Guaranteeing Non-Disruptiveness and Real-Time Deadlines in an Incremental Garbage Collector (corrected version)

Fridtjof Siebert
Oberfeldstr. 34B
76149 Karlsruhe, Germany
siebert@jamaica-systems.de

Abstract

For Garbage Collection (GC) to be a generally accepted means of memory management it is required to prove its efficiency. This paper presents a scheme that guarantees that an incremental Garbage Collector will have completed its collection cycle before the system runs out of memory. Furthermore, it is shown that the work that has to be done by the collector in one incremental step is limited by a small constant depending on the percentage of total memory used by the application program. This result then allows a suitable trade-off between memory demand and GC overhead to be found.

1. Introduction

The reason for applying an incremental Garbage Collector (as opposed to a disruptive one) is to reduce pauses imposed by the collector to a minimum and instead spread the collection work over small increments. This should be done so as to ensure that the collector makes sufficient progress that it is always able to satisfy an allocation by the application program without an unreasonably long delay.

Most Garbage Collection algorithms have to do a complete traversal of all or part of the memory allocated by the application before any garbage may be detected and considered free. This traversal and the freeing of the encountered garbage is called a GC cycle. An incremental collector splits the work to be done during a GC cycle into many tiny parts that are done during the execution of the application program, either in parallel in a separate thread or by occasionally stopping the application to do some GC work.

In this paper, memory is measured in units. These may be bytes, the minimum object size, whole objects (in a system with only a single size-class) or any other memory unit. We assume (for now) that our heap has a fixed size of M units. The part of the memory that is known to be free will be called F. The amount that has not been proven to be free is called allocated: $A = M - F$. The memory actually used by the application program is the reachable memory R. So we have the following variables describing the memory usage:

	M	*total memory (always known to the GC)*
	F	*free memory (always known to the GC)*
(1)	$A = M-F \leq M$	*allocated memory (always known to the GC)*
(2)	$R \leq A$	*reachable memory (usually unknown at run time)*

To simplify the analysis, we normalise these values:

	$m = M/M = 1$	*normalised total memory*
(3)	$f = F/M$	*normalised free memory*
	$a = A/M = 1-f$	*normalised allocated memory*
	$r = R/M \leq a$	*normalised reachable memory*

During a GC cycle, the collector does one or a small constant number of scans through the allocated memory A (or at least through the reachable memory R). Possible algorithms are presented in [3] for a Mark and Sweep collector or in [2] for a copying collector. The details of the collection algorithm and its incremental implementation are not dealt with in this paper. There are only two requirements: The collector has to guarantee to find all garbage that existed at the beginning of the collection cycle (this implies that it is exact); and during each increment of GC work progress (measured in a fraction of scanned memory) is made so that at the latest after scanning all allocated memory, a GC cycle is finished. In more detail, these requirements are:

We define g_c to be the amount of garbage found (and therefore memory recycled) by the collector during GC cycle c:

G_c	*amount of garbage found during GC cycle c*
$g_c = G_c/M$	*normalised amount of garbage found during GC cycle c*

We assume that at the beginning of GC cycle c we have A_c units of allocated memory and R_c units of reachable memory. The amount of garbage is $A_c – R_c$. Any useful incremental GC algorithm has to guarantee that, during cycle c, it will detect at least those garbage objects that were garbage at the beginning of the cycle, but it may as well find garbage that was created during the cycle. So we have

$G_c \geq A_c - R_c$	*memory that is garbage at the beginning of cycle c is guaranteed to be found during cycle c*
$g_c \geq a_c - r_c$	*normalised* G_c

When cycle c is finished, the garbage detected is taken from the allocated memory a and added to the free memory f. During this cycle, the application program will continue running and allocate Q_c units of memory.

Q_c	*amount of memory allocated by application during cycle c*
$q_c = Q_c/M$	*normalised* Q_c

At the end of one cycle, the garbage found will be added to the free memory, while the amount allocated during the cycle has become allocated memory. So we can use

(4)	$a_{c+1} = a_c + q_c - g_c$	*allocated memory at the beginning of the cycle c+1*

to determine the amount of memory allocated at the beginning of the next cycle.

A useful indicator for the progress made by an incremental step of the GC cycle is the amount of allocated memory scanned by the collector: Even if a scan has to traverse only the reachable memory R, we can still use the allocated memory A to measure progress, since R is usually unknown, but is known to be less than or equal to A. Our progress function may indicate less progress than is actually made, so the cycle might finish earlier than anticipated.

P_i	*progress made by collector in increment i (amount of allocated memory scanned)*

The collector has to guarantee that a GC cycle that started with increment i is finished when the sum of the progress made in each increment reaches the amount of allocated memory.

In this paper, free memory is considered to be available for any allocation, i.e., not fragmented and therefore unusable. This means that either the GC algorithm applied has to compact the memory, only a single size class can be used or the implementor must be optimistic enough to assume that fragmentation won't be a problem.

For the collector to be non-disruptive, we must ensure that the progress P_i to be made in each increment is small but also that it is big enough that the GC can guarantee that it finds and recycles sufficient garbage to satisfy all allocation requests done by the application. A useful function for P_i will be presented here.

2. Simple schemes for time distribution between application and collector

A simple scheme to distribute the available processing power between the application and the collector would be to allocate a fixed fraction of total CPU time to a collection process, e.g. *10%* or *25%*. On a multiprocessor system, it is also tempting to allocate one processor for the collector, while leaving the remaining ones for the main application. This scheme ensures that only a limited amount of time will be spent for GC and ensures non-disruptiveness during normal execution. But it does not at all ensure that sufficient GC work will be done for the system not to run out of free memory, so the danger of having to halt the application for a complete GC cycle persists. Its likelihood can be reduced by increasing the percentage of time spent on GC, but there is no guarantee that sufficient progress will be made.

Furthermore, this method will spend a fixed fraction of CPU time garbage collecting, even if this is not required because the application uses little memory or does not do any allocations for a long period of time. Since no GC work has to be done as long as the application does not do any allocations, a better scheme might base the amount of work to be done by the collector on the amount of allocation done by the application. Here, a fixed ratio can be selected as well: for each allocation of n units of memory, we can have the collector scan $p \cdot n$ units of allocated memory, where p is a constant like *3* or *10*. This policy avoids wasting time doing GC while the application is not doing any allocations, and the incremental work to be done by the collector on an allocation is small and bounded. But we still cannot guarantee that the system won't run out of free memory so that a complete GC cycle becomes necessary.

As shown in [6], for an application with a maximum amount of reachable memory l (i.e., the amount of reachable memory is always less than this constant: $r \leq l$), completion of the GC cycle can be guaranteed by selecting $p \geq 2 \cdot l/(m-l)$. Note that in [6] it is assumed that objects that are allocated during the GC cycle don't have to be scanned (are allocated 'black') and that only reachable objects have to be scanned, while some collectors require to scan all allocated memory (as for the sweep phase of a Mark and Sweep collector).

Since l depends on the application program, this solution cannot be applied directly in a generic Garbage Collector that has no special knowledge about the application program, as in a Java virtual machine [1].

For a Garbage Collector that requires to scan all allocated memory, and that might have to scan objects allocated during one GC cycle, completetion and sufficient progress can be guaranteed by selecting $p = 2/(1-(l/m))$[1]. In such a system, not all memory m has to be allocated from the operating system at the beginning, instead these allocations can be delayed until an allocation cannot be satisfied using memory freed by the collector. The guarantee for sufficient GC progress then guarantees that the ratio l/m will never exceed the value preselected for calculation of p. So, when $l/m = 1/3$ was selected, which results in $p = 3$, the total memory requested from the operating system will never exceed three times the reachable memory. Accordingly, the amount of reachable memory can be guaranteed to be at least 90% of the total memory by selecting $p = 20$. This allows for a trade-off between GC and memory overhead even for applications with unknown memory requirement l, but the constant factor p will cause a constant and high GC overhead that is required only during the time of highest memory demand by the application.

3. Flexible time distribution between application and collector

To find a better function for the amount of progress the GC has to do, we want to exploit the information available on the current state of the memory. Since the collector is required to make progress only when the application allocates memory, we want to derive a function $P()$ for the progress to be made on the allocation of one unit of memory.

An easily determinable indicator for the state of the memory is the current amount of free memory F. The allocation of one unit of memory reduces the free memory by just this unit: $F_{i+1} = F_i - 1$. We want to avoid running out of free memory, so we want to finish a GC cycle (and hopefully free some memory) before F becomes zero. That means we have to finish the cycle during the next F allocations. We do not want to make an assumption on what state the GC is in; in the worst case it has just begun a new cycle. A complete cycle might mean scanning all allocated memory A. If we wanted to spread the work to be done during this cycle evenly over the next F allocations, we would have to scan at least A/F units of memory now. Additionally, the memory allocated during the cycle might have to be scanned as well, so we have to add one unit of memory in the required progress function and we get

$$P = (A/F) + 1 \quad \textit{Progress the GC must make on allocation of one unit of memory}$$

With a few conversions using (1) and (3) we get $P()$ as a function of the free memory f and allocated memory a:

$$(5) \quad P(f) = 1/f$$
$$P(a) = 1/(1-a)$$

With this simple scheme, the work to be done by the GC is very small as long as there is sufficient free memory,

[1] The memory allocated at the beginning of one cycle a_c is limited by

$$(1) \qquad a_c \leq l \cdot ((p-1)/(p-2))$$

this can be shown by induction, prooving (1) to hold after cycle c for a_{c+1} assuming the precondition that it holds for $a_0 = 0$ and a_c: The garbage found during cycle c is $g_c \geq a_c - l$. Cycle c needs q_c allocations to terminate, with $q_c \leq a_c/(p-1)$ (one is subtracted from p to allow scanning of memory allocated during this cycle). So we get

$$\begin{aligned} a_{c+1} &= a_c + q_c - g_c \\ &\leq a_c + a_c/(p-1) - (a_c - l) \\ &\leq l/(p-2) + l \\ &= l \cdot ((p-1)/(p-2)) \end{aligned}$$

The total memory allocated is limited by the amount of memory allocated at the beginning of one cycle a_c plus the maximum amount allocated during a cycle q_c:

$$\begin{aligned} a &\leq a_c + q_c \leq a_c + a_c/(p-1) = a_c \cdot (p/(p-1)) \\ &= l \cdot ((p-1)/(p-2)) \cdot (p/(p-1)) = l \cdot (p/(p-2)) \\ &= l/(1-2/p) \end{aligned}$$

With $p = 2/(1-(l/m))$, we get

$$\begin{aligned} a &\leq l/(1-2/(2/(1-(l/m))) \\ &= l/(1-(1-(l/m))) = m \end{aligned}$$

Which completes the proof.

but it increases hyperbolically as we run out of memory. Figure 1 shows the graph for $P(a)$.

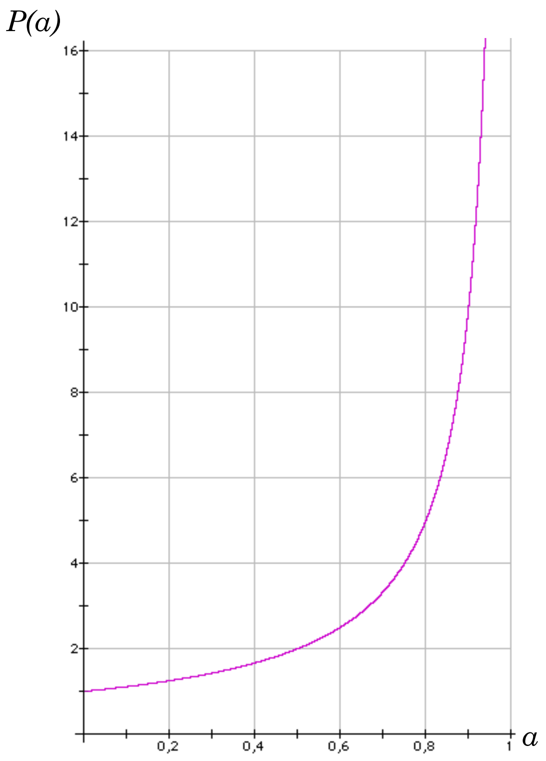

Figure 1: Progress to be made by collector on allocation as a function of allocated memory: $P(a) = 1/(1-a)$.

In the next section, it will be shown that, using this scheme for any application whose reachable memory is always less than the total memory, sufficient memory will be freed before the free memory is exhausted. Next, an upper bound for *P()* will be given that shows that the actual overhead to be done on every allocation is bounded and small.

4. Guaranteeing sufficient progress

For a collector to be non-disruptive or real-time, the overhead of each incremental step must be limited, but it also has to be ensured that the progress made on these incremental steps is sufficient to free some allocated memory before the free memory is exhausted. Here, we first want to show that sufficient progress is guaranteed by requiring a minimal constraint on the application program.

We want to guarantee that we never run out of free memory, i.e.,

(6) $f > 0$

always holds. Assume that an application program uses the total memory as reachable objects at some time t :

$$r_t = m$$

With *(1)* and *(2)*, we see that all memory is allocated and none is free:

$$a_t = m$$
$$f_t = 0.$$

This violates constraint *(6)*, so in this case the collector has no way to guarantee never to run out of free memory. For the collector to be able to ensure constraint *(6)*, we therefore have to require that at no time all memory is allocated as reachable objects, i.e.,

$$r < m$$

must be guaranteed by the application. This means there exists an upper bound k for r with

(7) $r \leq k \qquad \text{with } 0 \leq k < 1.$

The fraction of memory used by the application as reachable objects must be at most $k < 1$. But this factor is not known by the collector.

For the following analysis, we need a further constraint on k:

(8) $k \geq \frac{1}{4}$

Requiring this lower bound for k imposes no additional requirement on the application program.

Assume we have a_1 allocated memory when a new collection cycle starts. We want to determine how much memory q_1 has to be allocated for the sum of the progresses made on each allocation to be sufficient to complete this cycle. To do this, we determine a function $U(a_1)$ that gives an upper limit for q_1. With the progress function given in *(5)* and written as a function of the allocated memory a

(9) $P(a) = 1/(1-a)$

the collection cycle will be finished as soon as the cumulative progress made by the collector exceeds the initially allocated memory a_1 plus the amount q_1 of memory allocated meanwhile:

$$a_1 + q_1 \le \int_{a_1}^{a_1+q_1} P(a)\cdot da$$
$$\le \int_{a_1}^{a_1+q_1} 1/(1-a)\cdot da$$
$$= ln(1-a_1) - ln(1-a_1-q_1)$$

so we get for q_1:

$$a_1 + q_1 = ln((1-a_1)/(1-a_1-q_1)))$$

which is equivalent to

$$(q_1 - 1 + a_1)\cdot e^{(q_1-1+a_1)} = (a_1-1)/e$$

this equation can be solved using Lambert's $W()$ function [7], which is the solution of $w\cdot e^w = x$, such that $w = W(x)$. So we get as an upper limit for q_1

$$q_1 = U(a_1)$$

with

$$U(x) = 1 - x + W((x-1)/e)$$

Whenever a collection cycle starts while the amount of allocated memory is a_c, this cycle will be finished after $U(a_c)$ memory has been allocated. Figure 2 illustrates this function.

We now want to show by induction that for every GC cycle c, the amount of allocated memory a_c at the beginning of the cycle will be less or equal to $k+U(k)$:

(10) $\forall_c\, a_c < a_{begin\_max} := k + U(k)$

The graph shown in Figure 2 gives us a motivation for constraint *(8)*: For $k \ge \frac{1}{4}$ the function $U(k)$ is falling. This fact is required in the proof.

We distinguish two cases:

First case:

(11) $a_c \le k$

We cannot guarantee to find any garbage during this cycle since all allocated memory could be reachable, constraint *(7)* is of little help:

$$g_c \ge a_c - r_c \ge 0$$

The amount of memory to be allocated before the next cycle finishes is limited by $U(a_c)$:

$$q_c \le U(a_c) = 1 - a_c + W((a_c - 1)/e)$$

Applying *(4)* gives us for the allocated memory after this cycle:

$$\begin{aligned} a_{c+1} &= a_c + q_c - g_c \\ &\le a_c + q_c \\ &\le a_c + U(a_c) \\ &= a_c + 1 - a_c + W((a_c - 1)/e) \\ &= 1 + W((a_c - 1)/e) \end{aligned}$$

since $W(x)$ increases monotonically for $x>1/e$, we get:

$$\begin{aligned} &\le 1 + W((k - 1)/e) \\ &= k + 1 - k + W((k - 1)/e) \\ &= k + U(k) \\ &= a_{begin\_max} \end{aligned}$$

Second case:

(12) $k < a_c \le a_{begin\_max}$

Since $r_c \le k < a_c$ implies $r_c < a_c$, we will find some garbage in this cycle:

$$g_c \ge a_c - r_c \ge a_c - k > 0$$

The amount of memory to be allocated before the next cycle finishes is

$$q_c \le U(a_c) = 1 - a_c + W((a_c - 1)/e)$$

Applying *(4)* gives us for the allocated memory after this cycle:

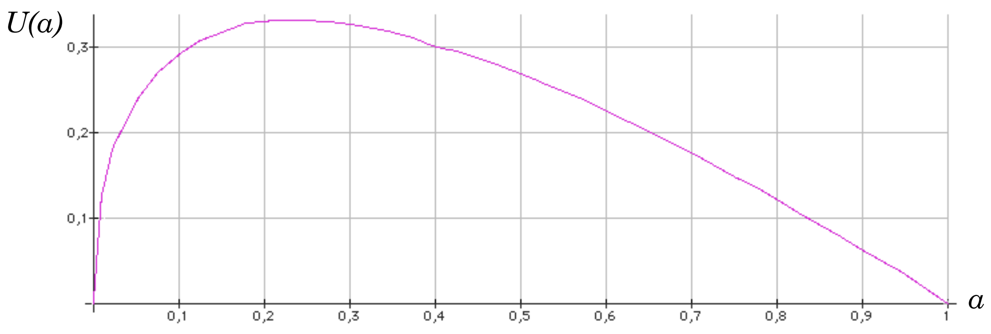

Figure 2: Amount of allocation required to complete GC cycle: $U(a) = 1 - a + W((a-1)/e)$

$$\text{(13)}\quad a_{c+1} = a_c + q_c - g_c \le a_c + U(a_c) - (a_c - k) = k + U(a_c)$$

(8) guarantees that *U(x)* is falling monotonically for $k \le x \le 1$, the derivative U'(x) is negativein this range of values.

This monotonicity and *(12)* allow us to continue the estimation for a_{c+1} from *(13)*

$$a_{c+1} \le k + U(a_c) \le k + U(k) = a_{begin\_max} \qquad \text{with } k \ge \frac{1}{4} \text{ and } a_c > k$$

The starting condition $a_0 = 0$ and induction over all $c > 0$ yields that a_c will never surmount a_{begin_max}.

This completes the proof.

Now that we have a proven upper bound for the amount of allocated memory at the beginning of each cycle, as given in *(10)*, we would like to have an upper bound of allocated memory for the application independent of the state of the collector.

Since every cycle takes at most $U(a_c)$ allocations to finish, the upper bound on a_c implies an upper bound of allocated memory at the end of every cycle, and therefore a total upper bound for the allocated memory:

$$\text{(14)}\quad a \le a_{max} = a_{begin\_max} + U(a_{begin\_max}) = k + U(k) + U(k + U(k)) = 1 + W(W((k-1)/e)/e)$$

Figure 3 illustrates this upper bound for the maximum amount of allocated memory a_{max} as a function of the upper limit of reachable memory *k*. Table 1 presents the maximum amounts of allocated memory for some selected maximum fractions of reachable memory.

***k*:**	0.0	0.3	0.4	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95	0.975	1.0
a_{max}:	0.0	0.838	0.876	0.906	0.931	0.952	0.961	0.970	0.978	0.985	0.993	0.997	1.0

Table 1: Upper bound for the maximum amount of allocated memory

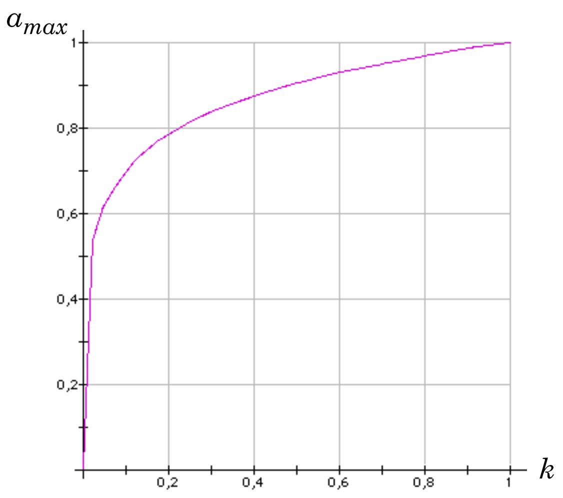

Figure 3: Upper bound for the maximum amount of allocated memory $a_{max} = k + U(k) + U(k + U(k))$

5. Upper bound for work to be done by GC on allocation

With the upper bound for the allocated memory *a* as given in *(14)*, we can use (5) to determine an upper bound for the progress the current GC cycle has to make on the allocation of one unit of memory:

$$\text{(15)}\quad P_{max} = 1/(1-a_{max}) = 1/((1-k)\cdot e^{(1-k)\cdot e^{-k}-1-k})$$

Figure 4 shows this required progress as a function of the maximum amount of reachable memory. Some values for P_{max} are given in Table 2.

For the example of *k=80%*, an application that uses at most *80%* of the total memory as reachable objects, on the allocation of one unit of memory at most *27.65* units of allocated memory must be scanned by the collector. The average amount of memory that has to be scanned is signifacantly lower: the maximum amount is reached at

***k*:**	0.0	0.3	0.4	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95	0.975	1.0
P_{max}:	1.0	6.173	8.085	10.67	14.47	20.71	25.67	33.10	45.45	70.11	144.0	291.8	∞

Table 2: Upper bound for required GC progress per unit of memory allocated

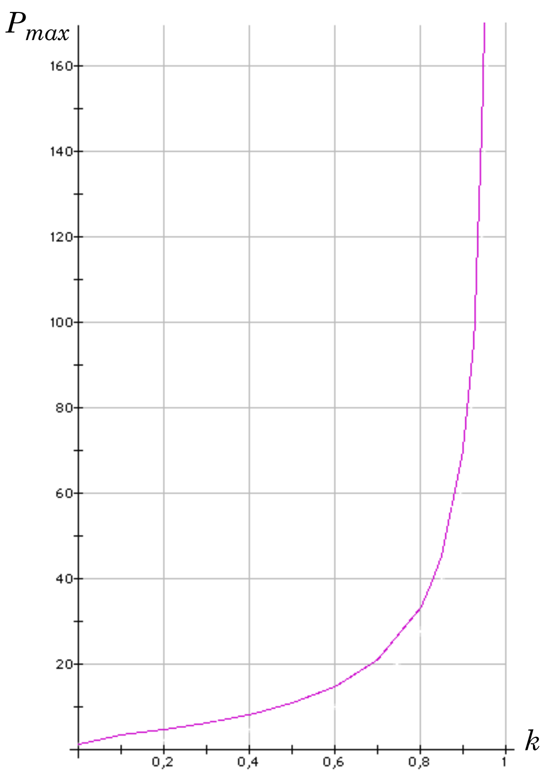

Figure 4: Upper bound for required GC progres per unit of memory allocated

$P_{max}(k) = 1/W(W((k - 1)/e)/e)$

the end of one GC cycle that reaches the maximum extent of allocated memory. At the beginning of this cycle, only *11.13* units need to be scanned and at the beginning of the next cycle, at most 6.84 units need to be scanned. Of course, whenever the amount of reachable memory is less than 80% of the total memory, less collection work will be required accordingly.

6. Exploiting these results

Applying this flexible scheme to control the collector's work seems to be especially beneficial for implementation of modern general purpose programming languages like Eiffel [4] or Java [1] that require efficient garbage collection and are applied to a wide and unknown variety of applications.

A straightforward implementation of the scheme will result in very little garbage collection overhead for most applications, while the overhead increases with the memory requirements of the application. Only applications that use all or nearly all of the available memory will encounter a noticeable slowdown by the collector. The scheme has the pleasant feature that adding memory to such a system will prevent this slowdown and will also reduce the required GC work on all applications.

Another application of this scheme is to find a reasonable trade-off between garbage collection time and memory requirement. Whenever the required progress to be made by the collector reaches a certain limit, the memory manager might decide that it would be better to allocate some additional memory from the operating system instead of doing too much collection work itself. The relation between the maximum amount of allocated memory a_{max} and k as shown in *(14)*, Table 1 and Figure 3 can serve as an upper limit for the amount of unused memory:

Assume that the work to be done by the garbage collector on an allocation of one unit of memory should never be more than scanning $P = 15.72$ units of memory. This will be the case when the allocated memory reaches $a = 0.936$. From Figure 3 we see that this amount of allocated memory can not be attained by an application whose reachable memory is less than 70% of the total memory. In this situation, at least 70% of the total memory must have been reachable by the application; the amount of unused memory is at most 30%.

If the memory manager decides to allocate more memory from the operating system now, it can guarantee the upper limit of $P = 15.72$ for the collection work, while also guaranteeing not to waste more than 30% of the total memory (assuming that the amount allocated from the operating system is small compared to the total memory).

Table 3 shows this trade-off between memory usage and GC overhead for several selected values of P_{max}, a_{max}, k_{min} and wasted memory w_{max},. If we allow a significant GC overhead of scanning up to 285,8 units of memory per unit allocated, we can reduce the amount of wasted memory to 2,5%; while limiting the collection work to 6,61 times the allocated memory, up to 50% of the total memory might be unused by the application program.

A detailed description of an Eiffel implementation and how the presented scheme can be used for its incremental Mark and Sweep collector is shown in [5]. This description also deals with many implementation dependend aspects that were not subject to this paper, like incremental scanning of the root set or an efficient write-barrier. The implementation of the collector is not finished yet so that practical experiences with it are still missing.

We expect the implementation to perform comparable or even better than existing incremental Garbage Collec-

P_{max}:	1.0	6.173	8.085	10.67	14.47	20.71	25.67	33.10	45.45	70.11	144.0	291.8	∞
a_{max}:	0.0	0.838	0.876	0.906	0.931	0.952	0.961	0.970	0.978	0.985	0.993	0.997	1.0
k_{min}:	0.0	0.3	0.4	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95	0.975	1.0
w_{max}:	100%	70%	60%	50%	40%	30%	25%	20%	15%	10%	5%	2.5%	0%

Table 3: Trade-off between maximum GC overhead P_{max} and wasted memory w_{max}. Also shown is maximum amount of allocated memory a_{max} in this case and lower bound k_{min} for the reachable memory of the application program.

tors. But unlike currently used techniques that reduce the frequency or likelihood of disruptive breaks caused by the collector, an implementation using the presented scheme can guarantee that disruptive breaks do not occur. Furthermore, an adequate trade-off between GC overhead and memory requirements can be found easily.

7. References

[1] Ken Arnold and James Gosling: *The Java Programming Language*, 2nd edition, Addison Wesley, 1998

[2] Henry G. Baker: *List processing in Real Time on a Serial Computer.* Communications of the ACM 21,4 (April 1978), p. 280-294.

[3] Edsgar W. Dijkstra, L. Lamport, A. Martin, C. Scholten and E. Steffens: *On-the-fly Garbage Collection: An Exercise in Cooperation,* Communications of the ACM, 21,11 (November 1978), p. 966-975.

[4] Bertrand Meyer: *Eiffel: The Language,* Prentice Hall International (UK) Ltd, Hertforshire, 1992

[5] Fridtjof Siebert: *Implementierung eines Eiffel-Compilers für SUN/SPARC*, Diplomarbeit Nr. 1484, Institut für Informatik, Universität Stuttgart, 1997

[6] Paul R. Wilson: *Uniprocessor Garbage Collection Techniques.* Submitted to the ACM Computing Surveys. 1998.

[7] R.M. Corless, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey, D. E. Knuth: *On the Lambert W Function*, Advances in Computational Mathematics, Volume 5, 1996, pp. 329-359.