CHAPTER 5

EVALUATION: COPYING COST

In this chapter we focus on the copying cost of collection to the exclusion of other cost factors. We look at the measured copying cost of the various age-based algorithms, including the traditional generational youngest-first collectors and the newly proposed deferred older-first collector. We examine the contribution of excess retention to the copying cost, and search for the lower bounds of copying cost given the constraint of heap size, by looking at a hypothetical optimal collector and at different tentative adaptive collectors.

5.1 Method for evaluating different configurations of a collector

The copying cost of a given garbage collection scheme depends primarily on the size of the available heap space and the settings of configuration parameters. The available heap space is described by a single number, V, since in this study that space is constant. For collection schemes with fixed size of the collected region (FC), that size is the only configuration parameter, and is conveniently expressed as a fraction g of total heap volume V, thus $g = \frac{C}{V}$.

For each benchmark, a suitable set of values is chosen for V. The heap volume must be at least equal to the maximum amount of live data in the program. The performance of any collector, however, can be expected to be extremely poor if V is too close to this minimum. We let V assume up to 9 discrete values between 1.2 and 6 times the minimum, in geometric increments of 20%. Note that fewer than 9 heap sizes are reported for some benchmarks: when the criterion for benchmark relevance discussed above (more than 10 non-generational collections) is not met for the largest tested heap sizes, they are not reported. A suitable set of values is chosen for g as well: we let the fraction assume 20 discrete values between 0 and 1.

For each pair of V and g values, we performed a simulation of the benchmark. For a given heap size V, by varying the configuration parameter g (how *much* data is collected each time), we explore its effects on copying cost, within a fixed strategy of *where* to choose the data to collect.

Similarly, for the generational schemes, specifically 2GYF and 3GYF, the fraction of heap space dedicated to the nursery is the primary configuration parameter. For 2GYF it is the only parameter for a given heap size, and for 3GYF there is an additional parameter in the choice of division of the remainder of the heap between the middle and the oldest generation, but, as we shall see, its influence on performance is less pronounced.

5.1.1 Results

We present the results for each benchmark in turn, and for each benchmark we give a series of plots, ordered by increasing heap size. For each heap size, we show in separate plots (a) the performance of the FC schemes as a function of the fixed fraction collected, and (b) the performance of the GYF schemes as a function of the nursery size.

Each of the figures in this section (Figures 5.7–5.110) consists of two plots, on the left (a) the plot for FC schemes and on the right (b) the plot for GYF schemes. The vertical axis (the ordinate) has the same meaning in both: it gives the relative mark/cons ratio ρ, which is the mark/cons ratio of a particular scheme relative to the mark/cons ratio of the non-generational collector for the same heap size. The scale on the vertical axis is the same for FC and for GYF, and for all heap sizes for the same benchmark. The horizontal axis (the abscissa) in the FC plots gives the size of the collected region expressed as the fraction g of heap size. In the GYF plots, the horizontal axis gives the size of the nursery as a fraction of heap size. It would be misleading to show both in the same graph, since the size of the collected region in GYF is occasionally the whole heap. In plots (a), we have drawn the curves for the following schemes:

FC-TOF, FC-TYF, FC-DOF, FC-DYF, and FC-ROF.[1] In plots (b), we have drawn a curve for the scheme 2GYF, whereas the measurements for the 3GYF scheme are presented as a scatter plot, since the same nursery size can be combined with different divisions of the remaining space into the two older generations. These plots permit us to analyze how the choice of configuration affects the copying cost. Furthermore, for FC schemes, the configuration determines the maximum amount that can be copied on each collection, thus it (approximately) bounds pause times.

5.1.1.1 The FC-TOF collector

We observe that in all benchmarks save the two Tree-Replace programs, FC-TOF performs poorly. Its copying cost is generally higher than that of non-generational collection: $\rho > 1$. When the size of the collected region is close to the whole heap size, FC-TOF performs as well as non-generational, which is to be expected, but when the collected region is reduced, the collection cost rises dramatically, and ultimately, the collector fails. This result is easily explained by the presence of some amount of permanent data in the heap: permanent objects are the oldest objects, hence the collector copies them repeatedly, and if the collected region is so small that it contains nothing but permanent objects, the collector finds no garbage and fails.

For the programs Tree-Replace-Binary and Tree-Replace-Random, on the contrary, FC-TOF is the best scheme, and its copying cost is as low as one half that of non-generational collection. In these (admittedly synthetic) benchmarks, the amount of permanent data is relatively small—only the first several layers of a tree from the root are permanent. Moreover, the subtrees are chosen for replacement *at random*, which means that older subtrees do eventually become garbage. Our intuitive argument in Chapter 1 and Figure 1.1, that the longer the collector waits the more likely it is that an object is garbage, holds in these programs.

[1] If a curve is entirely missing from a plot, it is because no configuration of that scheme successfully completes execution. (See Section 4.3.1.)

5.1.1.2 The FC-ROF collector

The FC-ROF collector behaves just like the FC-TOF collector for large collected region sizes. However, at smaller collected region sizes, the FC-ROF collector diverges from FC-TOF, and is sometimes able to avoid its poor behavior. For instance, in benchmark Swim, with a heap size of 38733 words (Figure 5.53(a), p. 158), whereas the performance of FC-TOF degrades rapidly as g decreases below 0.6, FC-ROF only degrades to $\rho = 1.5$, and then returns to lower levels—but never below 1. An interesting reversal occurs for the two programs Tree-Replace-Binary and Tree-Replace-Random, where FC-TOF works well, and FC-ROF degrades precisely when the size of the collected region is reduced below 0.6 (Figure 5.80(a), p. 169). This phenomenon deserves further investigation.

5.1.1.3 The FC-TYF collector

The FC-TYF collector joins FC-TOF and FC-ROF in the class of collectors that only work well when the collected region is nearly the whole heap. Otherwise, this collector almost always performs worse than the non-generational collector. This is not surprising; if this collector could work in a steady state, it would never examine objects to the left of its collection window. In other words, once a young object survives one nursery collection, it is "tenured". It is surprising that this collector works at all, and this must be due to the fact that programs show phase behavior and do not enter a truly steady state.

5.1.1.4 The FC-DOF collector

The FC-DOF collector is consistently better than other FC schemes, except on the two programs Tree-Replace-Binary and Tree-Replace-Random. It generally works better with *smaller* collected region sizes, except for Toba and Lambda. For many examined programs there exists, for sufficiently large heaps, a characteristic region size below which the FC-DOF collector works particularly well. This passage from acceptable performance to exceptionally good performance is sometimes abrupt, as for StandardNonInteractive at region size $g = 0.9$ (Figure 5.25(a), p. 145), and sometimes gradual, as for Richards, when region size is reduced from

$g = 0.8$ to $g = 0.4$ (Figure 5.85(a), p. 171). The FC-DOF collector is the best among the FC schemes, and we shall investigate its behavior in greater detail in the remainder of this chapter and in Chapter 6.

5.1.1.5 The FC-DYF collector

The FC-DYF collector usually performs better than non-generational collection. However, it is always substantially worse than FC-DOF, from which it differs only in the direction of collection window motion. Its copying cost is remarkably insensitive to the region size parameter for those values where the collector succeeds; these are, however, limited to medium values, whereas for small windows as well as for large windows, this collector is prone to failure. Both the poor performance and the failures (as the ultimate manifestation of poor performance) are caused by the fact that the collection window sweeps the heap too fast. The intuition that survivors of one collection should not be in the collected region of the following collection conspires with the younger-to-older window motion direction in FC-DYF to cause fast window motion, and too many visits to the far ends of the heap where there are many survivors.

5.1.1.6 The GYF collectors

The curve of the 2GYF collector copying cost and the patterns of the 3GYF scatter points reflect the presence of different modes of operation of the generational collector, corresponding to different numbers of younger-generation collections between successive older-generation collections (Figure 5.50(b), p. 157). The resultant 2GYF copying cost is therefore sometimes lowest for medium values of nursery size, as in InteractiveAllCallsOn (Figure 5.12(b), p. 140), sometimes it is lowest for small values of nursery size, as in Swim (Figure 5.50(b), p. 157), and sometimes it is lowest for *large* values of nursery size, as in Lambda-Fact6 (Figure 5.44(b), p. 154).

It will also be observed that the scatter points for 3GYF are clustered closely, and usually not far below or above the line corresponding to 2GYF. Thus, the copying cost of 3GYF is usually comparable to that of 2GYF. Occasionally we see a configuration of 3GYF with only half the copying cost of 2GYF with the same nursery size, as in Lambda-Fact6 (Figure 5.45(b), p. 154); on the other hand, there are programs where no 3GYF configuration is better than 2GYF, as in Toba (Figure 5.102(b), p. 179), hence the added overhead[2] of 3GYF is only detrimental.

5.2 Method for comparing various collection schemes

We now summarize the findings of the preceding section. In order to compare different collection schemes, we shall restrict each to operate within the same heap size, but we shall let it assume the *best* configuration. In other words, we give each scheme the benefit of the best static choice of the size of the collected region, or the sizes of the various generations. The plots in Figures 5.111–5.124 (pp. 183–190) show the copying cost for these best configurations. Along the horizontal axis is the heap size allotted, and along the vertical axis the relative mark/cons ratio ρ. There are 8 curves in each plot, for the collection schemes FC-TOF, FC-TYF, FC-DOF, FC-DYF, FC-ROF, NGYF,[3] 2GYF, and 3GYF.

In the plots for Interactive-TextEditing (Figure 5.112), StandardNonInteractive (Figure 5.113), HeapSim (Figure 5.114), Richards (Figure 5.121), and Bloat-Bloat (Figure 5.123), there is a clear separation between the schemes FC-TOF, FC-TYF, FC-DYF, and FC-ROF, which exhibit high copying cost, and the schemes FC-DOF, NGYF, 2GYF, and 3GYF, which exhibit low copying cost. In other programs the separation is not as clear, and indeed for Tree-Replace, the FC-TOF and FC-ROF schemes work best.

There is not a uniform dependence of ρ on heap size (for a specific collection scheme): whereas in Richards ρ decreases with heap size for the four better-performing schemes, in Bloat-Bloat and Swim it increases. Note that the *absolute* copying cost μ nevertheless decreases—ρ is relative to the cost of non-generational collection, which itself is decreasing. This fact is

[2] Recall the elision of the write barrier at collection time (Section 3.2.2).

[3] Since NGYF has no configuration parameters, it was not discussed in the preceding section. Heap size uniquely determines the value of ρ reported here for NGYF.

evidenced in Figures 5.125–5.138, pp. 191–198, which display the *absolute* copying cost, expressed as the mark/cons ratio μ, for the best configuration of each scheme.

With respect to copying cost, the newly introduced scheme FC-DOF is competitive with generational collectors for all examined programs, and markedly better for some (Interactive-AllCallsOn, Interactive-TextEditing, StandardNonInteractive, and Richards). For Richards with a heap size of 4914 words, the copying cost of 3GYF is 10.37 times higher than that of FC-DOF.

Observing the performance of all the collectors, we note that most programs permit region-based schemes to cut down the copying cost of collection significantly with respect to non-generational collection: for Interactive-AllCallsOn to the fraction 0.13; for Interactive-TextEditing to 0.18; for StandardNonInteractive to 0.045; for Richards to 0.029; and for HeapSim, Swim, Tomcatv, and Bloat to the vicinity of 0.2. However, the copying cost of Toba is not reduced below 0.6, Tree-Replace-Binary and Tree-Replace-Random are only reduced to about 0.5, and the most intransigent Lambda-Fact5 and Lambda-Fact6 are not reduced below 0.7.

5.3 Method for recognizing and measuring the excess retention cost

Let us examine a garbage collector that considers a collected region C smaller than the entire heap, in order to limit the amount of work done at each collection. The maximum copying cost is bounded by the size of the collected region, since in the worst case all objects within it may be live. The exact computation of the set of live objects, however, requires an examination (pointer tracing) of the entire heap, and not just C, at a cost not much lower than the cost of *collecting* the entire heap. Instead, collectors make a conservative approximation: they assume that all objects in the uncollected region U are live, hence that every pointer that crosses from U to C lies on a path from a root, even if that is not actually the case. *Excess retention* is that part of the collection copying cost which is caused by the inaccuracy of this assumption. On a single collection, the excess retention is manifested as an increase in the volume of survivors. Since the volume freed is correspondingly reduced, the next collection must take place sooner, and the overall number of collections is increased.

To measure how large the excess retention part of the copying cost is, we devised a variant of collector simulation (deployed in all the simulated schemes) that does not assume liveness for the uncollected region. Instead, the survivor set S among the collected objects C is precisely the set of objects that are not *known-dead*. This manner of simulation is possible because our trace contains accurate liveness information, and can thus act as an oracle guiding the collection. The difference between the actual copying cost, and the copying cost as it would be with this oracle, gives the excess retention cost.

5.3.1 Results

Each of the figures for this section (Figures 5.139–5.242, pp. 199–243) consists of two plots, on the left (a) the plot for FC schemes and on the right (b) the plot for GYF schemes. The vertical axis (the ordinate) shows the excess retention for the configuration, expressed as the ratio of the actual copying cost to the oracular copying cost. Note that the variation of the excess retention ratio is so large that we could not maintain the same scale in the plots, thus visual comparisons between FC and GYF are discouraged. The horizontal axis (the abscissa) in the FC plots gives the size of the collected region expressed as the fraction g of heap size. In the GYF plots, the horizontal axis gives the size of the nursery as a fraction of heap size.

Let us first note that in some rare configurations, both for the FC and for the GYF schemes, the excess retention ratio is below 1, for instance in Interactive-AllCallsOn, Figure 5.141, p. 200. It may seem odd at first glance that the collector without the benefit of an oracle performs better than the one with it, but recall that, if the oracle makes an actual difference in the determination of the survivor set S, then the two collectors will perform their collections at different instants, and, as we discussed in Section 4.2.2.2, the choice of collection instants introduces a slight "random" effect. Having dispatched this anomaly, we note that excess retention displays widely different behavior for the several benchmarks, and for different schemes.

The excess retention of the GYF schemes is low, below 10% (ratio below 1.1) for most programs, with Lambda (Figures 5.166–5.181, pp.211–217), Tree-replace-Random (Figures 5.205–5.212, pp.228–230), and Toba (Figures 5.234–5.242, pp.240–243) as exceptions.

Excess retention does play a critical role in determining the performance of collectors for some programs. For instance, for Lambda-Fact6, we noticed that none of the collectors has a copying cost much lower than non-generational (Figure 5.48, p. 155), and now we can observe in the corresponding plot of excess retention (Figure 5.180, p. 216) that for a nursery size of one-half heap size, the excess ratio of 2GYF is 2, hence with an oracle the copying cost would be halved from $\rho = 1.5$ to $\rho = 0.75$. But the effect on FC-DOF is far stronger: the excess ratio is 50 for collected region size equal to 0.2 of heap size, hence with an oracle the copying cost would be reduced from $\rho = 3$ to $\rho = 0.06$. The previously noted intransigence of Lambda-Fact6 with respect to copying cost thus has its origin in a pointer structure that causes very high excess retention for those schemes (GYF, DOF) that we found to work well on other programs.

On the other hand, consider the benchmark Richards with a heap size of 4032, for which the copying costs are shown in Figure 5.85 on p. 171, and the excess retention in Figure 5.217 on p. 232. The actual copying cost of FC-DOF decreases as the size of the region collected is decreased from 0.8 down to 0.3, *even though* the excess retention ratio increases sharply from 1 to 8. Meanwhile the 2GYF collector enjoys negligible excess retention, yet its copying cost is much higher than that of FC-DOF.

In summary, excess retention sometimes dominantly influences performance, but sometimes its effects are negligible. Whereas it could have been surmised that schemes that collect regions in the older parts of the heap suffer from prohibitively high excess retention because of the supposed predominance of younger-to-older pointers, this case does not occur in our benchmarks.

5.4 Methods for examining the lower limits of copying cost and the feasibility of adaptation

We have seen that the GYF and FC-DOF schemes compete for lowest copying cost, and that FC-DOF is sometimes considerably better. We now examine if copying cost can be reduced further, and how. We remain within the class of FC schemes—thus each collection collects a region of fixed size—but now we allow more latitude in the choice of where the collected region lies within the heap.

5.4.1 Locally-optimal copying cost

We described the locally-optimal schemes in Section 3.1.3. It is straightforward to simulate them in an object-level garbage collection simulator. On each collection, we place the collection window at all possible positions in the heap, and for each position we perform a collection as usual, except for the final step of deleting the garbage found. Thus for each possible position we calculate the corresponding mark/cons ratio, then choose the position with the lowest one, and complete the collection with the window at that position.

The number of possible positions is at most the number of objects in the heap, and somewhat smaller because the window must wholly lie within the heap. However, it can be a large number if the heap consists of many small objects, hence this simulation can only be carried out for smaller benchmarks. In plots in Figures 5.243–5.266, pp. 244–253 (at the end of this chapter), we show the copying costs of the locally optimal schemes together with the age-based schemes considered before, for three representative benchmarks, StandardNonInteractive, Lambda-Fact5, and Tree-Replace-Random.

For StandardNonInteractive, the locally optimal collector only matches the performance of FC-DOF, and is in some configurations not as good as FC-DOF (recall from Section 3.1.3 that FC-OPT is not necessarily optimal globally!). For Lambda-Fact5, on the other hand, FC-OPT is significantly better than any of the realistic FC collectors, and any of the generational collectors, and achieves a copying cost of less than half of the non-generational collector's cost. The situation is similar for Tree-Replace-Random, where FC-OPT has much lower cost than FC-TOF, the best realistic scheme considered for this benchmark. The explanation is straightforward in this case: the collected region is positioned to include exactly the objects of the subtree that has just been replaced (provided that the size of the collected region is not much larger than the subtree).

5.4.2 Window motion

In order to understand the behavior of FC collectors, we observe how the position of the collected region within the heap changes from one collection to the next. We call this *window motion*. We shall use window motion analysis to guide the design of adaptive collection in Section 5.4.5.2. A window motion plot (such as Figure 5.1, p. 132) shows the location of the collection window in the heap in successive collections. The window is contained between the two marks of the old end and the young end, shown along the vertical axis. The horizontal axis show the progression of collections, and is gauged according to the amount of allocation, so that the window motion graph can be compared against the live profiles (Section 4.2.3) and the detailed heap profiles (next section). The *horizontal displacement* of the window from one collection to the next equals the amount allocated between collections. The *vertical displacement* of the window, for the DOF regime of window motion, equals the amount of survivors of the preceding collection. Therefore, for DOF, the *slope* of the window motion curves equals the instantaneous mark/cons ratio. Hence good performance shows as shallow slope, and ideal performance would show as a flat window motion curve.

5.4.3 Demise point analysis

Our fully accurate trace permits simulation and evaluation of any garbage collection algorithm. The algorithms differ in the choice of the instants when they perform collections, and, at each collection, the choice of the collected region. These differences cause the heaps to contain different amounts of garbage data and in different positions. The sets of live objects, however, are necessarily the same. Objects in the heap are ordered according to their time of allocation, hence the live objects are not only equal as a set, but also as an ordered list, across all collectors.

When the collection algorithm decides which region it should collect, its goal is to choose that region which has the greatest garbage content. We have observed that garbage collectors differ in the amount and position of garbage that they keep in the heap. However, the component of garbage that just came into being—objects that have most recently died—is equal for all collectors. Indeed, if a collector succeeds in collecting objects immediately after they become garbage (or very soon), that will be the only component of garbage. Therefore, the focus of our method is to identify when and where fresh garbage arises.

To answer the temporal question, we imagine a collector that *immediately* discovers all objects that become garbage, and immediately collects them. Clearly, the information in the trace as described in Section 4.1 suffices to simulate this collector. To answer the spatial question, recall that ordering by age is assumed. Therefore, in a linear list of objects currently in the heap, the collector can mark those it has just discovered are garbage, and report their position before collecting them away. The position is given as the number of words from the young end of the heap, for each word of a garbage object.

Since the purpose of the method is to guide designers of garbage collection algorithms in their analysis of collector performance, we turn now to the task of presenting visually the findings of the described ideal collector.

5.4.3.1 Visualization: demise maps

We observed that the ideal collector reports the demise of each object as soon as it happens. More precisely, it reports the demise of each *word* of the object, giving the time of demise exactly, as the amount of allocation in the program up to that point. It is then natural to construct a two-dimensional grid, with one dimension corresponding to the allocation time, and the other to the position of the word in the heap, and the grid point is set if the word at the corresponding heap position died at the corresponding time. We lay the time along the longer, horizontal, axis, and heap position along the vertical axis. The heap position is measured from the *young*, most recently allocated end of the heap in age order. A demise point is shown as a black dot. Thus we have constructed a *demise bitmap*.

In practice, however, we must take an additional step to produce visual displays of demise. Note that the dimensions of the demise bitmap are (Total words allocated) $\times$ (Maximum words live), which, even for the relatively small benchmarks in our suite, can be of the order $10^6 \times 10^8$. Such a large bitmap cannot be graphically rendered directly. Instead, we divide both the time and the space axis into clusters, compute the number of set bits of the demise bitmap in each cluster, and assign greyscale levels to each cluster accordingly. Having considered the capability of the rendering device and the size of the final images, and having experimented with different granularities, we settled on a 100×660 greyscale map. Thus obtained demise greyscale maps for our benchmark are presented in Figures 5.267–5.280 on pp. 254–267. Each panel is in three aligned displays: the middle display shows the demise points, while the top and bottom display show heap profiles, which we presently discuss (the top display shows the curves of equal time of allocation; and the bottom display shows the curves of equal age).

5.4.3.2 Visualization: heap profiles

The demise map faithfully presents temporally local information. We know which words have just died in the sense that the map shows *where* they were. It does not tell us the identity of these words—at which point were they allocated. However, the ideal trace can be used to produce a telling visual display of this information, in the form of a *heap profile*. The duration of program execution is divided into a number of equal-sized segments, and the heap profile tracks the motion of each segment in the heap from the time of allocation onwards. Since the heap is age-ordered, the objects allocated during one segment remain contiguous in the heap. Thus it is possible to draw nonintersecting lines of constant allocation time for each of the times at the boundary of two allocation segments. If we again show the heap positions with youngest end at the bottom, then subsequent segments are stacked below previous segments in a banded plot. The height of a band indicates the currently surviving amount of data from a given segment of allocation. The total height of all bands indicates the total amount of live data in the heap.

The information in this plot is, in principle, equivalent to that in the demise map, but it has different visual properties. For instance, in benchmark Interactive-AllCallsOn, Figure 5.267, it is easy to see a correspondence between the sharp fall of the lines in the heap profile and the stripe of demise points at time 10200. However, it would be difficult to discern in the heap profile a corresponding feature for the stripe of demise points at time 1000, because it happens when the lines in the heap profile are mostly rising. Thus the two forms of display are complementary.

Lastly, the bottom displays of Figures 5.267–5.280 show the heap profiles using lines of constant *age*, allowing us to observe the relative participation of younger and older objects in the heap, as execution progresses.

Heap profiling is not a new concept. It has been used in analyzing heap usage in lazy functional programming languages [Runciman and Wakeling, 1992; Runciman and Röjemo, 1995; Sansom, 1994; Sansom and Peyton Jones, 1994; Röjemo and Runciman, 1996], where, however, the grouping criterion of interest is not the age of objects, but the function-point that allocates them. We earlier proposed a related three-dimensional visualization based on object age [Stefanović, 1993a; Stefanović and Moss, 1994].

5.4.3.3 Visualization: position mortality

The demise maps and heap profiles provide a graphical representation of the time-varying behavior of a program, but we nevertheless desire a time-averaged summary. Even if the demise maps show that garbage will occasionally arise anywhere in the heap, we want to know where most of it should be expected, in the hope that a collector might work well by concentrating its efforts there. A useful summary is obtained by counting the number of demise points that occur at each heap position. Note that if we ascribe demise not to an object, as we usually do, but instead to the heap position the object occupies, then we can speak of *position* mortality, as the likelihood that the word at certain position (whichever that word is) will become garbage upon the next time increment. In the frequency interpretation of probability, this likelihood is exactly given by the count of demise points. We show the plots of this quantity in Figures 5.281–5.294 on pp. 268–275. The horizontal axis corresponds to heap position, with the young end at 0. The vertical axis, on a logarithmic scale, gives the number of demise points at each heap position.

5.4.4 Case studies

We now look in detail at the spatio-temporal behavior of benchmarks Richards and Lambda-Fact5. In addition to discussing the information directly available in the graphs obtained by the method of the preceding section, we shall relate it to the object lifetime statistics, and the observed performance of different age-based garbage collection schemes. This analysis will serve as the basis for improving the copying cost of collection in the following section.

5.4.4.1 Benchmark Richards

The benchmark Richards is characterized by a small amount of live data in comparison with its long duration. The average lifetime of objects is short, which is reflected in the strong band of demise points (black) in the demise map (Figure 5.277) along the bottom edge of the map, corresponding to the demise of recently allocated objects. A long-term periodic pattern is clear as well, both in the demise map, and in the heap profile, with about four periods (of duration approximately 900000 words) completing. The end of a period is marked in the demise map by a sudden demise of a large number of words in the mid-to-older part of the heap.

The heap profile is revealing: the main feature is the wide band of oldest data, which consists of objects allocated entirely within the first segment or about 60000 words. Once the size of this segment has dropped to about 500 words, it remains constant for the remainder of program execution. In other words, these are the objects that the program uses permanently:

its “static” data. It is worth noting that these objects represent a large fraction of the total live amount, above one third. Indeed, in almost all the benchmarks we examined, there is a significant amount of static data. In addition to the band that survives from the very start of the program, within each period there is an additional accreting set of bands, which die together at the end of their period. For the duration of the period, these bands also act as static data. Underneath we observe a region of objects that survive a short time, enough to reach position 200-300 in the heap, and then die.

The presence of a large quantity of static data as the oldest data in the heap has a strong detrimental effect on the performance of garbage collector algorithms that ever choose to collect the older region of the heap. It is remarkable, however, that the FC-DOF algorithm, which repeatedly visits the oldest part of the heap, once for every rightward sweep (Section 3.1.1), works quite well on the Richards benchmark, and outperforms all examined age-based schemes, including generational collection—in spite of the presence of static data. It does so by sweeping the static data *fast*, and then focusing on the region of high demise. In Section 5.4.5.1, we modify that algorithm to avoid the static data and find that its copying cost improves further.

5.4.4.2 Benchmark Lambda-Fact5

The benchmark Lambda-Fact5 is of particular interest because we found that none of the examined garbage collection algorithms performs well on it. Examination of the heap profile, Figure 5.271 reveals that there is a very large fraction of static data, which hampers older-first algorithms. The static data, allocated during the first 10% of program execution, account for over half of the live amount present in the heap. In addition, there is some accumulation of objects allocated later on, seen as an increasingly wide band of narrowly spaced lines.

On the other hand, the demise map shows significant demise areas in middle heap positions, especially as a periodic pattern during the latter half of program execution. The presence of these demise points precludes the dominance of points in the very youngest part of the heap and hampers youngest-first algorithms, such as traditional generational collection. The plot of the distribution of demise points by position in Figure 5.285 emphasizes this observation.

Generational collection, unless given a very large heap space, works poorly with small nursery (youngest generation) sizes because too much data is promoted out of the nursery (Figure 5.36(b)); if the nursery size is increased, the above-mentioned demise regions are included, and the performance improves, but even then it only begins to approach non-generational collector performance (when the nursery is so large that the collector behaves almost as a non-generational collector). Similarly, age-based collectors with constant collected region size perform poorly on this benchmark (Figure 5.36(a)), no better than the non-generational collector. However, the hypothetical FC-OPT collector is capable of markedly better performance, with copying cost only 27% of the non-generational for heap size 11402 (Figure 5.254).

5.4.5 Adaptive FC collection

In this section we examine how to improve the performance of collection by modifying the idea of sweeping the heap with the collection window in successive collections. We are guided by the demise map, and by the observed motion of the collection window in the locally-optimal collector. We undertake exploratory case studies of *adaptive* policies within the class of age-based collectors with a fixed collected region size. On the example of the Richards benchmark we shall examine how to remove the permanent data of a program from consideration, and on the example of the Lambda-Fact5 benchmark we shall examine how to adjust the position of the collection region adaptively in response to collection results. In both cases, we shall find it possible to reduce the copying cost of collection over fixed policies.

5.4.5.1 Eliminating the "static" data in Richards

We noted in Section 5.4.4.1 that the presence of a substantial amount of permanent, "static" data in the oldest region of the heap degrades the performance of DOF collection, because collections that examine the oldest region find little garbage, or even fail if they find no garbage. If, however, the collector knew that the oldest region contains such permanent data, it could skip over them, never trying to collect them. It would thus be able to sweep over only the part of the heap where there is some object demise, and potentially incur a lower copying cost. The danger, of course, is if the "static" region is incorrectly identified as such: if some objects in the region that the collector never considers do in fact become garbage, then two penalties are paid. First, the space occupied by these objects is not reclaimed, and the collector works with a heap effectively smaller than it needs to be; and second, since these objects are considered live, any pointers they might have into the collected region cause excess retention. These dangers are familiar from the parallel situation of *premature tenuring* of objects in generational garbage collection [Ungar, 1984; Ungar, 1986]. Most programs, however, do have a region of permanent data, and the question is only how big it is.

The collector should not immediately exclude the oldest region. The heap profiles (Figures 5.267–5.280) show that a significant amount of allocation occurs before the permanent data consolidate, and during this initial period, there is garbage to be collected from among the oldest data as well. Hence, we introduce an adaptive thresholding policy to recognize when the oldest region begins to contain mostly permanent data. The collector initially operates in the usual FC-DOF manner. On each collection upon a window reset, i.e., a collection of an oldest region, the survivor ratio for the collection is computed. As soon as the ratio is above a threshold value θ, all subsequent window resets skip the oldest region.

We take the benchmark Richards for our case study, with the heap size of 4032 words. We use a window size of 1088 words, or $g = 0.27$, for which the FC-DOF collector achieves a copying cost of $\mu = 0.03181$. With the same heap size, the NONGEN collector has a copying cost $\mu = 0.3937$, the best configuration of the 2GYF collector has $\mu = 0.2198$ (with nursery size 2056 words), the best configuration of the 3GYF collector has $\mu = 0.1221$ (with nursery size 2214 words), and the best configuration of the FC-DOF collector has $\mu = 0.02975$ (with window size 1572 words, or $g = 0.39$).

Our first experiment is to supply to the collector the size of the region of permanent data. This value can be read off the heap profile, and for Richards it is 490 words. We expect that if θ

is chosen too low, then the region of permanent data will be taken out of consideration too soon, whereas if θ is chosen too high, the region will be excluded too late to improve performance. Experimentation with different values of θ shows that results are not exceedingly sensitive to the choice, as shown in Table 5.1.

Table 5.1. Copying cost dependence on the adaptation threshold

θ	μ
0.2	0.02759
0.3	0.02734
0.4	0.02734
0.5	0.02734
0.6	0.02760
0.7	0.02830
0.8	0.03260

The best range of threshold values, $0.3 \leqslant \theta \leqslant 0.5$ includes the value $\frac{490}{1088} = 0.45$, and results in a copying cost of $\mu = 0.02734$. Thus, it was possible to reduce the copying cost by 14% with respect to the ordinary FC-DOF collector with the same window size, and in fact by 8% with respect to the best configuration of the ordinary FC-DOF collector.

In the second experiment, the collector is not told the amount of permanent data, but tries to guess it using the following heuristic: each time the first collection following a window reset results in a survivor ratio above the threshold, multiply the amount of *survivors* by a fixed fraction φ, and add the result to the current estimate of the amount of permanent data. The collector guesses that the high survivor ratio for this collection arose because there were permanent data in the collected region, and that they were the oldest data in the region. It is willing to subject the remaining fraction $1 - \varphi$ of survivors to examination at the following reset. Thus the heuristic has two parameters: the threshold θ and the fraction φ. Note that this heuristic is only able to *increase* the estimate, starting from an initial zero, but never to revise the estimate downward. Therefore the danger is that if θ is too low, or if φ is too high, the oldest region may prematurely be considered permanent. We found that the best values are 0.8 for the threshold θ and 0.5 for the fraction φ, in which case the copying cost is $\mu = 0.029$. This result leads us to believe that it is possible to devise adaptive schemes to sense the presence and gauge the size of permanent data, although our simplistic heuristic is too sensitive to the choice of the two parameters to be reliably used.

5.4.5.2 Choosing the collected region in Lambda-Fact5

Instead of restricting the window motion to the region recognized as non-permanent data, we can modify the behavior of the DOF collector more subtly. The window motion of DOF is based on the intuitive premise that survivors of a collection should not be subject to the immediately succeeding collection. Therefore the window moved rightward over the survivors (Figure 3.9). This policy has excellent performance when the collection survivor ratio is zero or extremely low (as illustrated in Figure 3.10), which is the case in practice for several benchmarks, including Richards and StandardNonInteractive (Section 5.1.1.4). This policy suffers if the survivor ratio is not very low, however, as in Lambda-Fact5. In spite of the many demise points in the middle range of the heap for an ideal collector (demise points map in Figure 5.271), the survivor ratios in actual runs of FC-DOF are high. In Figure 5.2 we show the measurements of the survivor ratios for each collection of the FC-DOF collector on the Lambda-Fact5 benchmark, with a heap size of 11402 words and a window size of 4446 words, for which a copying cost of $\mu = 1.27$ is achieved. The FC-DOF collector performs a total of 150 collections, and each shows up as a point in the scatter plot. The horizontal axis gives the position of the window (indicated by the amount of data between the old end of the window from the old end of the heap), and the vertical axis gives the survivor ratio of the collection (the ratio of the amount of survivors to the window size). The corresponding window motion diagram is in Figure 5.1. Thus, FC-DOF suffers because its window moves too fast across the middle region where it might encounter acceptable survivor ratios, and is too often reset to the old end, window position 0, where survivor ratios are high. Contrast this with the run of the FC-OPT collector with the same heap size and window size, which achieves $\mu = 0.198$. Its window motion graph is in Figure 5.3, and the survivor ratio scatter plot in Figure 5.4. Observing that FC-OPT moves its window much more slowly, we shall try to emulate its behavior using the following heuristic window motion policy, called the *floating window*.

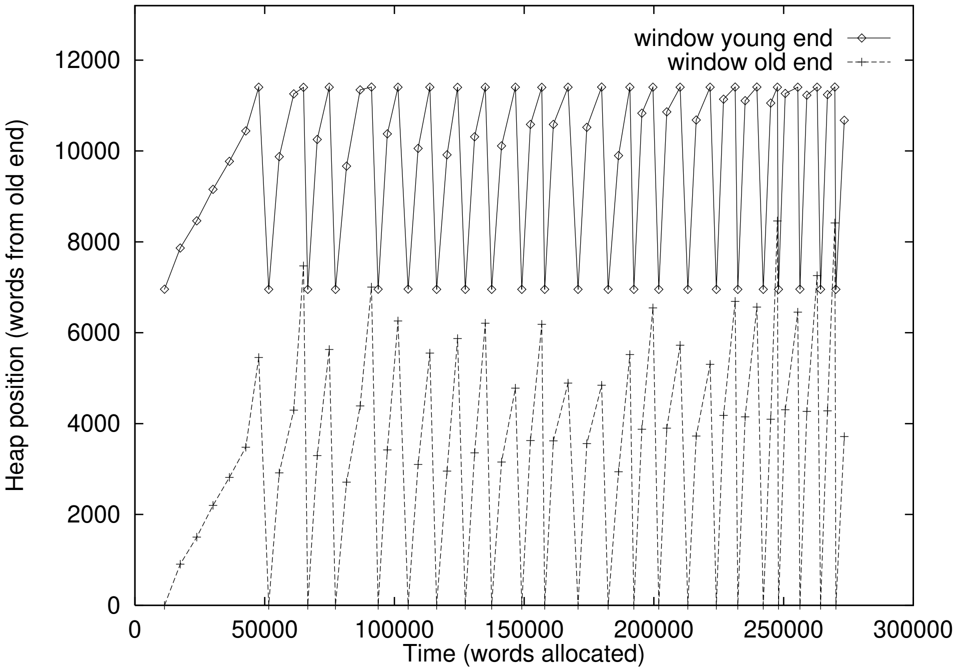

Figure 5.1. Window motion in the FC-DOF scheme.

The position of the window of collection is defined as the amount of data in the age-ordered heap between the old end of the heap and the old end of the window of collection. Let X be the current position of the window of collection. Let W be the size of the window of collection. Upon collection, we find that an amount S has survived out of W. We decide the position for the next collection as $X' = X + \Delta X$ (but resetting X' to zero after the young end of the heap is reached). The increment ΔX is determined as follows. If $S = 0$, then $\Delta X = 0$. Otherwise, note the position p_i of each surviving word in the collection window relative to the old end of the window. The average survivor position is $\bar{p} = \frac{1}{S}\sum_S p_i$. The relative bias is the relative difference between $\bar{p}$ and the value $\frac{W}{2}$ that would be obtained if survivors were evenly spread in the collection window: $r = \frac{\frac{W}{2} - \bar{p}}{\frac{W}{2}}$. Thus $-1 \leqslant r \leqslant 1$. Given an additional fixed external bias e, we set $\Delta X = (r+e)S$. The window is initially placed at X equal to one-half heap size.

In other words, the motion of the window is governed by two forces. An external bias pushes the window in a fixed direction. In particular, the value 1 for the external bias cor-

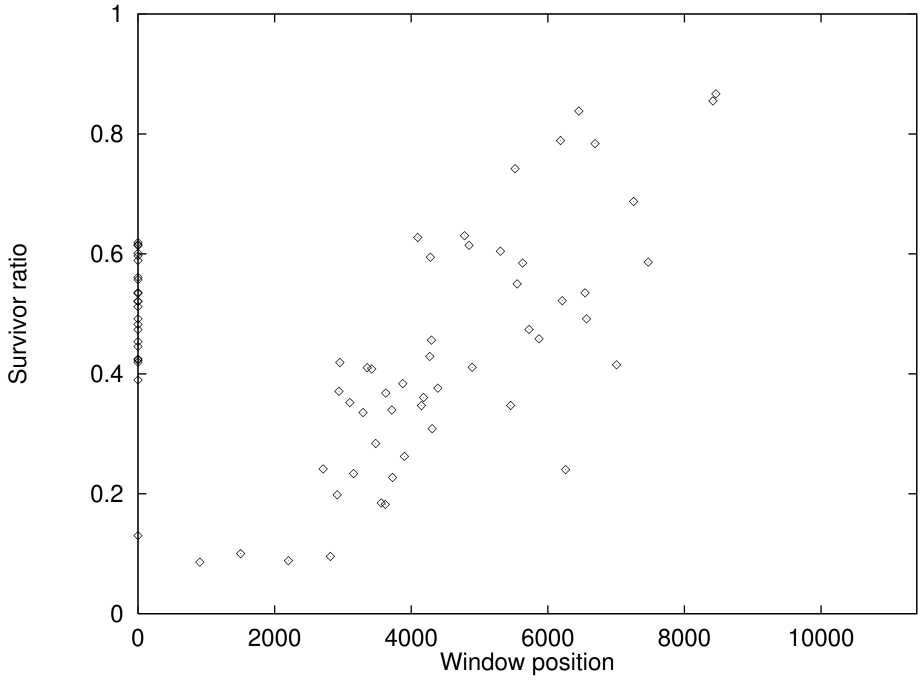

Figure 5.2. Window survivor ratios in the FC-DOF scheme.

responds to the ordinary FC-DOF collector, pushing the window in the direction of younger objects. The second force uses feedback information from the completed collection. Intuitively, if most of the survivors of the collection were found in one part of the window, then the window should be pushed in the opposite direction. For example, if the window straddles the region of permanent data, then most of the survivors will have come from the older part of the window, and it should be pushed towards younger data. We calculate a simple heuristic measure of the desired push as the average position of the survivors within the window. If the average survivor position is left of center, the force is rightwards, and vice versa.

Recall that our test case is the Lambda-Fact5 benchmark, with a heap size of 11402 words and a window size of 4446 words, or $g = 0.39$, for which the FC-DOF collector achieves a copying cost of $\mu = 1.27$. With that heap size, the NONGEN collector has a copying cost $\mu = 0.59$, the best configuration of the 2GYF collector has $\mu = 0.55$, the best configuration of the 3GYF collector has $\mu = 0.56$, and the best configuration of the FC-DOF collector has $\mu = 0.52$. Experiments with different values of the external bias e produce the following results, shown in Table 5.2.

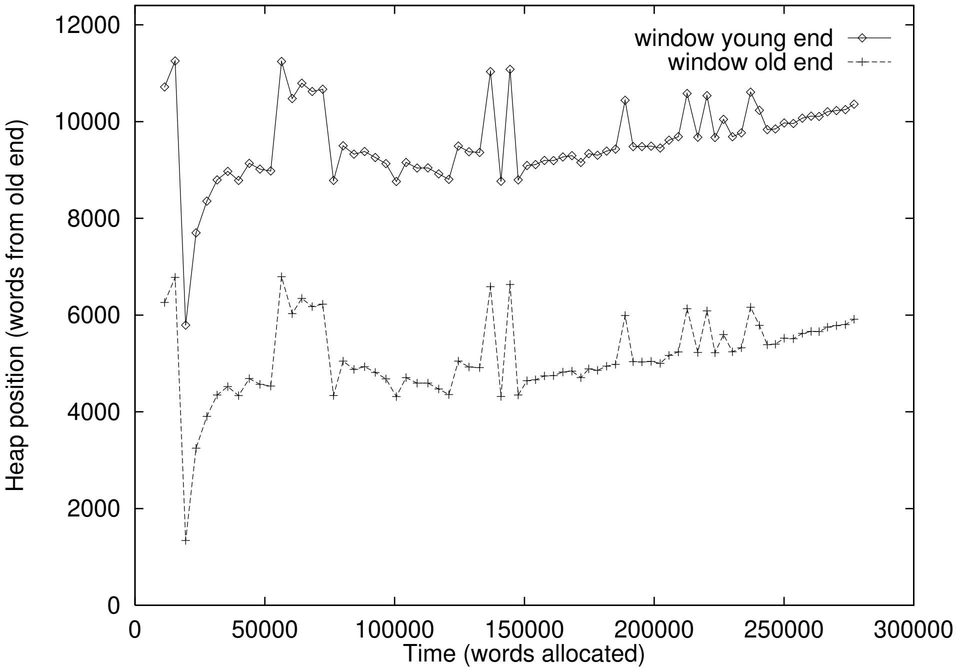

Figure 5.3. Window motion in the FC-OPT scheme.

The lowest achieved copying cost with the floating window policy is $\mu = 0.37$. This is 28% below the best of the fixed schemes for any configuration at the same heap size, namely $\mu = 0.52$ for FC-DOF with a window size of 10135 ($g = 0.88$). We observe the window motion of the adaptive scheme in Figure 5.5, and its scatter plot of window position vs. survivor ratio in Figure 5.6.

Table 5.2. Copying cost dependence on external bias

e	μ
-0.2	0.64
-0.1	0.43
0.0	0.52
0.1	0.46
0.2	0.44
0.3	0.37
0.4	0.40
0.5	0.48

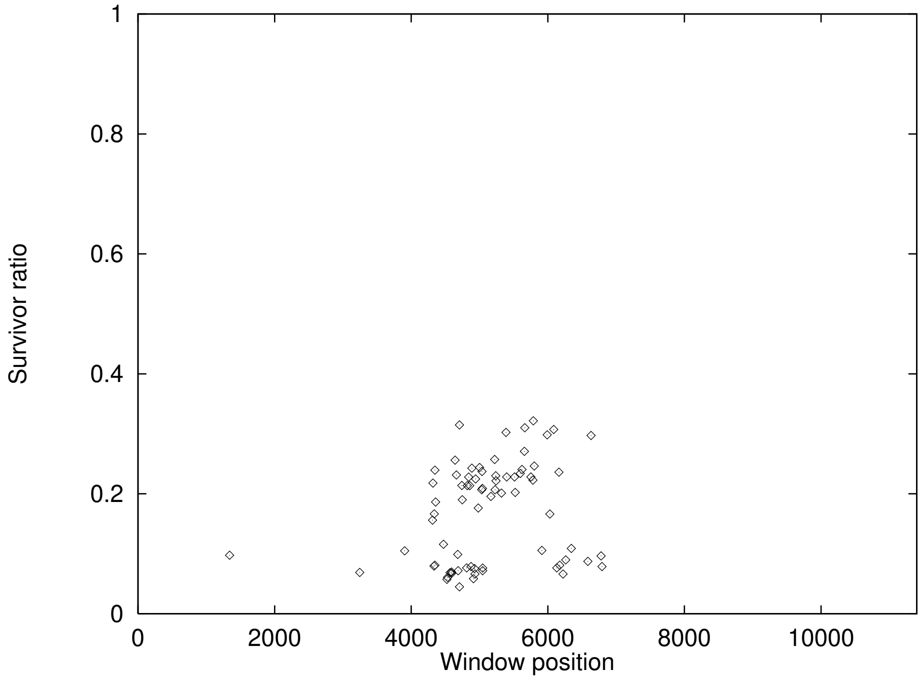

Figure 5.4. Window survivor ratios in the FC-OPT scheme.

The floating window policy is relatively robust with respect to the choice of the external bias, since for the entire region $-0.1 \leqslant e \leqslant 0.5$, the scheme is better than any of the fixed schemes. On the other hand, the FC-OPT copying cost of $\mu = 0.198$, for the same heap size and the same window size, is still another 47% lower.

A final word of caution is in order. We have seen that adaptive variants of FC schemes, and of FC-DOF in particular, can lower the copying cost of collection, and that there exists, for some programs, still more room for improvement, as indicated by the locally-optimal schemes. However, to achieve this further reduction of copying cost requires window motion policies more general than, for instance, the predictable linear sweep of FC-DOF. Recall from Section 3.2.2 that effective pointer filtering at the write barrier exploited the predictability of window motion. Hence, the ultimate reduction of copying cost may well be offset by an increase of pointer management costs.

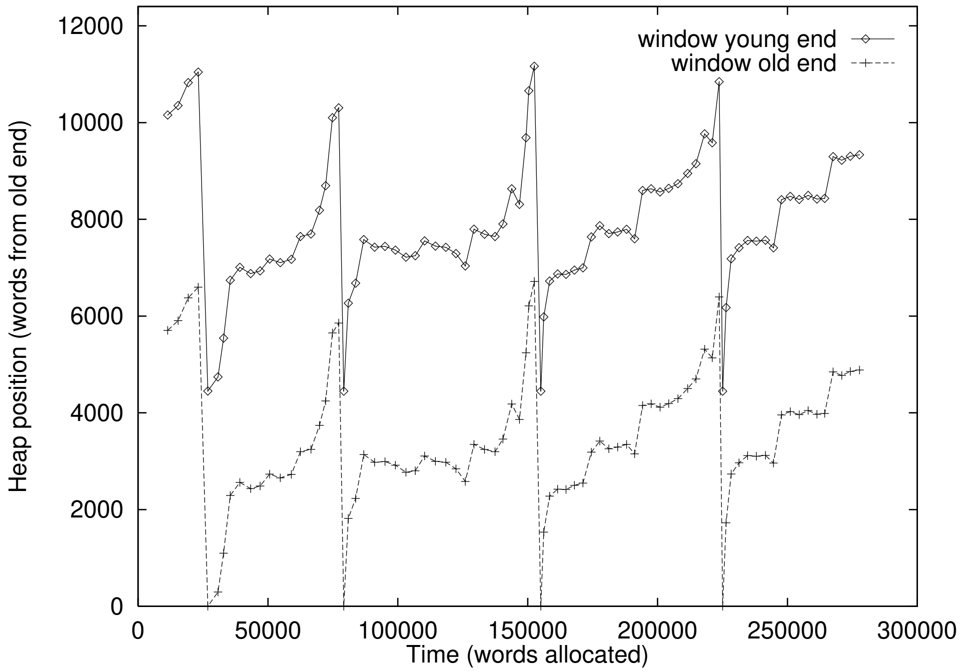

Figure 5.5. Window motion in the adaptive scheme.

5.5 Summary

This chapter examined the copying cost of garbage collection for several age-based algorithms. The main finding is that the promise of low copying cost of our proposed FC-DOF collector is delivered on many actual allocation loads. This result could not be expected given the prior understanding of generational collector performance. Particularly revealing is the fact that excess retention effects, even when very pronounced, do not by themselves rule out using non-youngest-first collectors for our test programs.

The measurements of the performance of generational collectors are interesting in their own right, as they shed light on collector behavior under constraints on total heap size. It is not surprising that small nursery sizes lead to excessive promotion into the older generation, but it is surprising that in many cases the optimal configuration assigns more than half the heap to the nursery. It is possible that more sophisticated generational schemes could perform better—for instance, requiring an object to survive more than one collection in the younger generation before promotion, or, equivalently, dividing the younger generation into steps, possibly combined with adaptive policies, such as Ungar and Jackson's feedback mediation [Ungar and

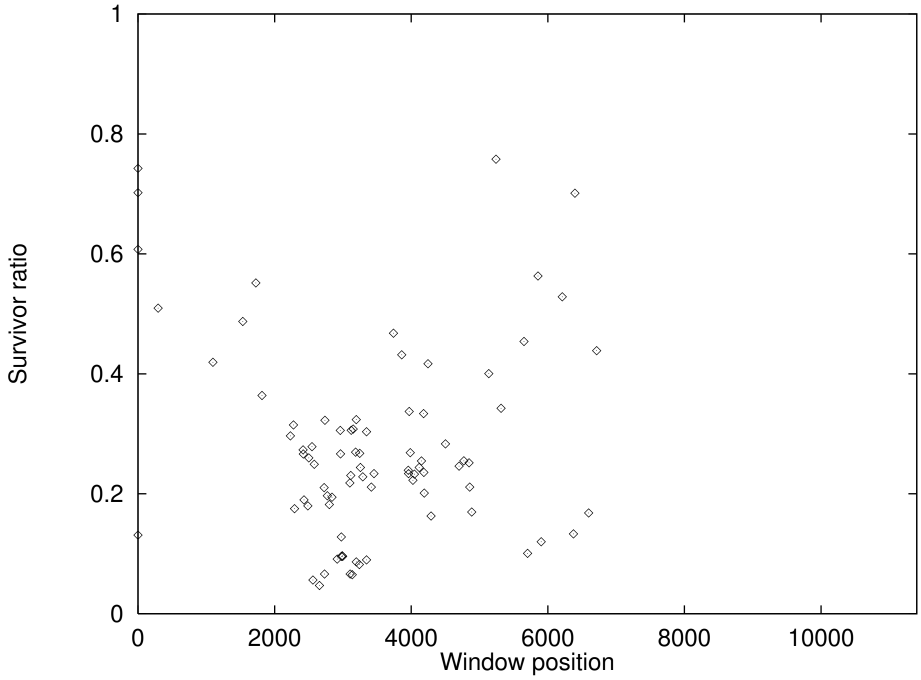

Figure 5.6. Window survivor ratios in the adaptive scheme.

Jackson, 1992]. However, each collection adds to the copying cost, and the final balance cannot be predicted. Unfortunately, the vast dimensions of the space of possible configurations makes it difficult to these collection schemes. (Thus, the selection of their many configurable parameters [Hudson *et al.*, 1991] has largely been a matter of judgement by experience.)

The proposed non-youngest-first schemes can also benefit from more sophisticated configurations. We observed that adaptive positioning of a window of fixed size can reduce copying cost; it is likely that it would be more effective in conjunction with adaptive selection of window size. Most important, however, could be the heuristic discovery of permanent (static) data, and their elimination from consideration by the collector, a task implicitly performed by a multi-generational collector.