CHAPTER 6

EVALUATION: COSTS OTHER THAN COPYING

In this chapter we examine the non-copying costs of collection, focusing on the cost of maintaining the information needed for accurate region-based collection: the write barrier and remembered set overheads. The evaluation uses a realistic, block-allocation collection simulator, based on our prototype implementation. We compare the generational youngest-first collector with the deferred older-first algorithm, which was shown in the preceding chapter to be the most promising with respect to copying cost among the alternatives to generational collection that we considered.

6.1 Method for evaluating the pointer maintenance cost

We use an instrumented version of the block simulator (Section 4.3.3) to obtain the statistics of pointer management operations. We discussed in Section 3.2 the design of pointer management for age-based collection, and noted alternative models of pointer store filtering and of remembered set organization. The actual costs of operations depend on these alternative design decisions. Indeed, the operation counts are not invariant themselves: if two filtering tests are executed in sequence, the outcome of the first influences the number of times the second is executed. The division of labor between the run time and the garbage collection time is also a matter of choice. Here we use for the generational collector the traditional model of operation with a unified remembered set and generational filtering, and we show that it is very efficient. For the DOF collector we use block-local and directional filtering. All of the operation counts reported are accurate for these design choices, while some of them are also universally applicable regardless of the design choice.

For the write barrier, we report the operation counts for pointer stores, and the number of non-null pointer stores. Non-null stores are further divided into stores filtered by the specified filtering mechanism, and those not filtered. The non-filtered stores are divided into unique and duplicate, with respect to the prior content of the remembered set into which the stores are inserted. Note that the write-barrier statistics include both the run time and the garbage-collection time pointer stores.

For remembered set processing at garbage-collection time, we report the number of pointers from the uncollected region into the collected region, as well as the number of remembered set entries actually processed, which can be larger.

6.1.1 Results

In Table 6.1 we report the statistics that are independent of the collection algorithm or its configuration. The columns "Words alloc." and "Ptr. st." repeat the information of Table 4.1. The column "Alloc./st." gives the ratio of the number of words allocated to the number of pointer stores for the program. This value is an indication, albeit crude, of the frequency of pointer manipulations, or the "object-orientedness" of the program, as well as of the expected influence of pointer maintenance operations on overall memory management performance. The last two columns give the number of non-null pointer stores as an absolute number, and as a percentage of all pointer stores. In many implementations of the write barrier, the check for a null pointer is a separate first step. As we can see in our numbers, there are few null pointers, and this test rarely eliminates the pointer from consideration. Hence a write barrier that integrates the null-pointer test into a general address test, as proposed in Section 3.2.4, is to be preferred.

In Tables 6.2–6.15, pp. 295–382 (at the end of this chapter) we report the pointer management statistics for the different configurations of the FC-DOF and the 2GYF collectors. The first group of tables (Tables 6.2–6.8) is for the FC-DOF collector. The first three columns in Tables 6.2–6.8 indicate the configuration. The first column, having values between 9 and 20,

Table 6.1. Properties of benchmarks used: run-time allocation and pointer stores

Benchmark	Words alloc.	Ptr. st.	Alloc./st.	Non-null ptr. st.	%
Smalltalk					
Interactive-AllCallsOn	18 359	855	21.47	134	15.7
Interactive-TextEditing	22 747	3 509	6.48	3 241	92.4
StandardNonInteractive	204 954	7 251	28.26	5 882	81.1
HeapSim	3 791 306	30 298	125.13	30 233	99.8
Lambda-Fact5	277 940	91 877	3.03	87 403	95.1
Lambda-Fact6	1 216 247	404 670	3.01	387 037	95.6
Swim	1 533 642	134 355	11.41	120 427	89.6
Tomcatv	2 117 849	286 032	7.40	285 056	99.6
Tree-Replace-Binary	209 600	39 642	5.29	32 986	83.2
Tree-Replace-Random	925 236	168 513	5.49	140 029	83.1
Richards	4 400 543	763 626	5.76	611 767	80.1
Java					
JavaBYTEmark	1 161 949	49 061	23.68	48 835	99.5
Bloat-Bloat	37 364 458	4 927 497	7.58	4 376 798	88.8
Toba	38 897 724	3 027 982	12.85	2 944 672	97.2
Geometric mean			10.27		80.2

gives the block size: for value k, the block size is 2^k bytes. The second column gives the allotted heap size, in terms of blocks. The third column gives the size of the collected region (the window size), also in terms of blocks. The fourth column gives the absolute copying cost as the number of words copied. The fifth column gives the number of collector invocations. The sixth column gives the number of remembered set insertion operations at the write barrier, i.e., the number of pointer stores that were *not* filtered out. The seventh column gives the number of insertion operations that are not duplicates, and hence actually add entries to remembered sets. The eighth column gives the *sum* of the sizes of all remembered sets processed at garbage collection time (i.e., the sets belonging to blocks within the collected region). The ninth column gives the sum over all collections of the sizes of filtered unions of sets within the collected region.

The design used to produce the statistics is the following. Each block has its own remembered set (Section 3.2.2). The write barrier code is executed on every pointer store, both at run time and at garbage collection time. The write barrier consists of three filtering checks: first, the check for null pointers; second, the check for block-local pointers; and third, the directional check (as in Figures 3.15 and 3.16). If none of the checks eliminates the pointer store, then the write-barrier code inserts it into the remembered set of the pointer target block, and adds it to the tally for table column six, "W.b. insert." The remembered set may already contain an entry recording the same pointer source address; otherwise, a new entry is added to the remembered set, and to the tally for table column seven, "W.b. add."

At collection time, following the processing of global roots, the collector processed the remembered sets of the blocks in the collected region one by one. Table column eight, "R.s. proc," reports the sum of the sizes of all remembered sets processed over all collections. On the other hand, the sum of the sizes of remembered sets for the collected region can be greater than the number of pointers actually crossing into the collected region from the uncollected region. The latter number is computed as the union of the remembered sets involved, less any pointers that have both source and target in the collected region. The sum of these numbers over all collections is reported in column nine, "R.s. proc."

The second group of tables (Tables 6.9–6.15) is for the 2GYF collector. The first four columns in Tables 6.9–6.15 indicate the configuration. The first column gives the block size, the second column gives the allotted heap size in blocks, the third column gives the size of the younger generation (the nursery) in blocks, and the fourth column gives the size of the older generation in blocks. The fifth column gives the absolute copying cost as the number of words copied. The sixth column gives the number of collector invocations. The seventh column gives the number of remembered set insertion operations at the write barrier, and the eighth gives the number of these that are not duplicates. The ninth column gives the sum of the sizes of processed remembered sets.

The design used to produce the statistics is the following. There is a single remembered set, and it records the pointers from the older generation into the younger generation. The write barrier code is executed on every pointer store at run time. No write barrier is needed at garbage collection time, because, with only two generations, all objects are in the oldest generation following a collection. The write barrier consists of two filtering checks: first, the check for null pointers; and second, the generational check. If neither filter eliminates the pointer store, it is inserted into the remembered set, and added to the tally for column seven, “W.b. insert.” If the pointer source is not already present in the remembered set, a new entry is added to the set, and to column eight, “W.b. add.”

At collection time, the collector processes the single remembered set. Table column nine, “R.s. proc”, reports the sum, over all collection, of the sizes of the remembered set. There is no need for a “R.s. union” column for 2GYF, because its meaning is subsumed by “R.s. proc.”

Each row of a table corresponds to one configuration of a collector, and thus to one simulation run. The configurations are ordered by block size, and for each block size, they are ordered by heap size. For a given heap size, the configurations are ordered by the size of the collected region (in the case of FC-DOF), and by the size of the younger generation (in the case of 2GYF).

Certain rows are printed in boldface for ease of reference in both the FC-DOF and the 2GYF tables; we shall use these configurations as examples to discuss the comparative costs of the two schemes.

6.1.2 Block size

We first examine the effect of block size on copying and pointer management costs. For the FC-DOF collector, we find that increasing the block size reduces the operation counts for the write barrier and remembered set processing. Consider benchmark Lambda-Fact5 in Table 6.4, p. 310. We shall compare block sizes 2^9 bytes and 2^{10} bytes, and, in particular, configurations, printed in italic boldface font, for block size 2^9 bytes and heap size 240 blocks, and for block size 2^{10} bytes and heap size 120 blocks. These configurations are pairwise identical with respect to heap size and collected region size.

The differences in copying cost are small in some pairs, but significant in others, such as 30236 words copied for configuration $[9, 240, 108]$ vs. 44507 words copied for configuration

[10, 120, 54]. There are two reasons for the differences. First, blocks of different size cause different fragmentation at block boundaries. However, since blocks are already substantially larger than typical object sizes present in the trace, the effect of fragmentation should be small, and the effect of differences in fragmentation even smaller; moreover, larger blocks should suffer less from fragmentation than smaller blocks, whereas we see that the copying cost is lower for smaller blocks. The second reason, which applies here, is that positioning of the collection window in the FC-DOF scheme is constrained by block boundaries and is less flexible with larger blocks, and, as Section 5.4.5.2 has shown, window positioning is important for good performance of FC-DOF.

If we focus on pairs with similar copying cost, we note that larger blocks enjoy lower counts in columns "W.b. insert," "W.b. add," and "R.s. proc," which is to be expected. With larger blocks, a greater portion of pointer stores are block-local, hence filtered out. Note, however, that doubling the block size reduces the number of write-barrier and remembered set operations by only about 10% in these examples. The column "R.s. union" is not reduced by increasing block size since the number of external pointers into the collected area does not depend on its internal division.

For the 2GYF collector, Table 6.4, p. 310, again comparing the configurations for block size 2^9 bytes and heap size 240 blocks, and for block size 2^{10} bytes and heap size 120 blocks, we find that differences in copying cost are generally not large, and in some cases favor larger, whereas in other cases smaller blocks. The pointer management operation counts are not systematically affected by block size, which is to be expected, since the generational write barrier check ignores block boundaries, except to the extent that it is indirectly influenced by fragmentation.

6.1.3 Comparison between collectors

We compare the FC-DOF and the 2GYF collector with respect to copying and pointer management costs. Consider the benchmark StandardNonInteractive with a block size of 2^9 bytes and heap size of 25 blocks, i.e., 3200 words (Table 6.2, p. 295). Among the configurations of the FC-DOF collector for this heap size, the lowest copying cost of 3902 words is achieved with a collected region size set to 4 blocks. We abbreviate this configuration as $[9, 25, 4]$. Among the configurations of the 2GYF collector for this heap size (Table 6.9, p. 335), the lowest copying cost of 15699 words is achieved with a younger generation set to 17 blocks, and older generation set to 8 blocks, abbreviated $[9, 25, 17, 8]$. Thus the copying cost of the FC-DOF collector is significantly (4.02 times) lower than that of the 2GYF collector, which confirms, in the context of a block-based implementation, the observations we made in Sections 5.1 and 5.2.

The number of pointer management operations, however, is significantly higher for FC-DOF than for 2GYF. The number of non-filtered pointer stores is $\frac{688}{13} = 52.9$ times greater, and the number of processed remembered set entries is $\frac{615}{12} = 51.5$ times greater. It should come as no surprise that pointer management costs are higher in the FC-DOF scheme. The "R.s. union" column for FC-DOF indicates that the total number of pointers into the collected region (summed over all collections) is 392, whereas the corresponding number for 2GYF is just 12. Therefore, regardless of the efficiency of pointer filtering, when collected regions are chosen in the FC-DOF manner, the number of incoming pointers is high. The causes are in the small size of the region (and large size of the remainder), and the fact that the region visits all parts of the heap regularly, and some parts of the heap have many incoming pointers; we investigate pointer structure further in the following Section 6.2.

We can remark then on the efficiency of the FC-DOF filtering scheme of Section 3.2.2: out of 7251 pointer stores in StandardNonInteractive (Table 6.1), only 688 or 9.5% are inserted into remembered sets. This reduction in the number of pointer that are recorded is certainly not as good as 13 or 0.2% for the generational filtering of 2GYF. On the other hand, we noted that keeping a large number of pointers is a necessary price to pay in order to choose collected regions in a particular way. The true measure of the efficiency of the filtering scheme is therefore given by the ratio of the "R.s. proc" column and the "R.s. union" column: $\frac{615}{392} = 1.56$, thus 56% more pointers than is absolutely necessary are processed at garbage collection time.

We see that the copying cost is lower for the FC-DOF collector, and the pointer management cost is lower for the 2GYF collector. We must look at absolute numbers to determine which collector achieves a lower total cost. The difference in copying cost, in favor of FC-DOF, is 11797 words copied. The difference in pointer management cost, in favor of 2GYF, is 675 remembered set insertions (636 non-duplicate), and 603 processed entries at garbage collection time. Which of these is more expensive depends on the details of implementation. For the sake of a simple analysis, we subsume the different components of pointer management under the single appellation of *pointer processing cost*, which we associate with the highest operation count, that of the non-filtered pointer stores, thus erring conservatively in favor of the scheme with lower pointer management cost. If the cost of processing a pointer is greater than $\frac{11797}{675} = 17.5$ times the cost of copying a word at garbage collection time, then the 2GYF collector will have the advantage for benchmark StandardNonInteractive. (These break-even ratios are summarized for all examined configurations in Figures 6.1– 6.9, pp. 389–393 which we discuss below.)

If we are interested in *incremental* operation of the collector, then we should compare configurations that result in similar (and *large*) numbers of collector invocations. In this case, if the 2GYF collector is configured with the younger generation at 4 blocks, and the older generations at 21 blocks, then it will perform 418 collections, the same as the FC-DOF collector with the collection window of 4 blocks. The copying cost advantage of FC-DOF is now greater, 23510 words, but the pointer management advantage of 2GYF is only slightly smaller, 673 remembered set insertions (633 non-duplicate), and 600 processed entries at garbage collection time; the pointer/copy cost ratio would have to be over 34.9 for 2GYF to have the advantage.

Similar results obtain for the larger block size of 2^{10}: compare the configuration $[10, 13, 3]$ of FC-DOF with configuration $[10, 13, 7, 6]$ of 2GYF.

We turn our attention to benchmark HeapSim (Tables 6.3, p. 301 and 6.10, p. 343), at block size 2^{13} bytes and heap size 123 blocks, or 251904 words. The best configuration of FC-DOF is $[13, 123, 4]$, with copying cost 262984, with 4645 non-filtered pointer stores (of which 3731 non-duplicate), and 3297 processed entries at garbage collection time. The remembered set union total is 1628, thus the filtering efficiency is assessed at an overhead of 102% more processed pointers than is absolutely necessary. The best 2GYF configuration is $[13, 123, 41, 82]$, with copying cost 341702, with 22 non-filtered pointer stores (of which 12 non-duplicate), and 12 processed entries at garbage collection time. The copying cost advantage of FC-DOF is 89798 words; the pointer/copy cost ratio would have to be over 19.4 for 2GYF to have the overall advantage. Similar results obtain for the larger block sizes of 2^{14} (compare the configuration $[14, 62, 3]$ of FC-DOF with configuration $[14, 62, 20, 42]$ of 2GYF) and 2^{16} (compare $[16, 16, 4]$ with $[16, 16, 6, 10]$).

In benchmark Richards (Tables 6.5, p. 322 and 6.12, p. 370), we consider the block size 2^{10} bytes, and heap size 20 blocks, or 5120 words. The best configuration of FC-DOF is $[10, 20, 8]$, with copying cost 56758, with 30270 non-filtered pointer stores (of which 29775 non-duplicate), and 29686 processed entries at garbage collection time. The remembered set union total is 11354, thus the filtering achieves an overhead of 161%. The best 2GYF configuration is $[10, 20, 11, 9]$, with copying cost 631920, with 3754 non-filtered pointer stores (of which 1541 non-duplicate), and 1541 processed entries at garbage collection time. The copying cost advantage of FC-DOF is 575162, and its pointer management disadvantage is 28614 insertions (26021 for non-duplicates) and 28145 processed entries. The pointer/copy cost ratio would have to be over 20.1 for 2GYF to have the overall advantage. With twice as large blocks (configurations $[11, 10, 4]$ and $[11, 10, 6, 4]$) the results are similar, and the pointer/copy cost ratio break-even point is 27.0.

The situation is somewhat different for Lambda-Fact5 (Tables 6.4, p. 310 and 6.11, p. 354). Recall that no collector worked much better than non-generational for this benchmark in object-level simulations of Section 5.1, and that differences among collectors were small. We now look at block size 2^9 bytes and heap size 90 blocks, or 11520 words. The best configuration of FC-DOF is $[9, 90, 67]$, with copying cost 134723, with 21778 non-filtered pointer stores (of which 21769 non-duplicate), and 21087 processed entries at garbage collection time. The remembered set union total is 4365, which means that filtering is unusually inefficient here, at an overhead of 383%. The best 2GYF configuration is $[9,90,67,23]$, with copying cost 158419, with 801 non-filtered pointer stores (of which 558 non-duplicate), and 557 processed entries at garbage collection time. The copying cost advantage of FC-DOF is 23696, but its disadvantage is 20977 insertions (21211 for non-duplicates) and 20530 processed entries. It is sufficient for the pointer/copy cost ratio to be over 1.12 for 2GYF to have the overall advantage.

Similarly, in Bloat-Bloat (Tables 6.7, p. 327 and 6.14, p. 376), we find that the copying cost difference between the collectors is very small, but the pointer management costs are noticeably higher for FC-DOF. For instance, with a block size of 2^{16}, and a heap size of 28 blocks, or 458752 words, the best configuration of FC-DOF is $[16,28,14]$, with copying cost 2308104 and 387199 non-filtered pointer stores, whereas the best configuration of 2GYF is $[16,28,13,15]$, with copying cost 2498540 and 33304 non-filtered pointer stores. In this case it is sufficient for the pointer/copy cost ratio to be over 0.53 for 2GYF to have lower total cost.

In Toba (Tables 6.8, p. 331 and 6.15, p. 382), with 2^{16}-byte blocks and a heap size of 40 blocks, the difference in copying cost in favor of FC-DOF configuration $[16,40,25]$ is 756188 words, and the difference in pointer cost in favor of 2GYF configuration $[16,40,23,17]$ is 288379 operations, giving a pointer/copy cost ratio threshold of 2.62.

Benchmark JavaBYTEmark stands out (Tables 6.6, p. 326 and 6.13, p. 375). We look at block size 2^{18} bytes and a heap size of 6 blocks, or 393216 words. One of the (tied) best configurations of FC-DOF is $[18,6,3]$, with copying cost 17634 words, with 23 non-filtered pointer stores (out of 49061 pointer stores), 19 non-duplicate, and 19 processed entries at garbage collection time. The best 2GYF configuration is $[18,6,5,1]$, with copying cost 37842 words, with 135 non-filtered pointer stores (of which 102 non-duplicate), and 16 processed entries at garbage collection time. Here both collectors have negligible pointer management costs, but FC-DOF is better, by more than a factor of 2, with respect to copying cost.

A summary of the break-even ratios of pointer-operation cost to word-copying cost is presented in Figures 6.1–6.11, pp. 389–394 (at the end of this chapter), for all simulated heap sizes, including the exemplars discussed above. In these graphs, the horizontal axis indicates the heap size (in words), and the vertical axis indicates the break-even ratio, shown on a logarithmic scale. Each graph contains several data lines, one for each block size. For example, Figure 6.1 has three lines, for block sizes 2^9, 2^{10}, and 2^{11} words. If in an implementation the ratio of the cost of a pointer operation to the per-word cost of copying is above the point in the graph, then 2GYF has a lower cost overall than FC-DOF.

6.1.4 Additional observations

In addition to the comparison between FC-DOF and 2GYF, we can observe a trade-off between copying cost and pointer-maintenance cost in the configurations of FC-DOF itself. Consider for instance the configurations $[16, 28, *]$ of benchmark Bloat-Bloat. As the size of collected region is increased from 11 to 26 blocks, the copying cost first briefly falls from $2.6 \cdot 10^6$ at 11 blocks to the minimum of $2.3 \cdot 10^6$ at 13 blocks (the configuration we compared against 2GYF above), then rises to $5.4 \cdot 10^6$ at 24 blocks, and briefly falls to $5.2 \cdot 10^6$ at 26 blocks. Although the copying cost is not strictly monotone increasing, it shows a clear increasing trend. At the same time, the "R.s. union" column shows a decreasing trend, from 106683 entries in the remembered set union with collected region of 11 blocks, down to 29426 with 26 blocks. The actual processing costs in the "R.s. proc" column are also nearly monotone decreasing.

An examination of all the tabulated results for the FC-DOF collector reveals that there are remarkably few duplicate entries: the values in the "W.b. add" column are close to those in the "W.b. insert" column. Therefore, it is not essential to eliminate duplicates early in pointer management. A simple sequential store buffer representation of the remembered set, which does not remove duplicates, may be adequate.

With the single generational remembered set of the 2GYF collector, the number of duplicates is significant: the values in the "W.b. add" column are typically well below those in the "W.b. insert" column. When the proportion of duplicate entries is large, it is worth eliminating them early and cheaply, e.g., by using a card-marking write barrier, with cards flushed into sets without duplicates.

Since the proportion of pointer stores retained is higher in the FC-DOF collector than in the 2GYF collector, the amount of auxiliary space needed for the remembered sets is higher, which raises the concern about the *total* memory usage of the collector. It turns out, however, that the space needed for remembered sets is typically below 2% of heap size. In particular, assuming that a remembered set entry uses one word, the total size of remembered sets for all heap blocks is 0.62% of total heap size for StandardNonInteractive (in the configuration discussed above), 0.49% for Richards, 1.38% for Bloat-Bloat, and 1.02% for Toba. Therefore, the space overhead of pointer maintenance in FC-DOF is low.

The measured values for individual copying and pointer-tracking operations (Section 3.2.5) permit a more detailed estimation of the total cost of collection. However, our trace-based simulation does not simulate stack contents, and therefore, while its copying counts are accurate, it cannot model the pointer-scanning counts: the number of scanned pointers in to-space that do not result in object copying depends on the contents of the stack. We can establish lower and upper limits, however. The lower limit corresponds to approximately one duplicate pointer per object, that one being the pointer to the class object. The upper limit corresponds to all words being duplicate pointers. We assume individual operation costs as in Section 3.2.5, and, for a given heap size, we allow each collector to assume its best configuration with respect to the total cost estimate. For the heap sizes as in the examples discussed above, the lower and upper bounds for the ratios of total processor cycles spent on garbage collection by 2GYF and by DOF are as follows: for benchmark StandardNonInteractive, between 2.66 and 3.28; for benchmark Richards, between 3.84 and 5.70; for benchmark Lambda-Fact5, between 0.98 and 1.04; for benchmark JavaBYTEmark, between 2.16 and 2.48; for benchmark Toba, between 0.98 and 1.08; for benchmark Bloat-Bloat, between 0.94 and 0.98. Therefore, pointer management costs do not significantly alter the qualitative performance relationship between the two collectors that we established when considering the amounts copied in Chapter 5: DOF remains significantly better than 2GYF on one set of benchmarks, while not significantly worse than 2GYF on the remaining benchmarks.

6.2 Methods for examining the directions of pointers in the heap

An evaluation of the cost of remembered-set maintenance provides us with raw numbers, such as the number of executed write-barrier checks, the number of processed remembered-set entries, and an estimate of the time penalty of these operations. Upon seeing these numbers, we can assess the effect of pointer tracking on different collectors, but we are at a loss about the origins of the effect. Just as it is necessary to recognize where (in which region of the heap) the high copying cost was incurred, it is necessary to recognize *where* in the heap the pointer-tracking costs arise, if we are to understand and remedy the situation. Our purpose is to establish quantitative measurements where the literature so far provides speculative statements.

6.2.1 Pointer stores in the ideal heap

We begin with an examination of pointer store directions in an absolute sense, without reference to a specific collection algorithm, assuming instead that the heap is perfectly collected, as we did for the heap profiles in Section 5.4.3. For each non-null pointer store, we note the heap position of both the pointer source and the pointer target. Recall that the heap position of an object is expressed as the distance from the young end of the heap, i.e., the amount of data that was allocated since that object was allocated and is still live. We define pointer *distance* as the difference between the heap position of the pointer target and the heap position of the pointer source. Hence, the distance is positive for younger-to-older pointers, and negative for older-to-younger pointers. Aggregating over all stores, the distribution of the positions is shown in Figures 6.12–6.25, pp. 395–408 (at the end of this chapter). Each figure contains three plots: (a) a histogram of the distribution of heap positions of pointer sources; (b) a histogram of the distribution of heap positions of pointer targets; and (c) a cumulative distribution graph for pointer distances and directions.

The pointer source histograms for all programs have a peak in the low positions, the youngest objects. A majority of pointer stores are into the most recently allocated objects. We emphasize that the histogram excludes null pointer stores, therefore the default initializations of pointer fields to the null pointer do not skew the distribution. Only the actual initializations by assignment are reflected in the histogram. The pointer *target* distributions also exhibit sharp peaks for the youngest objects, and indeed the fact that pointer distances are very short means that the source and target distributions must be similar. Thus a majority of pointers are from one object in the youngest part of the heap to another.

We now understand that the generational write barrier works well because for most pointer stores, both the source and the target are in the youngest generation, and thus the store need not be recorded. The putative explanation, that older-to-younger pointers are rare relative to younger-to-older pointers, is neither true (as the pointer distances graphs show, both directions are common), nor relevant.

Two programs with exceptional behavior are Swim and Tomcatv; they have secondary peaks of the pointer source distribution among the oldest objects. Both of these programs use a common implementation of floating-point matrices. A floating-point value in Smalltalk is not represented as an immediate register value, because the necessary tag bit(s) would both limit precision and require expensive conversions to the hardware floating-point format. Instead, a level of indirection is used, and each floating-point value is allocated in the heap. Computation creates new floating-point objects and updates the elements of a matrix to point to them. Since the matrix itself is allocated early in program execution, computation produces a stream of old-to-new pointer stores This peculiarity of the Smalltalk floating-point representation, and similar situations with frequently updated data structures with *boxed* representations in general and their interaction with the write barrier, have been noted before [Wilson, 1990; Chambers, 1992; Wilson, 1992] as a potential problem for the generational write barrier.

6.2.2 Pointer stores and remembered sets in age-based collectors

We now examine the pointer distributions that arise as a result of the execution of particular collection algorithms. Our frame of reference is an age-ordered heap, divided into blocks of equal size. The heap position of the youngest block (i.e., the current allocation block) is numbered 0, and older blocks are given successive numbers. We are interested in the distribution of pointer-tracking events by heap position. In other words, we are interested, having observed the positional distribution with an ideally collected heap, in any bias that the actual collection scheme introduces.

The resulting pointer source and target distributions are shown in plots (a)–(d) of Figures 6.26–6.31, pp. 409–414, for the benchmarks and their configurations discussed in the preceding Section 6.1.3. Plots (a) and (c) on the left reflect all non-null pointer stores, whereas plots (b) and (d) on the right reflect only those stores not eliminated by filtering, i.e., those inserted into remembered sets. Plots (a) and (b) show the histogram of positions of the pointer source, whereas plots (c) and (d) show the histogram of positions of the pointer target.

Additionally, the bottom plots (e) of Figures 6.26–6.31 show the positional distribution of the remembered set entries chosen for processing at garbage collection time.

The plots (a) and (c) show the same patterns as the corresponding plots (a) and (b) for the perfectly collected heap in the preceding section. Therefore, the FC-DOF collector, owing to its regular full sweeps, does not maintain large quantities of uncollected ("tenured") garbage, which, if present, would visibly distort the distribution relative to the perfectly collected heap.

Of greater interest is the distribution of heap position among the *non-filtered* pointer stores, in plots (b) and (d). First note that the absolute numbers are much smaller, since filtering eliminates most pointer stores.

A majority of the non-filtered stores still have their source in the youngest block (position 0). The non-filtered targets in the youngest block are few, because of the block-local filtering. However, there are many targets tailing off from block 1 towards older blocks—the directional filter could not eliminate the pointers from block 0 into these older blocks. In Lambda-Fact5 there is also a large number of pointers with source in block 0 and target among the very oldest objects.

When a filter cannot eliminate a pointer store, it inserts it into the remembered set of the target block. Thus, most pointers are inserted into the remembered sets of blocks at low positions (but not position 0). However, some time elapses between the pointer store and the time when the target block becomes part of the collected region of a collection, and during that time the block ages and moves toward higher heap positions. Therefore, in the distribution of heap positions of *processed* remembered set entries, the peak is in the middle or higher (older) regions. In Lambda-Fact5, for instance, just the oldest of the 90 blocks accounts for 30% of all processed remembered set entries (Figure 6.29(e), p. 412). Recall that the oldest blocks contain permanent data (Figure 5.271, p. 258), hence collections immediately following the resetting of the window not only incur a high copying cost, but also a high pointer processing cost.

Benchmark JavaBYTEmark is an exception: the filtering is so successful that very few pointers remain non-filtered, and, with their number so small, it is not warranted to discuss their positional distribution.

6.3 Method for evaluating the collection start-up cost

In addition to the copying cost of collection, which is directly related to the efficiency of the collector in finding low-survival regions, and the remembered-set maintenance cost, which is related to the ability of the collector to approximate the structure of the uncollected region safely and cheaply, there are other costs that are incurred on every collection.

First, there is a direct overhead of switching system activity from the mutator to the collector. This overhead depends on the implementation details. In some implementations, the collection is triggered by a page fault on accessing a protected page past the end of the available nursery space, or it may involve saving architectural state because the mutator is written in a different language from the garbage collector itself, and in these cases the start-up may be costly. On the other hand, this direct overhead is minimal in virtual machine implementations. The change of program focus from the mutator to the collector also has an adverse effect on reference locality—the performance of the instruction cache, and possibly the data cache, degrades.

Second, each collection requires establishing the global roots. These are found in the registers (machine or virtual) and in the stack (if the implementation uses an explicit stack). Whether this task is a large fraction of total garbage collection cost depends on the implementation: in the case of an explicit stack, has the implementation provided stack maps in a form that permits efficient stack scanning [Diwan *et al.*, 1992]?[1]

The cost of stack scanning is not directly dependent of the garbage collection algorithm that is subsequently employed. However, the cost of one stack scan depends on the stack state (how many activation records there are on the stack), hence it depends on the moment at which it is examined. Since different collector algorithms result in different moments of collector invocation, there is an indirect dependence of total scanning cost on collection algorithm, beyond the mere number of invocations, but with many invocations spread over the execution, with somewhat asynchronous points of collection, this additional dependence is probably very weak.

In summary, there is a certain fixed cost to be paid for each collector invocation. If we are primarily concerned with the total running time of the program, then we must be careful in the choice of the garbage collection algorithm lest it cause too many collector invocations. For many programs, we found that the lowest copying costs are obtained with smaller collected regions or smaller nurseries, and consequently with larger numbers of collector invocations. Clearly, when some rather large number is reached [Stefanović and Moss, 1994], the invocation costs will dominate.

---

[1] In the case of a stack hidden within the heap, stack scanning is implicitly accomplished during the Cheney scan phase of garbage collection, so it is difficult to isolate the stack scanning cost.

If, on the other hand, the primary consideration is that of minimizing pause times, then the invocation cost represents a fixed component that cannot be avoided. Since this cost is independent of the collection algorithm, except as mentioned above, it is still appropriate to choose an algorithm that minimizes the other costs.

6.3.1 Results

The invocation cost is shown in Figures 6.32–6.135, pp. 415–459. The vertical axis shows the number of collector invocations *relative* to the non-generational collector. (This permits us to use the same scale for all plots for a given benchmark.) As before, the horizontal axis 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.

A very regular pattern emerges, for all benchmarks, and all heap sizes: when the size of the collected region (or nursery size) is close to the full heap size, the behavior is non-generational and the relative invocation count is close to 1; but as the size of the collected region decreases, the invocation count increases. This result is intuitively clear: the amount of garbage found by a collection cannot be larger than the size of the collected region, thus the number of collector invocations must rise as the reciprocal of that size, if we ignore for a moment the influence of that size on the proportion of survivors.

6.4 Summary

We have shown that pointer-maintenance costs in a generational collector are very low, confirming past research. The analysis of pointer stores reveals that the explanation lies in the location of stored pointers (in recently allocated data) and not in the temporal direction of pointers (younger-to-older or older-to-younger).

The FC-DOF collector, because it collects smaller regions in all parts of the heap, incurs a considerably higher pointer maintenance cost, both in the higher number of pointer stores that must be recorded at run time (even after efficient filtering), and in the high number of incoming pointers processed at garbage collection time. Our trade-off analysis suggests that the copying cost advantage of DOF outweighs the pointer cost disadvantage for several tested programs, but the final verdict must await a full implementation.

Let us note, however, that the focus of our study on the application of collectors to the entire heap precluded an investigation of the space of older (mature) objects only, in which the distributions of pointer stores as well as the distribution of lifetimes is less favorable to the generational approach (to the extent that this can be plausibly inferred from observations of relatively longer-lived objects in our programs). Moreover, it can be argued, pending further measurements, that, with remembered set costs having already been absorbed in the mature object space [Hudson and Moss, 1992], deferred older-first becomes a collection *policy* worth exploring. It is a different policy from the *train* algorithm [Hudson and Moss, 1992; Seligmann and Grarup, 1995]; in particular, it does not guarantee reclamation of arbitrary cycles. On the other hand, the train algorithm leaves certain promotion decisions for objects moved out of a train underspecified; whereas locality (for the purpose of exploiting *key object opportunism* [Hayes, 1991]) is suggested as a criterion for making these decisions (thus: moving an object close to the object that keeps it live by pointing to it), age is a permissible criterion as well.