# A Generational On-the-fly Garbage Collector for Java

Tamar Domani* Elliot K. Kolodner† Erez Petrank‡

**Abstract**

An *on-the-fly garbage collector* does not stop the program threads to perform the collection. Instead, the collector executes in a separate thread (or process) in parallel to the program. On-the-fly collectors are useful for multithreaded applications running on multiprocessor servers, where it is important to fully utilize all processors and provide even response time, especially for systems for which stopping the threads is a costly operation.

In this work, we report on the incorporation of generations into an on-the-fly garbage collector. The incorporation is non-trivial since an on-the-fly collector avoids explicit synchronization with the program threads. To the best of our knowledge this incorporation has not been tried before. We have implemented the collector for a prototype Java Virtual Machine on AIX, and measured its performance on a 4-way multiprocessor.

As for other generational collectors, an on-the-fly generational collector has the potential for reducing the overall running time and working set of an application by concentrating collection efforts on the young objects. However, in contrast to other generational collectors, on-the-fly collectors do not move the objects; thus, there is no segregation between the old and the young objects. Furthermore, on-the-fly collectors do not stop the threads, so there is no extra benefit for the short pauses obtained by generational collection. Nevertheless, comparing our on-the-fly collector with and without generations, it turns out that the generational collector performs better for most applications. The best reduction in overall running time for the benchmarks we measured was 25%. However, there were some benchmarks for which it had no effect and one for which the overall running time increased by 4%.

**Keywords:** Programming languages, Memory management, Garbage collection, Generational garbage collection.

## 1 Introduction

Garbage collectors free the space held by unreachable (dead) objects so that this space can be reused in future allocations. On multiprocessor platforms, it is not desirable to stop the program and perform the collection in a single thread on one processor, as this leads both to long pause times and poor processor utilization. Several ways to deal with this problem exist, the two most obvious ways are:

1. Concurrent collectors: Running the collector concurrently with the mutators. The collector runs in one thread on one processor while the program threads keep running concurrently on the other processors. The program threads may be stopped for a short time to initiate and/or finish the collection.

2. Parallel collectors: Stopping all program threads completely, and then running the collector in parallel in several collector threads. This way, all processors can be utilized by the collector threads.

*IBM Haifa Research Lab. E-mail: tamar@il.ibm.com.

†IBM Haifa Research Lab. E-mail: kolodner@il.ibm.com.

‡Computer Science Dept., Technion - Israel Institue of Technology. This work was done while the author was at the IBM Haifa Research Lab. E-mail: erez@cs.technion.ac.il.

In this paper we discuss a concurrent collector; in particular, an *on-the-fly* collector that does not stop the program threads at all.

The study of on-the-fly garbage collectors was initiated by Steele and Dijkstra, et al. [27, 28, 8] and continued in a series of papers [9, 14, 3, 4, 20, 21] culminating in the Doligez-Leroy-Gonthier (DLG) collector [11, 10]. The advantage of an on-the-fly collector over a parallel collector and other types of concurrent collectors [1, 13, 24], is that it avoids the operation of stopping all the program threads. Such an operation can be costly. Usually, program threads cannot be stopped at any point; thus, there is a non-negligible wait until the last (of many) threads reaches a safe point where it may stop. The drawback of on-the-fly collectors is that they require a write barrier and some handshakes between the collector and mutator threads during the collection. Also, they typically employ fine-grained synchronization, thus, leading to error-prone algorithms.

Generational garbage collection was introduced by Lieberman and Hewitt [23], and the first published implementation was by Ungar [29]. Generational garbage collectors rely on the assumption that many objects die young. The heap is partitioned into two parts: the young generation and the old generation. New objects are allocated in the young generation, which is collected frequently. Young objects that survive several collections are "promoted" to the older generation. If the generational assumption (i.e., that most objects die young) is indeed correct, we get several advantages:

1. Pauses for the collection of the young generation are short.
2. Collections are more efficient since they concentrate on the young part of the heap where we expect to find a high percentage of garbage.
3. The working set size is smaller both for the program, because it repeatedly reuses the young area, and the collector, because it traces over a smaller portion of the heap.

## 1.1 This work

In this paper we present a design for incorporating generations into an on-the-fly garbage collector. Two issues immediately arise. First, shortening the pause times is not relevant for an on-the-fly collector since it does not stop the program threads. Second, traditional generational collectors partition the heap into the generations in a physical sense. Namely, to promote an object from the young generation to the old generation, the object is *moved* from the young part of the heap to the old part of the heap. On-the-fly garbage collectors do not move objects; the cost of moving objects while running concurrently with the program threads is too high. Thus, we have to do without it.

Demers, et al. [6] presented a generational collector that does not move objects. Their motivation was to adapt generations for conservative garbage collection. Here, we build on their work to design a generational collector for the DLG on-the-fly garbage collector [11, 10].

We have implemented this generational collector for our JDK 1.1.6 prototype on AIX, and compared its performance with our implementation of the DLG on-the-fly collector. Our results show that the generational on-the-fly collector performs well for most applications, but not for all. For the benchmarks we ran on a multiprocessor, the best reduction in overall program runtime was 25%. However, there was one benchmark for which generational collection increased the overall running time by 4%.

Several properties of the application dictate whether generational collection may be beneficial for overall performance. First, the generational hypothesis must hold, i.e., that many objects indeed die young. Second, it is important that the application does not modify too many pointers in the old generation. Otherwise, the cost of handling inter-generational pointers is too high. And last, the lifetime distribution of the objects should not fool the partitioning into generations. If most tenured objects in the old generation are actually dead, no matter what the promotion policy is, then we will not get increased efficiency during partial collections. If collecting the old generation frees the same fraction of the objects as collecting the young generation, then we may as well collect the whole heap since we do not care about pause times. Furthermore, the overhead paid for maintaining inter-generational pointers will cause an increase in the overall running time of the application.

We used benchmarks from the SPECjvm benchmarks [25] plus two other benchmarks as described in Section 8.2. Benchmarks for which overall application performance improves with generational collection are Anagram (25% improvement), _213_javac (15% improvement) and _227_mtrt (10% improvement). The improvement for Multithreaded RayTracer ranges between 1%-16%, depending on the number of application threads running concurrently. The application that does not do well is _202_jess, for which there is a 4% increase in the overall running time. The two reasons for this deterioration are that lots of objects in the old generation have to be scanned for inter-generational pointers and that most of the objects that get tenured die (become unreachable) in the following full collection.

### 1.2 Card marking

Hosking, Moss and Stefanović [16] provide a study of write barriers for generational collection. Among other parameters, they investigate the influence of the card size in a card marking barrier on the overall efficiency. For most of the applications they measured, the best sizes for the cards were 256 or 512 bytes, and the worst sizes were the extremes, 16 or 4096 bytes.

Note that the advantage of small cards is that the indication of where pointers have been modified is more exact, and the collector does not need to scan a big area to find the inter-generational pointers that it needs on the card. However, small cards require more space for the dirty marks, which reduces locality.

In the process of choosing the parameters for our collector, we have run similar measurements with various card sizes. As it turns out, the behavior of an on-the-fly generational collector is different. The best choice for the card sizes is at one of the extremes, depending on the benchmark. We chose to set the card size to the minimum possible. This was the best for most benchmarks and not far from best for the rest. We suspect that the primary reason that our results differ from those of Hosking, et al. [16] is that our collector does not move objects. We provide the details in Section 8.5.3.

### 1.3 Techniques used and organization

We start with the state of the art DLG on-the-fly collector [11, 10], which we briefly review in Section 2. We then construct our generational collector similar to the work of Demers, et al. [6], presenting it in Section 3. We augment DLG to work better with generations, both by utilizing an additional "color" in Section 4 and also by using a color-toggle trick to reduce synchronization in Section 5. A similar trick was previously used in [21, 17, 7, 22, 19]. Our first promotion policy is trivial: promote after an object survives a single collection. We also study options to promote objects after several collections in Section 6 below. In Section 7 we provide the code of the collector and lower level details appropriate for an implementer. In Section 8 we report the experimental results we measured and justify our choice of parameters. We conclude in Section 9.

## 2 The collector

We build on the DLG collector [11, 10]. This is an on-the-fly collector that does not stop the program to do the collection. There are two important properties of this collector that make it efficient. First, it employs fine-grained atomicity. Namely, each instruction can be carried out without extra synchronization. Second, it does not require a write-barrier on operations using a stack or registers. The write barrier is required only on modifications of references inside objects in the heap.

The original papers also suggest using thread local heaps, but the design assumes an abundant use of immutable objects as in ML. We did not use thread local heaps.

We start with a short overview of the DLG collector. For a more thorough description and a correctness proof the reader is referred to the original papers [11, 10]. The collector is a mark and sweep collector that employs the standard three color marking method. All objects are white at the beginning of the trace, the root objects are then marked gray, and the trace then continues by choosing one gray object, marking it black, and marking all its white sons gray. This process continues until there are no more gray objects in the

heap. The meaning of the colors is: a black object is an object that has been traced, and whose immediate descendants have been traced as well. A gray object is an object that has been traced, but whose sons have not yet been checked. A white object is an object that has not yet been traced. Objects that remain white at the end of the trace are not reachable by the program and are reclaimed by the sweep procedure. Shaded (gray or black) objects are recolored white by sweep. A fourth color, blue, is used to identify free objects.

To deal with the fact that the collector is on-the-fly, i.e., it traces the graph of live objects while objects are being modified by the program, some adjustments to the standard mark and sweep algorithm are required. The collector starts the collection with three handshakes with the mutator threads. On a handshake, the collector changes its status, and each mutator thread cooperates (i.e., indicates that it has seen the change) independently. After responding to the first handshake, the write barrier becomes active and the mutators begin graying objects during pointer updates. The second handshake is required for correctness; the behavior of the mutators does not change as a result. While responding to the third handshake, each mutator marks its roots gray, i.e., the objects referenced from its stack The mutators check whether they need to respond to handshakes regularly during their normal operation. They never respond to a handshake in the middle of an update or the creation of an object. The collector considers a handshake complete after all mutators have responded. After completing the three handshakes, the collector completes the trace of the heap and then sweeps it.

The mutators gray objects when modifying an object slot containing a pointer until the collector completes its trace of the live objects. The amount of graying depends on the part of the collection cycle. Suppose a reference to an object $A$ is modified to point to another object $B$. Between the first and the third handshake, the mutator marks both $A$ and $B$ gray. After the third handshake and until the end of the sweep, the mutator marks only $A$ as gray.

The mutators also cooperate with the collector when creating an object. During the trace, objects are created black, whereas they are created white if the collector is idle. During sweep, objects are created black if the sweep pointer has not seen them yet (so that they will not be reclaimed). If the sweep pointer has passed them, they are created white so as to be ready for the next collection. If the sweep pointer is directly on the creation spot, the object is created gray. Some extra care must be taken here for possible races between the create and the sweep. However, a simple method of color-toggle allows avoiding all these considerations. We discuss it in Section 5 below.

# 3 Generational collection without moving objects

We describe an approach to generational collection that does not relocate objects. We call a collection of the young generation a *partial collection* and a collection of the entire heap a *full collection*.

Our design is similar to the Demers, et al. [6] design for a stop-the-world conservative collector. However, we incorporate features necessary to support on-the-fly collection: clearing the card marks without stopping the threads, an additional color for objects created during a collection and a color toggle to avoid synchronization between object allocation and sweep.

Instead of partitioning the heap physically and keeping the young generation in a separate place, we partition the heap logically. For each object, we keep an indication of whether it is old or young. This may be a one bit indication or several bits giving more information about its age.

The simplest version is the one that promotes objects after surviving one collection. We begin by describing this simpler algorithm. We discuss an aging mechanism in Section 6 below. Demers [6] notes that if an object becomes old after surviving one collection, then the black color may be used to indicate that an object is old. Clearly, before the sweep, all objects that survived the last collection are black. If we do not turn these objects white during the sweep, then we can interpret black objects as being in the old generation.

During the time between one collection and the next, all objects are created white and therefore considered young. At the next partial collection (i.e., collection of the young generation) everything falls quite nicely into place. During the trace, we do not want to trace the old generation, and indeed, we do not trace black

objects. During the sweep, we do not want to reclaim old objects, and indeed, we do not reclaim black objects. All live objects become black, thus, also becoming old for the next collection.

Before a full collection (a collection of the old and young generation), we turn the color of all objects white. Other than that, full collections are similar to partial collections.

### 3.1 Inter-generational pointers

It remains to discuss inter-generational pointers, pointers in old objects that point to young objects. Since we do not want to trace the old generation during the collection of the young generation, we must assume that the old objects are alive and treat the inter-generational pointers as roots.

How do we maintain a list of inter-generational pointers? Similarly to other generational collectors, we may choose between card marking [26] and remembered sets [23, 29]. (See [18] for an overview on generational collection and the two methods for maintaining inter-generational pointers.) In our implementation, we only used card marking. The reason is that in Java we expect many pointer updates, and the cost of an update must be minimal. Also, we did not have an extra bit available in the object headers required for an efficient implementation of remembered sets.

In a card marking scheme, the heap is partitioned into cards. Initially, the cards are marked "not dirty". A program thread (mutator) marks a card dirty whenever it modifies a card slot containing a pointer. The collector scans the objects on the dirty cards for pointers into the young generation; it may turn off a card mark if it does not find any such pointers on the card. Card marking maintains the invariant that inter-generational pointers may exist only on dirty cards.

The size of the cards determines a tradeoff between space and time usage. Bigger cards imply less space required to keep all dirty marks, but more time required by the collector to scan each dirty card to find the inter-generational pointers. We tried all powers of 2 between 16 and 4096 and found that the two extremes provided the best performance (see Section 8.5.3).

### 3.2 The collector

A partial collection begins by marking gray all young objects referenced by inter-generational pointers; in particular, the collector marks gray all white objects referenced by pointers on dirty cards. At the same time, all card marks are cleared. Clearing the marks is okay since all surviving objects are promoted to the old generation at the completion of the collection, so that all existing inter-generational pointers become intra-generational pointers. For a more advanced aging mechanism (as in Section 6) we would have to check to determine whether a card mark could be cleared.

After handling inter-generational pointers, all mutators are "told" to mark their roots using the handshake mechanism. This is followed by trace, which remains unchanged from the non-generational collector, and then sweep. Sweep is modified so that it does not change the color of black objects back to white.

A full collection begins by clearing card marks, without tracing from the dirty cards. The collector also recolors all black objects to white, allowing any unreachable object to be reclaimed in a full collection. After that, the mutators are "told" to mark their roots and the collector continues with trace and sweep as above.

### 3.3 Triggering

We use a simple triggering mechanism to trigger a partial collection. A parameter representing the size of the young generation is determined for each run, and a partial collection is triggered after allocating objects with accumulating size exceeding the predetermined size[1]. To trigger a full collection, we use the standard method of starting the concurrent collection when the heap is "almost" full.

[1] With our heap manager, we cannot trigger exactly at this time. Thus, the predetermined bound serves as a lower bound to the trigger time.

# 4 Dealing with premature promotion

When promoting all objects that survive a collection, there are infant objects created just before the start of the collection, which are immediately made old. These objects may die young, but they have already been promoted to the old generation, and we will not collect them until the next full collection. In an on-the-fly collection, objects are also created during the collection cycle; thus, compounding this promotion problem. We have added a simple mechanism to avoid promoting objects created during the collection to the old generation. A more advanced mechanism that keeps an age for each object is described in Section 6 below.

This is done by introducing a new color for objects created during a collection cycle. Instead of creating objects white or black depending on the stage of the collection as in the DLG algorithm, we create objects yellow during the collection. Yellow objects are not traced by the collector, and the sweep turns yellow objects back to white (without reclaiming them). Thus, the collector does not promote them to the old generation. One subtle point, which we discuss in the more technical section (see Section 7 below), forces an exception to the rule. In particular, between the first and the third handshakes of the collector, the mutators also mark yellow objects gray.

# 5 Using a color-toggle

Recall that during the collection, mutators allocate all objects yellow. Trace changes the color of all reachable white objects to black. In the design described so far, sweep reclaims white objects and colors them blue (the color of non-allocated chunks), and changes the color of yellow objects to white. Thus, at the end of the sweep, there are no remaining white objects.

Instead of recoloring the yellow objects, sweep can employ a color toggle mechanism similar to previous work [21, 17, 7, 2, 22, 19]. The color toggle mechanism exchanges the meaning of white and yellow, without actually changing the color indicators associated with the objects. Thus, live objects remain either black or yellow, and mutators go on coloring new objects yellow, so that yellow plays the role of white from the previous collection cycle. When a new collection begins, the mutators begin coloring new objects white, so that white begins playing the role of the yellow color from the previous cycle.

To implement the color toggle, we use two color names: the *allocation color* and the *clear color*. Initially, the allocation color is white, and the clear color is yellow. At all times, objects are allocated using the allocation color. At the beginning of the collection cycle, the values of the allocation color and the clear color are exchanged. In the first cycle this means that the allocation color becomes yellow and the clear color becomes white. During the trace, all reachable objects that have clear color are turned gray. Objects that have the allocation color are not traced and their color does not change. During the sweep, all objects with clear color are reclaimed.

Using this toggle we do not need to turn yellow objects into white during the sweep, but more important, we avoid the race between the create and the sweep. We do not need to know where the sweep pointer is in order to determine the color of a new object. A newly allocated object is always assigned the current allocation color.

**Remark 5.1** *Our discussion here is adequate for the generational collector, but one may easily modify the original collector to run with the same improvement by toggling the black and the white colors. In the comparison between a collector with and without generations, we feel that it is not fair to let only the generational collector enjoy this improvement. Therefore, we have also added this modification to the collector that does not use generations. Thus, the comparison we make has to do with generations only.*

# 6 An aging mechanism

In the algorithm described so far, the age indication is combined with the colors and we promote all objects that survive one collection. This promotion policy is extremely primitive, and the question is whether a

parameterized promotion policy may help. To do that, we keep an age for each object, i.e., the number of collections that it has survived. This age is initialized to 0 at creation and is incremented at sweep time. We also fix a predetermined parameter determining the threshold for promotion to the old generation. After an object reaches the threshold, the sweep procedure stops incrementing its age. We chose to fix a predetermined threshold, but dynamic policies could easily be implemented.

Using the aging mechanism, old objects continue to be colored black. However, the trace colors reachable objects black, whether they are young or old. Thus, a modification to sweep is required: sweep recolors reachable objects, which are young (age less than the threshold), to the allocation color, and continues to leave old objects black, and reclaim objects with the clear color. The pseudo-code for the sweep procedure appears in Figure 5.

Several changes to the card marking mechanism are also required to support aging. Simple clearing of the card marks at the beginning of each collection no longer works, since inter-generational pointers in the current collection cycle may remain inter-generational pointers in the next cycle. Furthermore, we must also ensure that inter-generational pointers are recorded correctly during the collection cycle. A race may occur between setting and resetting the card marks. We elaborate on the race in the more technical section, see Section 7 below.

At the beginning of a partial collection, the collector scans the card table and colors gray all young objects referenced by pointers on dirty cards. If no young object is referenced from a given card, then the collector clears the card's mark. Then the collector toggles the allocation and clear colors and continues with the handshakes, trace and sweep.

For a full collection, the collector does not trace inter-generational pointers. Instead, it recolors all black objects with the allocation color. Then it toggles the allocation and clear colors and continues with the handshakes, trace and sweep. In the initialization done before a full collection (see InitFullCollection in Figure 6) we do not clear the dirty bits. The reason is that they indicate dirty cards with inter-generational pointers that may still be relevant in the following partial collections.

An implementation question is where to keep the age. One option is with the object, and the other is in a separate table. We chose to keep it in a separate table. We did not have room in the objects headers. More importantly, note that sweep (for both partial and full collections) goes through the ages of all objects to increase them. Thus, for reasons of locality, it is better to go through a separate table, then to touch all the objects in the heap. We keep a byte per age (although two or three bits are usually enough). We could locate the age in the same byte with the card mark or with the color. However, that would require synchronization while writing the byte, e.g., via a compare and swap instruction. Empirical checks show that such synchronization is too costly for a typical Java application. Note that such a synchronized instruction would be required for a good fraction of all pointer modifications.

# 7 Some technical details

In this section we provide pseudo-code and some additional technical details. This paper is written so that the reader may skip this section and still get a broad view of the collector.

Our purpose in presenting the code is to show how the generational mechanism fits into the DLG collector. Thus, our presentation of the code concentrates on the details related to generations. We do not present details of the mechanism for keeping track of the objects remaining to be traced, nor do we present details of a thread-local allocation mechanism necessary to avoid synchronization between threads during object allocation. See the DLG papers [11, 10] for the details of these mechanisms. One other difference with DLG is that we separate the handshake into two parts, *postHandshake* and *waitHandshake*, instead of using a second collector thread.

Figure 1 shows the mutator routines, which are influenced by the collector: the write barrier (update routine), object allocation (create routine), and the cooperate routine, which the mutator must call regularly (e.g., backward branches and invocations). In the code the notation $heap[x, i]$ denotes slot $i$ of the object at address $x$. Figure 2 shows the overall collection cycle and in Figure 3 we present routines called by the

collector . We refer to the code below.

We assume that the reader is familiar with the DLG collector [11, 10], and we use the following terminology taken from their paper. The period between the first handshake and the second is denoted *sync1*, the period between the second handshake to the third is denoted *sync2*, and the rest of the time, i.e., after the third handshake and up until the beginning of the next collection cycle is denoted *async*. Each mutator has its own perception of these periods, depending on the times that it has cooperated with the handshake.

The most delicate issue for the generational collector is the proper handling of the card mark: how to set and reset it, properly avoiding races and maintaining correctness. We partition the discussion to the simple algorithm and to the aging algorithm. We assume a table with a designated byte for each card holding the card mark. The byte does not have any other use.

## 7.1 The simple algorithm

First, we consider the handling of the card marks for the simplest algorithm, without the yellow color or the color toggle, in particular the algorithm of Section 3. Using this algorithm, the collector marks all live objects black and promotes them. Thus, an inter-generational pointer can be created only after trace is complete. Thus, card marks can be cleared at the beginning of the cycle without fear of losing a mark due to a race condition with a mutator.

Now we add the yellow color (Section 4). The collector does not trace objects, which are created yellow during the cycle. Thus, it must keep a record of any pointer referencing a yellow object from any other object. (Actually, we are only interested in pointers from black objects, but we do not perform this filtering in our collector.) To solve the problem of keeping correct card marks for parents of yellow objects, it is enough to make sure that the order of operations at the beginning of a collection cycle is as follows: scan the card table and clear the dirty marks and only after that start creating yellow objects. Notice that *ClearCards* (code in Figure 3) precedes *SwitchAllocationClearColors* (code in Figure 3) in the collection cycle (code in Figure 2).

Next we add the color toggle (Section 5). There is a window of time between the check of an object $A$ for inter-generational pointers during the scan of the card table and the color toggle. If after the collector checks $A$, a mutator creates a new inter-generational pointer in $A$ referencing a yellow object $B$, the collector will miss this pointer during the current collection. Furthermore, after the color toggle, the object $B$ becomes white (i.e., having the clear color) and it might be collected in the current (partial) collection.

To solve this, we make an exception to the treatment of yellow objects by the DLG write barrier and treat them the same as white objects during sync1 and sync2 (between the first and third handshakes). This means that in this (usually short) period of time, whenever the DLG write barrier would shade a white object gray, it will also shade a yellow object gray. See *MarkGray* in Figure 1.

An additional point that needs to be verified is that the tracing always terminates. Without the yellow color modification, all (live) objects turn from white to gray and from gray to black. Since the number of live objects is finite, all of them turn black in the end, and the tracing always terminates. This is still the case here. A yellow object either stays yellow till the end of the trace, or it may turn gray and later black.

After performing these necessary modifications, we note that there is no need for card marking during sync1 and sync2. Thus, we get a small gain in efficiency: card marking is required only during the async stage. Notice that *MarkCard* is called only during async in the write barrier code in Figure 1.

To summarize, card marking occurs only during async. The clearing and checking of the card marks by the collector is done after the first handshake, and before the second handshake. After clearing the card marks, the collector toggles the (clear and allocation) colors; thus, mutators create new objects with the "yellow" color. Yellow objects may be shaded gray by the write barrier in sync1 and sync2.

## 7.2 The aging algorithm:

Next, we discuss the aging algorithm. Here, the collector must keep careful track of inter-generational pointers during all collector stages. We have two concerns. First, the choice of which card marks to clear

**Update(x,i,y):**
If ($status_m \neq async$) then
    MarkGray(heap[x,i])
    MarkGray(y)
  else if (Collector is tracing) then
    MarkGray(heap[x,i])
    MarkCard(x)
  else
    MarkCard(x)
heap$[x, i] \leftarrow y$

**Create:**
Pick $x \in free$.
$color[x] \leftarrow allocationColor$
Return $x$

**Cooperate:**
If ($status_m \neq status_c$) then
  If ($status_m = sync2$) then
    For each $x \in roots$:
      MarkGray(x)
    $status_m \leftarrow status_c$

**MarkGray(x)**:
If ($color(x) = clearColor$) or
  ($color(x) = allocationColor \wedge$
    $status_m \neq async$) then
    $color(x) \leftarrow gray$

Figure 1: The mutator routines

clear: If (full collection)
    InitFullCollection
  Handshake($sync1$)
mark: postHandshake($sync2$)
  ClearCards
  SwitchAllocationClearColors
  waitHandshake
  postHandshake($async$)
  mark global roots
  waitHandshake
trace : While there is a gray object:
    Pick a gray object $x$
      MarkBlack(x)
sweep : For each object $x$ in the heap:
    if ($color(x) = clearColor$)
      $free \leftarrow free \cup x$
      $color(x) \leftarrow blue$

Figure 2: The collection cycle

```
ClearCards:
For each card c:
        If (dirty(c)) then
            ClearCardMark(c)
            For each object x on c
                If (color(x) = black) then
                    color(x) ← gray

SwitchAllocationClearColors:
temp ← clearColor
clearColor ← allocationColor
allocationColor ← temp

InitFullCollection:
For each object x in the heap:
    If (color(x) = black ∨ color(x) = gray)
                                              then
            color(x) ← allocationColor
For each card c:
    ClearCardMark(c)

MarkBlack(x):
If (color(x) ≠ black) then
     For each pointer i ∈ x do:
         MarkGray(i)
      color(x) ← black

Handshake:
postHandshake(s)
waitHandshake

postHandshake(s):
status_c ← s

waitHandshake:
For each m ∈ mutators
     wait for status_m = status_c
```

Figure 3: The collector routines

must be done with care. Not all are reset. Second, at the same time that collector clears a card mark, a mutator may set it. In this case, we must make sure that the card mark remains set if there is a pointer in an object associated with this card to a young object.

To solve the first problem, the mutators set the card mark throughout the collection, also during sync1 and sync2 (see Figure 4). In order to clear the card mark, the collector checks first that no pointer to a young object exists on the card, and then clears the mark. However, there could still be a race between the clearing by the collector and the setting by the mutator.

In particular, the following interleaving of mutator and collector actions is problematic (say the dirty mark in question is associated with card $A$):

1. The collector thread scans card $A$, finds out that there is no inter-generational pointer and determines that the card's mark can be cleared.

2. Before the collector actually clears the mark, the program thread writes an inter-generational pointer into $A$ and sets the card mark.

3. The collector clears the card mark since its check from Step (1) allows this.

The outcome of this course of events is that an inter-generational pointer is now located on an unmarked card. In the next (partial) collection, the referenced object may be skipped by the trace and reclaimed although it is live. To solve this race we let the collector and mutator act as follows. The collector acts in three steps instead of the naive two steps. In Step 1, the collector resets the card mark. In Step 2, it checks whether the card mark can be cleared, i.e., whether there are no young objects referenced from $A$. Finally, in Step 3, if the answer of Step 2 is "no", the collector sets the card mark back on. (This idea is encoded in the ClearCards routine in Figure 6.) The update of the mutator involves two steps. In Step 1 it performs the actual update, and in Step 2 it sets the card mark. The order of steps is important in both cases. (This can be seen in the Update routine in Figure 4.)

We claim that the race is no longer destructive. Suppose a mutator is updating a slot on card $A$, storing an inter-generational pointer. We assume that before the update the object did not contain other inter-generational pointers; thus, it is crucial to get the new update noticed with respect to recording an inter-generational pointer. At the same time, the collector is checking whether the dirty bit of $A$ can be erased and erases it if necessary. We assume that all processors see the stores of a particular processor in the same order. There are two possible cases:

- **Case 1: The mutator sets the card mark before the collector clears it.** Since the mutator sets the mark after doing the actual update, the mutator must have performed the update before the collector cleared the card mark. Since the collector checks for inter-generational pointers after clearing the card mark, we get that the update was performed before the collector checked for inter-generational pointers. Thus, the collector's check will find the inter-generational pointer and the collector will set the card mark.

- **Case 2: The mutator sets the card mark after the collector clears it.** In this case, the card mark will remain set as required.

In summary, if a new inter-generational pointer is created, then the card mark will be properly set and this pointer will be noticed during subsequent collections.

# 8 Experimental results

Our goal is to compare the on-the-fly collector with and without generations, and to compare the effects of choices for the parameters governing the generational version, e.g., size of cards, size of young generation, use of aging, etc. We implemented both the original on-the-fly collector[2] and the generational on-the-fly

[2] For a fair comparison, we also introduced a black-white color toggle in the original on-the-fly collector

```
Update(x,i,y):
If (status_m ≠ async) then
        MarkGray(heap[x,i])
        MarkGray(y)
else if (Collector is tracing) then
        MarkGray(heap[x,i])
heap[x, i] ← y
MarkCard(x)

MarkGray(x):
If (color(x) = clearColor)
        color(x) ← gray
```

Figure 4: Aging version: modified mutator routines

```
clear: If (full collection)
           InitFullCollection
       Handshake(sync1)
mark: postHandshake(sync2)
       SwitchAllocationClearColors
       ClearCards
       waitHandshake
       postHandshake(async)
       mark global roots
       waitHandshake
trace : While there is a gray object:
            Pick a gray object x
               MarkBlack(x)
sweep : For each object x in the heap:
            if (color(x) = clearColor)
               free ← free ∪ x
               color(x) ← blue
            elseif(age(x) < oldestAge)
               color(x) ← allocationColor
               age(x) ← age(x) + 1
```

Figure 5: Aging version: The collection cycle

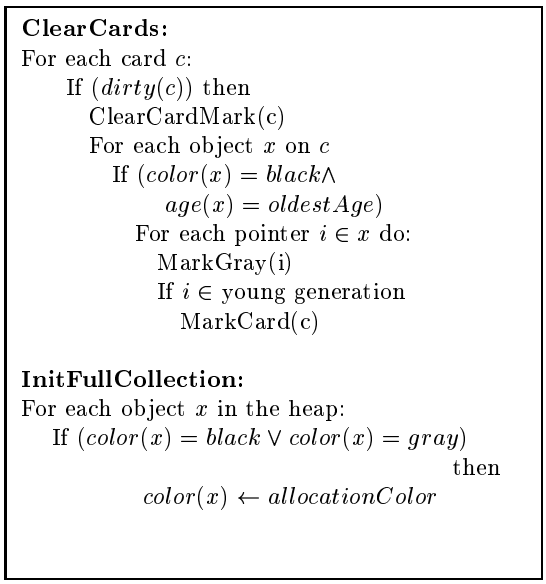


Figure 6: Aging version: modified collector routines

collector in a prototype AIX JDK 1.1.6 JVM. Measurements were done on a 4-way 332MHz IBM PowerPC 604e , with 512 MB main memory, running AIX 4.2.1. Additional measurements on a uniprocessor were run on a PowerPC with 192 MB main memory, running AIX 4.2.

All runs were executed on a dedicated machine. Thus, although elapsed times are measured, the variance between repeated runs is small. All runs were done with initial heap size of 1 MB and maximum heap size of 32 MB. The calculation of the trigger for a full collection was the same with and without generations. We verified that the working set for all runs fit in main memory, so that there were no effects due to paging.

## 8.1 Measuring elapsed time for an on-the-fly collector

A delicate point with an on-the-fly collector is how to measure its performance. If we run a single-threaded application on a multiprocessor, then the garbage collector runs on a separate processor from the application. If we measure the elapsed time for the application, we do not know how much time the collector has consumed on the second processor.

In a real world, the server handles many processes and the second processor does not come for free. In order to get a reasonable measure of how much CPU time the application plus the garbage collector actually consume, we ran four simultaneous copies of the application on our 4-way multiprocessor. This ensured that all the processors would be busy all the time, and the more efficient garbage collector would win. Each parallel run was repeated 8 times, and the average elapsed time was computed.

In addition, we measured the improvement of generational collection on a uniprocessor. This is not a typical environment for an on-the-fly collector, but it was interesting to check whether generations help in this case as well (and they usually do).

## 8.2 The benchmarks

Most of our benchmarks are taken from the SPECjvm benchmarks [25]. Descriptions of the benchmarks can be found on the Spec web site [25]. We ran all the SPECjvm benchmarks from the command line and not through the harness. For all tests we used the "-s100" parameter.

We also used two additional benchmarks. The first is an IBM internal benchmark called *Anagram* [15]. This program implements an anagram generator using a simple, recursive routine to generate all permutations

| No. of threads | 2 | 4 | 6 | 8 | 10 |
|---|---|---|---|---|---|
| Improvement | 1.3% | 2.6% | 10.6% | 16.0% | 11.7% |

Figure 7: Percentage improvement (elapsed time) for multithreaded Ray Tracer on a 4-way multiprocessor

| Benchmark | Multiprocessor Improvement | Uniprocessor Improvement |
|---|---|---|
| Anagram | 25.0 % | 32.7% |

Figure 8: Percentage improvement for Anagram

of the characters in the input string. If all resulting words in a permuted string are found in the dictionary, the permuted string is displayed. This program is collection-intensive, creating and freeing many strings.

The second is a code modification of the _227_mtrt [5] from the SPECjvm benchmarks [25] in order to make it more interesting on a multiprocessor machine. The program _227_mtrt is a variant of a Ray tracer, where two threads each render the scene in an input file, which is 340 KB in size [5]. _227_mtrt runs on matrices of $200 \times 200$ and uses 2 concurrent threads. We modified it to run on a bigger matrix of dimensions $300 \times 300$ and we also parametrized the number of rendering threads. We call this modification *multithreaded Ray Tracer*. The modified code is available on request for SPECjvm licensees.

## 8.3 The choice of parameters

For each application, a different choice of the parameters governing the generational collection seems to yield best performance. On the average, the best choice of parameters turns out to be object marking (i.e., card marking with 16 bytes per card) without the advanced aging mechanism and the best size of the young generation turns out to be 4 megabytes (we also tried 1, 2 and 8 megabytes for the young generation). In the next section (Section 8.4), we present results for this set of parameters. In Section 8.5 below, we justify our choice by comparing the performance of the algorithm with aging and for various settings of the other parameters.

## 8.4 The results

In Figure 7 we present the percentage improvement for the multithreaded Ray Tracer benchmark, described in Section 8.2 above. The number of application threads varies from 2 to 10. Generations perform very well for it.

Next, in Figure 8, we present the improvement generational collection yields for the Anagram benchmark. Here, generational collection is also very beneficial. In Figure 9 we examine the applications of the SPECjvm benchmark. As one may see, for most applications generations do well. We omit the results for the benchmarks _200_check and _222_mpegaudio, since they do not perform many garbage collections and their performance is indifferent to the collection method.

The performance of the benchmarks either gains a boost from generational collection or remains virtually unchanged, except for two benchmarks, _202_jess and _228_jack, which suffer a performance decrease. To account for the differences between the applications, we measured several runtime properties of these applications. As expected, an application performs well with generational collection if many objects die young and if pointers in the old generation do not get frequently modified. The decrease in performance for _202_jess and _228_jack originates from several reasons, some of them are shown in our measurements: First, the lifetime of objects was not typical to generations - they die soon after being promoted, unless one makes a huge young generation. Second, for _202_jess 36.2% of the objects that are scanned during partial collection are scanned because they are dirty objects in the old generation. This is a high cost for

| Benchmark | Multiprocessor Improvement | Uniprocessor Improvement |
|---|---|---|
| _227_mtrt | 7.0% | 25.2% |
| _201_compress | 0.0% | 2.0% |
| _209_db | -0.9% | 0.7% |
| _202_jess | -3.7% | -2.5% |
| _213_javac | 17.2% | 15.3% |
| _228_jack | -2.12% | -7.7% |

Figure 9: Percentage improvement for SPECjvm benchmarks

| Benchmark | Percent time GC active | No. partial GC | No. full GC | Percent time GC active w/o generations | No. of GC w/o generations |
|---|---|---|---|---|---|
| _227_mtrt | 21.5% | 36 | 0 | 30.5% | 26 |
| _201_compress | 1.7% | 5 | 15 | 1.2% | 17 |
| _209_db | 2.4% | 15 | 1 | 3.4% | 15 |
| _202_jess | 13.3% | 70 | 2 | 14.8% | 51 |
| _213_javac | 23.8% | 36 | 16 | 43.3% | 82 |
| _228_jack | 7.7% | 45 | 4 | 6.3% | 35 |
| Anagram | 62.8% | 152 | 8 | 78.9% | 56 |

Figure 10: Use of garbage collection in application.

manipulating inter-generational pointers. However, note that the success or failure of generational collector is influenced also by factors that we did not measure. For example, the increased locality of the heap, caused by frequent collections is hard to measure.

We now present measured properties from the runtime. In the remainder of this section, we present measurements of the applications properties. These measures were taken on the multiprocessor in running a single copy of the application. We start in Figure 10 with the amount of time spent on garbage collection. These numbers indicate how much a change in the garbage collection mechanism may affect the overall running time of the application. For example, the program that spends the most time garbage collecting during the run is Anagram, whereas programs that spend a small part of their time in garbage collection are _201_compress and _209_db. We also include the number of collection cycles executed in each of the applications.

Next, in Figures 11 and 12 we measure the "generational behavior" of the benchmarks involved. In particular, we measure how many objects are scanned during the collection, how many of them are scanned due to inter-generational pointers and what percentage of the objects are freed. For partial collection, we report what percent of the objects of the young generation that are collected. For the full collection, we report what percentage of the allocated objects in the whole heap that are reclaimed (allocated objects are counted as the sum of the objects freed and the objects that survive the collection). For example, in the benchmark _201_compress, objects do not tend to die young. However, for most of the other applications almost all objects die young. Next, we consider the maintenance of inter-generational pointers. We see, for example, that for _202_jess 36.2% of the objects scanned during partial collection are dirty objects in the old generation. This high cost for manipulating inter-generational pointers is one of the reasons for the deterioration in performance. Finally, we look at how many objects are reclaimed in partial and in full collections. For the applications _228_jack and _202_jess, objects that got tenured in the old generation did not survive long. We can see that almost all objects were collected during the full collections. This non-generational behavior is another reason why generations did not perform well for _202_jess and _228_jack. If non-generational collections can free a similar percentage of objects as partial collections, then we do not gain efficiency with the partial collections, whereas we do pay the overhead cost for maintaining inter-generational

| | Avg. No. of old objects scanned for inter-gen pointers | Avg. No. of objects scanned partial collections | Avg. No. of objects scanned full collections | Avg. No. of objects scanned in collection w/o generations |
|---|---|---|---|---|
| _227_mtrt | 280 | 1023 | N/A | 238703 |
| _201_compress | 3 | 168 | 4789 | 4778 |
| _209_db | 7 | 399 | 294534 | 287522 |
| _202_jess | 1373 | 3797 | 25411 | 25446 |
| _213_javac | 16184 | 53833 | 213735 | 194267 |
| _228_jack | 151 | 4890 | 14972 | 11241 |
| Anagram | 1 | 863 | 273248 | 271453 |

Figure 11: Generational characterization of the applications - Part 1.

| | percentage of bytes freed in partial collections | percentage of objects freed in partial collections | percentage of objects freed in full collections | percentage of objects freed in collections w/o generations |
|---|---|---|---|---|
| _227_mtrt | 99.89% | 99.54% | N/A% | 52.3% |
| _201_compress | 19.29% | 40.43% | 2.6% | 2.3% |
| _209_db | 97.66% | 99.77% | 22.2% | 43.1% |
| _202_jess | 98.02% | 97.88% | 87.2% | 86.3% |
| _213_javac | 71.25% | 68.67% | 44.7% | 26.8% |
| _228_jack | 91.63% | 96.58% | 90.8% | 94.7% |
| Anagram | 86.22% | 93.43% | 14.2% | 13.2% |

Figure 12: Generational characterization of the applications - Part 2.

pointers.

Next, in Figure 13 and Figure 14, we look at the cost and performance of partial and full collections for the various benchmarks. The cost is the time required to run the collection, and the performance is the number of objects collected (or their accumulated size). Note that for a mark and sweep algorithm, the cost of sweep is similar for the partial and the full collections. It is only the tracing times that get shorter. Thus, the partial collections take less time but not drastically less.

Figure 10 shows the number and types of collection cycles for the benchmarks. For all benchmarks the number of full collections when using the generational collector is less than the number of full collections when using the non-generational collector.

Finally, we examine the number of pages touched by the collector during the various collections, see Figure 15. We measure the pages touched during trace and sweep, including all the tables the collector uses (such as the card table.) Naturally, the number of pages touched during the partial collections are smaller than the number of pages touched during full collections. The smallest ratio is for the Anagram benchmark, where the number of pages touched during partial collections is about 20% of the number touched during full collections. The largest ratio is for the _213_javac benchmark. There, the number of pages touched in partial collections is about 70% of the number of pages touched during full collections. These positive results match similar measurements in Demers, et al. [6].

## 8.5 Tuning parameters

In this section we explain the choice of parameters. We compare the various card sizes, the method of aging versus the simple promotion method, and we evaluate various sizes for the young generation. For the aging

| | Avg. time active partial GC (ms) | Avg. time active full GC (ms) | Avg. time active GC (ms) w/o generations |
|---|---|---|---|
| _227_mtrt | 99 | N/A | 260 |
| _201_compress | 17 | 35 | 31 |
| _209_db | 80 | 270 | 215 |
| _202_jess | 61 | 116 | 87 |
| _213_javac | 145 | 367 | 249 |
| _228_jack | 60 | 95 | 71 |
| Anagram | 52 | 429 | 346 |

Figure 13: Ellapsed time of collection cycles

| | Avg. No. of objects freed in partial collection | Avg. No. of objects freed in full collection | Avg. No. of objects freed in collection w/o generations | Avg. space freed in partial collection | Avg. space freed in full collection | Avg. space freed in in collection w/o generations |
|---|---|---|---|---|---|---|
| _227_mtrt | 161441 | N/A | 261305 | 4008271 | N/A | 6517749 |
| _201_compress | 112 | 112 | 111 | 1057472 | 6922551 | 67953331 |
| _209_db | 170175 | 187882 | 217685 | 3914861 | 6196926 | 5188449 |
| _202_jess | 106185 | 166720 | 160458 | 3934524 | 6759448 | 5982237 |
| _213_javac | 82536 | 178289 | 71024 | 2863730 | 5788769 | 2387539 |
| _228_jack | 133671 | 186370 | 202109 | 3677861 | 6905298 | 5841292 |
| Anagram | 12251 | 30088 | 41370 | 3515684 | 13279332 | 12590566 |

Figure 14: Average gain from collections

| | Pages touched by | | w/o |
|---|---|---|---|
| | partial | full | generations |
| _227_mtrt | 1489 | N/A | 3355 |
| _201_compress | 76 | 124 | 109 |
| _209_db | 944 | 2794 | 2827 |
| _202_jess | 1304 | 2227 | 2048 |
| _213_javac | 2607 | 3709 | 3080 |
| _228_jack | 1199 | 2052 | 1767 |
| Anagram | 1082 | 4938 | 5054 |

Figure 15: Average no. of pages touched by a GC

| Number of threads | 2 | 4 | 6 | 8 | 10 |
|---|---|---|---|---|---|
| Block marking with 1m young generation | -3.9 | -8.8 | 5.0 | 9.0 | 8.2 |
| Block marking with 2m young generation | 0.8 | -7.1 | 6.0 | 9.8 | 8.7 |
| Block marking with 4m young generation | 1.1 | -2.5 | 6.6 | 9.8 | 7.4 |
| Block marking with 8m young generation | -0.9 | 4.7 | 7.7 | 10.9 | 8.8 |
| Object marking with 1m young generation | -4.7 | -2.6 | 4.3 | 14.0 | 13.0 |
| Object marking with 2m young generation | 1.4 | -4.4 | 5.9 | 11.3 | 8.6 |
| Object marking with 4m young generation | 1.3 | 2.6 | 10.6 | 16.0 | 11.7 |
| Object marking with 8m young generation | 1.9 | 8.0 | 13.2 | 18.8 | 15.4 |

Figure 16: Tuning the size of the young generation: percentage of improvement of generations for multi-threaded Ray Tracer.

| | Block marking | | | | Object marking | | | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | 1m | 2m | 4m | 8m | 1m | 2m | 4m | 8m |
| _201_compress | -0.41 | 0.19 | -0.05 | 0.46 | -0.04 | 0.11 | 0.02 | 0.29 |
| _202_jess | -22.44 | -12.97 | -5.05 | -1.55 | -13.77 | -8.72 | -3.7 | -5.66 |
| _209_db | -0.50 | 0.44 | -0.97 | 0.15 | -1.00 | 0.11 | -0.91 | -0.22 |
| _213_javac | -16.73 | -3.11 | 10.89 | 20.85 | 7.21 | 13.24 | 17.23 | 19.57 |
| _227_mtrt | -2.16 | 5.36 | 9.49 | 0.09 | -5.48 | 5.45 | 7.01 | -0.40 |
| _228_jack | -12.14 | -6.27 | -2.83 | -14.84 | -6.85 | -3.45 | -2.12 | -2.23 |
| Anagram | 14.43 | 30.03 | 37.17 | 38.73 | -8.67 | 12.06 | 24.67 | 26.42 |

Figure 17: Tuning the size of the young generation: percentage of improvement of generations for the SPECjvm benchmarks

method, we compare performance for various tenuring thresholds. The results are summarized in several tables, as described below.

### 8.5.1 Size of the young generation

We begin by evaluating various sizes of the young generation. We compare the sizes 1, 2, 4, and 8 megabytes as possible alternatives for the size of the young generation. We present measurements for the two extreme cases of card sizes: block marking, where the card size is 4096 bytes, and object marking, where the card size is 16 bytes. We will see in Subsection 8.5.3 below that these card sizes are the best for most applications. The results for multi-threaded Ray Tracer can be found in Figure 16 and for the SPECjvm benchmarks [25] in Figure 17. The results do not point a single best size for all benchmarks, but on the average, the best performance is obtained for a size of 4 megabytes for the young generation. In the sequel we fix the young generation to 4 megabyte, except when evaluating the aging mechanism.

### 8.5.2 The aging mechanism

The results for aging are disappointing. as can be seen from the results in Figure 18 and Figure 19. We vary the size of the young generation (1, 2, 4, and 8 megabytes) and the age threshold for promotion to the old generation (4, 6, 8, and 10). Recall that an object is allocated with age 1, and its age gets increased for each collection it survives. We chose the card size to be the smallest possible, which is justified by the analysis of card sizes in Section 8.5.3 below.

Note that if we use the simple promotion mechanism, each object gets old at the age of 2. Thus, it is possible to compare the overhead of the aging method itself by comparing the simple promotion mechanism with aging having the old age being 2. It turns out that our aging method does have a big overhead. See Figure 20. It shows the percentage of improvement (actually deterioration) when using aging with 2 ages

| Benchmark | Age 4 is old | | | | Age 6 is old | | | |
|---|---|---|---|---|---|---|---|---|
| | 1m | 2m | 4m | 8m | 1m | 2m | 4m | 8m |
| _201_compress | 0.3 | 0.1 | -0.5 | 0.4 | 0.5 | 0.2 | -2.0 | 0.1 |
| _202_jess | -17.7 | -15.8 | -10.1 | -7.8 | -12.6 | -13.7 | -10.3 | -9.2 |
| _209_db | -2.4 | -0.7 | -1.4 | -0.4 | -3.1 | -1.3 | -1.1 | -0.1 |
| _213_javac | -14.7 | -3.6 | -5.9 | 17.2 | -21.2 | -8.7 | 3.9 | 17.1 |
| _227_mtrt | -21.0 | -13.4 | 1.1 | -1.9 | -21.2 | -8.0 | -2.6 | -2.7 |
| _228_jack | -11.4 | -6.7 | -1.8 | -1.5 | -12.6 | -6.4 | -2.5 | -0.9 |
| Anagram | -10.8 | 1.9 | 20.0 | 29.6 | -11.2 | 0.8 | 18.3 | 26.7 |

Figure 18: Percentage of improvement for the aging mechanism over a non-generational collector for the SPECjvm benchmarks (part 1)

| Benchmark | Object Mark With Aging | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Age 8 is old | | | | Age 10 is old | | | |
| | 1m | 2m | 4m | 8m | 1m | 2m | 4m | 8m |
| _201_compress | 0.8 | 0.2 | -0.2 | 0.1 | 0.7 | 0.5 | -0.3 | 0.2 |
| _202_jess | -14.6 | -17.3 | -5.1 | -3.8 | -17.6 | -9.4 | -4.9 | -3.6 |
| _209_db | -3.0 | -1.5 | -1.2 | 0 | -3.5 | -2.0 | -1.7 | -0.3 |
| _213_javac | -27.0 | -13.1 | 3.6 | 17.4 | -33.5 | -16.2 | 3.2 | 15.5 |
| _227_mtrt | -10.3 | -8.0 | -3.1 | -2.8 | -22.9 | -10.6 | -1.7 | -1.4 |
| _228_jack | -11.6 | -3.5 | -2.0 | -0.4 | -14.4 | -4.2 | -2.6 | -1.2 |
| Anagram | -11.8 | -0.4 | 16.1 | 23.9 | -11.7 | -1.6 | 14.9 | 23.4 |

Figure 19: Percentage of improvement for the aging mechanism over a non-generational collector for the SPECjvm benchmarks (part 2)

| Benchmark | 1m | 2m | 4m | 8m |
|---|---|---|---|---|
| _201_compress | 0.09 | -0.18 | -0.97 | -0.16 |
| _202_jess | -3.21 | -3.43 | -3.54 | -1.24 |
| _209_db | -1.38 | -0.99 | 0.16 | 0.34 |
| _213_javac | -14.06 | -10.69 | -7.51 | -0.62 |
| _227_mtrt | -14.40 | -11.57 | -9.06 | -1.74 |
| _228_jack | -3.01 | -2.88 | -1.48 | 0.40 |
| Anagram | -2.11 | -9.10 | -3.63 | 3.34 |

Figure 20: The percentage of improvement (or the cost) of the aging mechanism with 2 ages over the simple promotion method.

instead of the standard method. (As before, we use object marking, i.e., the smallest card size.) It may be possible to improve the performance of the aging algorithm by changing the algorithm or data structures. This is something that we have not attempted in this work. Perhaps a simple modification, such as locating the value of the age inside the object instead of keeping a table with the ages, may help by improving the locality of reference. In light of the results, we have chosen not to use aging.

### 8.5.3 Choosing the size of the cards

Finally, we ran some measurements to find out what the best card size is. We varied the size from 16 to 4096, including all powers of 2. The best card size depends on the behavior of the application. Note that since we do not move objects in the heap, the objects of the young and old generations are not segregated.

There is an interesting phenomena about the scanning of the cards. If the dirty objects are concentrated in the heap in a specific location (and it can be big or small), than smaller cards do not shorten the scan. For example, if the first 1/4 of the heap contains dirty objects, then if we take cards whose size is a quarter of the heap or cards whose size is 16 bytes, then we'll have the same objects to actually scan on dirty cards. However, if the dirty objects are spread randomly in the heap than refining the card sizes is useful. The finer the cards are, the less objects we scan. Thus, the nature of the application determines how useful small cards are.

But there are more considerations. For example, smaller cards imply a bigger card table. The card table is accessed on each pointer modification and may influence the locality of reference. A big table that is accessed frequently in a random manner decreases locality. Here, it seems that the consideration is the opposite of the previous one. If the heap access of the application is randomly distributed, then a big table is bad, so bigger cards are required. If the heap accesses are concentrated, then the access of the card table will be concentrated even for a big table, so smaller cards are fine. The big question of which consideration is dominant is the frequency of accesses. Note that a card gets dirty even if touched only once, and that is the only relevant issue for the consideration of the previous paragraph. However, for locality of reference it matters how frequently the cards are touched. The frequency may determine which of these considerations wins and what card size is the best for the application.

The actual results are given in the following tables. In table 21 we specify the improvement of generational collection versus non-generational collection for all benchmarks and the various card sizes. We used a young generation of 4 megabytes and object marking. To get some impression of what influences the results we also present Table 22 the percentage of cards that were dirty in the collection, and in Table 23 the area that got scanned due to dirty cards.

In most cases, the size of the card did not make a significant impact on the running time. The biggest impact can be seen with the benchmarks Anagram, _213_javac, and _202_jess. The impact of card sizes on these benchmark was not the same. For Anagram the bigger card size, the better. For _213_javac the smaller the better, and for _202_jess the two extremes (16 and 4096 bytes) are best. We chose to use the smallest card size (denoted object marking) for the rest of the tests.

| | Object Mark with 4m young generation | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Benchmark | 16 byte | 32 byte | 64 byte | 128 byte | 256 byte | 512 byte | 1024 byte | 2048 byte | 4096 byte |
| _201_compress | 0.11 | 0.16 | 0.10 | -0.41 | 0.25 | 0.33 | 0.40 | 0.46 | 0.62 |
| _202_jess | -4.25 | -4.02 | -6.64 | -9.17 | -7.24 | -7.17 | -6.96 | -7.01 | -6.65 |
| _209_db | -0.45 | -0.87 | -0.30 | -0.03 | -0.70 | 0.06 | -0.12 | 0.33 | -0.63 |
| _213_javac | 18.82 | 16.22 | 15.50 | 14.78 | 13.88 | 13.21 | 12.22 | 11.87 | 11.83 |
| _227_mtrt | 9.05 | 7.72 | 9.58 | 8.36 | 9.11 | 9.63 | 8.24 | 8.78 | 8.90 |
| _228_jack | -7.43 | -6.24 | -7.01 | -6.12 | -6.79 | -7.16 | -6.78 | -6.72 | -6.50 |
| Anagram | 23.61 | 18.92 | 24.04 | 28.59 | 31.35 | 33.09 | 33.41 | 34.48 | 35.24 |

Figure 21: Percentage of improvement for SPECjvm benchmarks for the various card sizes

| | Object Mark with 4m young generation | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Benchmark | 16 byte | 32 byte | 64 byte | 128 byte | 256 byte | 512 byte | 1024 byte | 2048 byte | 4096 byte |
| _201_compress | 0.01 | 0.01 | 0.02 | 0.04 | 0.05 | 0.08 | 0.11 | 0.18 | 0.27 |
| _202_jess | 15.81 | 30.70 | 42.85 | 50.16 | 53.43 | 56.65 | 59.46 | 59.08 | 61.18 |
| _209_db | 19.96 | 19.97 | 20.20 | 20.41 | 20.58 | 20.64 | 20.55 | 20.80 | 21.36 |
| _213_javac | 9.58 | 17.54 | 26.41 | 32.18 | 38.51 | 43.67 | 48.47 | 52.81 | 59.49 |
| _227_mtrt | 1.76 | 3.73 | 4.92 | 6.90 | 9.33 | 12.59 | 17.40 | 23.54 | 29.99 |
| _228_jack | 17.66 | 28.71 | 32.51 | 34.47 | 35.19 | 38.41 | 40.01 | 40.53 | 44.11 |
| Anagram | 1.14 | 0.78 | 2.07 | 1.22 | 1.22 | 1.25 | 1.22 | 1.23 | 1.31 |

Figure 22: Tuning the parameters:Card size - percentage of dirty cards from allocated cards

Looking at Tables 22 and 23, we see that there are almost no dirty cards scanned for Anagram, which is one of the properties of Anagaram that make generational collection appropriate for it. Note that for Anagram, it is best to have a large card size. This is probably due to the smaller card table, since it does not influence the actual scanning, which is negligible. For _209_db the size of the card has practically no influence on the size of the area scanned for collection. This is probably due to concentration of the dirty objects as discussed above.

| | Object Mark with 4m young generation | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Benchmark | 16 byte | 32 byte | 64 byte | 128 byte | 256 byte | 512 byte | 1024 byte | 2048 byte | 4096 byte |
| _201_compress | 1 | 2 | 4 | 6 | 9 | 13 | 19 | 31 | 47 |
| _202_jess | 1237 | 2421 | 3426 | 3888 | 4191 | 4387 | 4499 | 4626 | 4780 |
| _209_db | 2696 | 2724 | 2772 | 2754 | 2775 | 2785 | 2807 | 2841 | 2893 |
| _213_javac | 1524 | 2616 | 3850 | 4873 | 5773 | 6537 | 7477 | 8027 | 9427 |
| _227_mtrt | 231 | 462 | 651 | 896 | 1197 | 1611 | 2227 | 3015 | 3854 |
| _228_jack | 1309 | 2059 | 2319 | 2450 | 2562 | 2717 | 2821 | 2983 | 3226 |
| Anagram | 107 | 175 | 170 | 168 | 167 | 170 | 165 | 167 | 178 |

Figure 23: Tuning the parameters:Card size - Area scanned for dirty cards

# 9 Conclusion

We have presented a design for incorporating generations into an on-the-fly garbage collector for Java. To the best of our knowledge such a combination has not been tried before. Our findings imply that generations are beneficial in spite of the two "obstacles": the fact that the generations are not segregated in space since objects are not moved by the collector, and the fact that obtaining shorter pauses for the collection are not relevant for an on-the-fly collector.

It turns out that for most benchmarks the overall running time was reduced by up to 25%, but there was one benchmark for which generational collection increased the overall running time on our multiprocessor by 4%.

The best performing variant of generational collection out of the variants we checked, was the one with the simplest promotion policy (promoting an object to the old generation after surviving one collection), a quite big young generation (4 megabytes), and a small size of cards for the card marking algorithm (16 bytes per card).

In most collections, less pages are touched by the generational collector. Thus, one should especially consider using generations for an on-the-fly collector when the applications run in limited physical memory.

# 10 Acknowledgments

We thank Hans Böhm for helpful remarks. We thank Alain Azagury, Katherine Barabash, Bill Berg, John Endicott, Michael Factor, Arv Fisher, Naama Kraus, Yossi Levanoni, Ethan Lewis, Eliot Salant, Dafna Sheinwald, Ron Sivan, Sagi Snir, and Igor Yanover for helpful discussions.

# References

[1] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280-94, 1978.

[2] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66-70, March 1992.

[3] Mordechai Ben-Ari. On-the-fly garbage collection: New algorithms inspired by program proofs. In M. Nielsen and E. M. Schmidt, editors, *Automata, languages and programming*. Ninth colloquium (Aarhus, Denmark), pages 14-22, New York, July 12-16 1982. Springer-Verlag.

[4] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333-344, July 1984.

[5] Jeff Chan and Nik Shaylor. Multithreaded Ray Tracer. Sun Microsystems, private communications.

[6] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel G. Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In Conference Record of the *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, January 1990. ACM Press, pages 261-269.

[7] J. DeTreville. Experience with Concurrent Garbage Collector for Mudula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, November 1990.

[8] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science*, No. 46. Springer-Verlag, New York, 1976.

[9] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965-975, November 1978.

[10] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In Conference Record of the *Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, 1994, pages 113-123.

[11] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In Conference Record of the *Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1993.

[12] T. Domani, E. Kolodner, and E. Petrank A Generational On-the-fly Garbage Collector for Java. Technical Report 88.385 IBM Haifa Reesrach Lab. Web access: http://www.cs.technion.ac.il/~erez/gen.ps

[13] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report DEC-SRC-TR-25, DEC Systems Research Center, Palo Alto, CA, February 1988.

[14] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921-930, December 1977.

[15] Randy Heisch. An Anagram Generator. Private communications.

[16] Antony L. Hosking, J. Eliot B. Moss, Darko Stefanović. A Comparative Performance Evaluation of Write Barrier Implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp 92-109 (Vancouver, Canada, October 1992). ACM SIGPLAN Notices 27(10), October 1992.

[17] Paul Hudak and Robert M. Keller. "Garbage Collection and Task Deletion in Distributed Systems. In *ACM Symposium on Lisp and Functional Programming*, pp. 168-178, Pittsburgh, PA, August 1982.

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

[19] E. K. Kolodner and E. Lewis. Using a Color Toggle to Reduce Synchronization in the DLG Collector. Private Communcations, 1998.

[20] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120-131. IEEE Press, 1977.

[21] L. Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 50-54, 1976.

[22] L. Huelsbergen and P. Winterbottom. Very Concurrent Mark-&-Sweep Garbage Collection without Fine-Grain Synchronization. In *Proceedings of the 1998 International Symposium on Memory Management*, pages 50-54, 1998.

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

[24] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, TX, August 1984, ACM Press, pages 235-245.

[25] SPECjvm. Spec - The Standard Performance Evaluation Corporation. Web access **http://www.spec.org/osg/jvm98/**.

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

[27] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495-508, September 1975.

[28] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495-508, September 1975.

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