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.

Information for the conference committee: Contact author: Erez Petrank, email: erez@cs.technion.ac.il, Tel. 011-972-4-829-4321, Fax: 011-972-4-822-1128, Postal address: Department of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel.

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.

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]

*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.

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 7.2. 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 dete-

rioration 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 the full paper [12].

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 (see our full paper [12] for these results). In Section 6 we provide the code of the collector and lower level details appropriate for an implementer. In Section 7 we report the experimental results we measured, and we conclude in Section 8.

1.4 What is missing

Due to lack of space, some important items, which we report in our full paper [12], are missing from this extended abstract. In the full paper we describe our advanced aging mechanism. This mechanism did not give us the improvements we hoped for. We also report more runtime measurements explaining the behavior of the applications. Finally, we explain the choice of the parameters (for example, the size of the cards for card marking) by providing comparisons for various values of the parameters.

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 our full paper [12]. 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 our full paper [12] for these measures).

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 (such as in our full paper [12]) 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.

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 our full paper [12].

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

[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.

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 6 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.

6 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 assume a table with a designated byte for each card holding the card mark. The byte does not have any other use.

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.

```
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])
        MarkCard(x)
    else
        MarkCard(x)
heap[x, i] ← y

Create:
Pick x ∈ free.
color[x] ← allocationColor
Return x

Cooperate:
If (status_m ≠ status_c) then
    If (status_m = sync2) then
        For each x ∈ roots:
            MarkGray(x)
        status_m ← status_c

MarkGray(x):
If (color(x) = clearColor) or
   (color(x) = allocationColor ∧
                    status_m ≠ async) then
        color(x) ← gray
```

Figure 1: The mutator routines

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.

```
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 ← free ∪ x
               color(x) ← blue
```

Figure 2: The collection cycle

7 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 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.

7.1 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-

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

ClearCards:
For each card c:
    If $(dirty(c))$ then
        ClearCardMark(c)
        For each object x on c
            If $(color(x) = black)$ then
                $color(x) \leftarrow gray$

SwitchAllocationClearColors:
$temp \leftarrow clearColor$
$clearColor \leftarrow allocationColor$
$allocationColor \leftarrow temp$

InitFullCollection:
For each object x in the heap:
  If $(color(x) = black \vee color(x) = gray)$ then
      $color(x) \leftarrow allocationColor$
For each card c:
  ClearCardMark(c)

MarkBlack(x):
If $(color(x) \neq black)$ then
  For each pointer $i \in x$ do:
    MarkGray(i)
  $color(x) \leftarrow black$

Handshake:
postHandshake(s)
waitHandshake

postHandshake(s):
$status_c \leftarrow s$

waitHandshake:
For each $m \in mutators$
  wait for $status_m = status_c$

Figure 3: The collector routines

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).

7.2 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 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 200×200 and uses 2 concurrent threads. We modified it to run on a bigger matrix of dimensions 300×300 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.

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

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

Benchmark	Multiprocessor Improvement	Uniprocessor Improvement
Anagram	25.0 %	32.7%

Figure 5: Percentage improvement for Anagram

7.3 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 7.4), we present results for this set of parameters. In our full paper [12] we justify our choice by comparing the performance of the algorithm with aging and for various settings of the other parameters.

7.4 The results

In Figure 4 we present the percentage improvement for the multithreaded Ray Tracer benchmark, described in Section 7.2 above. The number of application threads varies from 2 to 10. Generations perform very well for it.

Next, in Figure 5, we present the improvement generational collection yields for the Anagram benchmark. Here, generational collection is also very beneficial. In Figure 6 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,

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 6: Percentage improvement for SPECjvm benchmarks

	No. GC		w/o
	partial	full	generations
_227_mtrt	36	0	26
_201_compress	5	15	17
_209_db	15	1	15
_202_jess	70	2	51
_213_javac	36	16	82
_228_jack	45	4	35
Anagram	152	8	56

Figure 7: Number of garbage collection cycles

_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 full paper: 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 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.

Figure 7 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.

	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 8: Average no. of pages touched by a gc

Finally, we examine the number of pages touched by the collector during the various collections, see Figure 8. 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].

In our full paper [12] we report many more results on the runtime behavior of the applications as well as tables comparing possible values for the parameters. In particular, we report how many pointers are scanned for inter-generational pointers, how many objects are scanned altogether during the trace, what percentage of objects are freed in partial and full collections, how much time it takes to perform partial and full collections, etc. We also check the performance of the algorithm for various sizes of the young generation, for various card sizes, and for various tenuring steps for the advanced aging mechanism (which is completely absent in this extended abstract).

8 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.

9 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.