Adaptive Scheduling of Incremental Copying
Garbage Collection for Interactive Applications

Roger Henriksson

LU-CS-TR:96-174
LUTEDX/(TECS-3070)/1-8/(1996)

Also published in: Proceedings of NWPER'96, Nordic Workshop on
Programming Environment Research, L. Bendix, K. Nørmark,
and K. Østerbye (Eds), Aalborg, Denmark, May 1996.
Aalborg University, Technical Report R-96-2019.

Department of Computer Science
Lund Institute of Technology
Lund University

P.O. Box 118, S-221 00 Lund
Sweden

Adaptive Scheduling of Incremental Copying Garbage Collection for Interactive Applications

Roger Henriksson

Dept of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
e-mail: Roger.Henriksson@dna.lth.se

Abstract. Incremental algorithms are often used to interleave the work of a garbage collector with the execution of an application program, the intention being to avoid long pauses. However, overestimating the worst-case storage needs of the program often causes all the garbage collection work to be performed in the beginning of the garbage collection cycles, slowing down the application program to an unwanted degree. This paper explores an approach to distributing the work more evenly over the garbage collection cycle.

1 Introduction

Garbage collection (GC) is the process of automatically identifying objects on the heap no longer accessible from the application and reclaiming the memory occupied by them for later use. Garbage collection is often combined with compaction of the heap, which eliminates the problem of memory fragmentation. The first GC algorithms caused the application program to halt for extended periods, seconds or even minutes, while memory was reclaimed, essentially making them suitable only for batch programs. The last decades have brought a number of techniques for GC in environments with higher demands on short response times, such as interactive and real-time applications.

Incremental algorithms perform GC in small increments interleaved with the execution of the application program. Long GC-induced pauses are thus avoided. An often used incremental GC algorithm is Brooks' algorithm [Bro84], which is an improved version of Baker's copying algorithm [Bak78]. The heap is divided into two semispaces called *fromspace* and *tospace*, see Figure 1. New objects are allocated at the top of tospace (at T). When tospace is filled up, a *flip* is performed. This changes the meaning of fromspace and tospace. The old from-

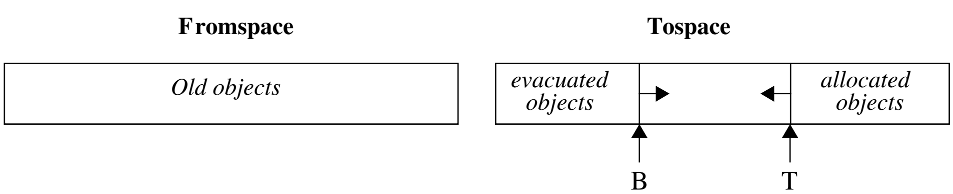

Figure 1 *The heap structure of Brooks' algorithm.*

space now becomes tospace and vice versa. The flip also initiates a new *GC cycle*. New objects are as usual allocated at the top of tospace. A small amount of GC work is performed in connection with every allocation. The garbage collector traverses the object graph starting from a set of root pointers. When an object located in fromspace is encountered, it is copied, or *evacuated*, into tospace. Evacuated objects are placed at the lower end of tospace (at B). When all live objects have been evacuated, the garbage collector is suspended awaiting tospace to be filled up. A new flip is then performed, starting a new GC cycle.

2 Static scheduling

A Brooks-style garbage collector must guarantee that all live objects are evacuated from fromspace before tospace fills up. Otherwise, a deadlock occurs. In order to allocate new objects, a flip must be performed, but a flip cannot be performed as long as live objects remain in fromspace. Since there is no room in tospace to hold the objects to evacuate, they cannot be evacuated.

To schedule the work of the garbage collector in such a way that it can be guaranteed that no deadlock occurs, some assistance from the programmer is required. Let E_{max} be the (programmer-specified) maximum amount of simultaneously live memory. Immediately after a flip, all live objects are located in fromspace. All of these may have to be evacuated to tospace in the worst case. Therefore, the maximum amount of memory needed to hold evacuated objects will be E_{max}. If the size of tospace is denoted S, the remaining memory available for new objects, F_{min}, will be equal to $S - E_{max}$. Let the GC work required in the worst case to evacuate all live objects from fromspace be W_{max}. Also, let W be the GC work performed so far during the cycle and A be the amount of newly allocated objects during the cycle[1]. In order to guarantee that a deadlock does not occur, the garbage collector must have performed W_{max} work before the amount of new objects reach F_{min}.

The traditional approach to distributing the GC work over the GC cycle is to perform an amount of work proportional to the size of the object to be allocated. The worst-case need is used a basis for calculating the amount of work. Enough work is performed after each allocation to make the following inequality hold:

$$W \geq \frac{A \cdot W_{max}}{F_{min}}$$

GC work can be measured in several ways [Hen96]. An often used method for interactive systems is to use the amount of evacuated objects as the metric for performed GC work. Then, the inequality becomes:

$$E \geq \frac{A \cdot E_{max}}{F_{min}}$$

If we plot the minimum amount of evacuated objects, E, as a function of the amount of new objects, A (equality in the above formula), we get a linear func-

1. When *amount of objects* is mentioned in this paper, it refers to the total amount of memory occupied by the objects, not the number of objects.

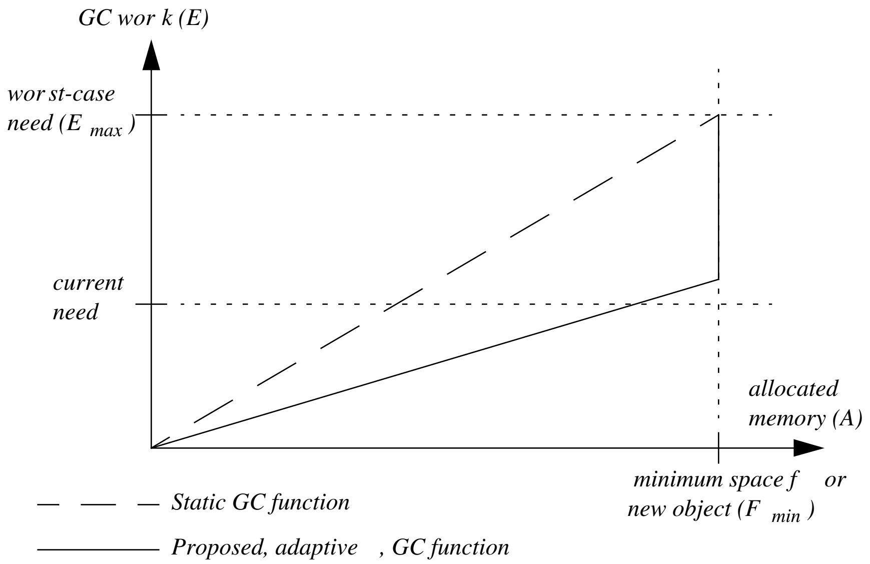

Figure 2 *Alternative GC functions. A GC function that continuously adapts itself to the current memory needs of the application distributes the GC work over time considerably better than a static GC function derived from the worst-case needs.*

tion. This is illustrated by the dashed line in Figure 2. Denote such a function a *GC function*.

When the actual amount of live objects in fromspace is close to the worst case, this strategy provides near-optimal distribution of the GC work. When the amount of live objects is significantly lower than in the worst case, the GC work is finished very early in the cycle. This is also illustrated in Figure 2. Instead of finishing the GC work immediately before the amount of new objects reaches F_{min}, the current need of GC work is reached considerably earlier (when the GC function reaches the *current need* level). The garbage collector is suspended for the rest of the cycle. The interference of GC on the execution of the application program will therefore vary considerably over the GC cycle. The application will run slowly in the beginning of each GC cycle due to a high GC rate, but will increase speed later in the cycle. This can lead to jerky behaviour of the application program. In Figure 2, the GC rate is given by the gradient of the GC function. It tells how much GC work must be performed for each allocated memory cell.

3 Adaptive scheduling

If we can make a better estimation of how much live objects fromspace contains just after a flip than using the worst-case amount, it is possible to reduce the GC rate accordingly. This paper proposes using the amount of objects evacuated during the last GC cycle as an estimate of how much objects are alive in fromspace. Let E_{last} denote this amount. Then, the total GC work that will have to be performed after an allocation is given by the inequality:

$$E \geq \frac{A \cdot E_{last}}{F_{min}}$$

However, it is quite conceivable that the actual amount of live objects is larger than E_{last}, which would lead to a deadlock if no actions were taken to counter it. It is therefore further proposed that the application program is halted if A should reach F_{min} before fromspace is fully evacuated, and that all the remaining GC work is performed before execution of the application program is resumed. This is an unwanted event though and should be avoided since it may cause a long delay. To allow for some change in the amount of live objects without causing such delays, a *variation term*, V, can be added to E_{last} when determining the GC function. We now get the inequality:

$$E \geq \frac{A \cdot (E_{last} + V)}{F_{min}}$$

This produces the GC function illustrated by the continuous lines in Figure 2. The variation term, V, can be implemented in several ways. The simplest approaches is to give it a constant value or to let it be a specific fraction of E_{last}. A more advanced approach is to again use an adaptive strategy. If the garbage collector observes that the estimation of the amount of live objects, $E_{last} + V$, is systematically to low, V should be increased, and vice versa.

The gain from using the proposed adaptive scheduling technique is that the GC rate can be reduced and the GC work thus better distributed over time when the actual amount of live memory differs significantly from the worst-case amount. The application program will then run more smoothly. Also, since GC work tends to be performed later in the GC cycle, the problem with *floating objects* should decrease somewhat. Floating objects are objects that are identified as live and therefore evacuated to tospace, but dies before the GC cycle ends. If objects are evacuated later in the cycle, more objects will have time to die before being evacuated. Therefore, the total amount of required GC work decreases.

4 Performance discussion

The proposed scheduling strategy assumes that the actual amount of live memory is significantly smaller than the worst-case amount specified by the programmer. Is this really a fair assumption, and how much can the GC rate be reduced? Will the memory usage be stable enough to make E_{last} a good estimate of the amount of live objects in fromspace, or will they not correlate?

The intention of introducing adaptive scheduling is to get interactive applications to execute more smoothly. Therefore, let us study the memory usage of an interactive application. To get realistic data, an interactive application, olvwm, the Open Look Virtual Window Manager was modified to record all allocation requests (malloc calls) and deallocations (free calls). A memory usage profile, which is shown in Figure 3, was compiled from the resulting memory

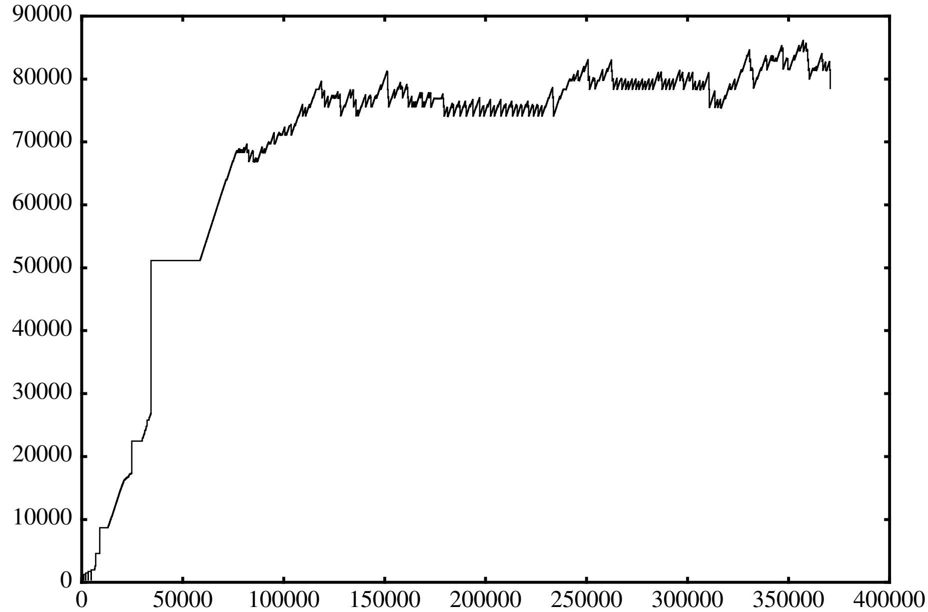

Figure 3 *Profile of memory usage in olvwm, the Open Look Virtual Window Manager. The amount of live memory is plotted as a function of the total amount of allocated memory.*

management trace. The figure shows how the amount of live memory varies during the execution of the program. The measurements show that the amount of live memory increases quickly as the program starts up (when no interaction with the user takes place), but is then quite stable during the lifetime of the program. It seems therefore reasonable to assume that E_{last} will indeed be a good estimate of the amount of live memory and that long pauses due to underestimation of the amount of live memory will be very rare.

As mentioned previously, the programmer must specify a maximum amount of live memory when designing the application. This is used by the garbage collector to ensure that deadlocks do not occur. The programmer-specified value do not have to be precise, but it must not be too small. When designing an interactive program like olvwm, it is very difficult to make a good estimate of the worst-case needs. Different users use the program differently and with different loads. In the case of olvwm, some users may have a lot of windows on the screen simultaneously while other only have a few. The programmer will have to design for the needs of the worst-case user. It can therefore be assumed that actual amount of live objects will often be small compared to the programmer-specified amount.

To assess what decrease in GC rate the strategy will yield, we study a possible scenario: Assume that the actual amount of live objects are stable at 50% of E_{max}. If we ignore the relatively small impact of V, the variation term, a statically calculated GC function based on E_{max} will finish the evacuation of tospace (reach the *current need* level in Figure 2) in half the time compared to an adaptive GC function. Thus, an adaptive garbage collector would reduce the GC rate by as much as a factor two.

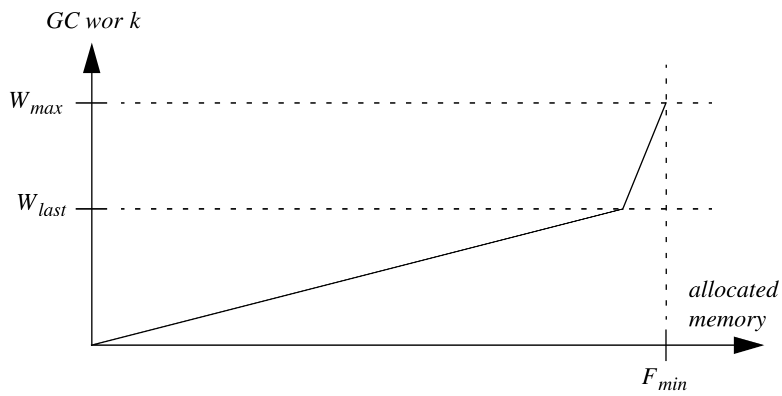

Figure 4 *An adaptive GC function for real-time applications. The gradient of the last part of the function is determined from the maximum allowed GC rate instead of going vertically upwards when the amount of allocated memory reaches F_{min}.*

5 Real-time systems

This paper has so far only discussed interactive systems, in which rare long GC-induced pauses can be tolerated. For interactive systems it is important that the average-case behaviour is good. Bad performance in the worst case can be tolerated if it is unlikely to occur. The proposed scheduling strategy employs this property of interactive systems to improve the average-case behaviour on the expense of a somewhat worse worst-case behaviour. However, the scheduling strategy can also be used to improve the average performance of *real-time systems*. A real-time system has very strict demands on response times and the worst-case behaviour must therefore govern the design.

The long GC-induced pauses that the proposed strategy may cause when underestimating the amount of live memory cannot be tolerated in a real-time system. A way to eliminate the long pauses is to construct the adaptive GC function somewhat differently. First, the highest GC rate that the application can tolerate without failing to meet its deadlines is determined. This also gives us the highest allowed gradient of the GC function. Let W_{last} denote the amount of GC work performed during the last GC cycle. The GC function is now constructed by drawing a line with the derived gradient through the point (F_{min}, W_{max}). The initial part of the GC function is a line from the origin to the point where the line through (F_{min}, W_{max}) intersects with the W_{last} level. A line from the intersection to (F_{min}, W_{max}) constitutes the latter part of the GC function. The resulting GC function is shown in Figure 4. The GC function thus constructed guarantees that GC will not stop the application from meeting its deadlines and improves at the same time the average-case performance. This strategy can also be applied to interactive systems to remove the rare long delays caused by large variations in the amount of live memory.

6 Comparison to previous work

Here, it is discussed how the proposed scheduling technique relate to two methods reducing the potentially disruptive impact of garbage collection.

6.1 Incremental generation scavenging

Generation scavenging [Ung84] relies on the observation that objects tend to die shortly after being created. Objects surviving this initial period of high mortality tend, on the other hand, to remain live for an extended period of time. New objects are therefore allocated in a separate, relatively small, section of the heap called the *young generation*. When an object is determined to be a long-lived object it is moved to another section of the heap called the *old generation*. The two sections of the heap is garbage collected separately. Using this scheme, most GC work can be confined to the young generation, which can be garbage collected quickly due to its small size and high garbage ratio. The old generation will only have to be garbage collected occasionally with longer delays as a result. The total amount of GC work can thus be reduced.

Incremental versions of generation scavenging exist [Ben90, LH83] which use algorithm similar to Brook's to reclaim the garbage in each generation. They combine the low-overhead advantage of generation scavenging with the very short GC-induced pauses of incremental algorithms. However, the GC work is scheduled using the static scheduling technique described in Section 2. Thus, they too suffer from the problem with clustering of GC work early in the GC cycles. The adaptive scheduling technique proposed in this paper should be directly applicable to these versions of generation scavenging.

6.2 The train algorithm

An example of a garbage collector using a variant of adaptive GC work scheduling is the train algorithm [HEM92] as implemented by Seligmann and Grarup [SG95]. The train algorithm is an improvement of the original generation scavenging technique where the old generation is garbage collected by a non-incremental garbage collector, resulting in occasional long delays. The old generation is divided into a number of equal-sized sections called *cars*. The cars are organized into ordered sequences called *trains*, which in turn are ordered. On each invocation, the garbage collector evacuates all live objects from the first car in the first train to other cars, reclaiming the space occupied by the car. The algorithm is thus incremental, but have a much coarser granularity than traditional incremental algorithms such as Brook's algorithm.

Evacuated objects, as well as new objects, are continuously added to the end of the existing trains as cars are reclaimed. The train algorithm can therefore not be said to work in a cyclic manner in the same sense as Brook's algorithm, which first evacuates all live objects from fromspace and then suspends itself until tospace fills up. Thus, the main problem is not to distribute the GC work evenly over the GC cycle, but rather to choose an invocation frequency that is high enough to guarantee that there will always be free memory available, but not so high that it wastes time unnecessarily moving live object around in memory. Seligmann and Grarup have chosen to let the application define a fixed *acceptable garbage percentage*. The actual percentage of garbage objects in the old generation is periodically estimated and the invocation frequency of the collector is modified. When the percentage of garbage is larger than the acceptable garbage percentage the invocation frequency is increased and vice versa.

7 Conclusions and future work

Incremental garbage collection algorithms often suffer from bumpiness since too much work is performed too early in the GC cycle. The cause of this is that the estimated worst-case memory need, which is used to determine the GC rate, correlates badly with the actual amount of used memory. The GC work can be distributed more evenly over the GC cycle if adaptive scheduling is introduced. Adaptive scheduling can lead to occasional long GC-induced pauses when the amount of live memory varies violently, but this is unlikely to happen in interactive applications since they tend to have only small variations in their memory usage needs. Adaptive scheduling can also be applied to real-time systems in order to improve the average-case behaviour of the system.

This work is still in a very early stage. To verify that the assumptions on the memory usage of interactive applications hold, a range of interactive applications should be studied. The memory usage traces thus acquired should then be used to simulate the behaviour of an adaptive garbage collector. Finally, a real implementation should be made and tested with actual interactive applications.

References

[Bak78] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4), April 1978.

[Ben90] M. Bengtsson. Real-Time Garbage Collection Algorithms. Licentiate thesis, Dept. of Computer Science, Lund University, 1990.

[Bro84] R. A. Brooks. Trading Data Space for Reduced Time And Code Space in Real-Time Garbage Collection on Stock Hardware. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*. Austin, Texas, August 1984.

[Hen96] R. Henriksson. *Scheduling Real-Time Garbage Collection*. Licentiate thesis, Dept. of Computer Science, Lund University, January 1996.

[HEM92] R. L. Hudson, J. Eliot, and B. Moss. Incremental Collection of Mature Objects. In *Proceedings of IWMM'92*, Springer-Verlag, LNCS 637. St. Malo, France, September 1992.

[LH83] H. Lieberman and C. Hewitt. A real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6), June 1983.

[SG95] J. Seligmann and S. Grarup. Incremental Mature Garbage Collection Using the Train Algorithm. In *Proceedings of ECOOP'95*. Aarhus, Denmark, August 1995.

[Ung84] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(3), May 1984.