-compact* techniques. Techniques in this family are not prone to *fragmentation* as mark-and-sweep collectors are, and do not require as much additional memory as do copying collectors. Furthermore, the techniques that use *sliding* preserve the *allocation order* between objects, and therefore their *relative age*, whereas copying collectors and mark-and-sweep collectors do not. By preserving the allocation order, two great advantages are gained:

- If the heap is part of a generational garbage collector, it is always possible to promote the oldest objects, whereas with a copying collector, a significant number of recently allocated objects may well be promoted, even though they are likely to die soon after being promoted.
- As Wilson [11] points out, objects that are allocated together, die together. Thus, if the allocation order is preserved, the heap is likely to have large intervals of live objects and large intervals of dead objects. When the allocation order is not preserved, the heap will more likely contain more and smaller intervals of live and dead objects.

The main reason that sliding techniques have largely been abandoned, is that they incur a much higher cost than both mark-and-sweep collectors and copying collectors, either in terms of additional computations or additional memory requirements, or both.

The method presented in this paper decreases the cost of compaction by avoiding the need to move the so-called *break table* used in the traditional sliding collector. Furthermore, we suggest using our method for collecting a per-thread *nursery* that fits in the processor cache. While a copying collector can use only half the available space, a sliding collector can use the entire available space. Because of the increased size, compared to a copying collector for the nursery, our method will allow more objects to die between two invocations, further reducing the overall cost of garbage collection.

2. PREVIOUS WORK

2.1 Haddon and Waite

In 1967, Haddon and Waite [4] designed the first algorithm for compacting garbage collection. The context of their work was assumed to be an existing mark-and-sweep garbage collector using a free list, and when that collector fails due to fragmentation, even though there is enough total space available, their compacting algorithm would take over and compact the heap. The cost of their algorithm was considered unimportant, because the alternative would be to fail by terminating the program.

Rather than invoking a marking procedure to determine live data, they imagined using the existing free list to determine areas of available storage.

An entry in their break table indicates a start address of a zone of live data and the total amount of free space below that address. There are many different possible variations on the exact contents of the break table, but they are all equivalent.

They show that, if each object requires at least two words of storage, then the total amount of free space in the entire heap is large enough to hold the break table. As a result, no additional space is required for their technique to work.

2.2 Other work

Of the traditional compacting algorithms, three types of algorithms preserve allocation order; the one by Haddon and Waite [4], the so-called Lisp2 algorithm, and the *threading* algorithms. The Lisp2 algorithm requires an additional field the size of a pointer in each object in order to hold forwarding information. This field is set after the marking phase. Pointers are then updated according to this information. Finally, the heap is compacted. Threading algorithms by Fisher [3], Morris [10], and Jonkers [7] work by reversing pointers. In their survey of different compacting algorithms, Cohen and Nicolau [2] conclude that the Lisp2 algorithm is the fastest. Threading algorithms perform the worst.

Abuaiadh et al [1] designed a compaction algorithm that uses forwarding information much like the Lisp2 algorithm. However, to avoid the cost of an entire pointer per object, they use a single word of forwarding information for around 256 bytes, which amounts to no more than a few bits per word. Although reported to be fast, unfortunately, their algorithm does not preserve the allocation order of objects. Their paper mentions that benchmarks have verified the importance of doing so. They report that compaction algorithms may cause significant pauses, and their algorithm reduces these pauses. The reason for these pauses is no doubt that the context of their work is a large global heap (they mention 1GB), whereas the context of the present work is the size of the nursery (around 4 MB).

Kermany et al [8] describe a compacting algorithm that uses memory-mapping operations in order to compact the heap. Their algorithm is parallel, concurrent, and incremental. However, in addition to the cost of marking, moving objects, and updating pointers, compared to the method described in this paper, there is an additional cost associated with the memory-mapping operations, and this cost is is non-negligible. Forwarding information is handled the same way as in the method used by Abuaiadh et al. Like the method described by Haddon and Waite, and the method presented in this paper, relative allocation order between the objects is preserved.

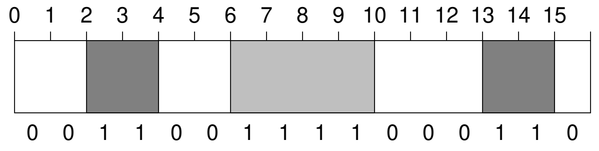

Figure 1: Example of initial heap.

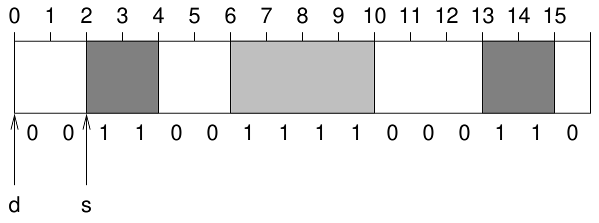

Figure 2: Pointers to source and destination locations.

3. OUR METHOD

Our method uses a *break table* just like the method by Haddon and Waite [4], but instead of building, moving, and sorting the table while objects are moved, we first move the objects and then construct the table. For that, additional space in the form of a bitmap is required. The bitmap has one bit per word of memory in the heap, which amounts to less than 2% additional memory on a 64-bit machine. The bits in the bitmap are set by the *mark* phase of the garbage collector.

Figure 1 shows a heap in which shaded areas indicate live objects and white areas indicate dead objects. The heap contains 16 word as shown by the addresses. At the bottom of the figure is shown the bitmap after the mark phase (phase 1) is complete.

In phase 2, the heap is compacted by sliding the live objects to the beginning of the heap. In this phase, two pointers are used, a *source* pointer pointing to words containing live objects, and a *destination* pointer pointing to words containing dead objects, as illustrated by Figure 2. Words are copied from the source location to the destination location. In each iteration, the destination location is incremented by one unit, whereas the source location is incremented until either it reaches the end of the heap, or a word containing a live object as indicated by the bitmap containing a 1.

Figure 3 shows the situation when phase 2 is complete.

In phase 3, a *break table* is built at the position of the destination pointer. The break table consists of a sequence $a_0, d_0, a_1, d_1, \ldots, a_n, d_n, a_{n+1}$ of alternating *addresses* and *deltas*. The value a_0 is always 0. The table has an odd number of elements, because it both starts and ends with an address. Each a_i (except possibly a_0 and a_{n+1}) is the index of the beginning of a zone of dead objects. Each d_i is the sum of the sizes of the dead zones preceding a_{i+1}.

Figure 4 shows the break table of the example heap in Fig-

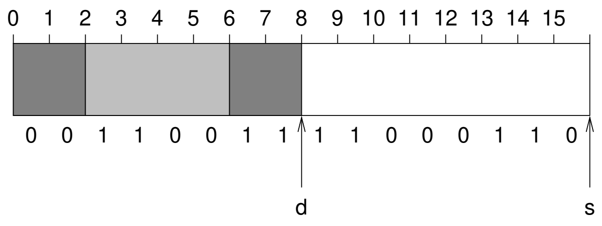

Figure 3: Heap after compaction.

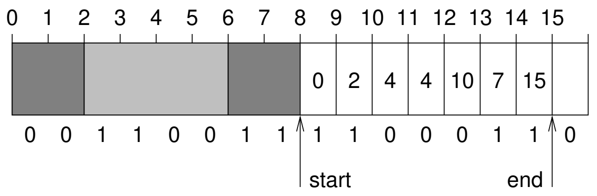

Figure 4: Break table.

ure 1.

Notice that if both the bottom zone and the top zone of the heap contain live objects, then our break table may require three additional words of storage compared to the total number of free words available. The reason for this additional requirement is that our table contains *sentinels* (the first two words and the last word) that are not strictly required, but by including them, we avoid special cases in our procedure for searching the table. We can easily make sure that three additional words are available by triggering a collection when granting a request for memory would leave fewer than three free words at the end of the heap. Haddon and Waite [4] did not have this luxury, because they assumed an existing mark-and-sweep collector. For that reason, their paper contains an extensive argument that their break table can fit in the available space.

The break table is built by scanning the bitmap from start to end. In practice, since the heap is likely to contain fairly large contiguous zones, the bitmap will contain long runs of 0s and long runs of 1s. It is therefore advantageous to scan the bitmap a word at a time, making this phase quite efficient.

In phase 4, the lower part of the heap is traversed word by word in order to adjust the pointers according to the contents of the break table. For each pointer value p, the table is searched for values a_i, d_i, a_{i+1} such that $a_i \leq p < a_{i+1}$. The pointer value p is adjusted by subtracting d_i from it. To find the entry, the break table is search using *binary search.*

While the overhead of the binary search may seem unacceptably high, two properties contribute to keeping this overhead low:

1. While the break table could contain as many as $N/4$ entries, where N is the number of words in the heap, in practice, it contains far fewer than that, again because the heap is likely to contain relatively few relatively large zones.
2. It is very likely that the entry in the break table required to adjust a particular pointer is the same as the entry required to adjust the pointer immediately preceding it. By testing this case first, the vast majority of full binary searches can be avoided.

4. PERFORMANCE

It is notoriously hard to test the performance of garbage collection algorithms. Nevertheless, we would like to get some idea of the time the various phases take. To that end, we created a few test cases which (together with some educated guesses) will give us some ballpark figures with respect to performance of our method, at least as compared to other methods.

For our tests, we chose a heap size of 2^{19} words of 8 bytes each. This heap size was not chosen randomly. It was chosen so that the entire heap would fit in the cache memory of the computer used for the tests (x86-64, 1.6GHz, GNU/Linux, SBCL), and research [9] indicates that a nursery this size is a good choice.

Thus, we have the following definitions valid for all the tests:

```
(defparameter *size* (ash 1 19))

(defparameter *heap* (make-array *size*))

(defparameter *bitmap*
  (make-array *size* :element-type 'bit))
```

4.1 Phase 1, marking

We did not attempt to test the marking phase, because it is no different from the marking phase of any other algorithm.

4.2 Phase 2, compaction

To test the compaction phase, we wrote the following compaction program:

```
(defun move (heap bitmap)
  (declare (type (simple-vector #.*size*) heap)
           (type (array bit (#.*size*)) bitmap)
           (optimize (speed 3) (safety 0) (debug 0)))
  (let* ((d (position 0 bitmap))
         (s (position 1 bitmap :start d)))
    (declare (type (integer 0 #.*size*) d s))
    (loop until (= s #.*size*)
          do (setf (aref heap d) (aref heap s))
             (incf d)
             (incf s)
             (loop until (or (= s #.*size*)
                             (= (sbit bitmap s) 1))
                   do (incf s)))))
```

The hypothesis here is that the time consumed by the compaction phase is determined by two elements:

1. A constant term that has to do with scanning the bitmap.

2. A term proportional to the number of elements that have to be moved.

To test the hypothesis, we executed the function `move` with three different bitmaps as shown in the following table:

Element count	CPU time (ms)
1	0.8
2^{18}	1.1
$2^{19} - 1$	1.4

To avoid measuring the performance of the function `position`, the first line in the table was executed with a bitmap where the element with index 1 was equal to 1. The second line in the table was executed with a bitmap where every other element was 1. The third line in the table was executed with a bitmap where only element 0 was 0. In all cases, the test was run in a loop with 1000 iterations. The loop was the argument of the `time` macro, and the right column of the table shows the result returned by `time` divided by 1000.

The hypothesis above seems confirmed where scanning the bitmap seems to take around 0.8ms and moving an element seems to take around 1ns per element.

Now, in a typical collection cycle, at least around half of the elements of the heap would be dead (if not, objects would be *promoted*). Furthermore, since objects that are allocated together die together, the bitmap will contain long runs of 0s and long runs of 1s, and because objects either die young or survive a long time, the bitmap will typically start with a run of 1s. A slightly more clever implementation of the bitmap would then test 64 bits at a time rather than individual bits like we do in our test. In such an implementation, the time for managing the bitmap would be negligible. In a typical collection cycle, significantly fewer than half of the elements would have to be moved. The expected compaction time would therefore be closer to 0.3ms.

4.3 Phase 3, building the table

In order to test the performance of building the table, we used the following function:

```
(defun build-table (heap bitmap start)
  (let ((acc 0)
        (end start))
    (declare (type fixnum acc start end)
             (type (simple-vector #.*size*) heap)
             (type (simple-array bit (#.*size*)) bitmap)
             (optimize (speed 3) (safety 0) (debug 0)))
    (loop for address of-type fixnum from 0
          for prev of-type bit = 1 then bit
          for bit of-type bit across bitmap
          do (when (= bit 0)
               (when (= prev 1)
                 (setf (svref heap end) address)
                 (setf (svref heap (1+ end)) acc)
                 (incf end 2))
               (incf acc))
          finally (when (= prev 1)
                    (setf (svref heap end) address)
                    (setf (svref heap (1+ end)) acc)
                    (incf end 2))
                  (return end))))
```

In that program, the parameter `start` contains the total number of live words in the heap, which is the same as the start index into the heap where the table should be built.

As for the compaction phase, the hypothesis here is that the time consumed by the compaction phase is determined by two elements:

1. A constant term that has to do with scanning the bitmap.
2. A term proportional to the number of entries in the table to be built, which is the same as the number of zones of dead elements.

We further hypothesize that the performance of this phase is independent of the *size* of the zones of dead elements.

We start by testing the last hypothesis. For that, we run the function `build-table` above with two different bitmaps, both containing 2^{10} zones of dead objects and the same number of zones of live objects. In the first test, the size of each live zone is a single word. In the second case, the size of each live zone is 2^8 words. The result of this test is shown in the following table:

Live zone size	CPU time (ms)
1	1.0
2^8	1.0

As we can see, the hypothesis that the sizes of the zones do not influence the performance of this phase is confirmed.

Next we tested the function `build-table` for different sizes of the table to be built. The size of each live zone was a single word. The following table shows the result:

Table size	CPU time (ms)
2^1	1.0
2^9	1.0
2^{11}	1.0
2^{13}	1.1
2^{15}	1.4
2^{17}	1.2

It appears that the time to build the table is entirely dominated by the time to traverse the bitmap. We have no explanation as to why the time does not increase monotonically with the table size, but we have run this test numerous times and we still get this surprising result. It appears that this anomaly does not manifest itself on computers other than ours.

With improved bitmap management code, we believe that building the table will typically take about the same as compacting the heap, i.e., around 0.3ms.

4.4 Phase 3, adjusting pointers

In the worst case, every live pointer must be updated, and each such pointer requires n iterations to search the break table, where n is the depth of that table. The number of live pointers is the difference of the size of the heap and the number of words required for the break table. In the worst case, the number of words required for the break table is as small as possible (which is 2^{n+1}), so that the number of live pointers is $2^N - 2^{n+1}$. The total amount of work required can therefore be expressed as $n \cdot (2^N - 2^{n+1})$. When $N = 19$, this function has a maximum for $n = 15$.

We can get an upper bound on the time to adjust the pointers by creating a heap with roughly $2^{19} - 2^{16}$ words containing random values in some interval $[0, K]$ followed by a break table with 2^{16} words, with addresses in the interval $[0, K + 1]$.

In order to search the break table, we used the following binary search function:

```
(defun binary-search (heap first last address)
  (declare (type (simple-vector #.*size*) heap)
           (type (integer 0 (#.*size*))
                 first last address)
           (optimize (speed 3) (debug 0) (safety 0)))
  (let ((f first)
        (l last))
    (declare (type (integer 0 (#.*size*)) f l))
    (loop until (= (- l f) 2)
          do (let* ((mid (1+ (ash (ash (+ l f) -2) 1)))
                    (elt (svref heap mid)))
               (declare (type fixnum elt))
               (if (< address elt)
                   (setf l mid)
                   (setf f mid))))
    (svref heap (1+ f))))
```

The following function was used to adjust the pointers of the heap:

```
(defun adjust ()
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (loop with w = (worst-frontier 19)
        with l = (1- (expt 2 19))
        with g = *g*
        for i from 0 below w
        do (incf (aref *g* i)
                 (binary-search g w l i))))
```

On our computer, this function takes approximately 30ms to run, again on a heap containing 2^{19} words. While this number may seem high, we should keep in mind that it is pessimistic in several different ways:

- This test assumes that almost 90% of the heap is live. When there are this many live objects, they will be *promoted*, so a typical number should be significantly less than 50%.
- It assumes that there are 2^{15} dead zones, which would make each zone very small. A more typical situation is that dead zones are significantly larger, so that there are significantly fewer dead zones.
- It assumes that every live word in the heap contains a pointer that needs to be adjusted. More typically, many words contain immediate data, or pointers to older generations, and these pointers do not need to be adjusted.

Furthermore, there is a simple optimization that we think can eliminate a large fraction of the table searches. By keeping the result (a_i, d_i, and a_{i+1}) between iterations and comparing the pointer to be adjusted to these values, we think that in many cases we will get a hit, simply because it is likely that two consecutive pointers belong to the same live zone and refer to objects in that zone.

An educated guess, then, is that a typical value for this phase would be closer to 5ms.

4.5 Overall performance

Taken together, the performance figures in the previous section suggest that the entire compaction phase of our suggested *mark-and-compact* sliding collector would take around 6ms on a fairly modest desktop, with an absolute worst-case of around 30ms.

Including the mark phase, there are two possibilities:

- Compaction dominates over marking. Then the figures given above are reasonable estimates of the total time for the entire collection.
- Marking dominates over compaction. Then the figures given above are sufficiently good that our method is no more expensive than others such as mark-and-sweep and copying.

In addition, we should not forget the reasons for choosing a sliding collector in the first place:

- Compared to a copying collector, only half the space is required, or, alternatively, the space available in the nursery is twice as big. When this is the case, the number of required collections decreases, a situation which gives more time for short-lived objects to die, thereby decreasing the total amount of work spent in garbage collection.
- Also compared to a copying collector, the relative ages of objects are kept intact, which decreases the risk of promoting objects that will die soon after a collection.
- Compaction eliminates the potential problem of fragmentation.

All these factor contribute to decreasing the total cost of garbage collection, although measuring the exact impact is of course very hard.

It is not advisable to use our algorithm on the global heap, say a heap that is three orders of magnitude larger than the one used in our tests. Even multiplying our timing results by a thousand would imply a significant pause, but since the break table requires a binary search, the time to search it would be proportional to the logarithm of the size of the heap, penalizing performance even more. To manage a global heap, we would recommend the technique by Kermany et al [8], which has a greater overhead, but has the advantage of being incremental, concurrent, and parallel, thus effectively eliminating significant pauses.

One might want to speculate about the performance of our algorithm when the size of the heap is significantly larger than the size of the cache, but much smaller than the global heap. Such a situation might be desirable if it turns out that the cost of cache misses is compensated by the fact that more time between collections will allow more objects to die. For that reason, we re-ran our tests with a heap that is 4 times the size of the heap used for the previous tests. We observed a factor 10 slowdown compared to the tests reported in this section. In order for this configuration to be advantageous, the number of objects that die between two garbage-collection cycles has to increase substantially, and whether this is the case depends on the type of application being executed. Only benchmarks in a final system or simulations with real allocation traces can determine whether this is the case. Note, however, that even though overall throughput *might* improve with a larger heap, the time for an invocation of the garbage collector becomes closer to what is tolerable for many so-called *soft real time* applications such as generation of sound or images in real time.

5. CONCLUSIONS AND FUTURE WORK

We have presented an improvement to the compacting and sliding collector invented by Haddon and Waite [4]. Our method is an improvement in that it does not require the partially built break table to be moved, because the entire table is built *after* the compaction phase is finished.

While it is extremely difficult to compare performance of different methods for garbage collection, in order to get an indication of the performance of our method, we created a number of tests. We believe these tests show that the method we suggest is not prohibitively expensive. The advantages of compaction to overall performance of the garbage collector are hard to measure or even estimate, but we think that they will compensate for the slight increase in cost of compaction compared to ordinary copying or mark-and-sweep.

We plan to use this algorithm in the per-thread nursery collector in our system SICL.[1] Inspired by the Multics system, we plan to instrument the system with a number of *meters* so that performance data can be collected at all times. Only then will we be able to obtain a final verdict concerning the performance of this method.

6. REFERENCES

[1] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 224–236, New York, NY, USA, 2004. ACM.

[2] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Trans. Program. Lang. Syst.*, 5(4):532–553, Oct. 1983.

[3] D. A. Fisher. Bounded workspace garbage collection in an address order preserving list processing environment. *Inf. Process. Lett.*, 3(1):29–32, July 1974.

[4] B. K. Haddon and W. M. Waite. A compaction procedure for variable length storage elements. *Computer Journal*, (10):162–165, Aug. 1967.

[5] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[6] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[7] H. B. M. Jonkers. A fast garbage compaction algorithm. *Inf. Process. Lett.*, 9(1):26–30, July 1979.

[8] H. Kermany and E. Petrank. The compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, New York, NY, USA, 2006. ACM.

[9] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. *SIGPLAN Not.*, 46(11):21–32, June 2011.

[10] F. L. Morris. A time- and space-efficient garbage compaction algorithm. *Commun. ACM*, 21(8):662–5, 1978.

[11] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.

[1]https://github.com/robert-strandh/SICL