CHAPTER 3

AGE-BASED GARBAGE COLLECTION ALGORITHMS

In this chapter we establish a classification of garbage collection algorithms that preserve the age ordering of objects in the heap. We introduce new algorithms within this classification. Unlike generational algorithms, the new algorithms do not always collect a youngest subset of heap objects. We proceed to discuss the repercussions of this design difference on the implementation of copying and pointer maintenance in the collector.

3.1 Conceptual design

The garbage collection algorithms we consider are all "generational" in the broadest sense of the word: when a collection happens, each scheme partitions the heap H into two regions: the collected region C, in which the collector examines the objects for liveness, and if live, they survive the collection; and the uncollected remainder region U, in which the collector assumes the objects to be live and does not examine them. The non-generational collector is a degenerate case in which the latter region is empty. The collector further partitions the set C into the set of survivor objects S and the set of garbage objects G, by computing root pointers into C and the closure of the points-to relation within C. To make the freed space conveniently available for future allocation, a copying collector manipulates the survivors S by moving them to the "to-space". This manipulation is a primary cost of collection [Jones and Lins, 1996].

Denoting by $||X||$ the total volume of objects in a set X, we use as a measure of the copying cost of collection the "mark/cons"[1] ratio, the ratio of the volume copied to the volume freed (i.e., available for allocation in the subsequent cycle): $\mu = \frac{||S||}{||G||}$.

It is desirable to minimize μ. If the initial partition into C and U is such that most of C falls into G, the copied volume of survivors $||S||$ will be small. However, if *arbitrary* schemes are considered, then clearly the best scheme is one that always makes the initial partition so that $G = C$, and thus $\mu = 0$. But is it realistic? The computational effort to perform the initial partition ideally is as great as to perform a non-generational collection—except if an oracle were available. We consider as realistic only those schemes which compute the initial partition very cheaply.

We restrict our attention to a class of schemes that keep objects in a linear order according to a certain *object timestamp*. Imagine objects in the heap as if arranged from left to right, with the oldest timestamp on the left, and the youngest timestamp on the right, as in Figure 3.1. The region collected, C, is restricted to be a contiguous subsequence of this sequence of heap objects, thus the cost of the initial partition, simply choosing the two boundaries, in some prescribed manner, is virtually nil. We call these schemes *age-based* collection.

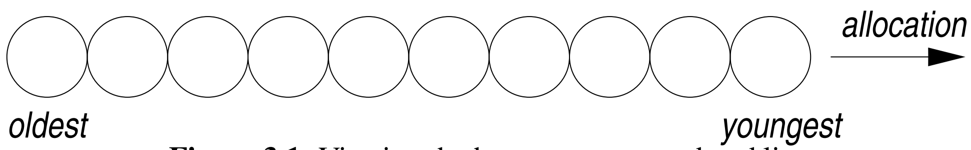

Figure 3.1. Viewing the heap as an age-ordered list.

We consider two notions of timestamp: *true age* and *renewal age*. In the first case, each object is given a permanent timestamp once, at the time of allocation. The relative order of objects in the sequence of heap objects never changes. In the second case, upon collection, the collector gives all survivors fresh, renewed, timestamps, as if they were newly allocated. It then moves these survivors to the young end of the sequence of heap objects.

[1]The name is used for historical reasons: in the original mark/sweep collector for LISP [McCarthy, 1960] the live objects were *marked* during collection, whereas the allocation consisted solely of LISP pairs, known as `cons` cells.

True-age timestamps permit the collector to focus attention on particular age groups, i.e., on sets of objects allocated consecutively. They arise naturally from considerations of object lifetime behavior—clusters of objects allocated together tend to expire together [Hayes, 1993]. On the other hand, the notion of renewal-age timestamps can arise from considerations of an *a priori* exponential lifetime distribution [Clinger and Hansen, 1997].

Traditional generational collection schemes are, in the main, age-based. They are based on true age, and the region collected is some subsequence that always includes youngest (most recently allocated) objects.[2] In the following, we introduce and categorize alternative collection schemes according to their choice of objects for collection. The age-based collection algorithm is also presented in the form of code in Figure 3.2.

3.1.1 True-age schemes

The simplest age-based collector and the base point of our comparisons is the non-generational collector. It will be labelled NONGEN in the discussion and figures below. The region of collection in NONGEN is always the entire heap: $C = H, U = \emptyset$.

We now consider age-based schemes, distinguished by different choices they make for the collected region. A *true youngest-first* (TYF) collector always chooses some youngest (rightmost) subsequence of the sequence of heap objects (Figure 3.3).

Generational collector schemes are variants of true youngest-first collection, differing in the size of C and in how they trigger collections [Barrett and Zorn, 1995]. In the basic design [Jones and Lins, 1996, p.147], new allocation is into one fixed-size part of the heap (the *nursery*), and the remainder is reserved for older objects (the *older generation*). Whenever the nursery fills up, it is collected, and the survivors promoted to the older generation (Figure 3.5). When the older generation fills up, then the following collection collects it together with the

[2] A slight divergence from the age-based criterion may arise in *copying* collectors, because the order of the objects in to-space is determined by the heap traversal (typically breadth-first, as in the standard Cheney scan [Jones and Lins, 1996; Cheney, 1970]), and is in general different from original order in from-space. Hence, the age order of objects is perturbed slightly within S for each collection. In *compacting* collectors reordering does not occur at all.

initialize:

available_space := *heap_size*

current_timestamp := 0

allocate (*num_words*):

if *available_space* $\geqslant$ *num_words* then

create new object o with

o.*size* := *num_words* and o.*timestamp* := *current_timestamp*

current_timestamp := *current_timestamp* + *num_words*

available_space := *available_space* - *num_words*

else

Garbage collection:

set timestamp boundaries t_{min}, t_{max} according to scheme

$C := \{o | o.timestamp \in [t_{min}, t_{max}]\}$

$U := H \setminus C$

$S_0 := \{o | o \in C \wedge (\exists r \in Roots \cup U) r \to o\}$

$S := TC(\to, C, S_0)$, where transitive closure is computed as:

$S_n = \{o | o \in C \wedge (\exists o' \in S_{n-1}) o' \to o\}$

$S = \bigcup_{n=0}^{\infty} S_n$(union finite)

$G := C \setminus S$

$available\_space := available\_space + \|G\|$

discard objects in G

if $\|G\| = 0$ then fail

fi

Figure 3.2. Pseudocode for the conceptual design.

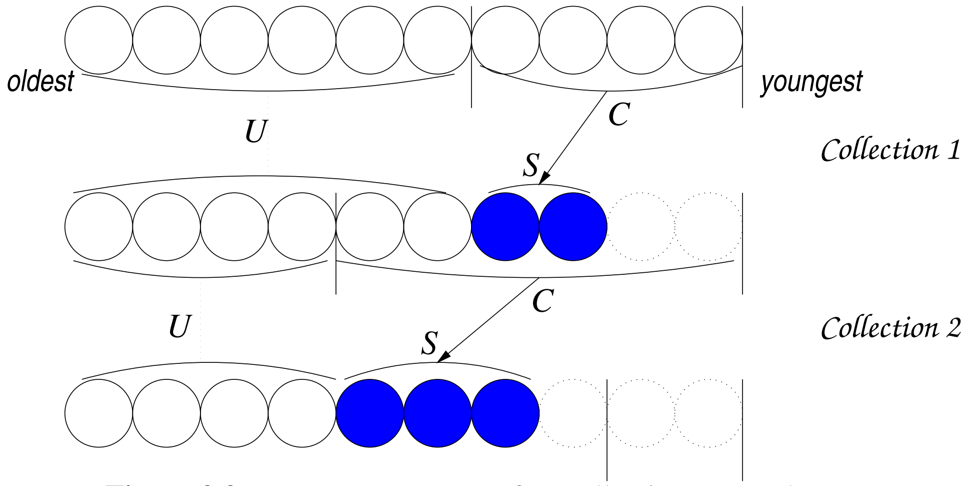

Figure 3.3. TYF: True youngest-first collection (general case).

$\|C\|$: collected region	U: remainder (region not collected)
S ●: region of survivors	○: area freed for new allocation

Two successive collections are shown. For sake of illustration, the surviving amount is 2 on collection 1, and 3 on collection 2. The collected region is enclosed in vertical bars.

Legend for Figures 3.3–3.12.

nursery. In a two-generation collector, that collection considers the entire heap. The collector deliberately reserves space, so that, unlike in TYF, the region chosen for collection contains exactly the objects allocated since the last collection (except for full heap collections). The operation of the GYF collector is summarized in the code of Figure 3.4.

We study two- and three-generation schemes: 2GYF (2 Generations; Youngest-First), and 3GYF. We stipulate that the size of each generation is strictly greater than 0, and therefore 3GYF never degenerates into 2GYF. When discussing generational schemes generally, we use the label GYF. We also examine a scheme in which the older generation is allowed to grow and use all the space of the nursery, and vice versa [Appel, 1989b], and no space is held in reserve; this scheme is denoted NGYF, and its operation is shown in Figure 3.6.

The simplest youngest-first collector is one that always chooses a constant amount of youngest objects to collect; we call this scheme FC-TYF (Fixed size of Collected region; True Youngest-First). Note, as is clear from the diagram in Figure 3.7, that survivors of the preceding collection remain in the collected region. One can expect this collector to be inefficient,

initialize:

  Generations 1–N:

  set $\sum_{g=1}^{N} nominal\_size_g = heap\_size$

  for $g = 1,\ldots,N$ do $Gen_g := \emptyset$

  $reserve := heap\_size$

  $next\_gen := 1$

allocate (*num_words*):

  if $\min(nominal\_size_1 - current\_size_1, reserve) \geqslant num\_words$ then

    create new object o in Gen_1 with

      $o.size := num\_words$

    $current\_size_1 := current\_size_1 + num\_words$

    $reserve := reserve - num\_words$

  else

    Garbage collection:

    $gen := next\_gen$

    $C := \bigcup_{g=1}^{gen} Gen_g$

    $U := H \setminus C$

    $S_0 := \{o | o \in C \wedge (\exists r \in Roots \cup U) r \to o\}$

    $S := TC(\to, C, S_0)$

    for $g = 1,\ldots,gen$ do $S^{[g]} := S \cap Gen_g$

    $Gen_1 := \emptyset$

    for $g = 2,\ldots,gen{+}1$ do $Gen_g := Gen_g \cup S^{[g-1]}$,

      except if $gen = N$ then $Gen_N := S^{[N]} \cup S^{[N-1]}$

    $reserve := heap\_size - \sum_{g=1}^{N} \|Gen_g\|$

    $next\_gen := \min g' : (\forall g \geq g') \|Gen_g\| \leqslant nominal\_size_g$

  fi

Figure 3.4. Pseudocode for GYF collectors.

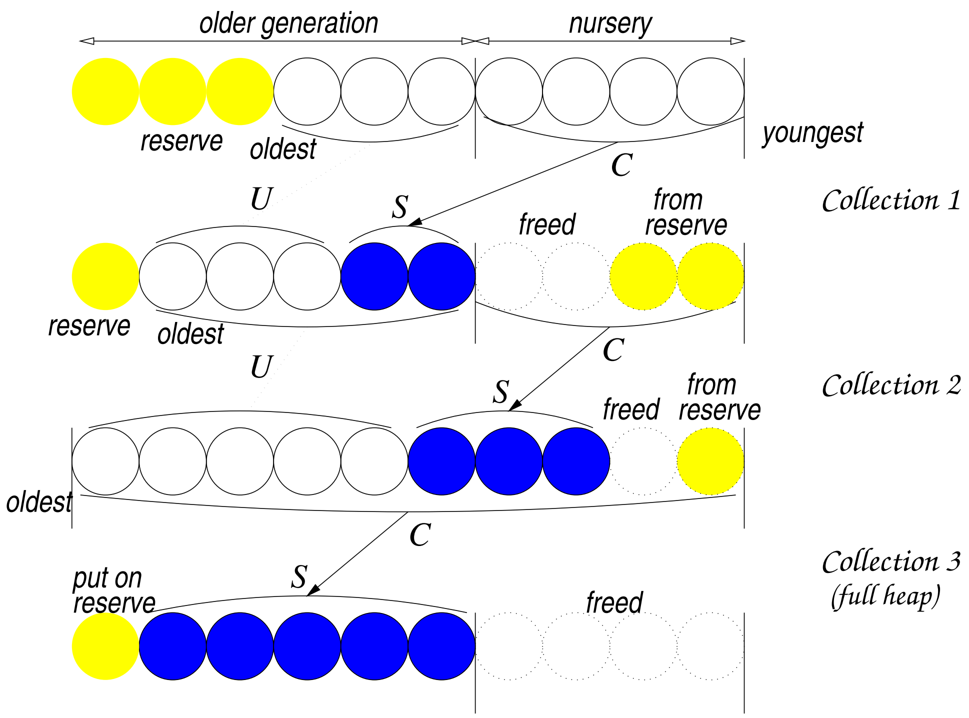

Figure 3.5. Generational youngest-first collection (2GYF).

because objects are repeatedly copied, and collections are frequent, because only the space freed is available for new allocation. Indeed, the older region remains untouched after it is initially allocated. Contrast this with the GYF collector, which does not use all the available space, putting some on reserve, but is then able to collect at regular intervals that correspond to the nursery size, and does not copy survivors of minor collections again until the next full collection.

A *true oldest-first* (TOF) collector always chooses an oldest (leftmost) subsequence of the sequence of heap objects (Figure 3.8). An FC-TOF collector considers a constant amount of oldest objects. It is clear that survivors of the collection will be collected repeatedly. If the oldest objects in the heap survive indefinitely, as in many programs, and the collected region is small and only encompasses such objects, FC-TOF necessarily fails because it finds no garbage in the collected region. However, we encountered some programs where FC-TOF, with a moderate size of the collected region, works better than other age-based collectors: intuitively, it gives objects enough time to die before they enter the collected region.

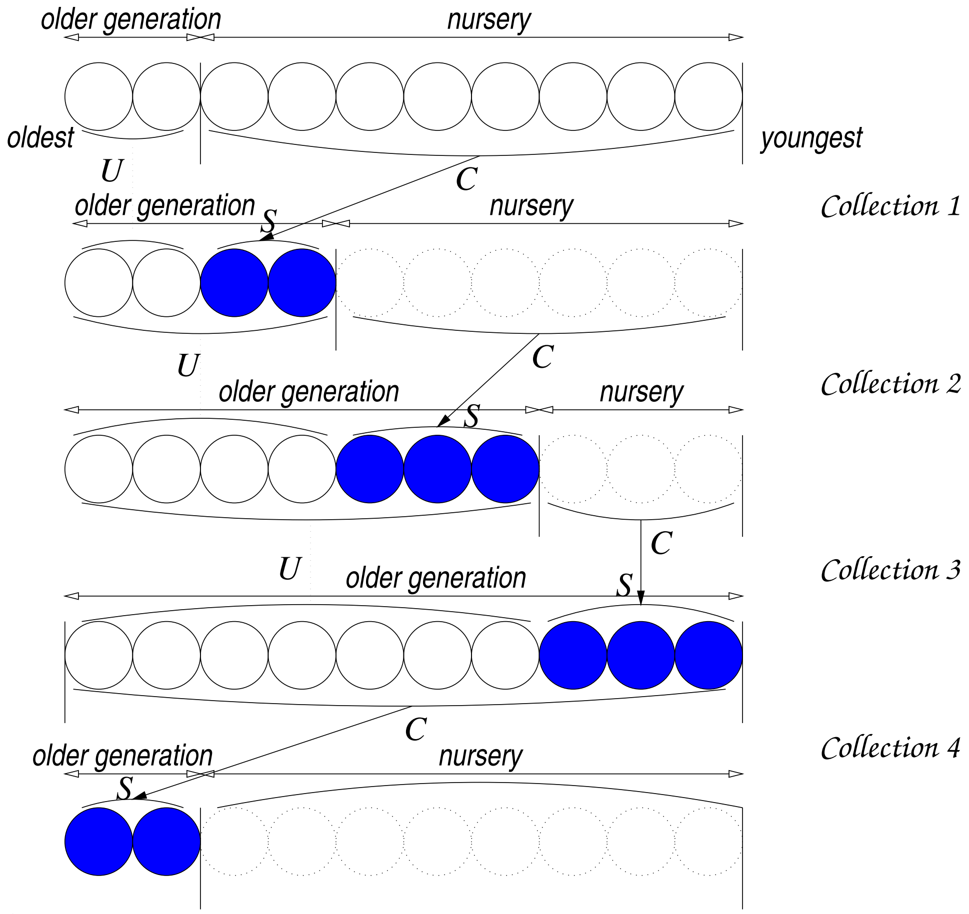

Figure 3.6. Generational youngest-first collection with flexible generation boundaries (NGYF).

The simplicity of FC-TYF and FC-TOF schemes is in the fact that collected regions are fixed. However, it is intuitively clear that performance suffers because some objects are subjected to collection repeatedly, while others are not subjected to collection soon enough. Thus, we must look for schemes that consider all parts of the heap at appropriate intervals. Note that GYF, which is known to work well in practice, revisits the entire heap, and at regular intervals, by means of full collections. Let us instead examine ways to revisit all of the heap piecemeal.

A *deferred older-first* (DOF) collector chooses a middle subsequence of heap objects, which is immediately to the right of the survivors of the previous collection (Figure 3.9). Thus the region of collection sweeps the heap rightwards, as a sliding window of collection. If the remaining area to the right is smaller than the window, the collector completes the sweep by

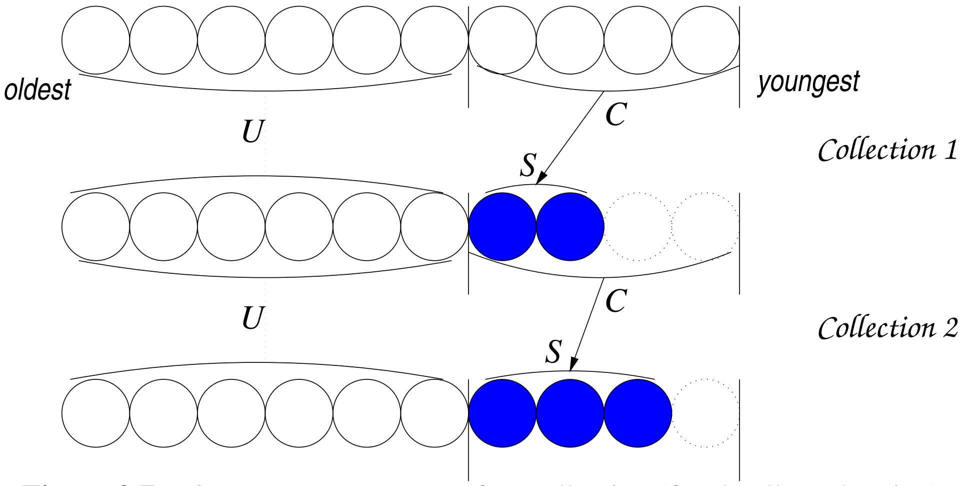

Figure 3.7. FC-TYF: True youngest-first collection (fixed collected region).

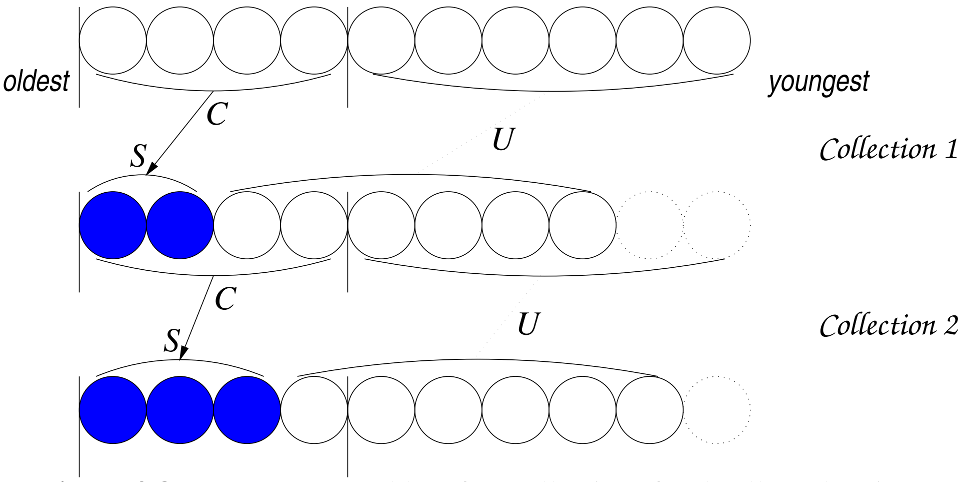

Figure 3.8. FC-TOF: True oldest-first collection (fixed collected region).

collecting the remaining area of youngest objects.[3] The FC-DOF collector employs a window of constant size.

The intuition for the potentially good performance of the FC-DOF collector can be gleaned from the diagram in Figure 3.10, which shows a series of eight collections, and indicates how the window of collection might move across the heap. The most important aspect is that if the window is in a position that results in small survivor sets, then the window moves by only a

[3]In the original description of the DOF algorithms, we proposed that as soon as the remaining area to the right is smaller than the window, the window is reset to the left end. However, we subsequently found that better worst-case copying cost is achieved with the current description. Additionally, full sweep is required for certain pointer filtering techniques (see below, Section 3.2.2).

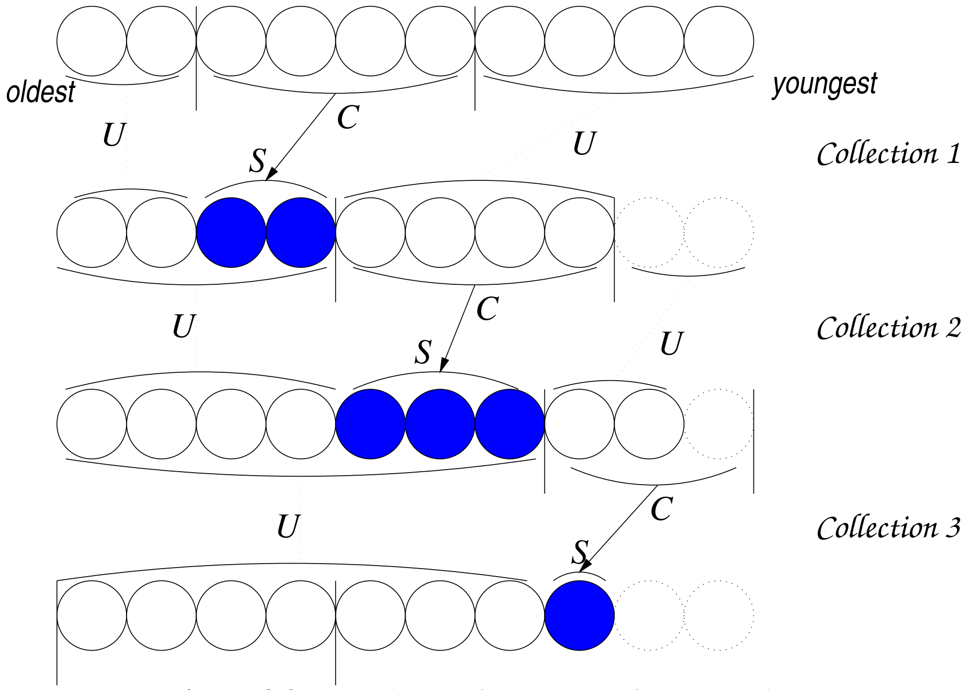

Figure 3.9. FC-DOF: Deferred older-first collection.

small amount from one collection to the next. If it continues to move slowly, then it is likely to remain for a long time in the same region, corresponding to the same age of objects, and it continues to enjoy small survivor sets (Collections 4–8). The outcome is that considerable progress is made in allocation while but a small amount is copied: μ is low. The question then is how long the window can remain in such a good position, and, once it has left it, how costly is the return, which involves finishing the rightward sweep, resetting to the left end, and sweeping over older objects back to the "sweet spot."

A *deferred younger-first* (DYF) collector similarly sweeps the heap, but leftwards (Figure 3.11). As soon as the remaining area to the left is smaller than the window, the window is reset to the right end. The FC-DYF collector employs a window of constant size. Despite the apparent symmetry of definition, DYF does not enjoy the window-motion benefit of DOF: the window moves leftward by one window size even when there are no survivors. We shall find that for this reason DYF never performs well.

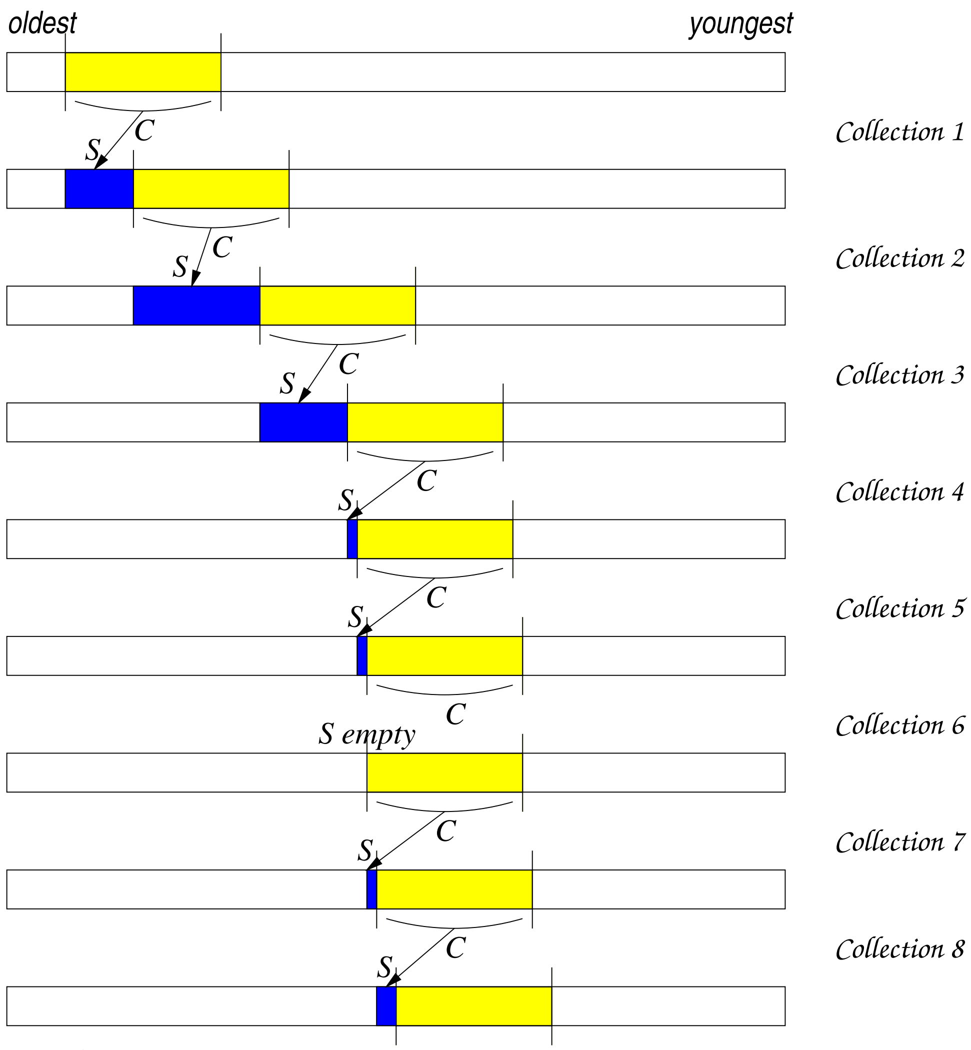

Figure 3.10. Deferred older-first collection, window motion example.

3.1.2 A renewal-age scheme

The collector proposed recently by Clinger and Hansen [Clinger and Hansen, 1997] and its generalization, the "oldest-first" collector [Stefanović *et al.*, 1998a], are also age-based, but according to renewal age, and older objects form the collected region. Similar ideas for circular management of heaps can be traced to earlier literature [Baker, 1978; Bekkers *et al.*, 1986; Lang and Dupont, 1987]. A *renewal-oldest-first* collector always chooses a leftmost subsequence of the sequence of heap objects—but the survivors are placed to the right of the uncollected remainder (Figure 3.12). Therefore, even though the collection window remains at the left, "old" end, the collector does not process the survivors of the collection again, until

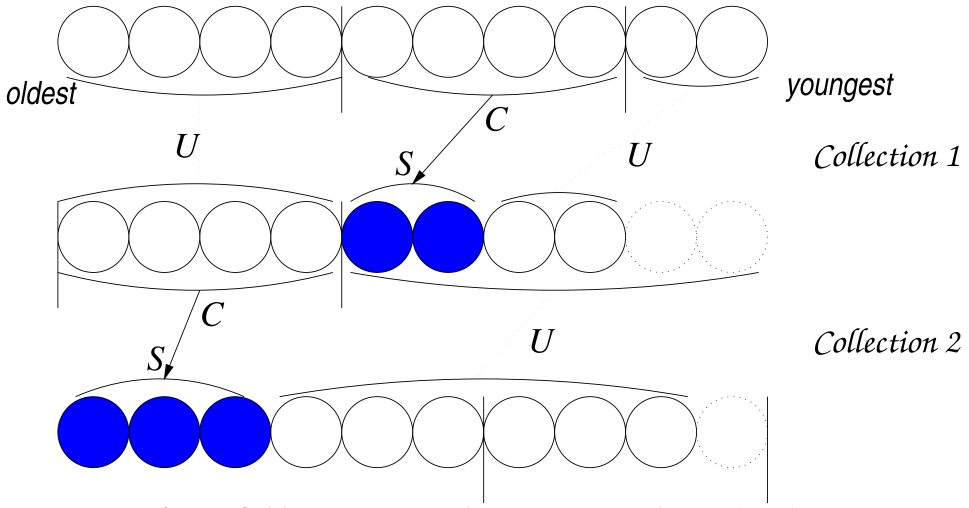

Figure 3.11. FC-DYF: Deferred younger-first collection.

they have been pushed back to the left by new allocation after several succeeding collections. Note, however, the conspicuous crossover of the logical paths of survivor and remainder objects, which results in irreversible mixing of true object ages in the heap, even in compacting collection.

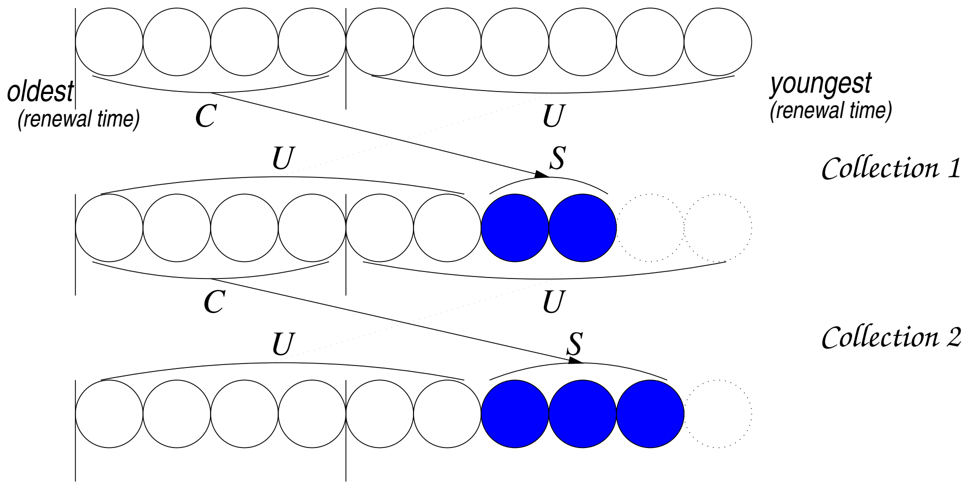

Figure 3.12. FC-ROF: Renewal-oldest-first collection.

3.1.3 Towards optimal schemes

There is a final category of schemes, clearly unrealistic for actual implementations, that we consider. Recall that a deferred oldest-first collector collects a middle subsequence of age-ordered heap objects. Contemplate for a moment a generalization: let us choose any subsequence of a given size, not necessarily starting from a known point, but instead whichever subsequence yields the lowest mark-cons ratio on this collection. Thus we arrive at the *locally-optimal* collector scheme FC-OPT.[4]

Is there a globally optimal scheme? The global optimum need not coincide with the aggregate outcome of locally optimal decisions. The overall copying cost can be reduced by selecting the amount of garbage released in such a way that the next collection comes at a particularly propitious time, namely when there are few live objects. However, the consideration of globally optimal schemes is beyond the scope of this investigation. Instead, we shall only use the results of the locally-optimal collector schemes as an upper bound on the minimum copying cost, for comparisons with realistic schemes.

3.1.4 Collection failure and recovery

It is possible, especially for an FC collector with a small setting for C, to find no garbage in the collected region. If that happens, we let the FC collector fail for the purposes of this study. An implementation could increase the heap size temporarily, or retry collection on another region. A moving-window scheme can retry collection at the next window position. Alternatively, the whole heap can be collected. Note that GYF schemes by design occasionally consider the whole heap, hence they enjoy an advantage over the FC schemes as simulated here, with the disadvantage that their maximum pause times are no lower than with non-generational collection.

3.1.5 Garbage cycles

A cycle of garbage objects can be so large that it cannot fit in the collected region. Even if the cycle is smaller than the size of the collected region, it can consist of objects of sufficiently

[4]Similarly, we could explore the limits of true oldest-first and true youngest-first collection: instead of fixing the amount to be collected, let the collector choose that amount which results in the lowest mark/cons ratio μ *for the current collection*, but we leave this exploration for future work.

diverse age that it is never entirely within the collected region.[5] In either case, the cycle cannot be reclaimed. Traditional generational collection, at the risk of long pause times, finds such cycles by collecting the whole heap occasionally. Since the presence of such cycles is liable to lead to poor performance and ultimately failure, the same remedy, suggested in the preceding section, applies. Alternatively, the purely age-based scheme can be modified, with provisions similar to the train algorithm for the *mature object space*, which tackles the problem of cycles while maintaining incrementality [Hudson and Moss, 1992; Seligmann and Grarup, 1995]. Note that the results presented here are based on a simulator and a prototype implementation which do not implement any mechanism to enforce reclamation of cycles.

3.1.6 Locality

Our study is based on extensive simulations, in order to explore a large number of collector configurations, and indeed a number of altogether different schemes. This focuses attention on the primary costs, namely copying and pointer maintenance. However, we cannot measure certain other costs, without integrating our implementation with a live object system. The chief of these costs is the effect on locality [Zorn, 1991; Reinhold, 1993; Gonçalves, 1995; Diwan *et al.*, 1995; Appel and Shao, 1996]. The schemes that we found to perform well with respect to copying cost also share the property that they *do not change* the relative order of objects; think of them as compacting. Therefore, they do not adversely affect the locality of access within the mutator. Unlike traditional generational collection, which repeatedly collects from the same memory region of young objects and *occasionally* the whole heap, some of the proposed schemes visit different parts of the heap more regularly. Although it is tempting to predict the memory hierarchy behavior of these schemes by observing the patterns of collected region motion, the complexity of interaction of allocation, copying and pointer maintenance

---

[5] This scenario is made unlikely by the fact that an unreclaimed cycle of garbage objects migrates to and concentrates in the oldest end of the heap. There its adverse effect on collector performance is indistinguishable from that of truly live but permanent objects (Section 5.4.3).

accesses with the memory hierarchy requires that the behavior should be measured in a fully implemented system instead.

3.2 Implementation

In preceding sections, we outlined several age-based garbage collection algorithms that we use to explore reducing copying cost beyond generational youngest-first collectors, by prescribing which portions of the heap should constitute the collected region. The following chapter will show that the promise of the DOF collector is actually delivered, but now we consider how to implement in practice a collector with a collected region that ranges outside the space of youngest objects.

If we look at all the age-based schemes together, we find two requirements for implementation. First, it must be possible to identify any set of objects according to their age, and use it as the collected region. Second, it must be possible to enumerate pointers into the collected set from outside, in order to collect it correctly.

3.2.1 Blocks

The first requirement can be fulfilled efficiently by dividing the heap into blocks and linking blocks into an age-ordered list (Figure 3.13). The collected region must consist of a number of blocks. This restriction on the freedom to choose the collected region is acceptable: constant overheads of collection will surely render too expensive any scheme that chooses extremely small collected regions. Therefore, imposing a block as the minimum size of the collected region, as well as the measure of granularity of window positioning, must be appropriate for some block size.

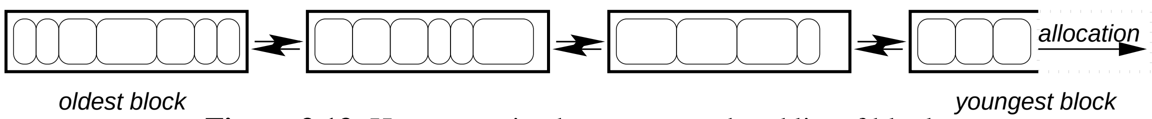

Figure 3.13. Heap organized as an age-ordered list of blocks.

Hence, the heap address space is divided into blocks. Each block consists of 2^b bytes, and is aligned on a 2^b-byte boundary. Assume for the time being that no object is larger than a block. Each object is entirely contained within one block.

The block table is an auxiliary data structure, an array that lists all the blocks available in the address space and records the status of each (free, or in use). Additional information for blocks in use is accessed indirectly, to avoid the space overhead for unused blocks.

Given the address of a heap object, or a field within an object, a constant b-bit-shift operation calculates an index into the block table corresponding to the block in which the object lies.

Blocks in use are doubly linked in age order, so that age-based collectors can easily follow their collection policies. Each block records the next free address within it, so that objects can be added to it: by the mutator in allocation, or by the garbage collection in survivor copying. During collection, each block records whether it is in the collected region or not. During the Cheney scan phase of garbage collection, a block also records the next address to be scanned. In generational schemes, the block additionally records the number of the generation to which it belongs.

The stipulation that whole blocks are subject to collection raises concern about possible fragmentation: if the survivors from an entire collected region must be placed in blocks themselves, then we can expect, on average, half a block of wasted space. The situation is easily remedied for the DOF collector, if the collector promotes survivors into any available space in the adjacent block to the left (youngest among uncollected blocks older than the collected region), as shown in Figure 3.14. Because of the rightward sweep of the collected region, any incomplete block left at the the end of one collection is filled in subsequent collections, and fragmentation is eliminated.

Objects larger than one block are allocated in a special area, the *large object space*, but each large object is logically assigned to a block. The collector never physically copies large objects, but rather logically moves them from a particular block to another by manipulating

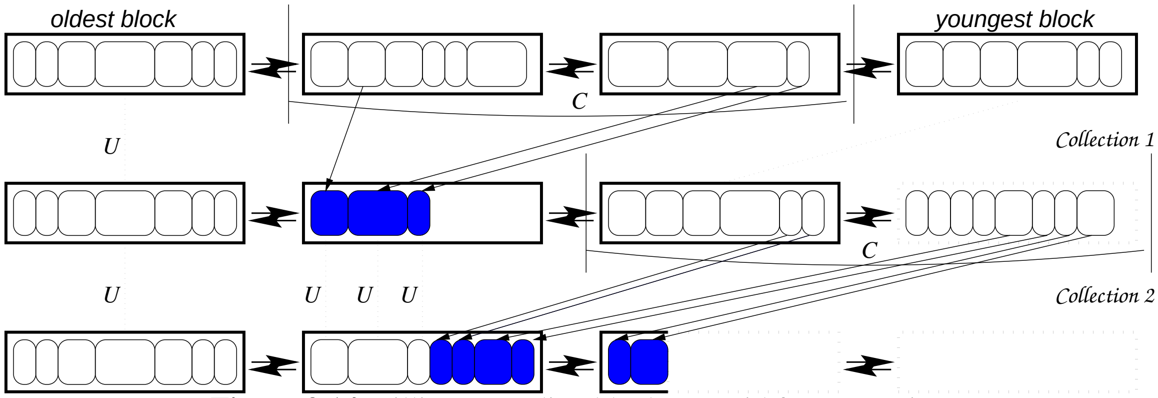

Figure 3.14. Filling preceding block to avoid fragmentation.

efficient indexing data structures. The implementation of large object space can be adapted from the one in the GC Toolkit [Hudson *et al.*, 1991]. The handling of objects in large object space is more expensive than of those in ordinary blocks [Hicks *et al.*, 1998]; an examination of the size distributions of objects (Section 4.2.3) shows, however, that large objects are a rare occurrence for our benchmarks.

3.2.2 Remembered sets

The second requirement is to be able to find pointers into the collected region, no matter which blocks, or how many blocks, are in the region. The most general solution is to track the set of pointers into each block (the remembered set), and then compute the union of the sets for all blocks in the collected region.

In contrast, note that a garbage collection algorithm that defines fixed boundaries for the collected region can safely record just the pointers that cross that boundary. For instance, a generational collector maintains one remembered set per generation, no matter how large the generation is. The need to maintain remembered sets for each block can cause significantly larger space overhead, as well as time overhead for recording and processing the pointers.

The generational collector does not even maintain truthful remembered sets for each generation: by design, a collection of generation n always collects all younger generations as well, therefore it is only necessary to know the pointers into generation n from *older* generations.[6] As we shall see in Section 6.1.3, this structure greatly reduces the number of pointers that must be tracked.

Therefore, while the design using per-block remembered sets achieves the desired functionality for the fully-general age-based schemes, we must also consider the implications of the particular age-based scheme on the sets of pointer stores that the system can safely ignore.

Note that the majority of stored pointers have the source and target within the same block, as we shall see in Section 6.2. Since a block is always entirely within the collected region, or entirely outside it, such local pointers do not need to be recorded in the remembered sets. This optimization applies to all age-based schemes. It eliminates a majority of pointer stores for reasonable block sizes at very low cost.

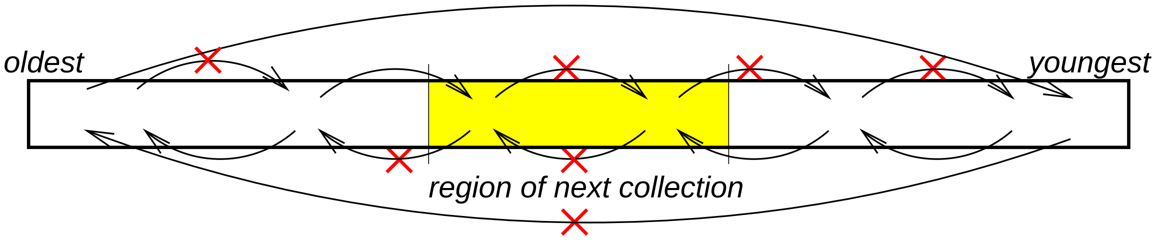

Figure 3.15. Directional filtering of pointer stores at run-time (DOF).

A more aggressive elimination criterion is possible. Suppose that the stored pointer is $p \to q$, and that the object p is subject to collection *before* object q. Then the pointer will be invalidated before q is collected, either because p is moved if live, or because p is dead. Hence the pointer $p \to q$ should not be recorded in the remembered set of q. Thus the general rule is to filter out those stores for which the source will be collected before the target, or at the same time as the target. Some age-based schemes have strict policies for the positioning of the collection window in the heap, and it is therefore possible to establish whether the source or the target of a pointer store will be collected first. Among them is DOF, with the full, cyclic,

[6]Note as a special case that in the two-generation collector the (single) remembered set is always empty following a collection, thus the write barrier at collection time can be elided.

sweep of the heap. The directional filtering for the DOF scheme is shown in Figure 3.15 for pointer stores in the mutator.

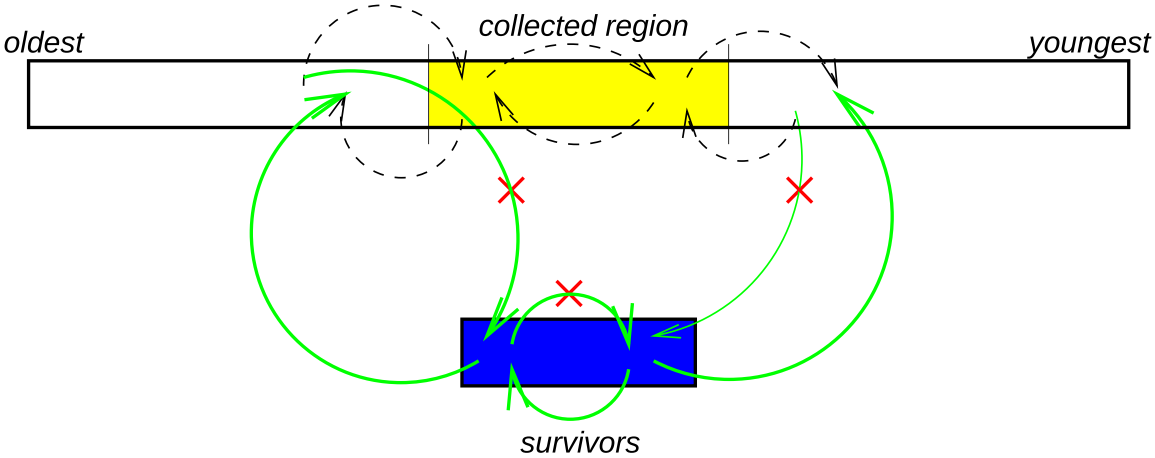

Figure 3.16. Directional filtering of pointer stores at collection time (DOF).

Distinguishing the region that will be collected next, and the regions older and younger than it, there are 12 cases of pointer stores at run-time. One half are eliminated by observing that the source will be collected before the target, and additionally, the case with both source and target within the region of the following collection is eliminated.

In Figure 3.16 we show the filtering scheme for pointer stores during collection. The dashed lines indicate the pointers as they existed before the collection, and the full lines indicate the pointers established during the collection, as survivor objects are copied into to-space. Here the filter eliminates 3 out of 6 cases. The actual benefits of directional filtering depend on the distribution of different pointer store directions, and we shall return to this question in Section 6.2.

In the design of remembered sets for the newly proposed collectors, there is an additional consideration that did not arise for generational collectors. Namely, when a collection copies objects from the collected set C into the survivors set S, then any pointers from C into the uncollected remainder U, recorded in the remembered sets of the blocks of U, must be discarded prior to collection. Otherwise, the remembered sets would contain entries for source addresses in the from-space of the collection, which, following the collection, are invalid. The collection then reestablishes remembered set entries for the surviving pointers from S into U. Therefore, the data structure for remembered sets must support deletion efficiently. (In generational collection, the pointers from C to U are younger-to-older, hence they are not recorded in remembered sets to begin with.)

3.2.3 Write barrier

We now consider how to construct remembered sets. In between collections, new pointers come into being by means of initialization of pointer-typed fields of newly allocated objects, and by updates of these fields in older objects. Pointer updates are tracked using a write-barrier mechanism. As for pointer initialization, either the initializing stores are tracked using the same write-barrier mechanism, as it is, in effect, done in the Smalltalk system, or the newly allocated area is scanned for pointers at garbage collection time.

We consider two write-barrier mechanisms. The first write-barrier mechanism immediately inserts stores into remembered sets. When a store occurs in the mutator, the following filtering actions occur. First, it is established that the store is of a pointer as opposed to a non-pointer value. In dynamically typed languages such as Smalltalk, this check is at run-time. In statically typed languages, such as Java and Modula-3, the write barrier code is only emitted for pointer stores, so this check disappears. Second, the write barrier verifies that the stored pointer value (the target) is not nil, since such stores do not create new pointers. Third, the write barrier verifies that the pointer crosses block boundaries. This check involves the comparison for equality of the high-order bits (full address length less b) of the pointer source and target. Fourth, if the particular age-based scheme permits further elimination of pointers according to the age of the source and target, then the write barrier may eliminate some additional stores. Fifth and last, the pointer source is inserted into the remembered set of the block corresponding to the target.

The second write-barrier mechanism is a hybrid of *card marking* and remembered sets [Hosking and Hudson, 1993]. The heap is logically divided into cards of size 2^c bytes, where

$c \leqslant b$. Experience with generational collectors suggests that cards are typically much smaller than blocks: e.g., while blocks of 64 KB were used in a Smalltalk implementation, the most efficient card sizes were in the range 256 bytes – 1 KB [Hosking *et al.*, 1992; Hosking and Hudson, 1993]. A card table is an array with an entry for each card of the heap. At the end of a collection, all entries of the card table are *clean*. In between collections, at every pointer update (the store of a pointer $p \rightarrow q$) the write barrier marks the entry for the card containing p as *dirty*. At the beginning of each collection, the collector scans the card table to find dirtied cards. For each dirtied card table entry, it then scans the corresponding card in memory, i.e., it examines all pointer-valued fields in all the objects in the card, and updates appropriate remembered sets—in our example, it is the remembered set of the block containing the address q. When cards are examined and summarized into remembered sets, the same filtering criteria apply as discussed above for the case of immediate insertion into remembered sets. One benefit of card marking is that *duplicate* entries are immediately eliminated. The price is paid instead in scanning the cards, since the location of the updated pointer source is recorded only imprecisely. (When scanning a card, the collector may examine many more pointers, depending on the size of the card, in addition to the one (or ones) that caused the card table entry to be dirtied.) We shall see, however, that there are few duplicates *within* a block (Section 6.1.3), thus card marking is not likely to be profitable.

3.2.4 A write barrier design for large address spaces

Returning to the write-barrier mechanism without card marking, we observe that it can be implemented especially efficiently if a large virtual address space is available. The goal is to make the sequence of write-barrier instructions short for the common case, namely that of stores that the barrier does *not* record. A large address space allows a heap organization that avoids the need to index into a block table and is able to make the pointer update filtering decisions solely from the two addresses of pointer source and target, which are already available in hardware registers for the store itself.

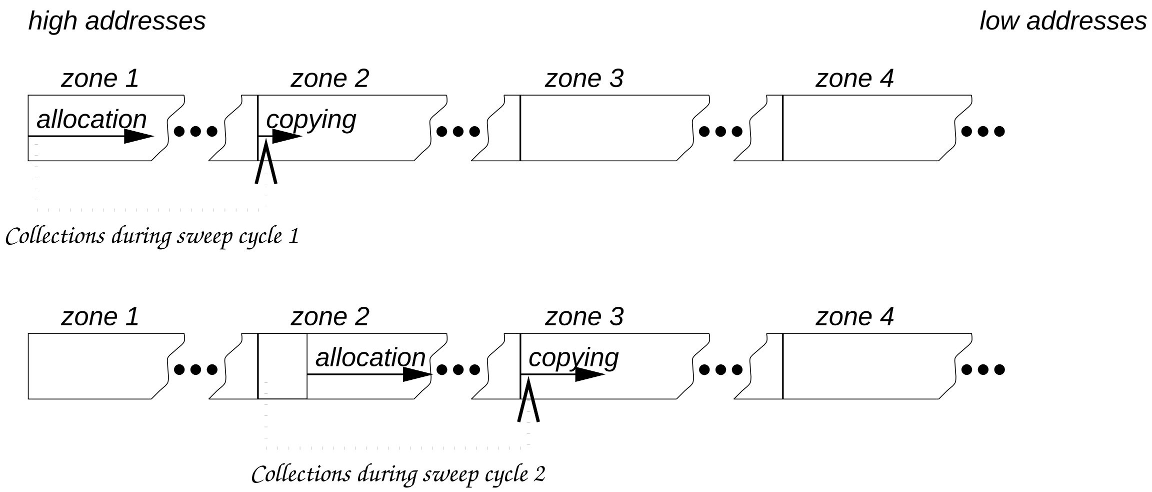

Figure 3.17. DOF heap organization in a large address space.

We illustrate the heap organization in Figure 3.17.[7] The address space is divided into zones, each zone consisting of a large number of blocks. At any moment exactly two adjacent zones are in use for the heap. Initially, all allocation is into zone 1, at the high end of the address space, whereas all copying by the collector is into the next lower zone 2. Both allocation and copying proceed from higher to older addresses within a zone. As soon as the amount allocated in zone 1 becomes equal to the specified heap size (expressed as a number of blocks), garbage collection begins in the DOF manner, with the collection window sliding over zone 1, from high addresses to low. The survivors of collection are copied over into zone 2. Subsequent collections are triggered when the total amount in the heap (i.e., in zones 1 and 2) again equals the specified heap size. This process continues until either: (a) the window of collection catches up with allocation in zone 1, or (b) the allocation in zone 1 reaches the limit, i.e., runs into zone 2. Case (a) corresponds to the natural end of the window sweep of the DOF collector (Figure 3.9). Case (b) is coerced into case (a) by enlarging the window to encompass the entire remainder of zone 1. Thus in both cases a full sweep of all objects allocated in zone 1 is completed. At this point, which corresponds to the resetting of the window in Figure 3.9, zone

[7]Note that the address-ordered layout of Figure 3.17 is logically equivalent to the age-ordered layout of Figure 3.9.

1 is abandoned, and its role is assumed by zone 2: allocation proceeds in zone 2, from the last copied object on towards lower addresses. Zone 3 is introduced for the survivors of subsequent collections.[8] In this fashion, allocation and collection can proceed until all zones have been exhausted. Under the assumption that we have a large virtual address space, most programs will not exhaust all available zones; if that does occur, however, the remedy is to move the data back into the high addresses, at the price of one non-generational collection (or relocation).

Choosing a good size for the zones is challenging. On the one hand, a zone should accommodate the amount allocated during one sweep of the collection window: if it does not, then the premature end of sweep will be forced (case (b) above), which will result in a longer-than-normal pause time and disturb the desirable operation regime of the DOF collector. On the other hand, when the DOF collector is in its most desirable regime, Figure 3.10, then the window moves slowly, and the amount allocated during a sweep is large. Therefore, since both the number of zones, and the size of a zone must be large, a large virtual address space is truly necessary for this organization to be effective.

The directional filtering at run-time, shown in the age-order layout in Figure 3.15, becomes very simple in this address layout, as shown in Figure 3.18: if the address of the pointer source is higher than the address of the target, the pointer store need not be recorded. Thus the directional filter can be implemented in a single comparison instruction. Note that the store of a null pointer, normally represented as a machine 0, is also caught by this filter, hence a separate prior check is not needed. The block-local filter remains as described above, but it can be coalesced with the directional check: one of the values compared in the directional check is first rounded to the nearest block boundary. In all, the filters require only a handful of instructions (a bit-masking operation, a comparison, and a conditional branch) and involve no memory accesses (Figure 3.20, p. 48, on the left).

[8]The operating system must cooperate by allowing the garbage collector to acquire and explicitly dispose of segments of the address space.

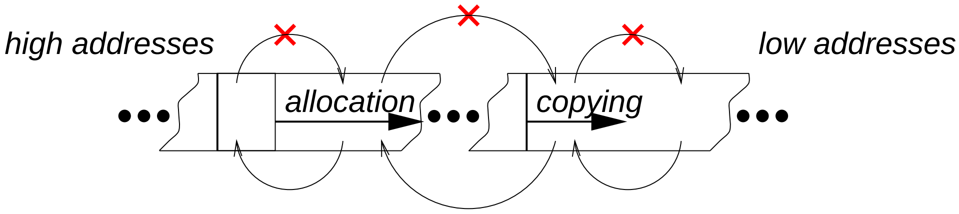

Figure 3.18. Directional filtering for DOF in a large address space.

Thus, the majority of pointer stores, those that do not need to be recorded, are handled quickly. We now consider the remainder. We have observed (as we report in Section 6.1.3) that there are very few duplicate pointer stores for the remembered set of a block in general, and, in particular, between two collections. Hence card marking is not advisable. However, if we wish to keep the cost of the write barrier predictable and low, we should not immediately insert the pointer store, that is, the address of the pointer source, into the remembered set of the pointer target. Instead we can write the same address into a sequential store buffer [Hudson *et al.*, 1991], and subsequently flush the store buffer into the remembered set. There will be a sequential store buffer for each block in current use.

At collection time, we form the set of remembered pointers into the collected region by considering the "union" of the remembered sets of all blocks in the region. We chose to maintain the remembered set of a block between collections as a sequential store buffer; now, rather than construct the remember set explicitly and then compute the union, we simply merge the store lists for the blocks involved, and eliminate pointers that go between two blocks both in the collected region.[9]

Recall that when remembered sets are explicitly maintained, it is necessary to purge the sets of uncollected blocks of entries for pointers from collected blocks, prior to collection. With the address layout proposed here, together with directional filtering, that is no longer necessary,

[9]Note that each store buffer is likely to be a nearly sorted list (with decreasing addresses), because most pointer stores are into the newly allocated area, and thus the sequence of pointer source addresses is correlated with the direction of allocation. This circumstance may influence the choice of the merge algorithm. Also note that an actual merge operation is not absolutely required: it is possible, and it may be less expensive, to recognize duplicate entries on the fly.

since entries cannot become stale: the collected region lies within the highest valid addresses, and no pointers from higher, now invalid addresses, were recorded in the remembered sets for the collected region, since such pointers would have been higher-to-lower address pointers.

The directional filtering described here is cheaper than the traditional generational test [Hosking *et al.*, 1992], which requires indexing into a block table to determine the generations of the pointer source block and the pointer target block, at the cost of two memory accesses. For fairness, however, under the assumption of a large address space, the generational test can also be simplified. We divide the address space into a number of very large blocks, and assign each generation to one block. We place the nursery at the highest addresses, and older generations at progressively lower addresses. With this heap layout, an address comparison acts as a generational test to eliminate younger-to-older stores, and simultaneously handles the null pointer store; a block-local filter is needed to eliminate older-to-younger stores within a generation.

Thus the write barrier cost components—the cost incurred for each pointer store and the additional cost incurred for each pointer store that must be recorded—will be comparable in implementations of the collectors studied. Similarly, the costs incurred at the time of processing each remembered set entry will be comparable. Therefore, in this study we focus on measuring the number of times different collectors invoke each of these operations. We find significant differences between the collectors. Chapter 6 describes these results in detail.

3.2.5 An estimation of copying and pointer-maintenance costs

In subsequent chapters we shall obtain accurate operation counts for the copying and pointer-maintenance actions of garbage collection. If in addition to the number of operations, we know the cost of each operation, we can calculate total collection costs, and by comparing with an actual implementation, validate the cost model.

We performed preliminary experiments with the proposed write barrier on a 292 MHz Alpha 21164 processor. The implementation is presented in Figure 3.19 as source code and in

Figure 3.20 as Alpha assembly code compiled using `gcc -O2`. The null-pointer, block-local, and directional filtering are accomplished using a single test. The estimated overhead of the barrier if the test eliminates the store is 2 cycles.[10] Stores that are not eliminated are recorded, without duplicate removal, in a sequential store list corresponding to the target block. A sequential store list holds a maximum of 15 entries. When a store list fills up, a new one is allocated from a large linear arena; thus each block owns a linked list of sequential store lists. The measured overhead of the barrier, including the writing into a sequential store list and management of sequential store lists, is 11 cycles per store.

Figures 3.21 and 3.22 present the code executed at garbage collection time to scan the list of sequential store lists of a block. The collector retrieves each entry, interprets it as a pointer source, and then reads the current target value of this pointer. (The target may have changed since the time when this store was recorded, so the collector verifies that the current target is within the collected region.) Finally, the collector retrieves the header word of the target object. If the header word is in fact a forwarding pointer, then the object need not, and must not, be copied again. If the header word is a true header, then this is the first time the collector has encountered the object in this collection, and therefore the collector copies the object into to-space. (A forwarding pointer is an object address and its least significant bit is 0, because objects are word-aligned, while addresses are expressed in bytes. We can define a true header to have 1 in its least significant bit to make headers and forwarding pointers distinguishable.) The cost of this entire sequence, sans copying, is 13 cycles per processed entry.

Thus, assuming that filtering succeeds in eliminating 95% of pointer stores on average, and assuming that all recorded stores are eventually processed, the average cost per pointer store is estimated at $0.95 \cdot 2 + 0.05 \cdot (11 + 13) = 3.1$ cycles.

---

[10]The measured overhead on the four-way issue Alpha 21164 processor ranges from 1 to 3 cycles for the three instructions in the test, depending on the surrounding code; the average and median of 2 is used in the absence of any assumptions about the surrounding mutator code.

In addition to the break-down of pointer-maintenance costs, we performed an experiment to measure the components of copying cost in detail. Whereas in the aggregate, the copying cost is roughly proportional to the amount copied, here we examine the contribution of individual operations to that cost. Our model assumes a copying collector, in which a first phase of root processing (processing of global roots and remembered sets) is followed by a Cheney scan of to-space. For each object copied, the collector pays the following three costs: the cost of processing the pointer that keeps the object reachable, the overhead of installing a forwarding pointer and preparation for copying the words of the object, and the overhead of preparation for scanning the pointer fields of the object. Together these three are the per-object cost α_{obj}. For each word copied, there is a cost α_{word}. Finally, in the scanning phase there is an additional cost for each scanned pointer that does not cause an object to be copied (whereas scanning a pointer that does cause an object to be copied is subsumed in α_{obj}). There are three circumstances when this happens: the pointer is null; the pointer points outside the collected region; or the pointer points to an object that has already been copied. In our large address space model, these three circumstances are identified as follows: category *null/lower* encompasses null pointers and pointers to addresses below the collected region (addresses above the collected region are not possible at collection time), and category *duplicate* encompasses pointers into the collected region to objects that have already been copied. The per-pointer-field costs for these two categories are $\alpha_{null/lower}$ and $\alpha_{duplicate}$, respectively. The full copying cost is then

$$c = \alpha_{obj} n_{obj} + \alpha_{word} n_{word} + \alpha_{null/lower} n_{null/lower} + \alpha_{duplicate} n_{duplicate}.$$

We measured the operation costs α in a synthetic copying experiment, in which heaps of different size, and with different object sizes and pointer contents were constructed and copied by a collector. We used an object format applicable to a typed language such as Java: an object header word points to a class record, which points to class-specialized methods to copy and scan the objects. We estimate the following values for the operation costs on the Alpha 21164, for operation within the primary cache: $\alpha_{obj} = 65$ cycles, $\alpha_{word} = 2.5$ cycles,

$\alpha_{null/lower} = 15$ cycles, and $\alpha_{duplicate} = 17$ cycles. These estimates could be used in conjunction with an instrumented garbage collector to validate the detailed model of copying cost.

One must be cautious with these numbers for processor cycles spent on particular operations. They were obtained in a tight loop, and may not reflect actual costs accurately. On the one hand, the cost of memory accesses is discounted, since block table entries are cache-resident. On the other hand, cycle counts may be overestimated, since the processor has no opportunity to interleave the write barrier instructions with other mutator instructions.

3.3 Summary

The proposed class of age-based collectors represents a logical generalization of the concept of generational collection. Age is the simplest criterion for choosing the collection region, and there are established techniques for encoding object age with sufficient accuracy and reasonable overhead; we have also suggested new techniques applicable to large address spaces that promise to reduce overheads further even in the more complex context of age-based collection.

The collectors as described are all of the stop-the-world variety. No explicit provisions for incrementality are suggested. However, we believe that collectors with a small fixed size collected region can provide sufficiently low maximum pause times that they can be considered truly incremental (even if they have to do so at the price of increased total time).

We described the collectors as operating within a unified heap, in which all objects lie. Hence, the same basic policy is applied to newly allocated objects as to older objects, and even long-lived semi-permanent objects. Surely the motivation that has led researchers in the past to consider dividing the heap into ephemeral and mature spaces remains valid, and different age-based schemes could be applied within each division.

We described simple oldest-first, youngest-first, and moving window schemes (Figure 3.23 summarizes the employed abbreviations). The space of other possible designs is vast, and

```
#define BLOCK_SHIFT 16
#define BLOCK_BYTES (1 << BLOCK_SHIFT)
#define BLOCK_MASK (~(BLOCK_BYTES - 1))
#define BLOCK_ADDR(addr) (addr & BLOCK_MASK)
#define SB_SHIFT 7
#define SB_BYTES (1 << SB_SHIFT)
#define SB_WORDS (1 << (SB_SHIFT-3))
#define SB_MASKLOW (SB_BYTES - 1)
typedef struct bte
{
    long int ** store_buffer_pointer; /* where to insert the next entry */
    ... other block information ...
} BTE;
#define BTE_SHIFT (BLOCK_SHIFT - log2 (sizeof(BTE)))
/* the store: */
*reg_source = reg_target;
/* the write barrier: */
if (reg_source < BLOCK_ADDR(reg_target)
{
    /* record the pointer: */
    btep = block_table_offset + (BLOCK_ADDR(reg_target)>>BTE_SHIFT);
    p = btep->store_buffer_pointer;
    p --;
    *p = reg_source;
    if ((p & SB_MASKLOW) == 0)
    {
        /* allocate new store list, set p to beginning of list: */
        previous_ssb_arena_pointer = ssb_arena_pointer;
        ssb_arena_pointer -= SB_WORDS;
        if (ssb_arena_pointer < ssb_arena_low_limit)
            {
            ... allocate new arena ...
            }
        *ssb_arena_pointer = previous_ssb_arena_pointer;
        p = ssb_arena_pointer;
    }
    btep->store_buffer_pointer = p;
}
```

Figure 3.19. Write barrier code.

```
register usage:
$0 is reg_target
$2 is BLOCK_ADDR(reg_target)
$2 is also p
$2 is also previous_ssb_arena_pointer
$3 is btep
$5 is block_table_offset
$10 is ssb_arena_pointer
$11 is reg_source
$13 is ssb_arena_low_limit

the store and the write barrier:
        zapnot $0,252,$2 ;; clear low 16 bits
        cmplt $11,$2,$1
        stq $0,0($11)
        bne $1,$42
$43: continuation of mutator

$42: record the pointer:
        sra $2,13,$1 ;; >>BTE_SHIFT
        addq $5,$1,$3
        ldq $2,0($3)
        subq $2,8,$2
        and $2,127,$1
        stq $11,0($2)
        bne $1,$46
        allocate new store list:
        bis $10,$10,$2
        subq $10,128,$10
        cmple $13,$10,$1
        ...conditional branch to allocate
        new arena (not shown)...
        stq $2,0($10)
        bis $10,$10,$2
$46:
        stq $2,0($3)
        br $31,$43
```

Figure 3.20. Write barrier assembly code.

may offer potential for better performance; for example, hybrid schemes can be envisaged to combine generational collectors' reserve space management with sweeping windows.

```
#define SB_SHIFT 7
#define SB_BYTES (1 << SB_SHIFT)
#define SB_WORDS (1 << (SB_SHIFT-3))
#define SB_MASKLOW (SB_BYTES - 1)
#define SB_CHUNK_ENTRIES (SB_WORDS-1);
#define SB_LINKMASK_VALUE (SB_CHUNK_ENTRIES*8);
#define FWD_POINTER_MASK 1
#define FWD_POINTER_TAG 0
p = actual_block_table [1] . store_buffer_pointer;
while (p != 0)
{
    while ((p & SB_MASKLOW) != SB_LINKMASK_VALUE)
    {
        reg_source = *p;
        reg_target = *reg_source;
        if (reg_target > reg_collection_window_low)
        {
            /* target is in the collected region */
            if ((*reg_target & FWD_POINTER_MASK) != FWD_POINTER_TAG)
            {
                /* the word in the object header is not a forwarding pointer,
                 because it does have the low bit set */
                reg_target_new = ...new address of target object after copying...;
                *reg_source = reg_target_new;
            }
        }
        p++;
    }
    p = *p;
}
```

Figure 3.21. Remembered set processing code.

```
register usage:
$2 is p
$11 is reg_source
$10 is reg_target
$14 is reg_collection_window_low

       lda $5,actual_block_table
$60:
       ldq $2,8($5)
       beq $2,$59
       br $31,$73
$66:
       ldq $11,0($2) ;; reg_source = *p
       ldq $10,0($11)
       cmpule $10,$14,$1 ;; target check
       bne $1,$67
       ldl $1,0($10) ;; load header word
       blbc $1,$67 ;; branch if low bit is 0
       ... forward pointer (not shown); new address is in $0 ...
       stq $0,0($11)
$67:
       addq $2,8,$2 ;; p++
$73:
       and $2,127,$1
       cmpeq $1,120,$1
       beq $1,$66
       ldq $2,0($2) ;; p = *p
       bne $2,$73
$59:
```

Figure 3.22. Remembered set processing assembly code.

- NONGEN: non-generational collector
- TYF: true youngest-first collector
- TOF: true oldest-first collector
- ROF: oldest-first collector with the renewal-age modification
- DOF: old-to-young sweeping collector (deferred-older-first)
- DYF: young-to-old sweeping collector (deferred-younger-first)
- OPT: locally-optimal collector
- FC: prefix for age-based schemes with a collection region of fixed size
- GYF: generational collectors in general
- 2GYF, 3GYF: generational collectors with generations of fixed size, with two and three generations, respectively
- NGYF: generational collector with two variable-sized generations

Figure 3.23. Age-based scheme name abbreviations.