Automatic Heapmanagement and Realtime Performance

H. Corporaal
Delft University of Technology
Faculty of Electrical Engineering
Mekelweg 4, P.O. Box 5031
2600 GA Delft, The Netherlands

Abstract

The debate whether to make heapmanagement partly or fully invisible to the programmer is still continuing. Although nobody disputes the programming advantages of automatic heapmanagement, many people believe that the runtime overhead of the necessary automatic garbage collection is unacceptable. On top of that, it seems that tight realtime constraints can no longer be met.

Recent progress has shown that the collection overhead can be minimised to a few percent by concentrating collection to specific storage areas (e.g., generation scavenging). This paper focusses on a particular garbage collection method, the incremental copying collector, which will probably be part of many future heapmanagement systems, because it can easily be adapted to make use of storage areas on stock hardware. Besides, this method can be tuned to meet severe realtime constraints. A detailed worst case time and space analysis, illustrated by a practical example, make clear how this can be done.

Keywords: heaporiented systems, realtime performance, garbage collection, incremental collection.

1 Introduction

Automatic heapmanagement relieves the programmer from reclaiming unused heapspace; reclaiming is done by a so called garbage collection process, which now and than steals cycles from the application in order to fulfill its job. Many language runtime systems do not incorporate automatic heapmanagement because of implementation complexity or expected runtime overhead. However developments in the late eighties showed that garbage collection can be efficiently implemented on stock hardware, with runtime overhead well below 10 percent (see e.g. [15,7,2,17]). The core of many of those implementations consists of a combination of a copying, compacting garbage collection algorithm with some sort of an area based tracing approach ([5]) 1. Using an area based approach may even enhance application behaviour by increasing the spatial locality. Hopefully these developments will encourage runtime designers to incorporate these sophisticated collection methods, even in more traditional oriented languages 2. This will raise the programming level.

One area however where garbage collection is considered totally unsuitable is formed by realtime systems. It is indeed hard to meet severe realtime constraints when the collection process is allowed to run several seconds at unpredicted moments. This paper focuses on these problems, in particular for the class of incremental copying collectors. Several people have studied performance of garbage collectors [16,3,4,9,10,17], however most of them analyse average behaviour, or use benchmarks to estimate performance. This paper gives a formal analysis of the worst case time and space behaviour. An example system is used to demonstrate how garbage collection can be tuned in order to meet realtime constraints. How far we succeed in this will be evaluated in the final section, where remaining problems are summarized. We assume the readers familiarity with the principles of garbage collection, especially the copying collection method. For an overview of the garbage collection design space see [6].

2 The garbage collector cycle

Standard stop & copy collectors divide memory into two equal parts called from- and to-space. Allocation is done in to-space until this space becomes full. The garbage collector then copies all reachable objects from the full to the empty part, after which the two spaces change their role (a flip), and the application process (called the mutator) may continue. The collection process starts with copying those objects which are referenced by the roots (this is called a root copy). From the roots all active objects can directly or indirectly be reached by following pointer chains. For the purpose of this paper we assume that roots live in registers, but in practise the stack and symbol table may also be part of the roots. At the old location of a copied object a forwarding pointer to its new location is left. If a root contains a reference to an already copied object (as can be detected by the forwarding pointer), the object is not copied, but the reference to this object is replaced by the forwarding pointer. This is called shorting. After the root copy tracing of the pointer fields of already copied objects starts, which will again cause objects to move. This process continues until all objects are copied.

Copying all reachable objects may take quite a while and therefore disturbs the application process. To overcome this the incremental copying collector was created ([3]). This collector traces only a limited number of objects (called the trace effort) after the root copy, and before every heap space allocation done by the mutator.

The time distribution of the incremental copying collector is shown in figure 1.

Figure 1: Time distribution of the incremental collector

The time distribution can be viewed as a function g(t), that gives the fraction of processor cycles used by the garbage collector at any time. The collector cycle starts at a flip at t_0 (in the sequel we...
assume \( t_0 = 0 \). The root copy ends at \( t_r \) (which is equal to the response time of this collector), after which the mutator and collector each run for short periods of time (interleaved mode), until all live objects are copied (at \( t_{free} \)). The system then enters free mode. The performance \( P(t) \) is defined by:

\[
P(t) = \frac{1}{t} \int_0^t 1 - g(r) \, dr
\]

where \( t \) is the time passed after a flip. Performance indicates the amount of useful work per unit time which can be done in a given time interval. Performance defined this way is a far better indicator for realtime behaviour than the response time (see [6] for an explanation). Figure 2 shows the performance for this type of collector.

![Figure 2: Performance of the incremental collector](image)

It will be clear that the average performance of the incremental collector is much higher than of the stop & copy variant. For very large intervals the performance converges to the long-term performance, which is called \( P_{\infty} \).

The bursty part of the function \( g(t) \) occurs during the interleaved mode. For worst-case performance considerations, the part of the function can be approximated by pretending that the collector and mutator do not alternate in this mode, but run concurrently. The collector speed in this mode is called \( s_{im} \) (the mutator speed then of course is \( 1 - s_{im} \)).

The collector speed in one cycle then becomes:

\[
g(t) = \begin{cases} 
1 & (0 \leq t \leq t_r) \\
\frac{s_{im}}{t_r} & (t_r < t < t_{free}) \\
0 & (t_{free} \leq t < t_{flip})
\end{cases}
\]  

(2)

Note that for the stop & copy collector \( t_r = t_{free} \). So, Equation 2 applies to both copying collectors. The actual values of \( t_r \), \( t_{free} \), \( t_{flip} \), and \( s_{im} \) depend on the datastructure in the heap.

### 3 Analysis of Time performance

The time analysis is based on the parameters listed in Table 1.

<table>
<thead>
<tr>
<th>Description</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of live objects in the heap</td>
<td>( N )</td>
</tr>
<tr>
<td>Heap size in words</td>
<td>( H )</td>
</tr>
<tr>
<td>Number of roots</td>
<td>( R )</td>
</tr>
<tr>
<td>Size of an object in words</td>
<td>( W )</td>
</tr>
<tr>
<td>Trace effort in object/second</td>
<td>( T )</td>
</tr>
<tr>
<td>Allocation rate in objects/time unit</td>
<td>( A )</td>
</tr>
<tr>
<td>Heap copy speed</td>
<td>( C_{ch}(W) )</td>
</tr>
<tr>
<td>Root/heap copy ratio</td>
<td>( \alpha(W) )</td>
</tr>
<tr>
<td>Short/heap copy ratio</td>
<td>( \beta(W) )</td>
</tr>
</tbody>
</table>

It is assumed that the heap is in equilibrium state. This means that both the number of objects allocated per time unit (\( A \)) and the number of live objects (\( N \)) in the heap are constant in time. It is also assumed that all objects of the same size (\( W \)).

The next subsections go into the details of time performance. To obtain formulas for realtime performance we have to consider the worst case situation. Therefore first the worst possible heap datastructure is derived. The following subsection calculates the collection and mutator time based on this datastructure. Finally the obtainable long-term performance is defined and calculated.

#### 3.1 The worst case heap structure

Section 2 described what happens when the heap becomes full and a flip occurs. First all objects that can be reached directly from the roots are copied to to-space. When there are few roots holding an object pointer, the root copy will be short. Few pointers per object means that the heap trace need not copy a lot. In that case the response time is short, and collector bursts are also short. This will result in a high realtime performance. In the opposite case, when all roots contain a pointer to an object, and all traced objects also contain pointers to objects in from-space, the garbage collection bursts are of maximal length and the realtime performance will be low. So, the performance of a garbage collector appears to depend on the structure of the data in the heap.

In this section a data structure will be devised that will cause a minimal realtime performance for the incremental copying collector. It is assumed that copying an object from a pointer in a root takes less time than from a pointer in an object in the heap, and that shorting a forwarding pointer in an object in the heap takes less time than copying an object from a pointer in the heap. This will be true for every implementation of a copying collector.

To create a worst case structure of \( N \) objects we start with one object in the first root and add objects one by one until all \( N \) objects are placed, choosing a place according to the following rules:

- First find a place where it has the greatest impact on the initial collector burst.
- Else, choose a place where it affects the next burst most; and so on for each following burst.
- When all objects are placed, fill the objects that do not contain pointers to objects with pointers to an arbitrary object.

With a trace effort \( T \), the first \( T \) objects that are copied to the heap are traced in the initial garbage collector burst. These objects must be copied there by the tracing of the first \( T \) registers. To obtain a maximal length for the initial burst, the objects that are traced must all contain pointers that cause an object to be copied. So, we place a pointer to the first object in the first register. Pointers to new objects are placed in that object until it is full. This same process is repeated until the first \( T \) roots are filled. Adding new objects will not contribute to the heap trace of the initial burst. Therefore pointers to new objects are placed in the remaining registers to add to the root copy. The structure that is created by these steps is shown in Figure 3.

What happens when this structure has to be garbage collected? First, the \( R \) roots are traced. The objects 1, 4, 7 and 8 are copied to the new heap. If the trace effort \( T \) is 2, 2 objects in the heap are traced (1 and 4), causing objects 2, 3, 5 and 6 to be copied. Now the initial burst has ended. The next objects to be traced would be objects 7
and 8. So, the next pointers to new objects are added into 7 and then into 8. The structure can be expanded by adding pointers to new objects starting at the top-right in the figure (object 2) and going down.

When all $N$ objects are added, all objects that do not yet contain a pointer to another object are filled with pointers to objects. Because there are no new objects that can be added, these pointers must point to objects that are already copied, causing a shorting of a forwarding pointer when it is traced. The pointers added in this last phase can well be recurrent pointers to the object that contains them.

The resulting structure has the following properties: (1) the initial collection burst takes maximal time, (2) the next heap traces first result in copying of all live objects and (3) the final heap traces do not copy objects; they only result in shorting forwarding pointers.

### 3.2 Garbage collection and mutator time

The time to copy an object generally depends on the size $W$ of the object. There are three different actions that the garbage collector can perform on objects in the heap:

- Copying an object that is pointed to by a register
- Copying an object that is pointed to by another object in the heap.
- Shorting a forwarding pointer in an object in the heap.

The times needed for these actions are $t_{cr}$, $t_{ch}$ and $t_{fh}$ respectively. Instead of using $t_{cr}$, $t_{ch}$ and $t_{fh}$ directly, the related parameters $C_{ch}(W) = 1/t_{ch}(W)$, $\alpha(W) = t_{cr}(W)/t_{ch}(W)$ and $\beta(W) = t_{fh}/t_{ch}(W)$ are used. The advantage of these parameters is that both $\alpha(W)$ and $\beta(W)$ are dimensionless parameters and depend only on the algorithm of the collector. Only $C_{ch}$ depends on the speed of the collector (and memory access).

The total time in one cycle used by the garbage collector to copy and update all live objects from one heap to the other is given by $t_{gc}$, the garbage collection time.

$$t_{gc} = N_c t_{cr} + N_{ch} t_{ch} + N_{fh} t_{fh}$$

where $N_c$ is the number of copies from a root, $N_{ch}$ the number of copies from a pointer in the heap, and $N_{fh}$ the number of forwarding pointers that are shorted in the heap. These parameters depend on the structure of the data in the heap.

To find the worst case garbage collection time these parameters must be found for the worst case heap structure. If $N \geq WT + R$, the number of forwarding pointers in the heap grows regularly with $N$, and is given by

$$N_{fh} = R + N(W - 1)$$

Note that $N \geq WT + R$ means that the first garbage collection burst is filled completely. The assumption that $N \geq WT + R$ is not a restriction on the use of this calculation, because real systems will certainly have parameter values satisfying this equation.

When the initial burst is filled, the number of copies from a root is equal to the number of roots $N$, so $N_c = R$. There are $N$ objects that must be copied to the other heap, therefore $N_{ch} = N - R$ must be copied from the heap.

Using this, equation 3 transforms into:

$$t_{gc} = \frac{Ra(W) + (N - R) + (R + N(W - 1))\beta(W)}{C_{ch}(W)}$$

This equation shows that the time to collect all garbage only depends on the data in the heap and the collector speed; it does not depend on the size of the heap.

The copying collectors use only half of the available heap space to allocate new objects, the other half is used for garbage collection. So, when the heap size is $H$ words and the size of an object is $W$ words, there is room for $\frac{H}{W}$ objects. When there are $N$ live objects, only $\frac{H}{W} - N$ new objects can be allocated. Under equilibrium conditions the allocation rate is constant, and equal to $A$. This means that the mutator allocates $A$ objects per unit time. So, the available heap space is allocated by the mutator in

$$t_{m} = \frac{\frac{1}{2}H - NW}{AW}.$$  

This is the mutator time, the time per cycle available to the mutator.

**Response time.**

Realtime response is a special case of realtime performance ($P(t) > 0$ for all $t > t_{max}$). In the first burst after a flip, all roots and $T$ objects in the heap are traced. In the worst case all these traces result in copying an object. The maximum length of this burst (the response time) is therefore

$$t_r = \frac{Ra(W) + WT}{C_{ch}(W)}, \quad N \geq WT + R$$

The response time therefore depends on the trace effort.

The interleaved mode. The interleaved mode starts immediately after the first collector burst and ends when all objects are copied to the other heap. With $N$ objects and a trace effort $T$, there are $[N/T]$ garbage collector bursts. The number of mutator bursts is then $[N/T] - 1$. With an allocation rate $A$, the length of a mutator burst is $1/A$. Therefore

$$t_{free} = t_{gc} + \frac{1}{A} \left( \left[ \frac{N}{T} \right] - 1 \right)$$

During interleaved mode, starting at $t_r$ and ending at $t_{free}$, the time used for collecting garbage is $t_{gc} - t_r$. So, the average collector

$^4$For the moment we assume all roots live in registers.

$^5$[x] means round up x to the nearest integer.

$^6$Remember that in worst case exactly $N$ objects have to be copied. On average this could be less, because mutator actions can make objects inactive during the collection period. Many implementations however, like in [27], do not allow the mutator to see references to from-space. In that case always $N$ objects are copied.
There are two equal semi-spaces, so the system can never be greater than \( N \) objects. With maximal tracing (\( T = N \), stop & copy collection), the smallest possible heap must be

\[
H_{\text{min}} = 2W(N + 1)
\]

and hence the maximal occupation is

\[
\eta_{\text{max}} = \frac{N}{2(N + 1)}.
\]

When the trace effort is smaller than \( N \), the garbage collection system can fail even for occupations smaller than \( \eta_{\text{max}} \). This happens when the heap becomes full before all live objects are copied to-space. From Equations 13 and 14, the occupation at which the system fails with a given trace effort (\( \eta_{\text{fail}} \)) is

\[
\eta_{\text{fail}} = \frac{N}{2[N(1 + \frac{1}{T})]}
\]

When \( N \gg T \) this becomes

\[
\eta_{\text{fail}} = \frac{T}{2(T + 1)}
\]

When just one object is traced per allocated object (\( T = 1 \)), the minimal heap even becomes \( 4NW \). So the maximal possible occupation grows with the trace effort.

### 3.3 The long-term performance

The long-term performance \( P_m \) is defined as the fraction of the processing time spent by the mutator when the total processing time \( t \) goes to infinity.

\[
P_m = \lim_{t \to \infty} \frac{t_{\text{mutator}}}{t}
\]

Taking \( t \) as a multiple of \( t_{\text{flip}} \), and using that \( t_{\text{flip}} = t_{gc} + t_m \), \( P_m \) becomes:

\[
P_m = \lim_{n \to \infty} \frac{n_{\text{mutator}}}{n(t_{gc} + t_m)} = \frac{t_m}{t_{gc} + t_m}
\]

Sometimes the long-term performance is called the productivity of the system, the fraction of processor time that the system is actually doing work ([10]). It is clear from this equation that \( P_m \) can reach the desired value of 1 arbitrary close, by increasing the amount of available heap memory. See in this respect [1], where it is argued that heap allocation can be cheaper than stack allocation.

### 3.4 Analysis of Space performance

The space performance of a collector describes how efficient the collector uses memory. It is also of importance how much memory the collector needs to maintain \( N \) live objects in the heap. For this purpose the occupation \( \eta \) of the heap is used. The occupation is defined as the fraction of the heap that holds live objects,

\[
\eta = \frac{N}{H}
\]

On systems with an incremental collector, memory can be allocated even when the collector is copying objects to the other heap space. When the heap becomes full, all live objects must be copied from-space to to-space or the system can not continue. When there are \( N \) live objects in the heap, and the trace effort of the collector is \( T \), all \( N \) objects are copied after \([N/T]\) allocations. Then the heap contains the \( N \) copied objects and the \([N/T]\) allocated objects. There are two equal semi-spaces, so the heap must be twice the size of a semi-space. Therefore, the smallest heap size is:

\[
H_{\text{min}} = 2W \left[ N \left(1 + \frac{1}{T}\right)\right]
\]

This means that incremental copying never uses less memory than stop & copy. When there are \( N \) live objects to copy, the trace effort can never be greater than \( N \) objects. With maximal tracing (\( T = N \), stop & copy collection), the smallest possible heap must be

\[
H_{\text{min}} = 2W(N + 1)
\]

and hence the maximal occupation is

\[
\eta_{\text{max}} = \frac{N}{2(N + 1)}.
\]

When the trace effort is smaller than \( N \), the garbage collection system can fail even for occupations smaller than \( \eta_{\text{max}} \). This happens when the heap becomes full before all live objects are copied to-space. From Equations 13 and 14, the occupation at which the system fails with a given trace effort (\( \eta_{\text{fail}} \)) is

\[
\eta_{\text{fail}} = \frac{N}{2[N(1 + \frac{1}{T})]}
\]

When \( N \gg T \) this becomes

\[
\eta_{\text{fail}} = \frac{T}{2(T + 1)}
\]

When just one object is traced per allocated object (\( T = 1 \)), the minimal heap even becomes \( 4NW \). So the maximal possible occupation grows with the trace effort.

### 5 Example heap management systems

Up till now we have derived several equations describing the behaviour of the incremental copying collector. This section applies the theory to 2 example heap oriented systems. In the first example the long-term performance has to exceed a certain lower bound; the second example considers a real-time constraint.

#### Long-term performance example

Suppose a heap oriented system with a garbage collector that is used to process batch jobs. There are no constraints on the real-time performance of the system, but the long-term performance of the system must be better than 80%. The jobs running on the system have an allocation rate of 25.0 objects/ms, and use 40 roots. The average number of live objects in the system is \( 10^5 \). Assume the object size is 3 words/object and every word contains a pointer. Which heap size is required to meet the desired performance?

Because the real-time behaviour is not important in this case, a stop & copy collector is chosen because it requires a smaller heap than the incremental collector. The collector has the following parameters:

\[
\alpha(3) = 0.8 \\
\beta(3) = 0.3 \\
C_{ch}(3) = 1000.0 \text{ objects/ms}
\]

The time required for a complete garbage collection of the heap is given by Equation 5: \( t_{gc} = 1600 \text{ms} \). Using equations 6 and 12 with the requirement that \( P_m \geq 0.8 \), the minimal required heap size can be calculated:

\[
H \geq 6.96 \times 10^5
\]

So, the heap must be at least \( 6.96 \times 10^5 \) words, and therefore the occupation \( \eta < 0.431 \).
Realtime performance example.

Suppose the same system, but now we require realtime behaviour. It is required that a job, using $W_{j, a} = 0.1 \text{ms}$ of processing time, will be completed within 1 ms. So the realtime performance has to satisfy: $P(1\text{ms}) \geq 0.1$.

The total garbage collection time is still 1600 ms. Because $t_{gc} > t_{j, ab} - W_{j, ab}$, a stop & copy collector can obviously not be used. Which trace effort and heap size are required to meet the realtime constraints?

The response time of the incremental collector is (Equation 7):

$$t_r = (32 + 3T) 10^{-3} \text{ ms}.$$  

The response time must obey

$$t_r \leq t_{j, ab} - W_{j, ab} = 0.9 \text{ ms} \Rightarrow T \leq 289$$

Any $T$ for which $1 \leq T \leq 289$ looks feasible, however, the realtime constraint gives an extra restriction on $T$. Because $t_r < t_{j, ab} < t_s$, $P(t)$ can be rewritten, using equations 1 and 10, as

$$P(t) = \frac{t_s - t_{gc, r}}{t_s} \times \frac{t - t_r}{t}.$$  

If we use the fact that $t_s > t_r$, this equation can easily be solved for $T$, giving $T < 83$. Therefore, it is not allowed to trace complete pages at ones, as some implementations require.

Table 2 shows the minimal heap size $H_{\text{min}}$, the maximal occupation $\eta_{\text{full}}$, the long-term performance $P_m$, and the realtime performance $P(1\text{ms})$.

<table>
<thead>
<tr>
<th>$T \times 10^3$</th>
<th>$\eta_{\text{full}}$</th>
<th>$P_m$</th>
<th>$P(1\text{ms})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.25</td>
<td>0.96</td>
<td>0.90</td>
</tr>
<tr>
<td>2</td>
<td>0.33</td>
<td>0.93</td>
<td>0.84</td>
</tr>
<tr>
<td>3</td>
<td>0.375</td>
<td>0.89</td>
<td>0.78</td>
</tr>
<tr>
<td>4</td>
<td>0.40</td>
<td>0.86</td>
<td>0.74</td>
</tr>
<tr>
<td>5</td>
<td>0.42</td>
<td>0.83</td>
<td>0.69</td>
</tr>
<tr>
<td>10</td>
<td>0.45</td>
<td>0.71</td>
<td>0.54</td>
</tr>
<tr>
<td>50</td>
<td>0.49</td>
<td>0.33</td>
<td>0.17</td>
</tr>
<tr>
<td>100</td>
<td>0.495</td>
<td>0.20</td>
<td>0.08</td>
</tr>
</tbody>
</table>

From this table we see that we can trade both real and long-term performance for memory. A good choice for this system is to take $T \geq 4$, requiring a heap space of $H = 7.5 \times 10^4$. This also gives an acceptable long-term performance. Figure 4 shows the performance for $T = 50$ and $H = 7.5 \times 10^4$.

6 Conclusions and Shortcomings

It appears to be possible to create a mathematical model that represents the worst case behaviour of the two semispace copying collector, whose parameters can easily be derived. The model can be used to predict the behaviour of a collector or to dimension a collector to special realtime constraints.

The analysis and the measurements we have done to check the model [12] reveal a number of interesting properties:

1. Realtime performance appears to include more than the response time of the collector or the fraction of the time that is needed for garbage collection. In fact the complete distribution of the garbage collection effort is of vital importance for a realtime garbage collector.

2. The time needed to collect all live objects only depends on the size and the structure of the data in the heap, and is therefore independent of the size of the heap. This is because only accessible (non-garbage) objects are traced; garbage objects are not accessed.

3. The required heap size only depends on the trace effort and the product $NW$ – the space occupied by the live objects. The allocation rate, collector speed and the structure of the data have no influence on it. So, a heap that is big enough to hold the live objects under some combination of circumstances can always hold those objects, even when the structure of the data or the allocation rate (mutator speed) changes (as long as the trace effort remains constant). This gives the Baker collector an enormous advantage over the current parallel garbage collectors; these fail if the mutator speed is too high for the collector to follow. Because of the interleaved operation of the Baker collector, its collector effort is increased when the mutator speed increases.

4. The realtime performance is high for low trace efforts, but the minimally required heap must be large. So, realtime performance can be traded for memory space. Note that the size of the heap itself has no direct influence on the realtime performance, but a large heap allows for a low trace effort to be used, which in turn increases the realtime performance. The trace effort has no effect on the long term performance. The heap size does affect the long-term performance. A large heap means high long-term performance. This is because the garbage collection time is independent of the heap size, but the length of a garbage collector cycle increases for a larger heap.

This paper showed how to derive realtime performance for the standard Baker algorithm. For practical implementations one has to be aware of the limitations of the presented model. Below we list several shortcomings and indicate possible refinements and improvements:

1. The model assumes objects of equal size. In practice varisized objects are used. Especially long objects (e.g. arrays) can destroy realtime performance. In that case objects must be partially copied during collection (see [14]).

2. The initial root copy can take very long if the roots include the stack and static data areas. These roots have to be traced incrementally.

3. It is possible to leave out the initial tracing directly after the root copy. This shortens the response time. Note that in such a case the root copy interval can become shorter than a normal garbage collection burst. When this happens, the longest garbage collection interval (which determines the response time) does not start on a flip, but at the longest normal burst after the flip.

4. Sometimes objects must be traced because of mutator actions. Remember, that the mutator does not have access to objects in from-space. This tracing can either be accounted for by the collector, in which case it reduces the next heartbeat, or it has to be considered as part of the work done by the mutator. In the latter case, the time calculated for time-critical jobs has to be increased, because of possible access delays.

5. We assumed constant mutator allocation rate. An allocation burst gives rise to very low realtime performance. However the mutator is written by the programmer himself; so he has more or less control on the allocation rate.

We have studied this collector, because we think it will be an ingredient of many fancy future garbage collectors. One of the most
promising extensions is to use several generations containing objects of different age. Older (and often larger) generations are collected less often; they contain more stable objects. Using incremental copying collection for all generations automatically generates this behaviour, because collection is tight to allocation. Another extension of copying collectors is the use of different areas of activity [7]. Objects traced because of mutator actions are put into high activity areas, objects traced by the collector into lower activity areas. This way, the application locality and therefore the performance can be improved considerably. Our method of calculating realtime performance remains applicable for these extensions.

References