Age-Based Garbage Collection*

Darko Stefanović
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
+1 609 258 5056
darko@princeton.edu

Kathryn S. McKinley
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
+1 413 545 2410
mckinley@cs.umass.edu

J. Eliot B. Moss
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
+1 413 545 4206
moss@cs.umass.edu

ABSTRACT

Modern generational garbage collectors look for garbage among the young objects, because they have high mortality; however, these objects include the very youngest objects, which clearly are still live. We introduce new garbage collection algorithms, called *age-based*, some of which postpone consideration of the youngest objects. Collecting less than the whole heap requires *write barrier* mechanisms to track pointers into the collected region. We describe here a new, efficient write barrier implementation that works for age-based and traditional generational collectors. To compare several collectors, their configurations, and program behavior, we use an accurate simulator that models all heap objects and the pointers among them, but does not model cache or other memory effects. For object-oriented languages, our results demonstrate that an *older-first* collector, which collects older objects before the youngest ones, copies on average much less data than generational collectors. Our results also show that an older-first collector does track more pointers, but the combined cost of copying and pointer tracking still favors an older-first over a generational collector in many cases. More importantly, we reopen for consideration the question where in the heap and with which policies copying collectors will achieve their best performance.

Keywords

Garbage collection, object behavior, write barrier, generational and copying collection.

1 INTRODUCTION

Dynamic memory management (management of heap-allocated objects) using garbage collection has become part of mainstream computing with the advent of Java, a language that uses and requires garbage collection. This wider use of garbage collection makes it more important to ensure that it is fast. Garbage collection has been investigated for decades in varying contexts of functional and object-oriented language implementation (e.g., Lisp, ML, Smalltalk). The consensus, for uniprocessor systems operating within main memory, is that a class of algorithms known as *generational copying collection* performs quite well in most situations. While the breadth of variation within the class is considerable, the algorithms have this in common: objects are grouped according to their age (time elapsed since object allocation), and the younger groups or generations are examined more often than older ones. In particular, the most recently allocated objects are collected first. In this paper, we present a new copying collection algorithm, called *Older-First*, that maintains the grouping by age, but chooses to collect older objects (following a particular policy which we describe in Section 2). Our algorithm achieves lower total cost, sometimes dramatically, than traditional copying generational collection for a number of Java and Smalltalk programs. Why does it improve performance?

Let us consider the costs that copying garbage collection imposes on the run-time system. First, there is the cost of copying objects when they survive a collection. Second, to allow the collector to examine only a *portion* of the heap at a time, bookkeeping actions must log changes to pointers (references) that go from one portion to another; we call this *pointer-tracking*. Some of the pointer-tracking is interleaved with program execution whenever the program writes a pointer (i.e., the *write barrier*), while some is done at garbage collection time. Third, the program itself and the garbage collection algorithm(s) have different cache and memory behaviors, which interact in complex ways. These effects are beyond the scope of this paper and are left for future work. In this paper, the *total cost* of collection refers to combined cost of the pointer tracking and copying collection.

Generational copying collection performs better than non-generational, i.e., full heap, copying collection because it achieves markedly lower copying costs. On the other hand, it must incur the cost of pointer tracking, whereas non-generational collection has no need to track pointers because it always examines the entire heap. Thus, generational collection incurs a pointer-tracking cost that is offset by a much reduced copying cost. We have discovered that there is a trade-off between copying and pointer-tracking costs that can be exploited *beyond* generational copying collection. Our Older-First (OF) algorithm usually incurs much higher pointer-tracking costs than generational algorithms, but also enjoys much lower copying costs. We find that most pointer stores and the objects they point to are among the youngest objects, and by moving the collected region outside these youngest objects, OF must track more pointers. However, OF lowers copying costs because it gives objects more time to die, and does not collect the very youngest objects which clearly have not had time to die. In the balance, its total cost is usually lower than the total cost of generational copying collection, in some cases by a factor of 4. In

*This work is supported in part by NSF grant IRI-9632284, and by gifts from Compaq Corp., Sun Microsystems, and Hewlett-Packard. Kathryn S. McKinley is supported by an NSF CAREER Award CCR-9624209. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other sponsors.

To appear in OOPSLA'99, Denver, November 1999.

itself, OF is very promising, but, more importantly, its success reveals the potential for other flexible collection policies to exploit this trade-off and further improve garbage collection performance.

In Section 2 we describe our new collection algorithm within a broader classification of age-based algorithms. We present our benchmark suite in Section 3, and assess the copying performance of the family of age-based collectors in Section 4. We then consider implementation issues, including a new fast write barrier in Section 5. Section 6 evaluates the combined costs of copying and pointer-tracking. The results call for a reevaluation of the premises and explanations of observed performance of copying collectors, which is the subject of Section 7.

2 AGE-BASED GARBAGE COLLECTION

Upon a garbage collection, each scheme we consider partitions the heap into two regions: the collected region C, in which the collector examines the objects for liveness, and if live, they survive the collection; and the uncollected remainder region U, in which the collector assumes the objects to be live and does not examine them. The non-generational collector is a degenerate case in which the uncollected region is empty. The collector further partitions the set C into the set of survivor objects S and the set of garbage objects G, by computing root pointers into C and the closure of the points-to relation within C. To make the freed space conveniently available for future allocation, the collector manipulates the survivors S by copying (or compacting) them.

The amount of work involved is, to a first approximation, proportional to the amount of survivor data, and so that should be minimized. Ideally we choose C so that S is empty; in the absence of some oracle, we must look for schemes that organize heap objects so that the partition into C and U is trivial, and then find heuristics that make S small.

We restrict attention to a class of schemes that keep objects in a linear order *according to their age*. Imagine objects in the heap as if arranged from left to right, with the oldest on the left, and the youngest on the right, as in Figure 1. The region collected, C, is restricted to be a contiguous subsequence of this sequence of heap objects, thus the cost of the initial partition is practically nil. We call these schemes *age-based* collection.

Traditional generational collection schemes are, in the main, age-based: the region collected is some subsequence of youngest (most recently allocated) objects. Copying collectors may reorder objects somewhat during copying since they typically follow pointers breadth-first instead of in age order. In compacting collectors, reordering does not occur.

In this paper, we introduce and categorize alternative collection schemes according to their choice of objects for collection. In all these collectors, we fix the size of the collected region rather than allowing it to vary during program execution, to simplify our analysis. Previous research shows that dynamically sizing the collected region can improve performance [29, 36, 34, 1, 5], but this investigation is beyond the scope of our paper.

A *youngest-only* (YO) collector always chooses some youngest (rightmost) subsequence of the sequence of heap objects (Figure 2). In our implementation, the YO collector fills the entire heap and then repeatedly collects the youngest portion of the heap including objects surviving the last collection. The time in allocation between collections is the amount the YO collector frees. This collector might have good performance if object death is only, or mainly, among the new objects.

Generational collector schemes are variants of youngest-only collection, differing however in how they trigger collections [5]. In the basic design [17, p.147], new allocation is into one fixed-size part of the heap (the *nursery*), and the remainder is reserved for older objects

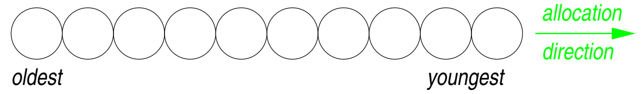
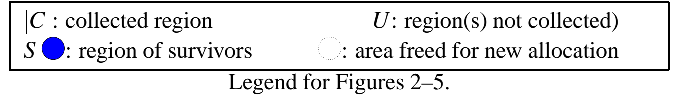


Figure 1: Viewing the heap as an age-ordered list.



Legend for Figures 2–5.

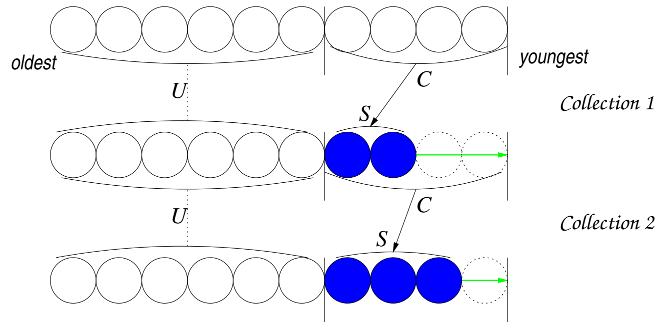


Figure 2: Youngest-only (YO) collection.

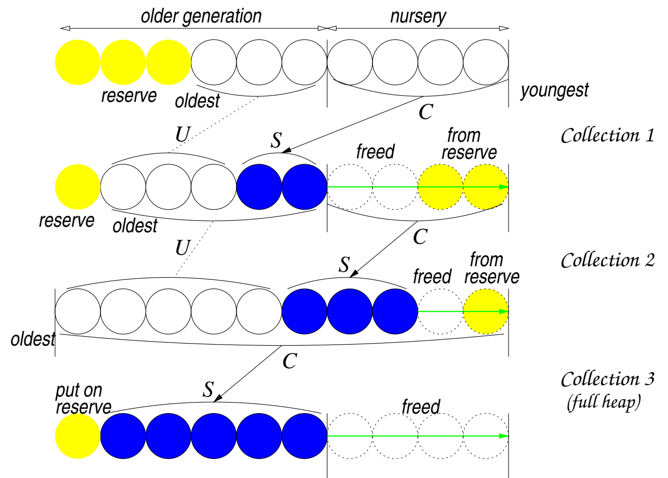


Figure 3: Generational youngest-only collection.

(the *older generation*). Whenever the nursery fills up, it is collected, and the survivors are promoted to the older generation (Figure 3). When the older generation fills up, then the following collection collects it together with the nursery. In a two-generation collector, that collection considers the entire heap.

Note that the generational collector deliberately does not allocate directly into the space reserved for the older generations, so that, unlike YO, the region chosen for collection contains exactly the objects allocated since the last collection (except for full heap collections). We study two and three generation schemes: 2G (2 Generations; youngest-only) and 3G. We assume the size of each generation is strictly greater than 0, and therefore 3G never degenerates into 2G.[1]

An *oldest-only* (OO) collector always chooses an oldest (leftmost) subsequence of the sequence of heap objects (Figure 4). In our implementation, the OO collector initially waits for the entire heap to fill and then repeatedly examines the oldest objects including those surviving the previous collection. As in the YO collector, only the resulting free amount is available for allocation. An object is more likely to be dead the longer we wait, hence the OO collector *might* have good performance. Of course, it will suffer if there are any objects that survive the entire length of the program because it will copy them repeatedly.

An *older-first* (OF) collector chooses a middle subsequence of heap objects, which is immediately to the right of the survivors of the previous collection (Figure 5). Thus the region of collection sweeps the heap rightwards, as a window of collection. The resulting free blocks of memory move to the nursery. Initially, objects fill the entire heap and the window is positioned at the oldest end of the heap. After collecting the youngest or right end of the heap, the window is reset to the left or old end.

The intuition for the potentially good performance of this collector can be gleaned from the diagram in Figure 6, which shows a series of eight collections, and indicates how the window of collection moves across the heap when the collector is performing well. If the window is in a position that results in small survivor sets (Collections 4–8), then the window moves by only that small amount from one collection to the next. The remaining window size is freed and becomes available for allocation. As the window continues to move slowly, it remains for a long time in the same region, corresponding to the same age of objects. A great deal of allocation takes place without many objects' being copied; almost a window size between successive collections. How long the window remains in a good position, and how long it takes to find this "sweet spot" again once it leaves, will determine the performance of the collector for a particular workload, heap size, and window size.

We refer to the OF, OO, and YO collectively as FC collectors (Fixed-size Collection window). The base point of our comparisons is the non-generational collector (NG), which considers the entire heap in each collection. Note that it is possible for an FC collector to find no garbage in the collected region. If that happens, we let the collector fail for the purposes of this study. (An implementation could increase the heap size temporarily, or retry collection on another region, perhaps the whole heap, or increase the window size adaptively.) Because generational schemes by design occasionally consider the whole heap, they enjoy an advantage over the new schemes as simulated here.

3 BENCHMARKS

Table 1 lists our benchmarks, which include Smalltalk and Java programs and their basic properties relevant to garbage collection performance: amount of data allocated in words (each word being 4 bytes), number of objects allocated, maximum live amount (which is also

[1]We also examined a scheme in which the older generation is allowed to grow into the nursery, and vice versa [1], but it performed similarly to 2G and 3G.

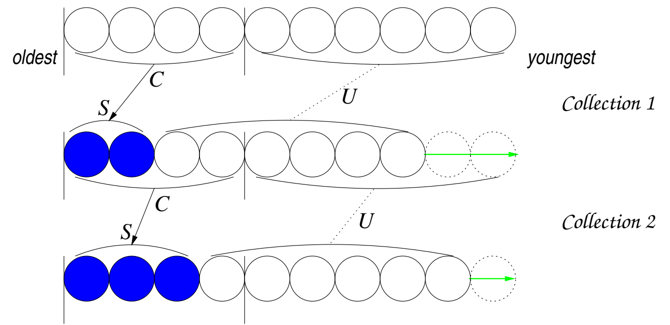


Figure 4: Oldest-only collection.

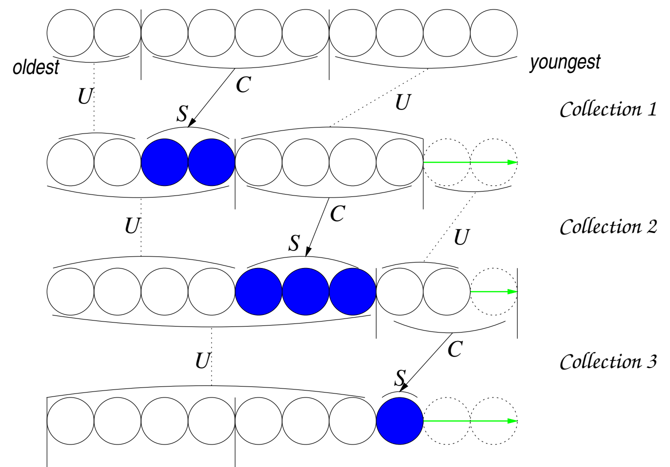


Figure 5: Older-first collection.

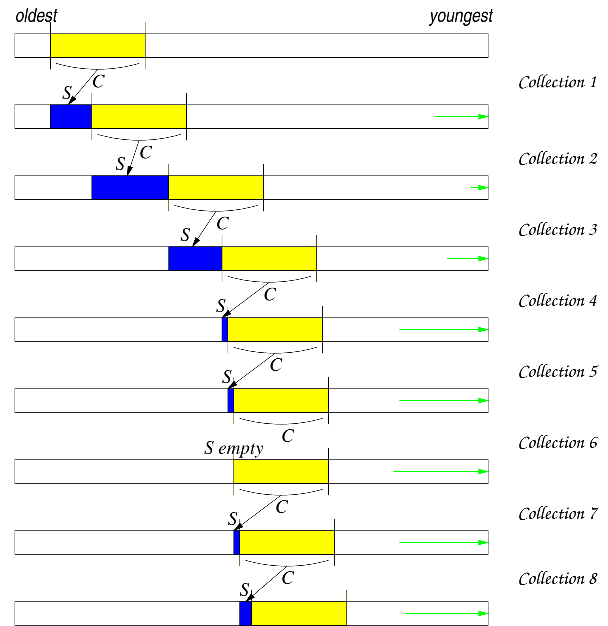


Figure 6: Older-first window motion example.

Benchmark	Words alloc.	Objects alloc.	Max. live	Pointer stores total	alloc./st.	non-null	%
Java							
JavaBYTEmark	1 161 949	109 896	59 728	49 061	23.68	48 835	99.5
Bloat-Bloat	37 364 458	3 429 007	202 435	4 927 497	7.58	4 376 798	88.8
Toba	38 897 724	4 168 057	290 276	3 027 982	12.85	2 944 672	97.2
Smalltalk							
StandardNonInteractive	204 954	46 157	1 160	7 251	28.26	5 882	81.1
HeapSim	3 791 306	1 084 650	93 026	30 298	125.13	30 233	99.8
Lambda-Fact5	277 940	53 580	6 295	91 877	3.03	87 403	95.1
Lambda-Fact6	1 216 247	241 864	13 675	404 670	3.01	387 037	95.6
Swim	1 533 642	444 908	17 542	134 355	11.41	120 427	89.6
Tomcatv	2 117 849	624 612	30 593	286 032	7.40	285 056	99.6
Tree-Replace-Binary	209 600	35 785	9 784	39 642	5.29	32 986	83.2
Tree-Replace-Random	925 236	189 549	13 114	168 513	5.49	140 029	83.1
Richards	4 400 543	652 954	1 498	763 626	5.76	611 767	80.1

Table 1: Benchmark Properties

the minimum required heap size to execute the program), total number of pointer stores, words of allocation per pointer store, number of non-null pointer stores, and the percentage of pointer stores that are non-null.

We now describe individual benchmarks, providing where possible details of their structure. Our set of Java programs is as follows:

- JavaBYTEmark. A port of the BYTEmark benchmarks to Java, from the BYTE Magazine Web-site.
- Bloat-Bloat. The program Bloat, version 0.6, [21] analyzing and optimizing the class files from its own distribution.
- Toba. The Java-bytecode-to-C translator Toba working on Pizza [22] class files [23].

Our set of Smalltalk programs is as follows:

- StandardNonInteractive. A subset of the standard sequence of tests as specified in the Smalltalk-80 image [12], comprising the tests of basic functionality.
- HeapSim. Program to simulate the behavior of a garbage-collected heap, not unlike the simplest of the tools used in this study. It is however instructed to simulate a heap in which object lifetimes follow a synthetic (exponential) distribution, and consequently the objects of the simulator itself exhibit highly synthetic behavior.
- Lambda-Fact5 and Lambda-Fact6. An untyped lambda-calculus interpreter, evaluating the expressions 5! and 6! in the standard Church numerals encoding [4, p.140]. Previously used in Ref. [15]. We used both input sizes to explore the effects of scale.
- Swim. The SPEC95 benchmark, translated into Smalltalk by the authors: shallow water model with a square grid.
- Tomcatv. The SPEC95 benchmark, translated into Smalltalk by the authors: a mesh-generation program.
- Tree-Replace-Binary. A synthetic program that builds a large binary tree, then repeatedly replaces randomly chosen subtrees at fixed height with newly built subtrees. (This benchmark was named Destroy in Ref. [15, 14].) Tree-Replace-Random is a variant which replaces subtrees at randomly chosen heights.
- Richards. The well-known operating-system event-driven simulation benchmark. Previously used in Ref. [15].

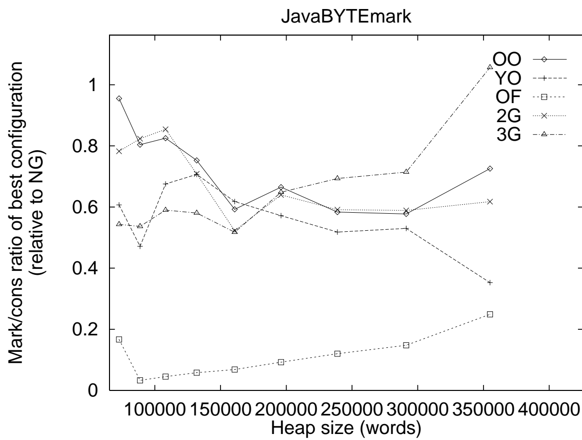
4 ESTIMATING COPYING COSTS

The idea of Older-First collection sufficiently diverges from established practice that it is instructive first to determine whether it is feasible in principle, before going into the details of an implementation. With the understanding that pointer-tracking costs are likely to be higher in older-first collection than in generational collection, we sought a quick estimate of copying cost to discover if the promise of Figure 6 is delivered on actual programs. We built an object-level simulator that executes the actions of each of the collectors exactly as depicted in Figures 2–5. The simulator is much simpler than the actual implementation: objects and collection windows of arbitrary sizes are allowed, the age order is perfectly preserved on collection, and pointers are not tracked. This simulator can produce the statistics of the amount of data copied over the run of a program, which, divided by the amount allocated, gives the "mark/cons" ratio, traditionally used as a first-order measure of garbage collector performance.

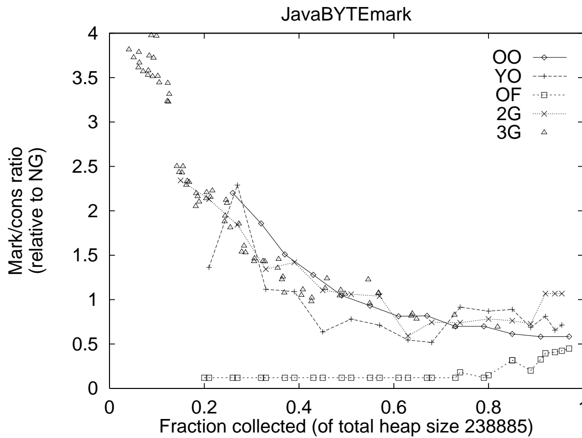
We now discuss the copying cost estimate results for two Java benchmarks, JavaBYTEmark and Bloat-Bloat, then summarize and make some general observations. Figures 7 and 8 each present two graphs: Graph (a) compares the best performance of each collection scheme (OO, YO, OF, 2G, 3G), plotting the mark/cons ratio (the copying cost that we would like to minimize), relative to NG, against heap size. Performance depends on the heap size available to the collector, which is laid along the horizontal axis. For each heap size, we simulated many configurations of each collection scheme. This graph only includes the *best* configuration of each collector. Graph (b) provides details of different configurations of each collector for one representative heap size, plotting the relative mark/cons ratio against the size of the collected region or nursery as fraction of the heap size.

JavaBYTEmark. For this program, the OF scheme copies significantly less data than all other schemes under all configurations. In fact, it copies over a factor of 10 fewer objects than the 3G collector. As we see in Figure 7(b), it attains this performance even while keeping the window of collection small: 20% of total heap size. In smaller heaps not shown here, the best window size for OF grows up to 40% of the heap. The generational collectors in Figure 7(b) only approach their best configurations when the nursery constitutes over 50% of the heap. Thus, the OF scheme copies much less using a smaller window size. Small window sizes are desirable because they contribute to keeping pause times for collection short, which is especially important in interactive programs.

The reason for this dramatic reduction in copying cost is exactly the scenario described in Figure 6. Many objects wait until middle age to die, and the OF collector is able to find them just as they die,

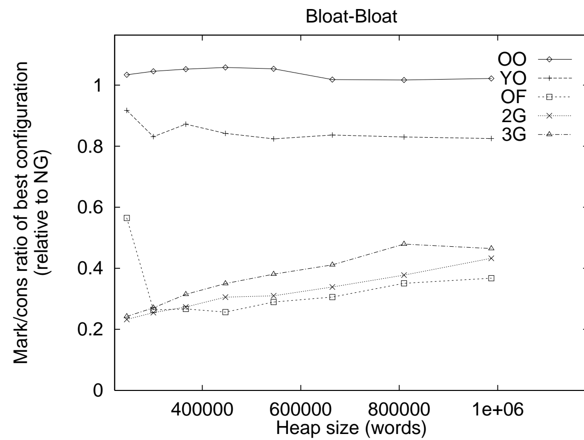


(a) Best configuration.

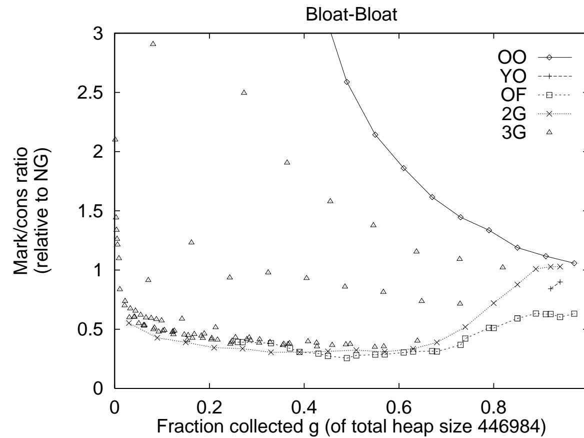


(b) Representative heap size.

Figure 7: Copying cost estimates, JavaBYTEmark.



(a) Best configuration.



(b) Representative heap size.

Figure 8: Copying cost estimates, Bloat-Bloat.

and to stay in a sweet spot for a long time. The OF collector does occasionally sweep through the heap and as a result revisits the oldest objects repeatedly. When we examine the lifetimes of the objects in this program [25] we find there are a number of long lived objects. Thus, the OF collector is repeatedly copying these objects (whereas generational collectors *by design* rarely copy these objects); nevertheless it copies a factor of 10 less data.

As it is the trend in most of the benchmarks, OF collection outperforms OO and YO. OF collection achieves similarly low copying costs that are also integer factors better than the generational collectors using a small window size on StandardNonInteractive, HeapSim, Richards, Lambda-Fact6, and Lambda-Fact5.

Bloat-Bloat. Figure 8(a) illustrates that the best configurations of OF, 2G, and 3G, all exhibit comparable and low copying cost. Furthermore, Figure 8(b) shows that these 3 collectors achieve close to or their minimums with a window size around 40% of the entire heap. The OF collector (as simulated for this study) fails with a window size below 20%, because long-lived data spans more than the collection window [25]. These results are representative of the remaining 8 programs. Comparing 2G with 3G collection in Figure 8(a) and (b) reveals no significant differences in the best configurations, but many configurations of the 3G collector perform worse, sometimes much worse, than the 2G collector.

4.1 Comparing 2 and 3 Generations

Several of the programs follow the trend we see in Figure 7(a) for JavaBYTEmark, in which 3G copies fewer objects than 2G. JavaBYTEmark is the program in our suite in which the 3G collector enjoys the largest advantage over the 2G collector. The more detailed presentation in Figure 7(b) reveals however that there are many configurations of the 3G collector that the 2G collector outperforms. This trend is true for the other programs as well, and demonstrates the difficulty of configuring generations well. For the remaining 9 programs, Toba, Bloat-Bloat, Lambda-Fact5, Lambda-Fact6, HeapSim, Swim, Tomcatv, Tree-Replace-Binary, and Tree-Replace-Random, the 2G collector copies the same amount or less than a 3G collector.

4.2 Comparing FC Collectors.

As it is demonstrated by JavaBYTEmark and Bloat-Bloat, the OF collector usually copies significantly less data than the OO and YO collectors. There are however a few programs for which the OO collector performs the best: Tree-Replace-Random and Tree-Replace-Binary. In these programs, there is very little long-lived data [25]. Random replacement of random subtrees or the interior node connected to the leaves of the binary tree does indeed imply that the longer the collector waits the more likely an object will be garbage. However, such synthetic programs are probably not representative of behaviors in users' programs, and most programs do have some very long-lived data [10].

4.3 Conclusion.

The copying cost estimates show great promise for the Older-First algorithm on a set of benchmarks. We therefore consider the issues involved in an actual implementation, and then proceed to the evaluation of a prototype. To simplify the investigation and the presentation, we will focus on the two-generation collector 2G (since we have found that it is usually comparable to the three-generation one) and the Older-First algorithm OF.

5 WRITE BARRIER

While OF collection reduces copying costs, it may increase write barrier costs. This potential increase prompted us to consider carefully which pointer stores need to be remembered in our prototype implementation. Generational collectors remember pointers from older to younger generations, but not within generations. Thus, stores into the youngest generation, including objects just allocated (in the nursery), never need to be remembered. The corresponding rule for OF collection is based on the following observation: when a store creates a reference $p \rightarrow q$, then we need to remember it only if q might be collected before p. Figure 9 shows diagrammatically which pointers an OF collector must remember, according to their direction between different regions of the heap. For example, the pointer store that creates the pointer $\boxed{p} \rightarrow \boxed{q}$ need not be remembered, because object $\boxed{p}$ will necessarily fall into the collected region earlier than $\boxed{q}$ will.

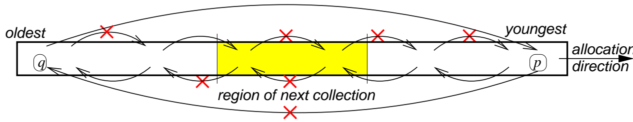


Figure 9: Directional filtering of pointer stores: crossed-out pointers need not be remembered.

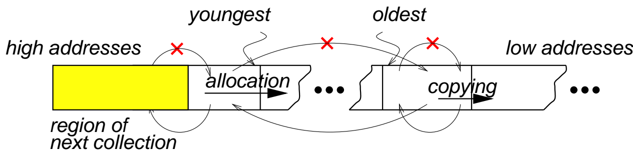


Figure 10: Directional filtering with an address-ordered heap.

At first glance, it would appear complex and expensive to do the filtering suggested by Figure 9, although not more than in flexible generational collectors [15]. However, if we reorder the regions of the heap physically as shown in Figure 10, then the test can be simpler still: we need only test if the store creates a pointer in a particular direction and possibly crossing a region boundary. A large zone of the virtual address space is set aside for allocation from higher addresses to lower. The collection region also moves from higher addresses to lower, but lags behind the allocation; the survivors are evacuated into the next similarly sized zone at lower addresses. If the collection region catches up with allocation (equivalent to reaching the right end in the logical layout of Figure 9), the former allocation zone is released, the former copying zone becomes the allocation zone, and a new copying zone is acquired. The organization of Figure 10 is especially attractive with a very large address space and with some cooperation from the operating system, to acquire and release address space as the heap progresses from higher to lower addresses.

Our implementation is based on allocating fixed-size *blocks* to the various heap regions, with the collector constrained to collect an integral number of blocks. This structure, with a block table, simply and quickly maps from addresses to remembered sets.

Since the block size is a power of two, blocks are aligned by block size, and the collection window moves from higher to lower addresses, we essentially test if `p < q`:

```
// write barrier for: *p = q
// mask == 2^k - 1
if (p < (q & ~mask))
  remember p in q's remset;
```

Adjusting one of the pointers using the mask eliminates stores within the same block. This test is important, since the vast majority of stores are to nearby objects, and thus tend not to cross block boundaries [25]. The directional test (<) also reduces the number of pointers remembered.

This write barrier, then, filters stores inline so that out-of-line code to remember a pointer is only executed for those cross-block pointers where the source block of the pointer may be collected after its target block. The test above also filters out stores of null pointers. In essence, it is treating the null pointer value of 0 as referring to an object that will never be collected, without the need for an additional explicit test.

Assuming that `p` and `q` are in registers and that the mask fits in the immediate field of an instruction, the above sequence requires only three instructions: mask, compare, and conditional branch. On the Alpha processor we indeed obtain such a sequence. The SPARC requires an additional instruction to construct the mask, since the immediate fields are too small for reasonable block sizes. One can dedicate a register to hold the mask, and thereby reduce the sequence to three instructions.

The slow path to remember a pointer at the write barrier consists of the following: determine the target object's block (shift the address right), index a block table (the base of which is in a register), load a pointer into the block's remembered set, decrement the remembered set pointer and check for underflow (explained in a moment), save the pointer to be remembered, and store the decremented remembered set pointer back into the block table. We organize each block's (generation's, in a generational collector) remembered set as a linked list of chunks, where each chunk holds 15 remembered pointers in sequential memory addresses. We allocate these chunks on aligned memory boundaries, so the underflow test consists of checking if some low bits of the remembered set pointer are all 0.

Garbage collection requires a space overhead for its auxiliary data structures for pointer remembering; since our evaluation of the time overhead is with respect to a given heap size, a fair comparison of different collectors requires the space allowed each collector for ordinary data to be diminished by the amount needed for auxiliary data (which it is difficult to do *a priori*). In our study, OF collectors have a greater space overhead than 2G because their pointer filtering is less efficient. However, we measured the space overhead of OF on our suite of benchmarks to be only 1% of heap size—therefore the consequent time overheads are negligible.

6 EVALUATING TOTAL COLLECTION COSTS

We evaluate our proposed collection algorithm and write barrier on our benchmark suite using a combination of simulation and prototyping. We obtained heap traces (described in detail below) from program runs in a Smalltalk and a Java virtual machine. *These traces are independent of the storage management scheme of the system from which they were collected.* For each collection algorithm we study, we process the traces using a driver routine, which performs relevant actions (such as object allocation and mutation) on objects in a heap. An actual implementation of the particular collection algorithm manages the heap. From this implementation, we obtain exact counts of various relevant quantities, such as the number of objects copied, number

of bytes copied, and write barrier actions, which we use to estimate execution times.

6.1 Obtaining Counts and Volumes

We now describe in more detail how we obtained the counts and volumes we report in our results.

Traces. Our traces indicate each object allocation (with the size of the object), each update of a pointer field of a heap object, and each object "death" (an object dies when it ceases to be reachable). Object death is *precise*—in the tracing system we perform a complete garbage collection immediately before each object allocation, and note in the trace the objects that have died since the previous allocation. While this tracing technique is time-consuming, it does mean that when we present the traces to any actual collection algorithm, we will observe *exactly* the collection behavior we would have obtained from the corresponding program (but without running the program).

Driver. The driver routine is straightforward in concept: it simply reads and obeys each trace record, by taking appropriate action on the prototype heap implementation. A key difference between the driver and a live program is that, since our traces do not include manipulations of local and global variables, the driver keeps a table (on the side) of *all live objects*. When the driver processes an object death record, it deletes the corresponding object from the table of live objects. From the point of view of the collector, the driver thus differs from a live program only in that more objects are referred to directly rather than reached only via other objects.

Prototype heap implementations and write barriers. All the heap implementations share some common infrastructure. Each heap consists of a collection of blocks, which are aligned, 2^k-byte portions of memory. We varied the block size in some experiments. Each heap also has remembered set data structures and write barriers appropriate to that heap. For example, the generational heap uses a generational comparison, whereas the OF heap uses the same-block and directional filtering. We note that these implementations are highly instrumented, so that we can tell how many pointer stores go down each filtering path of each write barrier. Likewise, the collector cores are highly instrumented to obtain accurate counts of copying actions. We do not obtain wall-clock timings from these prototype heap implementations.

6.2 Estimating Execution Times

Pending a complete implementation, we carefully implemented the write barriers and other actions and timed them. All code fragments have the same advantages, i.e., they execute in tight loops with important quantities in registers, so we argue that the *ratio* of their timings gives a reasonable order-of-magnitude estimate of the ratio we would expect in an actual implementation, even though the absolute values of the timings are optimistic.

We used a 292 MHz Alpha 21164. We took a cycle count measurement by running a piece of code, with and without the fragment we wished to measure, for many iterations of a loop, then taking the difference in times and dividing by the clock period.

Write barrier. Depending on the details of the loop in which we embedded the barrier, the fast path took 1, 2, or 3 cycles, which we expected since the original sequence is 3 instructions and the Alpha has an issue width of 4 (i.e., the alignment matters). We use 2 cycles in our estimates. Remembering a pointer on the slow path of the write barrier takes an average of 11 cycles (including the original test, and the time needed for chunk management on overflow). Finally, to fetch a remembered set entry, examine the target object, and possibly start to copy the object takes 13 cycles on average. Thus the total cost to create and process a remembered set entry, exclusive of copying its target object, is 24 cycles.

Copying timing. Object copying involves more than simply copying some bytes from one place to another. One must also: decode the object header, determine which fields of the object contain pointers, and handle each one of those pointers, thus accomplishing the transitive closure of the points-to relation in a breadth-first manner [9]. Since our prototype heaps were slightly simplified from actual language implementations (i.e., we did not deal with all special cases that arise in Java, such as finalization and locks), any comparisons are likely to underestimate copying cost, and thus underestimate the benefits of OF.

We modelled the total copying and collection processing costs using this equation:

$$c = \alpha_{obj} \cdot n_{obj} + \alpha_{w} \cdot n_{w} + \alpha_{skp} \cdot n_{skp} + \alpha_{dup} \cdot n_{dup}$$

Here the α are the costs per occurrence of each case and the n are the number of times that case occurs. The subscript *obj* concerns the number of objects processed, *w* the number of words copied, *skp* the number of pointer fields skipped because they are null or do not point into the collected region, and *dup* the number of pointers into the collected region but to objects already copied. Note that when we encounter a pointer to an object in the collected region but not yet copied, we charge our cost of discovery to the copying of that object.

We measured the following values (for operation with all data structures in primary cache): $\alpha_{obj} = 65$ cycles, $\alpha_{w} = 2.5$ cycles, $\alpha_{skp} = 15$ cycles, and $\alpha_{dup} = 17$ cycles. As an aside, we note that these costs indicate that copying the words is not a large component of the cost of processing pointer-rich objects.

Given our instrumentation to gather counts (the n as well as the number of times the different write barrier actions occur) and our careful estimates of the times for the various collector and write barrier operations, we can project cycle costs for each collection algorithm. As previously mentioned, we would not claim that the difference in predicted cycle counts would exactly match that in practice, but that *ratios* of predicted cycle costs would be reliable to an order of magnitude. Put another way, if we predict a ratio of collection costs of 2:1 or more, then it would be surprising if an implementation showed an inversion of costs of the schemes.

6.3 Results

We applied the block-based evaluator to our benchmark suite. We now examine the resulting evaluation of the older-first and generational collectors with the detailed cost model just described which takes into account both copying and pointer-tracking costs.

Similar to the mark/cons ratio plots we examined in Section 4, the plots of total cost in Figures 11–22 show the lowest total cost that each collector can achieve, among all examined configurations for a given heap size. The minimum heap size equals the maximum amount of live data, and evaluated heap sizes range from 2 to 6 times that minimum. While pointer costs work in favor of the 2G and against the OF collector, and diminish the advantages that OF enjoyed in the estimate of copying costs in Section 4, nevertheless they do not succeed in changing the qualitative relationship that we observed previously. On one subset of benchmarks (JavaBYTEmark, StandardNonInteractive, HeapSim, Lambda-Fact5, Lambda-Fact6,Richards) the OF collector has a clear advantage, except with very small heap sizes. On the remaining benchmarks, the performance of the two collectors is similar.

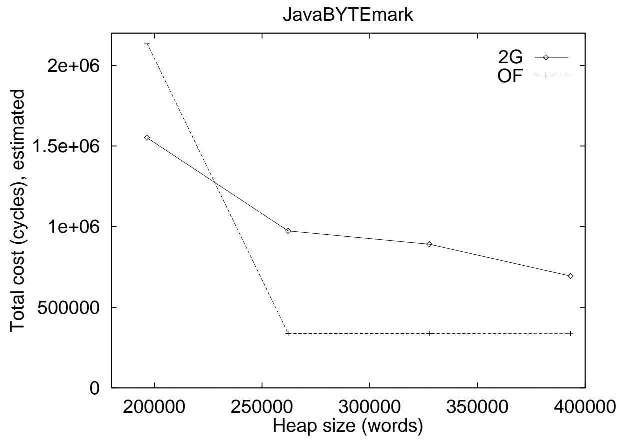


Figure 11: Total collection cost, JavaBYTEmark.

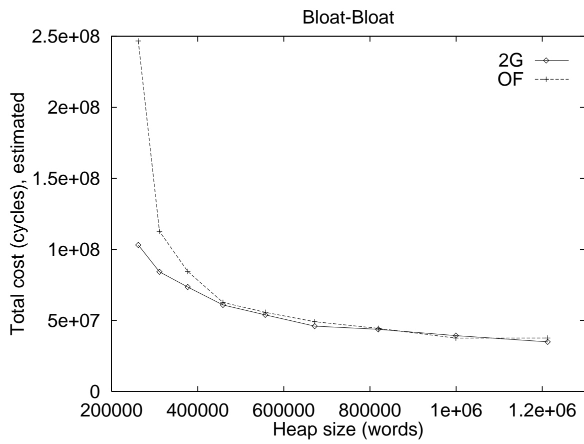


Figure 12: Total collection cost, Bloat-Bloat.

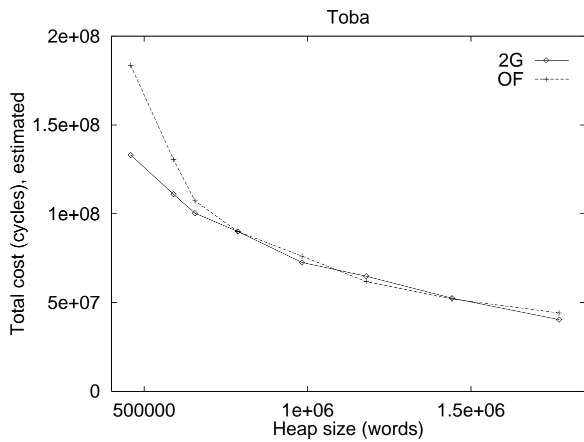


Figure 13: Total collection cost, Toba.

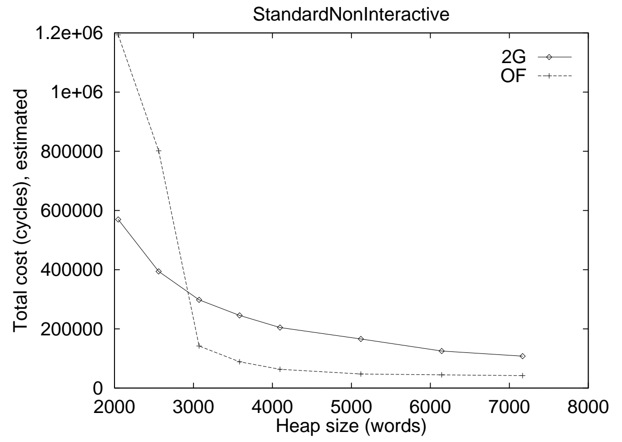


Figure 14: Total collection cost, StandardNonInteractive.

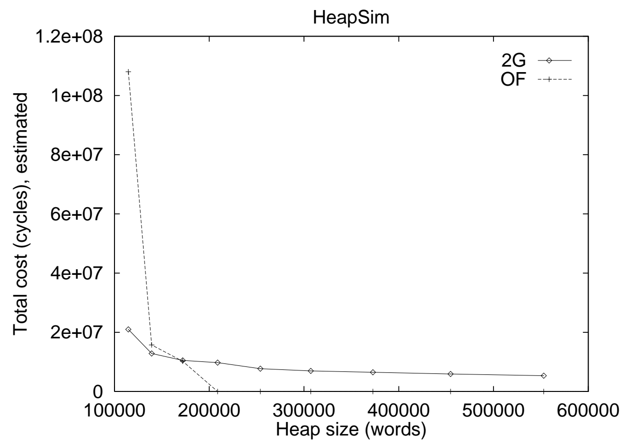


Figure 15: Total collection cost, HeapSim.

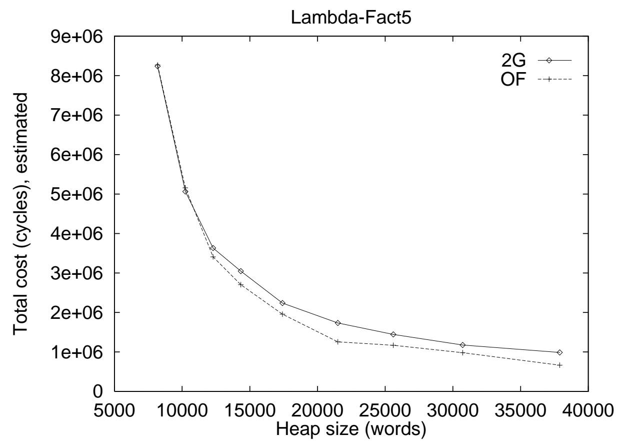


Figure 16: Total collection cost, Lambda-Fact5.

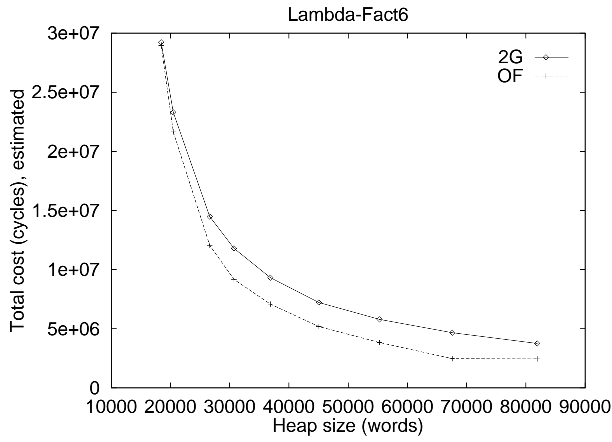


Figure 17: Total collection cost, Lambda-Fact6.

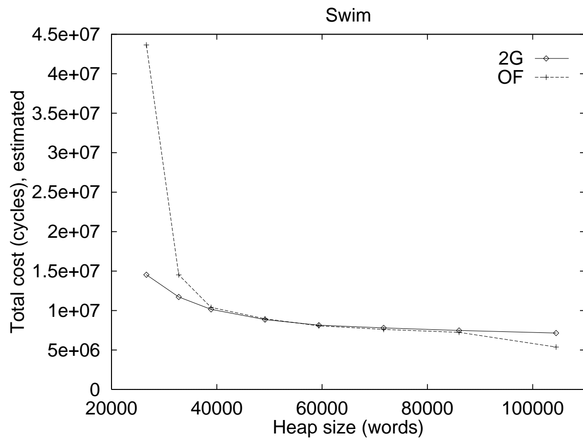


Figure 18: Total collection cost, Swim.

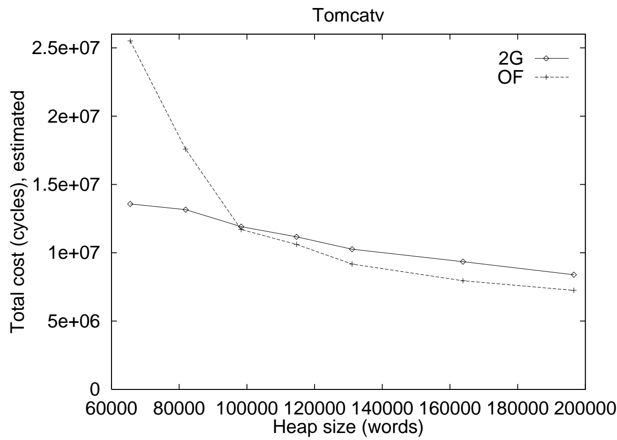


Figure 19: Total collection cost, Tomcatv.

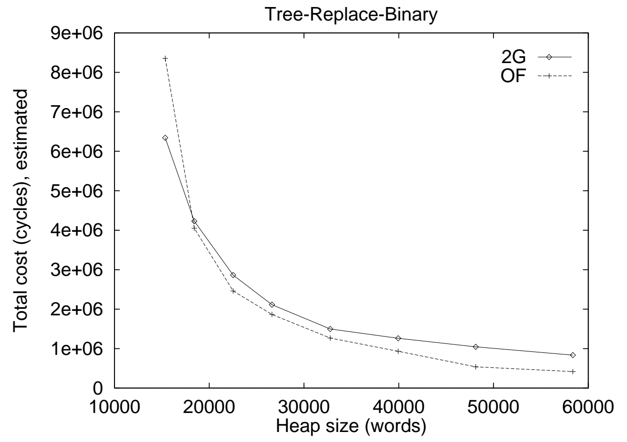


Figure 20: Total collection cost, Tree-Replace-Binary.

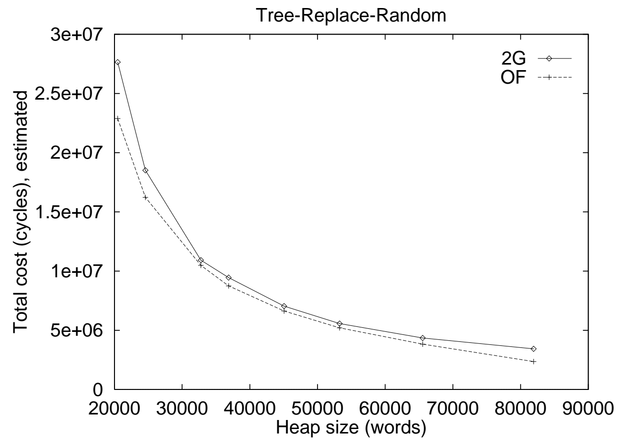


Figure 21: Total collection cost, Tree-Replace-Random.

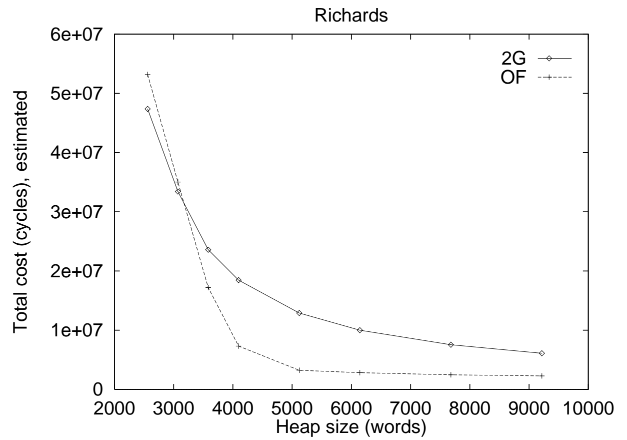


Figure 22: Total collection cost, Richards.

7 DISCUSSION

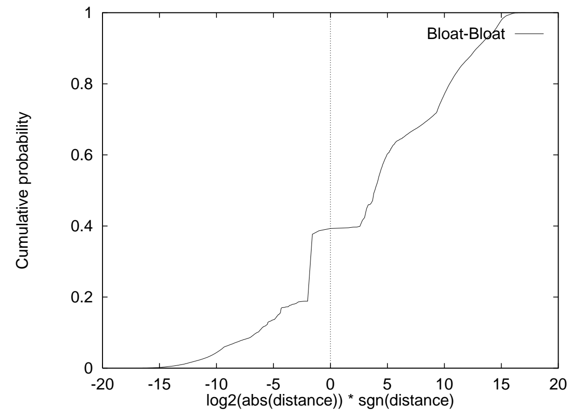
Comparing collectors. A straightforward comparison between OF and 2G collectors shows that OF achieves lower total costs in many cases. The main contributing factor is the reduction of copying cost; the supporting factor is the containment of the increase of pointer-tracking cost.

That copying costs can be markedly lower than with generational collection, in a collector that scavenges areas other than the youngest, is perplexing in the light of widely recognized good performance of generational collectors. Nevertheless, it is entirely in accord with the intuition that the very youngest objects are live, and to collect them is wasteful. In generational collection there is a tension between the need to increase the size of the nursery so as to reduce such wasteful copying of young objects, and the need to increase the size of older generations so that they are not collected frequently—a tension that cannot be resolved in a heap of finite size. In contrast, Older-First collection is able to focus on an age range where wasteful copying is minimized, which results in good performance on those programs where such a range prominently exists. Whereas our diagram in Figure 6 shows how this desirable behavior may arise, it is tempting to consider how a designer could encourage it. For example, further improvements may be achieved by dynamically (adaptively) choosing the size of the collection window, and, more ambitiously, looking at window motion policies more sophisticated than the one we have described.

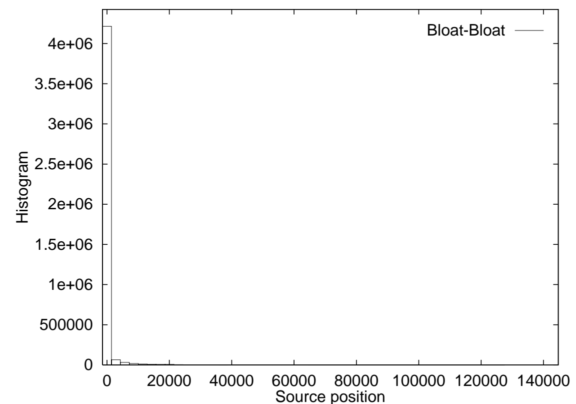
Pointer tracking. While an ever-increasing latitude in collection policy may further reduce copying costs below those of generational collection and the simple Older-First scheme, it will also be necessary to keep the pointer-tracking costs within reason. The pointer-tracking costs in OF, albeit high with respect to generational collection, are not excessive, because its window motion policy allows efficient pointer filtering. Any block-based collector can apply a filter to ignore pointer stores that do not cross block boundaries; we found that filter to eliminate about 60% of stores for reasonable configurations (note that blocks cannot be arbitrarily large lest the collector degenerate into a non-generational one). Directional filtering (Figure 9), ignores about 95% of stores: not as many as generational filtering, which ignores about 99%, but enough that the cost for the remaining, remembered, stores does not substantially offset the copying cost reduction.

As we developed our directional filtering scheme, we collected statistics of pointer stores, according to the position, in an age-ordered heap, of the pointer source and target (i.e., the object containing the reference, and the referent object), which shed new light on some long-held beliefs about the pointer structure of heaps. It has been widely assumed that pointers tend to point from younger objects to older ones. While this belief is surely justified for functional programs, it is not generally true of the object-oriented programs we examined. Both younger-to-older and older-to-younger directions are well represented, neither dominant, in most of our benchmarks. The supposed predominance of younger-to-older pointers is often cited as cause and justification of the efficacy of generational pointer filtering. A more faithful explanation arises from our observations: most pointer stores are to objects that are very young, and they install pointers to target objects that are also very young (whether relatively younger or older than the source), and a generational filter ignores these stores because they are between objects of the *same* generation. Figure 23 provides an example: (a) in Bloat-Bloat, older-to-younger pointers (negative age distances) account for 40% of the stores; however, the histogram of source positions (b) as well as that of target positions (c) show that most stores establish pointers between very young objects.

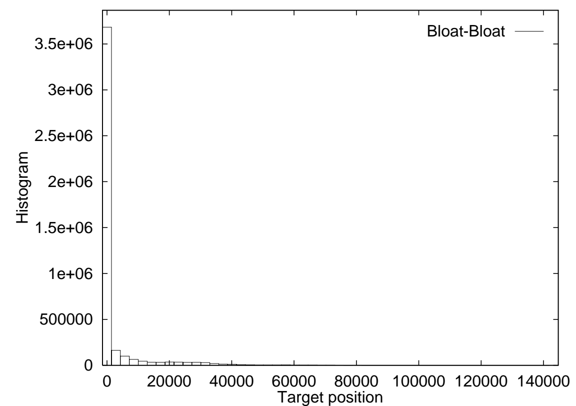
Caching and memory effects. Since copying collectors only touch the live data, and leave untouched newly dead objects, collectors that copy less should also have good locality. However, OF visits the entire heap more regularly as compared to generational collectors,



(a) Distribution of pointer age distances



(b) Distribution of pointer source ages



(c) Distribution of pointer target ages

Figure 23: Pointer store heap position: Bloat-Bloat.

which may decrease its locality in the cache and increase its paging activity. Clearly, we can only study these effects in the context of a complete implementation, and we will do so in future work.

8 RELATED WORK

The overwhelming consensus in the studies on generational garbage collection has been that a younger-first discipline should be used; i.e., that when the collector decides to examine one generation, it must at the same time examine all younger generations. The scheme that we introduce may be understood (if we ignore policy details), as similar to requiring an older generation to be collected apart from younger ones. This possibility is indeed mentioned, but dismissed both in Wilson's survey of garbage collection [32, p. 36] and in Jones and Lins' monograph [17, p.151], the two most accessible sources on the state of the art in uniprocessor garbage collection.

Generational garbage collection employs fixed boundaries between generations, in order to minimize the pointer-tracking effort needed for each such boundary. Barrett and Zorn explored the possibility of using flexible generation boundaries (remaining however within the youngest-first discipline), and found that the increase in pointer-tracking effort need not be excessive [5]. Our OF scheme uses flexible collection region boundaries, but we combine it with efficient mechanisms to keep pointer-tracking costs in check, even without the youngest-first discipline.

Clinger and Hansen proposed a collector for Scheme that does not base collection decisions on object age, but rather on the time elapsed since last collection [11], and focuses on objects for which that time is longest. (There have been historical precursors to this idea [2, 18, 6].) Although this algorithm is not age-based, it prompted us to investigate similarly flexible age-based ones; in the context of object-oriented languages that we examined, we found the latter to be superior.

More generally, schemes have been suggested that divide the heap into regions, not necessarily age-based, that can be collected independently and/or incrementally. Bishop proposed such segregation in accordance with the usage of objects [8], while Hudson and Moss's mature object space algorithm (for managing very-long-lived data) introduced policies that approximate the age-order criterion [16].

In garbage collection, there is an inherent trade-off between space and time overheads, and there is a trade-off between reducing the total time overhead and reducing the time of a single collection (for incremental operation). Different authors have applied different measures in their system evaluation. Our focus is on time overhead of collection within given space constraints. Therefore, without making specific comparisons, which are difficult when evaluation metrics as well as underlying languages are widely different, we recognize that our study draws on previous experience with generational garbage collection implementations [19, 27, 20, 24, 28, 35], their policies [29, 30, 31, 34, 1, 13], their write barrier mechanisms [33, 15, 14], and their evaluation with respect to object allocation and lifetime behavior [3, 26, 11].

Achieving performance improvements with generational collection critically depends on setting or adapting the configuration parameters right—incorrectly chosen generation sizes can cause performance to degrade severely. We have confirmed these matters in our observations of multi-generational collectors on our benchmark traces. Choosing a good regime of generations is not an easy task, and it is not yet fully understood despite numerous studies [29, 36, 34, 1, 5]. However, we can also say that it is a matter of tuning the performance *within* the class of youngest-only collection schemes. Our goal in this study has not been to examine how to tune a particular scheme, but instead to compare the schemes. Whether optimal configurations can be chosen *a priori*, or how a system might adaptively arrive at them are questions for separate investigation.

9 SUMMARY

Generational collection achieves good performance by considering only a portion of the heap at each collection. It achieves this good performance even while imposing additional costs on the mutator, namely a write barrier to track pointers from older to younger generations. We found that we can reduce copying costs further, in many cases dramatically, by not including the youngest objects in each collection, and we call this more general scheme *age-based collection* since it still determines which objects to collect based on age. We considered in detail a particular age-based algorithm that we term *older-first* (OF) and found that it never needed to copy substantially more data than generational collection, and copied up to ten times less for some programs. OF does require more write barrier work than generational collection, perhaps ten times more, but the savings in copying can outweigh the extra pointer tracking costs.

We obtained these results with exact heap contents simulation, prototype collector implementation, and careful timing of crucial code fragments. Given the factor by which OF outperforms generational collection—often a factor of 2 or more—it should also perform well in actual implementation. Integration with a Java virtual machine is in progress.

While improved performance is one measure of the significance of this work, we also feel that it contributes substantially to our understanding of memory usage and garbage collector behavior. Put another way, garbage collection has a long tradition of study, yet we have shown that the widely accepted state of the art, generational collection, leaves considerable room for improvement.

We also question some of the widely held beliefs about generational collection, offering new intuition. While we clearly agree with the tenet that one should wait for objects to die before collecting them, as it has been recognized in the considerable body of work concerning the avoidance of early "tenuring" of objects, we show that it is practical to avoid copying the very youngest objects and that doing so saves much work, even though it imposes a heavier burden on the running program. In the past the write barrier cost was thought too high to permit exploring algorithms like OF. Now we have results encouraging consideration of a wide range of new techniques.

Future work should include considering other window motion algorithms, dynamically changing the window size, using multiple windows (e.g., one for younger objects and one for mature objects as in mature object space collection), and more experimentation and measurement, for more programs, platforms, and languages.

Acknowledgements. We acknowledge with gratitude the assistance of David Detlefs and the Java Topics group with Sun Microsystems Laboratories, Chelmsford, Massachusetts, in collecting and providing traces for this work. We thank Margaret Martonosi and the anonymous referees for valuable comments on drafts of this paper.

References

[1] Appel, A. W. Simple generational garbage collection and fast allocation. *Software Practice and Experience 19*, 2 (1989), 171–183.

[2] Baker, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (1978), 280–94. Also AI Laboratory Working Paper 139, 1977.

[3] Baker, H. G. 'Infant Mortality' and generational garbage collection. *SIGPLAN Notices 28*, 4 (1993), 55–57.

[4] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier (North-Holland), Amsterdam, 1984. Second edition.

[5] Barrett, D. A., and Zorn, B. Garbage collection using a dynamic threatening boundary. In *Proceedings of SIGPLAN'95 Conference on Programming Languages Design and Implementation* (La Jolla, CA, June 1995), vol. 30 of *SIGPLAN Notices*, ACM Press, pp. 301–314.

[6] Bekkers, Y., Canet, B., Ridoux, O., and Ungaro, L. MALI: A memory with a real-time garbage collector for implementing logic programming languages. In *3rd Symposium on Logic Programming* (1986), IEEE Press, pp. 258–264.

[7] Bekkers, Y., and Cohen, J., Eds. *International Workshop on Memory Management* (St. Malo, France, Sept. 1992), no. 637 in Lecture Notes in Computer Science, Springer-Verlag.

[8] Bishop, P. B. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1977.

[9] Cheney, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (Nov. 1970), 677–678.

[10] Cheng, P., Harper, R., and Lee, P. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation* (Montreal, Québec, Canada, June 1998), vol. 33 of *SIGPLAN Notices*, ACM Press, pp. 162–173.

[11] Clinger, W. D., and Hansen, L. T. Generational garbage collection and the radioactive decay model. *SIGPLAN Notices 32*, 5 (May 1997), 97–108. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*.

[12] Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[13] Hayes, B. *Key Objects in Garbage Collection*. PhD thesis, Stanford University, Stanford, California, Mar. 1993.

[14] Hosking, A. L., and Hudson, R. L. Remembered sets can also play cards. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems* (Oct. 1993), E. Moss, P. R. Wilson, and B. Zorn, Eds.

[15] Hosking, A. L., Moss, J. E. B., and Stefanović, D. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, Canada, Oct. 1992), *SIGPLAN Notices 27*, 10 (Oct. 1992), pp. 92–109.

[16] Hudson, R. L., and Moss, J. E. B. Incremental collection of mature objects. In Bekkers and Cohen [7], pp. 388–403.

[17] Jones, R., and Lins, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, Chichester, 1996.

[18] Lang, B., and Dupont, F. Incremental incrementally compacting garbage collection. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques* (Inst. Rech. Informat. Automat. Lab., France, 1987), vol. 22(7) of *SIGPLAN Notices*, ACM Press, pp. 253–263.

[19] Lieberman, H., and Hewitt, C. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM 26*, 6 (June 1983), 419–429.

[20] Moon, D. Garbage collection in a large Lisp system. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Austin, TX, Aug. 1984), pp. 235–246.

[21] Nystrom, N. Bytecode-level analysis and optimization of Java class files. Master's thesis, Purdue University, West Lafayette, IN, May 1998.

[22] Odersky, M., and Wadler, P. Pizza into Java: translating theory into practice. In *POPL '97. Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming* (New York, 1997), ACM Press, pp. 146–159.

[23] Proebsting, T. A., Townsend, G., Bridges, P., Hartman, J. H., Newsham, T., and Watterson, S. A. Toba: Java for applications, a way ahead of time (WAT) compiler. (at http://www.cs.arizona.edu/sumatra/toba/), 1998.

[24] Sobalvarro, P. G. A lifetime-based garbage collector for LISP systems on general-purpose computers, 1988. B.S. Thesis, Dept. of EECS, Massachusetts Institute of Technology, Cambridge.

[25] Stefanović, D. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, Amherst, MA, Feb. 1999.

[26] Stefanović, D., and Moss, J. E. B. Characterisation of object behaviour in Standard ML of New Jersey. In *1994 ACM Conference on Lisp and Functional Programming* (Orlando, Florida, June 1994), ACM Press.

[27] Ungar, D. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, Apr. 1984), *SIGPLAN Notices 19*, 5 (May 1984), pp. 157–167.

[28] Ungar, D. M. *The Design and Evaluation of a High Performance Smalltalk System*. ACM distinguished dissertation 1986. MIT Press, 1986.

[29] Ungar, D. M., and Jackson, F. Tenuring policies for generation-based storage reclamation. *SIGPLAN Notices 23*, 11 (1988), 1–17.

[30] Ungar, D. M., and Jackson, F. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems 14*, 1 (1992), 1–27.

[31] Wilson, P. R. A simple bucket-brigade advancement mechanism for generation-based garbage collection. *SIGPLAN Notices 24*, 5 (May 1989), 38–46.

[32] Wilson, P. R. Uniprocessor garbage collection techniques. In Bekkers and Cohen [7], pp. 1–42.

[33] Wilson, P. R., and Moher, T. G. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Notices 24*, 5 (May 1989), 87–92.

[34] Wilson, P. R., and Moher, T. G. Design of the opportunistic garbage collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, Louisiana, Oct. 1989), *SIGPLAN Notices 24*, 10 (Oct. 1989), pp. 23–35.

[35] Zorn, B. Barrier methods for garbage collection. Tech. Rep. CU-CS-494-90, University of Colorado at Boulder, Nov. 1990.

[36] Zorn, B. G. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Dec. 1989. Available as Technical Report UCB/CSD 89/544.