Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine*

Darko Stefanović[§] Matthew Hertz[†] Stephen M. Blackburn[¶] Kathryn S. McKinley[‡] J. Eliot B. Moss[†]

[§] Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
darko@cs.unm.edu

[†] Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
{hertz,moss}@cs.umass.edu

[¶] Dept. of Computer Science
Australian National University
Canberra, ACT, 0200, Australia
Steve.Blackburn@cs.anu.edu.au

[‡] Dept. of Computer Sciences
University of Texas at Austin
Austin, TX, 78712
mckinley@cs.utexas.edu

ABSTRACT

Until recently, the best performing copying garbage collectors used a generational policy which repeatedly collects the very youngest objects, copies any survivors to an older space, and then infrequently collects the older space. A previous study that used garbage-collection simulation pointed to potential improvements by using an *Older-First* copying garbage collection algorithm. The Older-First algorithm sweeps a fixed-sized window through the heap from older to younger objects, and avoids copying the very youngest objects which have not yet had sufficient time to die. We describe and examine here an implementation of the Older-First algorithm in the Jikes RVM for Java. This investigation shows that Older-First can perform as well as the simulation results suggested, and greatly improves total program performance when compared to using a fixed-size nursery generational collector. We further compare Older-First to a flexible-size nursery generational collector in which the nursery occupies all of the heap that does not contain older objects. In these comparisons, the flexible-nursery collector is occasionally the better of the two, but on average the Older-First collector performs the best.

1. INTRODUCTION

Garbage collection for object-oriented programming languages automates memory management and thus relieves programmers of a source of errors and the burden of explicit memory management. Since most objects die quickly [15], generational copying collectors divide the heap into *generations* [3, 9, 15]. They collect the youngest objects frequently, and copy survivors into progressively older generations. When the heap fills, they collect the older generation together with the younger generation.

Generational collectors prematurely copy the very youngest objects because every object needs *some* time to die. The Older-First copying garbage collector [14] exploits this observation by avoiding collecting the youngest objects. It organizes the heap in order of object age and collects a fixed-size window that slides through the heap from older to younger objects. When the heap is full, Older-First collects the window, compacts the survivors (logically) in place, returns any free space to the nursery, and then positions the window for the next collection over objects just younger than those that survived. If it bumps into the allocation point, it resets the window to the oldest end of the heap.

Previous work describes a range of implementation possibilities for the Older-First collector (OF) [13, 14]. It also presents garbage collection simulation results that show OF performs much better than a fixed-size nursery generational collector. In this work, we describe an implementation of OF for Java in IBM's Jikes RVM, a well performing system [1, 2]. We present a variety of execution results that (1) validate our simulation model, (2) compare execution times and copying ratios of OF and generational collectors, and (3) explore *pause* times (the range of times for one collection) and the total collection time tradeoff.

These results show that Older-First delivers on its promise and outperforms a tuned fixed-size nursery generational collector by on average 5–25% for 10 Java programs on a wide range of heap sizes. We further show that these improvements are mostly due to reduced copying costs.

In our experiments [4] the flexible-nursery collection introduced by Appel [3] consistently performs better than a fixed-size nursery collector. Our results show that OF sometimes (in 7 of the 10 programs studied) does better than this generational collector as well, and further, OF almost always beats both generational collectors on maximum pause time, since the generational collectors must sometimes collect the whole heap, which OF never does.

We proceed in Section 2 with an overview of OF and design decisions we made in the implementation as part of our Garbage Collector Toolkit for the IBM Jikes RVM for Java. In Section 3, we give the details of the experimental setting and Section 4 gives a comparative performance evaluation of the collector with respect to throughput, and in Section 4.3 with respect to garbage collection pause times.

2. DESIGN AND IMPLEMENTATION OF THE OLDER-FIRST COLLECTOR

The Older-First collector organizes the heap by object age [14]. It collects a fixed-size window that slides through the heap from older to younger objects. When the heap is full, OF collects the window, makes any free space available for future allocation, and then positions the window for the next collection over objects just younger than those that survived. If it bumps into the allocation

*This work is supported by NSF ITR grant CCR-0085792, NSF grant ACI-9982028, DARPA grants F30602-98-1-0101 and F33615-01-C-1892, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

MSP'02, June 16, 2002, Berlin, Germany

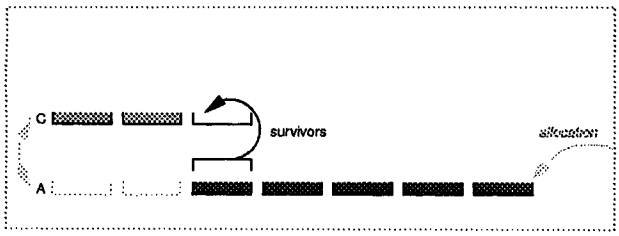


Figure 1: Design of the Older-First Collector

point, it resets the window to the oldest end of the heap.

This scheme is illustrated in Figure 1, which shows the allocation region A and the copy region C, each of which can be considered as queues of window-size groups of objects. OF allocates to the back of region A. Whenever all usable memory is consumed, OF collects a window from the front of region A, copying survivors to the back of region C. If all usable space is consumed and A is empty, then OF 'flips' the roles of the two regions, collects the first window in the new region A, and copies its survivors to the back of the new, now empty, region C. OF then continues to allocate to the back of the new region A.

As illustrated in the figure, the heap is organized from left to right, older to younger objects. The Older-First collector must *remember* pointers from any uncollected regions to the collected region, and during a collection assume that pointers into the collected region refer to live objects. OF need not remember *all* pointers between regions; it needs to remember a pointer between two regions only if it will independently collect the target before the source. A generational collector with two generations has just one such region boundary, but OF has many boundaries and thus its *write barrier* remembers more pointers.

Previous work discusses the design and several implementation strategies for OF [13, 14]. We make a few modifications to this design for our implementation. Most of these changes are necessary because the collector uses a 32-bit environment in Jikes RVM instead of the envisaged large address space, which would enable optimizations that reduce pointer maintenance costs. The original design had several other mechanisms to reduce pointer maintenance costs, and this implementation introduces a further enhancement to reduce the number of remembered pointers. The remainder of this section presents the high-level design of the Older-First algorithm implementation and discusses how this implementation deviates from the initial OF design.

2.1 Blocks and Frames

Jikes RVM currently supports only 32 bits of address space, which prevents us from using an address-ordered heap in which the write barrier can quickly compare virtual addresses to determine if it needs to remember a pointer. We instead simulate a larger address space by organizing objects within *frames*, which are mapped (by software) into a larger, logical, age-ordered address space. A frame is a contiguous aligned chunk of virtual address space of size 2^F, where F is set at system build time and was 26 for this work, giving up to 64 frames of 64 Mb each. (In practice we actually use a few less than 16 frames because of virtual address space restrictions imposed by the operating system, etc.) We call the high order bits of a frame's address the *frame number*.

A frame is the largest amount of contiguous space in which objects reside, and the frame size thus determines the maximum object size. A frame is also the minimum unit of collection, so in general we do not fill frames completely. For example, in the OF collector, the window size determines how much space we allocate within a frame.

The collector represents ages by associating a *time-of-death* (TOD) with each frame, using an array indexed by frame number. We use TOD because TOD values do not change as time passes, whereas ages do. The TOD corresponds to the frame's position in the larger logical address space, and allows us to apply an age-order write barrier that is analogous to (but not as efficient as) the address-order write barrier possible in a larger virtual address space in which the frames would be placed in virtual address space in age order.

As the program executes, first one hands frames out to the A region, filling each one with a windowfull of objects. When the heap is full, one starts handing frames out to the C region, filling them with survivors from collecting the oldest frames of the A region. After copying survivors from a collected frame, the frame becomes available to handout to either the A or C region. The actual order of the frames in virtual memory does not matter: the TOD table gives the necessary logical ordering both for collection and for the write barrier.

Space within frames is allocated in aligned chunks of size 2^B that we call *blocks*. For this study, $B = 17$, giving a block size of 128K. A block can be no smaller than a virtual memory page and no larger than a frame. We perform space accounting (i.e., enforce maximum heap size) in terms of blocks, and we map and unmap virtual memory dynamically to allocate space to frames dynamically and to recover the space at the end of each collection. Within a frame, allocation proceeds sequentially. When the next object allocation would cross a block boundary, we attempt to obtain an additional block for the frame (assuming the frame is not full). This will trigger collection if the block budget is exhausted.

2.2 Managing TOD Values

As we allocate frames to the A and C regions, the frames obtain increasing TOD values, drawing from one sequence for the A region and a higher numbered sequence for the C region. Now if the survival rate from the A region collections is low (say 1%), then the A region will consume TOD values much more rapidly than the C region grows (100 times as fast for 1% survival), which means that the A region TOD values can collide with the C region values.

In an actual large address space implementation, one would need to do something, and since we are trying to emulate the large address space case, what we do even in this implementation is collect the remaining frames of the A region all at once, and "flip" regions just as we do in the case of a window reset. In fact, we call this case a *hard window reset* ("hard" because it is forced; we also call normal window resets "soft").

When there is a "flip", we establish the starting TOD value of the new C region a certain amount higher than the starting TOD value of the (new) A region. We call that amount the *zone size*, and one can specify on the command line as a multiple of the maximum heap size (itself a command line parameter). For our runs we adjusted the zone size so that hard resets did not occur (so in fact we did not implement the hard reset case since we did not need to).

Another exceptional case is having so many window resets, hard or soft, that the TOD values themselves overflow. One way to handle this in a large address space implementation is to copy the entire heap to the other end of the address space, which we call a *zone reset*. As with hard resets, we chose the zone size so that that zone resets did not occur in our runs. We note that for long running programs, it may be impossible to avoid zone resets.

2.3 Write Barrier

Our implementation limits the number of remembered pointers while maintaining the simplicity of the OF write-barrier test. The OF write-barrier test needs to remember a pointer between two blocks only if it will collect the target block before the source block. This occurs only if the TOD value of the target block's frame is less than the TOD value of the source block's frame. Hence, it never needs to remember intra-frame pointers.

Our write barrier is complicated by the fact that we could not use a pure address-ordered write barrier, but had to use the logical ordering of frames (age). Making each frame hold a single collection window, we ensure that collections advance frame-by-frame. Therefore our write barrier can exclusive-or the source object's and target object's addresses to determine if they are in the same frame. (OF collects objects within the same frame at the same time and need not remember intra-frame pointers.) If the objects are from different frames, the barrier looks up the frames' ages in a table and uses the original age-based pointer filtering technique (OF always collects older objects before younger ones). Although more expensive than a pure address-ordered test, the implemented write barrier significantly limits the number both of expensive table lookups and of remembered pointers.

In the event, the following are the PowerPC [10] instruction sequences that the two barriers are compiled to. Figure 2 shows the code of the address-order write barrier, used in the generational collectors. Figure 3 shows the write barrier with age lookup in the frame table, used in the OF collector.

```
;; clear low-order 28 bits of pointer source:
rlwinm Rtemp, Rsource, 0x0, 0x0, 0x3
;; compare with pointer target:
cmp cr1, Rtarget, Rtemp
;; if comparison is favorable, skip remembering:
bge 1 label:do-not-remember-pointer
;; fall-through: remember pointer
```

Figure 2: Address-order write barrier

```
;; calculate frame numbers for source and target:
rlwinm Rtemp1, Rsource, 0x6, 0x1a, 0x1f
extsb Rtemp1, Rtemp1
rlwinm Rtemp2, Rtarget, 0x6, 0x1a, 0x1f
extsb Rtemp2, Rtemp2
;; intraframe pointers test:
cmp cr1, Rtemp1, Rtemp2
beq 1 label:do-not-remember-pointer
;; heap boundary test:
cmpi cr1, Rtemp2, 0xf
blt 1 label:do-not-remember-pointer
;; load base of TOD array:
lwz Rtemp3, a-static-offset(JTOC)
;; look up age of source and target:
sli Rtemp1, Rtemp1, 0x2
lwzx Rtemp1, Rtemp3, Rtemp1
sli Rtemp2, Rtemp2, 0x2
lwzx Rtemp2, Rtemp3, Rtemp2
;; age comparison test:
cmp cr1, Rtemp1, Rtemp2
ble 1 label:do-not-remember-pointer
;; fall-through: remember pointer
```

Figure 3: Write barrier with age lookup

3. EXPERIMENTAL METHOD

In this section, we describe the collectors, implementation environment, hardware platform, test programs, configuration parameters, and metrics we use to evaluate our work.

3.1 Collector families

In these experiments, the baseline is a two-generation collector with variable nursery size [3]. We refer to this collector as the *Appel* collector. It devotes all free space to the nursery. When the nursery is full, it copies surviving objects to the older generation, and then reduces the nursery size by this volume. It repeats this process until the older generation occupies the entire heap, at which point it performs a full heap collection, returning all free space to the nursery. We have found this to be the best performing generational collector [4].

Normalized to this collector, we compare two families of collectors with fixed window size—the traditional two generational fixed-size nursery collectors and the Older-First collectors. In a nursery of size k, the two generational fixed-size nursery collector sizes the nursery at k bytes, and collects every k bytes of allocation. It promotes surviving objects into the older generation, and when the heap is full, it collects the entire heap. We refer to this collector as the *generational* or *fixed-generational* collector.

We also include the non-generational semi-space collector which demonstrates that all of the collectors perform better than collecting the entire heap.

3.2 Experimental environment

We use and measure collectors in the Jikes RVM release 2.0.3 [2] (formerly Jalapeño), built and running with the optimizing compiler turned on. Jikes RVM has no interpreter, and all Java bytecode is translated to native PowerPC code before execution. The virtual machine is itself written in Java, and it translates its own bytecodes to native code [1]. This translation could be done at runtime, but to avoid obscuring the behavior of benchmark programs, the classes of the virtual machine are precompiled during the build stage of Jikes RVM. However, the measured execution includes the compilation of the application methods to native code.

Our version of Jikes RVM includes the recently developed and publicly available version of the UMass GC Toolkit. We believe the collector implementations to be well-tuned. The write barrier used in the Appel and fixed-generational collectors is an address-order write barrier with fast common case code [14, 5]. The write barrier used in the Older-First collector is somewhat less efficient, as discussed in the preceding section.[1]

The hardware platform is a Macintosh PowerMac G4 with a single 733 MHz PowerPC 7450 processor, 32 KB L1 data and instruction caches, 256 KB unified L2 cache, and 640 MB of memory, running Yellow Dog Linux 2.1 (Linux kernel 2.4.10). The machine is placed in single-user mode and disconnected from the network for the duration of the experiment.

[1]These results can still be improved, for the compiler does not fully optimize the write barrier code. Upon inspecting the compiled result, around the actual barrier code (Figure 3) we find some unneeded instructions and at least one extra branch. We expect that a fully optimized barrier would provide an additional 1-2% time savings, but, without significant reengineering of the optimizing compiler we cannot replace the compiled barrier with fully optimized code to verify this savings. On the other hand, we are currently developing a 64-bit version of Jikes RVM which will permit using the address-order write barrier for the Older-First collector as well.

3.3 Test programs

We use all 8 programs from SPECjvm98 [11, 8] without any modification. Using SPEC JBB [12] posed a challenge: in its original form, it is a throughput-based self-calibrating program. We modified the code so that it performs a fixed amount of work. To avoid confusion, this benchmark is named *pseudojbb*. In a similar fashion, *pseudojBYTEmark* was derived from the javaBYTEmark code. A summary of benchmark programs used is in Table 1.

Since the Appel collector serves as reference for performance measurements, we use it to determine the minimum heap size needed to run each benchmark (namely, we stipulate that under no circumstance will the memory manager request more memory from the operating system than a given amount—instead, the memory manager *fails* if it cannot satisfy all requests within that amount of memory). In many cases, other collectors need somewhat larger heaps to operate and thus data points for very small heaps will be absent. In the results below, we report heap sizes relative to this minimum heap size.

3.4 Configuration parameters

We vary the heap size between the minimum feasible size and 3.25 times that amount. This range reveals the space-time tradeoff which is *de rigueur* in garbage collection. In small heaps, the collector runs more frequently and in larger heaps, less. In very large heaps that are sparsely populated with live objects, paging results. Note that even the largest benchmark configurations operate in less than a third of available physical memory on the experimental platform, therefore we entirely avoid paging activity in these experiments. We cover the range 1–3.25 with 17 heap sizes, spaced more densely towards the low end.

For both generational and Older-First collector families, we vary the window size between 5% and 60% of maximum permissible size, which is roughly half the heap size. (Within this range, we find the best-performing window sizes for each family. With larger window sizes, both families quickly degenerate into the semi-space collector.) We use 9 window sizes in this range.

3.5 Metrics

The first performance metric is the mark/cons ratio. For copying garbage collectors such as the ones we consider here, this ratio is the total amount of data copied by the collector divided by the total amount of data freshly allocated by the program (last column of Table 1). If the expense of copying data is the predominant expense of garbage collection, and that cost is nearly proportional to the amount copied, the mark/cons ratio ought to be a good indicator of performance. Perhaps this expense solely determined collection costs early in the history of garbage collection work. In general, systematic differences arise between collector families because of copying costs and other costs, such as pointer tracking [14]. Nevertheless, the mark/cons ratio provides a clue into the copying cost, and remains the only direct metric derivable in simulation.

Counting the amount of data allocated imposes a significant overhead on each object allocation, and similarly for the amount of data copied. Therefore, we perform separate statistics-gathering experimental runs to obtain mark/cons ratios.

We use the total execution time of the program as the ultimate metric of garbage collection performance. Each reported time is the minimum over three measured runs for a given configuration. Total execution time includes costs incurred both at garbage collection time and within the mutator, including the cache locality effects of object motion and of write-barrier actions. Unfortunately, the total execution time does not provide an insight into the contribution of these various effects. We do report garbage collection times, and include a few results for write-barrier effects.

Garbage collection time is the sum of all collector induced pause times, which we measure in elapsed time using the Jikes RVM interface to the PowerPC Linux system clock, with an effective millisecond resolution. Note that the reported garbage collection time does not capture the *full* cost of memory management, since most allocation and write barrier actions take place during the execution of the mutator program, and not during garbage collection pauses. These actions are extremely short and interleaved with application code as a result of instruction scheduling and out-of-order execution; therefore it is not feasible to measure their cost directly.

4. RESULTS

Because the experimental results cover a large configuration space, we begin by considering in detail a single benchmark, and then present total time, mark/cons, and pause-time results for all benchmarks. We choose *pseudojbb* as our detailed study because it allocates the most and has the largest live data size in our set.

4.1 Results for benchmark pseudojbb

Figure 4 shows the mark/cons ratios obtained in statistics-gathering runs, with a graph for the generational collector, and a graph for the Older-First collector. Heap size (horizontal) is drawn to a logarithmic scale to provide details at smaller sizes close to the minimum feasible. Mark/cons ratios (vertical) are normalized with respect to the mark/cons ratio of the Appel collector at each given heap size.

For the fixed-generational collector, the relative mark/cons ratio is almost always above 1. Thus, the Appel collector copies significantly less and utilizes the heap better than a fixed-generational collector, for all choices of nursery size on a range of heap sizes. This result agrees with the goals of the Appel collector [3], but is only now appearing in the literature [4]. Figure 4(b) shows that the mark/cons ratio of the Older-First collector is both lower than the generational collector and usually lower than Appel. These results confirm our earlier simulation-based study comparing OF with the fixed-generational collector [14].

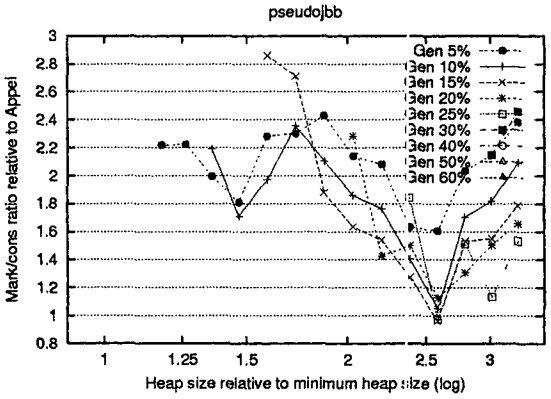
Looking more closely at the sizes of the collected region (nursery in the case of the generational collector, window size for OF), we note that there is considerable variation in which region size gives the lowest mark/cons ratio, as the heap size is varied, and also that efficient sizes tend to be small, but not too small. Many configurations of the generational collector operate well with nursery sizes from 5 to 15% of the heap size, but fail with large sizes. OF is more robust with respect to this parameter; it operates well with window sizes between 5 and 40% of the heap.

Figure 5 shows the garbage collection times for different garbage collectors, again for *pseudojbb*. Although there is variation in the fine detail, garbage collection times have the same behavior as the mark/cons ratios: the fixed-generational collector generally exhibits higher garbage collection times than Appel, and OF generally further lowers times, with relative differences diminishing towards larger heap sizes.

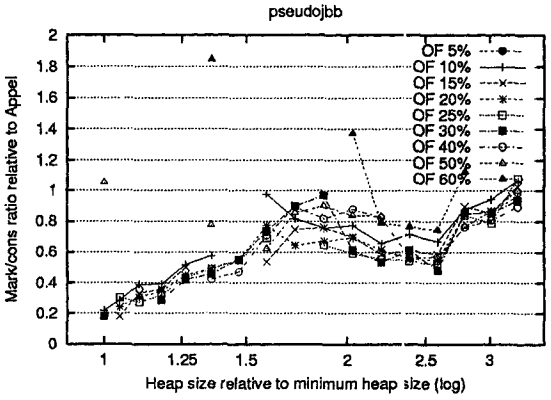
Figure 6 shows the total execution times for *pseudojbb*. Recall that total execution time comprises the time spent in garbage collection (previous figure), mutator or useful work time, and write-barrier time (incurred within the mutator but not measured separately). Comparison of Figure 5(a) and Figure 6(a) shows that the fixed-generational collector has a higher total execution time than the Appel collector, but the relative difference is not as pronounced as for garbage collection time alone. This dilution of differences is expected, because garbage collection time is considerably less than mutator time, especially for larger heaps, as shown in Figure 7 for the Appel collector.

Program	Description	MH	AL
SPEC_201_compress	Compresses and decompresses 20MB of data using the Lempel-Ziv method.	19	215
SPEC_202_jess	Expert shell system using NASA CLIPS.	12	508
SPEC_205_raytrace	Raytraces a scene into a memory buffer.	15	252
SPEC_209_db	Performs series of database functions on a memory resident database.	22	192
SPEC_213_javac	Sun's JDK 1.0.4 compiler.	28	639
SPEC_222_mpegaudio	MPEG audio decoder	10	153
SPEC_228_mtrt	Graphics ray tracer	21	255
SPEC_228_jack	Generates a parser for Java programs.	14	534
pseudojbb	Fixed-work version of the SPEC JBB benchmark	59	667
pseudojBYTEmark	Fixed-work version of the JavaBYTEmark benchmark	12	211

Table 1: Benchmark programs used in the experiment. MH is the minimum heap size needed to run the program using the Appel collector, and AL is the total amount of data allocated by the program. Both are expressed in megabytes.

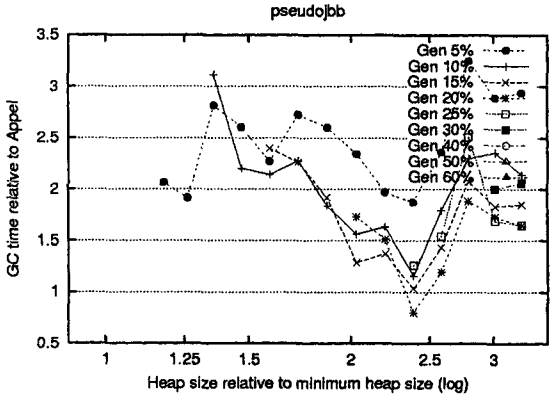


(a) Generational collector

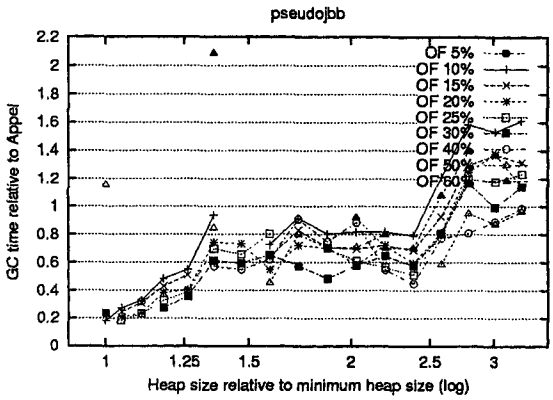


(b) Older-First collector

Figure 4: Mark/cons ratio: *pseudojbb*.

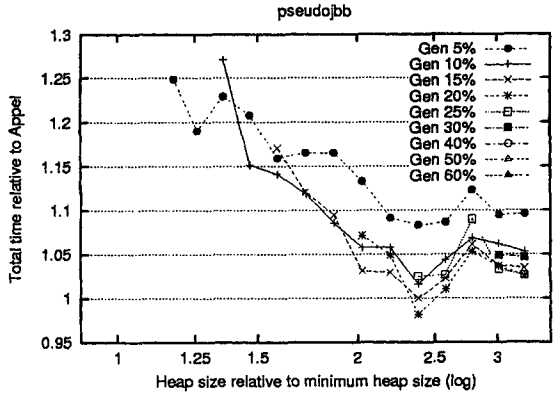


(a) Generational collector

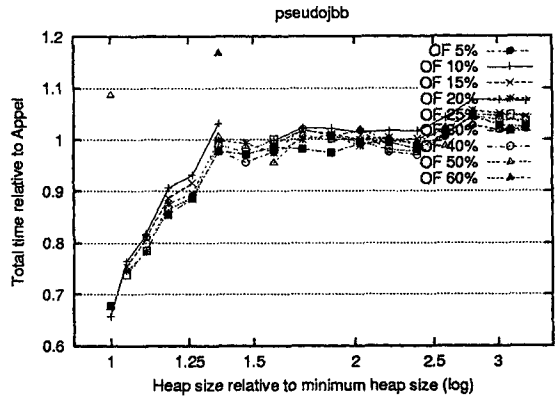


(b) Older-First collector

Figure 5: Garbage collection time: *pseudojbb*.



(a) Generational collector



(b) Older-First collector

Figure 6: Total execution time: *pseudojbb*.

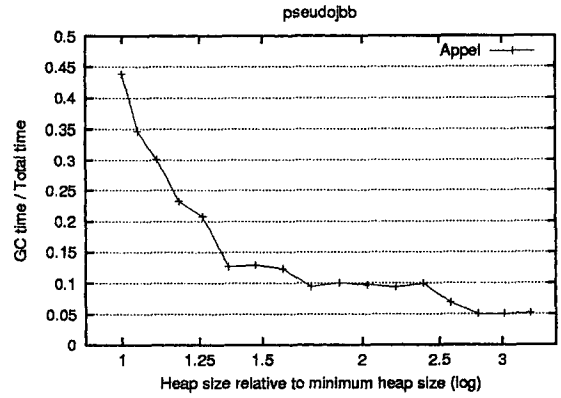


Figure 7: GC time as fraction of total execution time (Appel collector): *pseudojbb*.

Comparison of Figure 5(b) and Figure 6(b) shows that the range of heap sizes at which the total execution time of the Older-First collector is below that of the Appel collector (1–1.6) is reduced with respect to the corresponding range for garbage collection times alone (1–2.5). This result is explained by an unequal contribution of write-barrier times: OF must record more pointer stores than generational collectors, and its write barrier implementation is not as efficient.

To demonstrate these differences, we take as a case study the execution of *pseudojbb* in a heap of size 74 (relative heap size ≈1.25), comparing the Appel collector and the OF collector with window size 10%. Total execution times are 42.035s (OF) and 45.155s (Appel), giving the ratio 0.93 (as in Figure 6(b)). Garbage collection times are 5.148s (OF) and 9.378s (Appel), giving the ratio 0.55 (as in Figure 5(b)). Mark/cons ratios are 0.173 (OF) and 0.334 (Appel), giving the ratio 0.52 (as in Figure 4(b)). Now we look at pointer-maintenance costs. In each case, 98.2 million write barriers (code in Figure 3 or fig:write-barrier-address-order) were executed. The number of interesting pointers, which must be remembered, is 6.24 million for OF, but only 2.59 million for the Appel collector, giving a ratio of 2.41. A further difference arises at garbage collection time, when remembered pointers with target in the collected region are processed. In OF, a total of 10.32 million remembered pointers are processed, but only 2.59 million for the Appel collector, giving a ratio of 3.99. A further difference is in the number of garbage collections; OF performs 97 collections, the Appel collector 85. Although a larger number of collections may be good for reducing pause time (Section 4.3), it increases the execution time, since stacks must be scanned more often. Here is the tradeoff we have made. The flexible choice of garbage collection region as in OF has resulted in having to record approximately 2.5 times more pointers at write barriers, and to process approximately 4 times more pointers at garbage collection time; 14% more time is spent in stack scanning and other GC startup overhead. In spite of these measured factors, and the disadvantage of a slower write barrier "fast path" which we could not directly measure, the total execution time for OF is 7% lower, thanks to a halving of the mark/cons ratio. In other program runs (*db*) we noticed that OF achieved improvements that we could not explain entirely as a tradeoff between copying and pointer-maintenance costs, and that are most likely a consequence of improved cache performance. A detailed study of comparative cache behavior of these garbage collection algorithms in a Java virtual machine is called for but beyond the scope of the present paper.

4.2 Total Time and Mark/Cons Results for All Benchmarks

We now present total execution time data for all 10 benchmarks. For each benchmark, we summarize generational collector data into a single plot line, using the best possible nursery size for each heap size. Similarly, we present the best OF window size for each heap size. Automatically or adaptively choosing these region sizes is a very challenging problem that we do not explore here. Figures 8–10 present these two results together with the semi-space collector, and normalize them with respect to the total execution time of the Appel collector.

Performance results, namely total execution times, are surprisingly favorable to the Older-First collector. An earlier study based on fully accurate garbage collection traces and faithful simulation [14] found that the Older-First collector's mark/cons ratio was occasionally as low as one-tenth that of the fixed-generational collector using the best configuration of each. In such cases, the study predicted, according to an estimate of the write-barrier costs, that total *memory management* costs could be a factor of two to three

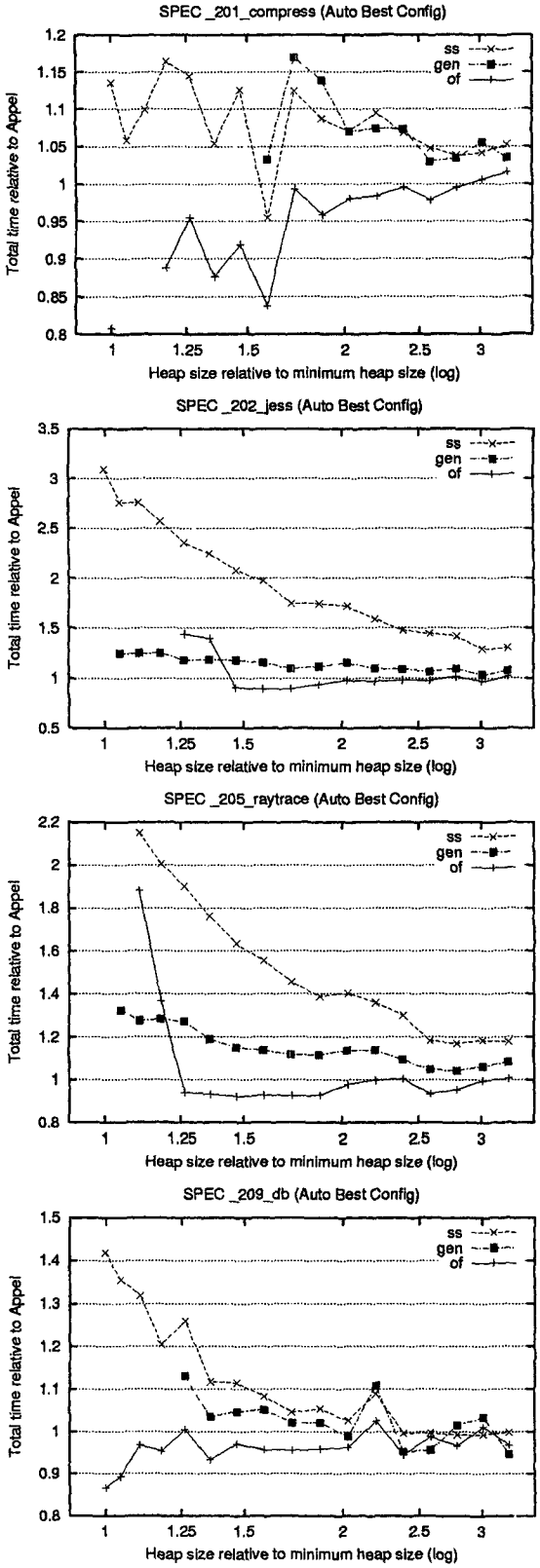


Figure 8: Total execution time, for best configuration.

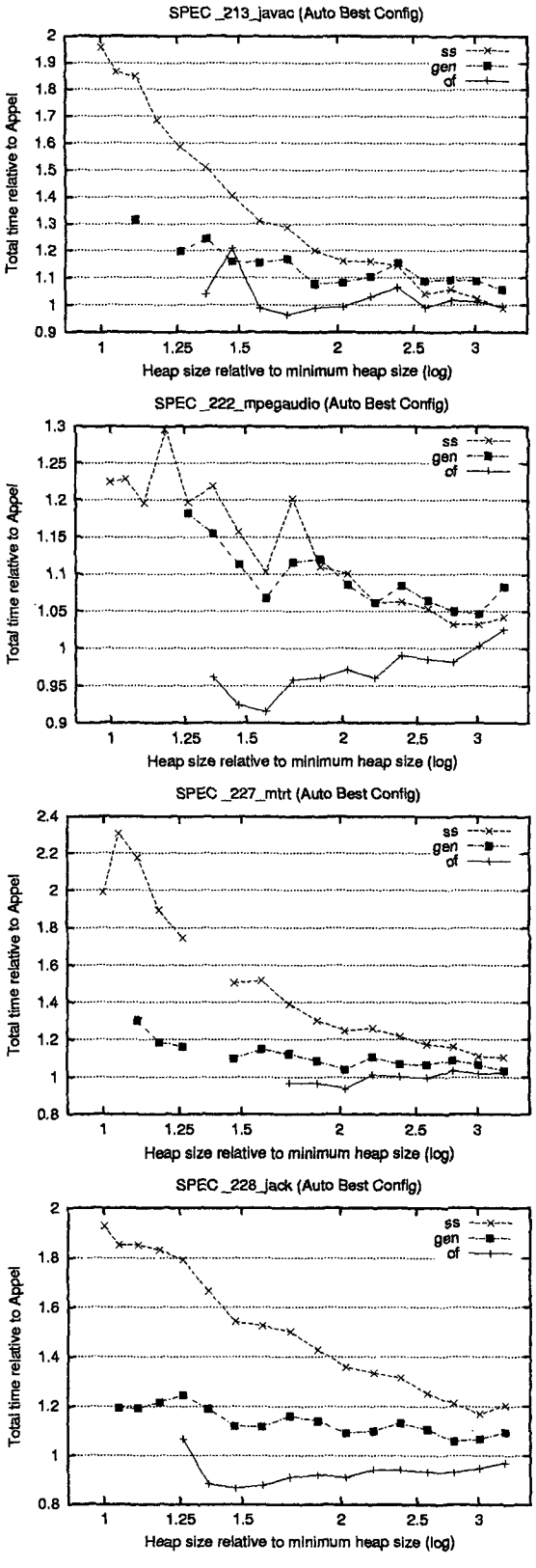


Figure 9: Total execution time, for best configuration (continued).

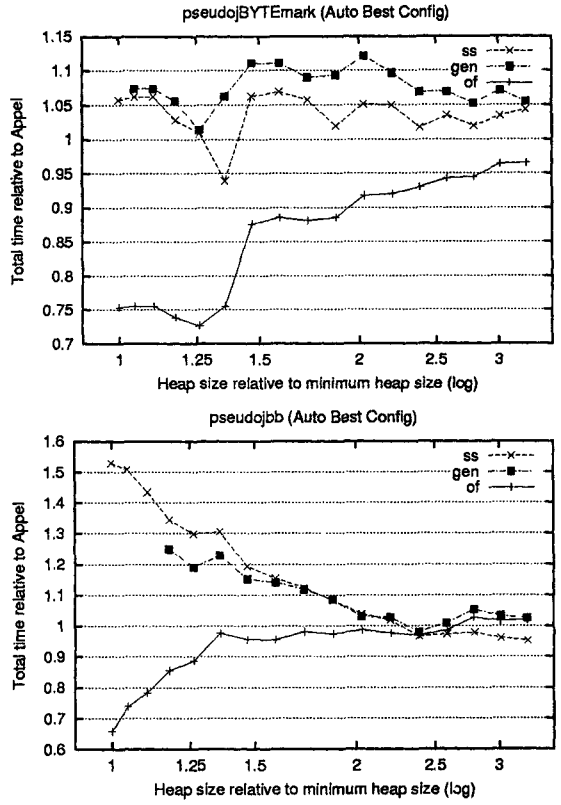


Figure 10: Total execution time, for best configuration (continued).

lower with OF. That study made no prediction about total execution times, but from these estimates and from Figure 7 we could extrapolate a reduction of between 2.5% (for larger heap sizes) and 20% (for small heap sizes). However, among the set of Smalltalk and Java program traces used in the earlier simulation study, only a few showed such dramatic reductions in the mark/cons ratio with OF; for many programs, it found no significant improvement of mark/cons ratios and estimated there could be only marginal improvement in total collection cost.

In the present live measurements, we observe more consistent reductions in the mark/cons ratio (Figures 11–13) with the Older-First collector compared to the generational collector, although not dramatically lower as in the earlier study. These improvements in the mark/cons ratio, however, translate into measurable reductions of total execution time (Figures 8–10). Of note is that on several benchmarks, the mark/cons ratio of Older-First is lower even than that of the Appel collector which is tuned to minimize copying. In 7 out of 10 benchmarks (*compress, jess, raytrace,db, jack, pseudojBYTEmark, pseudojbb*), the total execution time with the Older-First collector is ultimately lower than with the Appel collector for a wide range of heap sizes.

4.3 Measurements of Pause Times

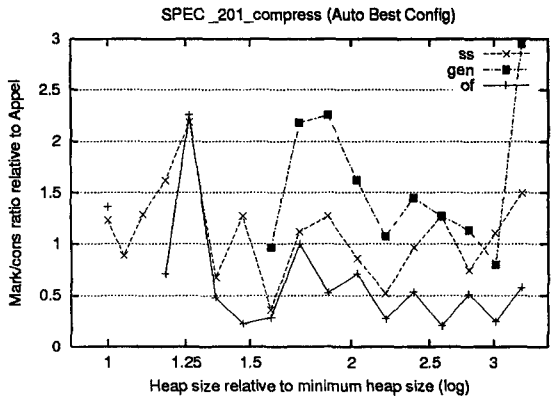
The design goal of OF is to improve throughput by reducing the mark/cons ratio, however OF achieves low mark/cons ratios with small window sizes. Since the amount of data copied at each collection is bounded by the window size, previous work predicted [14] that lower pause times would be an additional benefit when using OF as compared to the generational collectors which occasionally collect the entire heap. Similar reasoning applies to using a fixed-generational collector as compared with Appel, or a semi-space collector. We measure pause times for OF, fixed, and Appel in the Jikes RVM collector and we analyze them using the recently developed method of *mutator utilization* [4, 7].

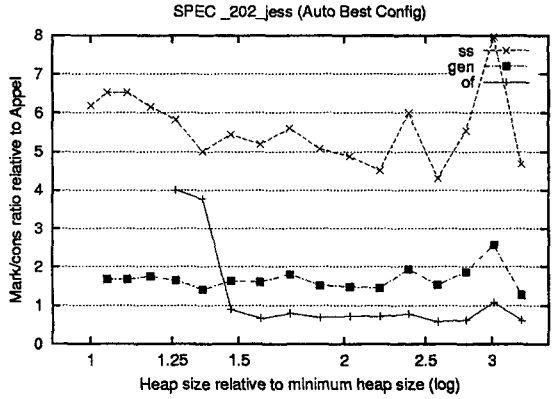
Reporting the duration of each garbage collection pause in a general timing run introduces a slight overhead. Whereas the time reported for each pause is accurate, over the execution of the entire program this reporting increases the total time. To avoid this problem, separate pause-timing runs were performed, whereas the timing runs described in the preceding section only reported the time once, at the end of execution.

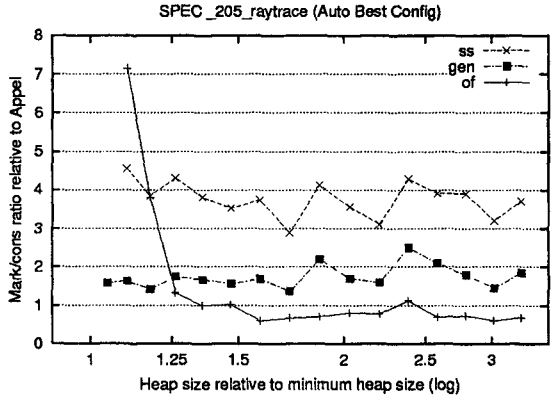
We first focus on maximum pause times and present an aggregate picture of *all* program runs (for all heap sizes and collector configurations) in Figure 14. Each scatter point corresponds to a single program run, and the marks distinguish the runs of generational, OF, Appel, and the semi-space collector. The horizontal coordinate gives the longest pause time incurred in the run, and the vertical gives the mutator utilization averaged over the entire run, i.e., the fraction of total execution time spent outside the garbage collector.

The semi-space collector points form a vertical band about 3 wide on the logarithmic scale, which is expected given that the span of heap sizes is 3.25. The shortest maximum pause times come from the generational collector when the heap is so large that it never performs a full heap collection. Most generational collector runs incur some long pauses when it collects the nursery together with the older generation. The Appel collector has some of the highest mutator utilization scores, but it too has high maximum pause times for major collections. The Older-First collector points are clustered in the favorable region of high mutator utilization and low maximum pause times, but there are also a number of runs with very long pause times.

A survey of maximum pause times does not capture the pause behavior of a collector completely. For interactive (or real-time) use







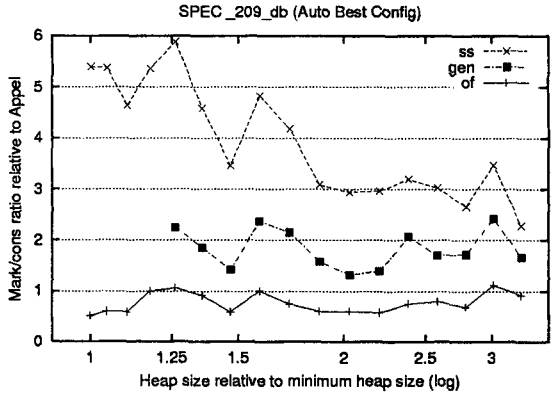
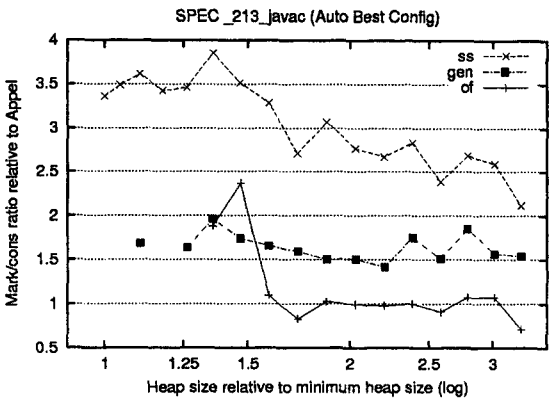
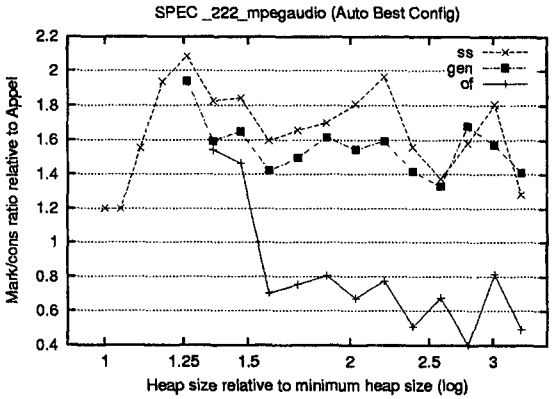
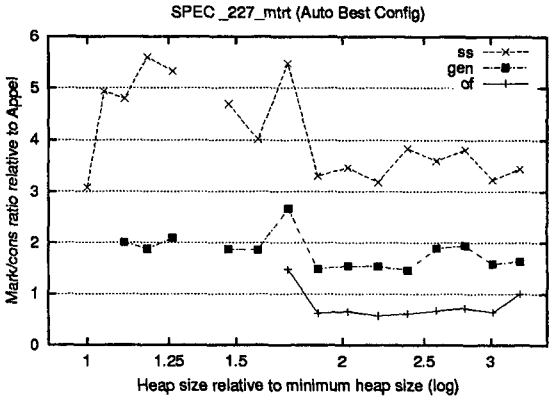


Figure 11: Mark/cons ratio, for best configuration.







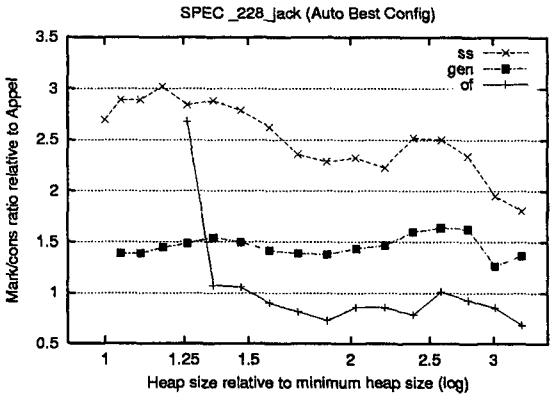


Figure 12: Mark/cons ratio, for best configuration (continued).

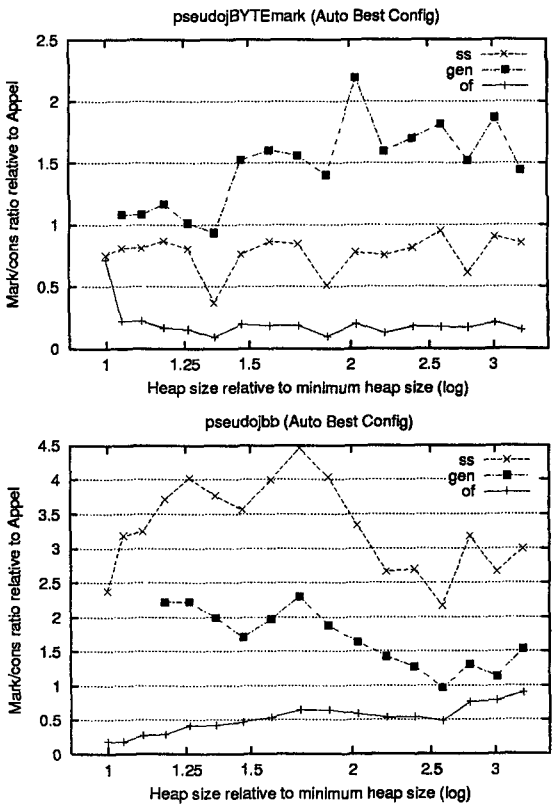


Figure 13: Mark/cons ratio, for best configuration (continued).

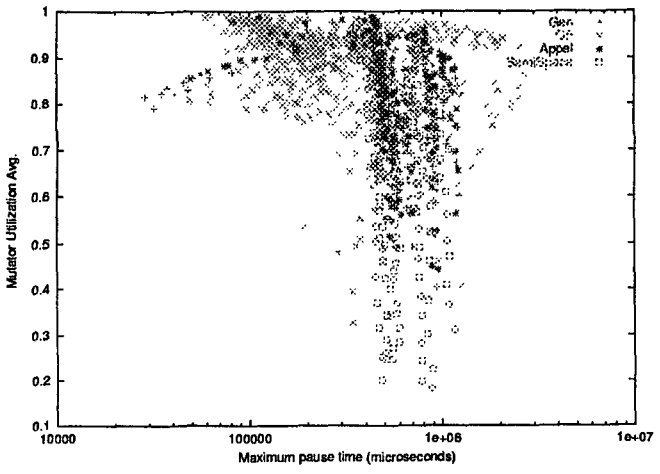


Figure 14: Mutator utilization vs. maximum pause time, all program runs.

it is important for the mutator to be able to make progress within any given period, and therefore it is important that garbage collection pauses do not occur in clusters. To quantify this progress, we examine all time intervals within a program execution. We say that *minimum mutator utilization* or MMU for interval length w is m if for all intervals of length equal to or greater than w, the mutator utilization in the interval is at least m.[2]

MMU plots are shown in Figures 15–17 for all 10 benchmarks, at the same relative heap size for each, twice the minimum heap size. For the fixed-generational collector, we show a representative well-performing nursery size of 15%, and a window of the same size for OF.

For six of the 10 benchmarks, *compress*, *jess*, and *raytrace*, *mpegaudio*, *pseudojBYTEmark*, and *pseudojbb*, the Older-First collector achieves both a higher average mutator utilization (y-intercept) and a lower maximum pause (x-intercept) than the Appel collector, and its MMU curve is everywhere above the Appel collector curve. For *pseudojbb*, there is a fivefold reduction in maximum pause time. In addition, for *mtrt*, the Older-First collector has higher mutator utilization and a lower maximum pause, but for a mid-range of pauses the Appel collector has greater MMU. For *javac* there is little difference among collectors with respect to maximum pause time, and the Older-First collector comes close to Appel with respect to average mutator utilization; however, its MMU curve in between the extremes is markedly lower than the Appel collector curve. For *db*, the Appel collector is the best both in terms of throughput and responsiveness. Finally, *jack* is an aberration with the semi-space collector having the smallest maximum pause time; we must investigate this further.

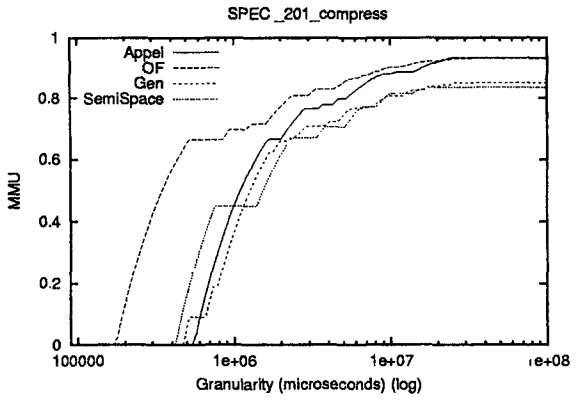
Note that none of the collectors we discuss provide any kind of real-time guarantee. Therefore, these results are only indicative of actual behavior, insofar as the benchmarks are representative of true workloads.

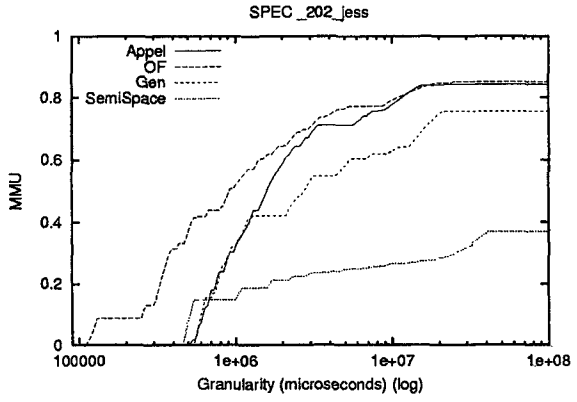
5. SUMMARY

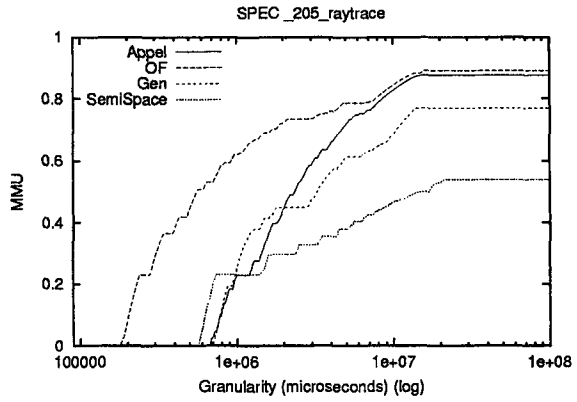
We present a first implementation of the Older-First collector, inside a Java virtual machine. We evaluate it against its natural competitor, the fixed-size nursery generational collector, as well as the Appel variable-size nursery generational collector. In the domain of throughput metrics, we find that the Older-First collector yields lower mark/cons ratios than the fixed-size nursery generational collector and is also lower than the Appel collector for a range of important, relatively small, heap sizes, across the SPECjvm benchmark suite. Moreover, this result is true of total program execution times, though the improvement over the Appel collector is never more than 30%. In the domain of pause-time metrics, we found that for many benchmarks, though not all, the Older-First collector achieves significantly lower maximum pause times than generational collectors.

We believe better implementations of OF are possible. For instance, profile-driven pretenuring provides immediate improvements to this basic collector organization [6]. We hope eventually to build OF in a 64-bit environment, in which OF will have the same fast write barrier as the generational collectors. The question of adaptive tuning of window size and other heap configurations remains open, as well as generalizations of the Older-First window motion policy.

[2]This formulation is exactly as in [4]. It is slightly different from [7] in that MMU curves are necessarily monotone increasing.







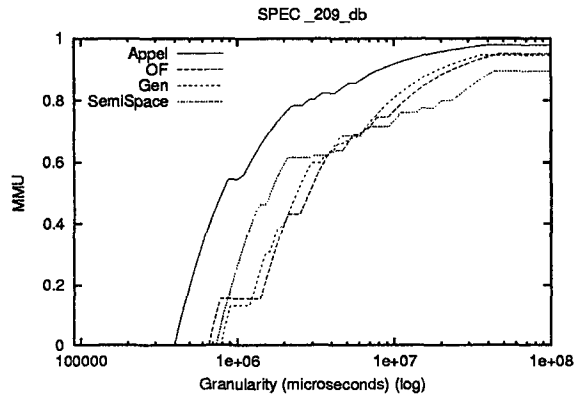
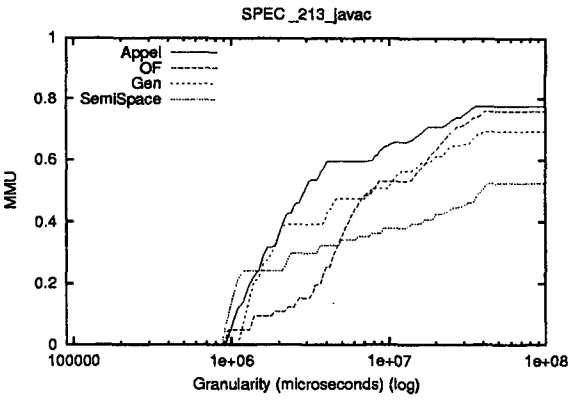
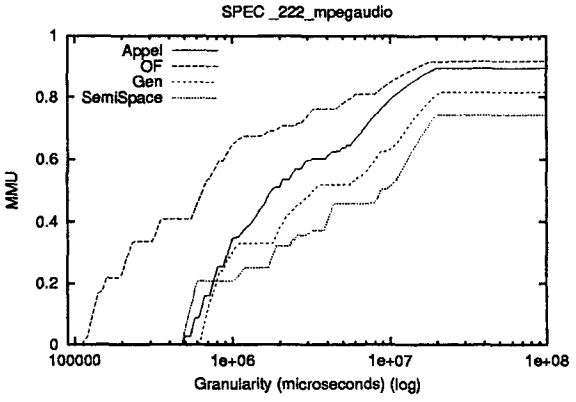
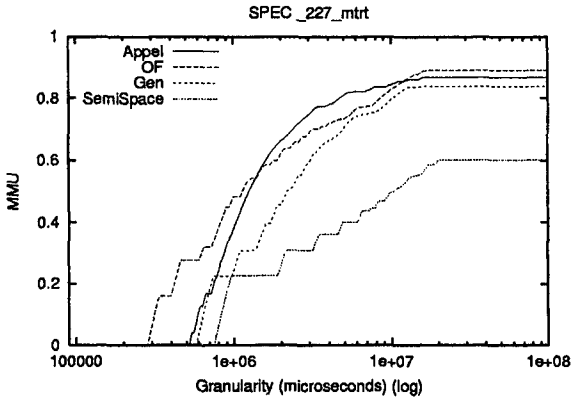


Figure 15: MMU.







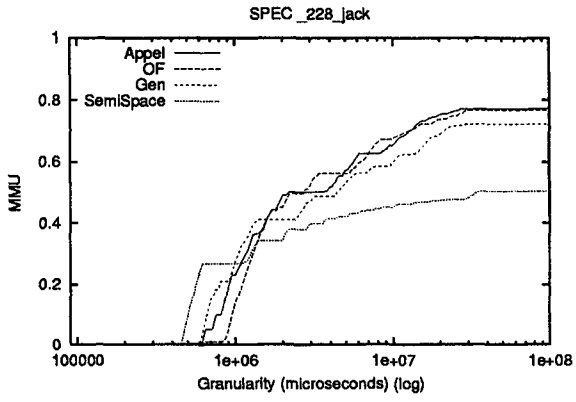


Figure 16: MMU (continued).

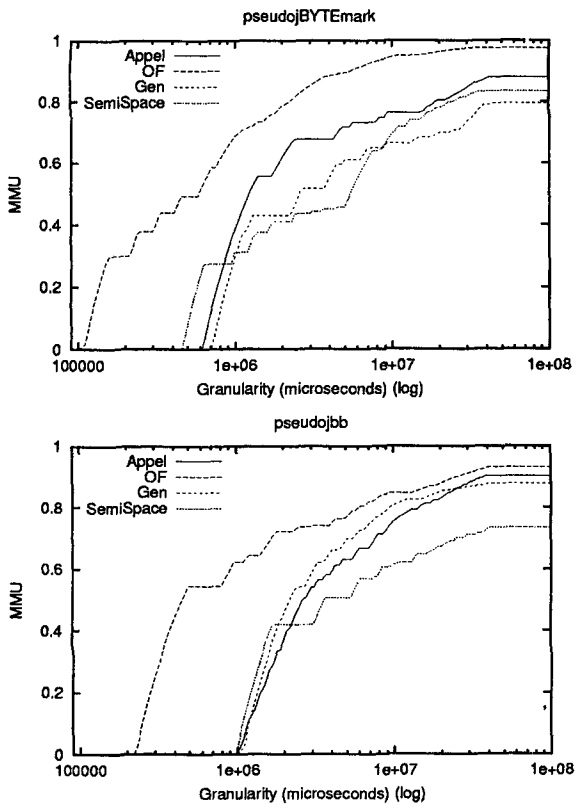


Figure 17: MMU (continued).

6. ACKNOWLEDGEMENTS

We thank Ben Andrews for help analyzing assembly code sequences, and the anonymous reviewers for their comments that helped us strengthen the paper.

7. REFERENCES

[1] ALPERN, B., ATTANASIO, C. R., COCCHI, A., LIEBER, D., SMITH, S., NGO, T., BARTON, J. J., HUMMEL, S. F., SHEPERD, J. C., AND MERGEN, M. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999* (October 1999), vol. 34(10) of *ACM SIGPLAN Notices*, ACM Press, pp. 314–324.

[2] ALPERN, B., ATTANASIO, D., BARTON, J., BURKE, M., CHENG, P., CHOI, J., COCCHI, A., FINK, S., GROVE, D., HIND, M., HUMMEL, S., LIEBER, D., LITVINOV, V., NGO, T., MERGEN, M., SARKAR, V., SERRANO, M., SHEPHERD, J., SMITH, S., SREEDHAR, V., SRINIVASAN, H., AND WHALEY, J. The Jalapeño virtual machine. *IBM Systems Journal 39(1)* (Feb. 2000).

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

[4] BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. E. B. Beltway: Getting around garbage collection gridlock. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2002), ACM SIGPLAN Notices, ACM Press. to appear.

[5] BLACKBURN, S. M., AND MCKINLEY, K. S. In or out? Putting write barriers in their place. In *Proceedings of the Third International Symposium on Memory Management, ISMM '02* (Berlin, Germany, June 2002), vol. 37 of *ACM SIGPLAN Notices*, ACM Press.

[6] BLACKBURN, S. M., SINGHAI, S., HERTZ, M., MCKINLEY, K. S., AND MOSS, J. E. B. Pretenuring for Java. In *Proceedings of SIGPLAN 2001 Conference on Object-Oriented Programming, Languages, & Applications* (Tampa, FL, Oct. 2001), vol. 36(10) of *ACM SIGPLAN Notices*, ACM Press, pp. 342–352.

[7] CHENG, P., AND BLELLOCH, G. A parallel, real-time garbage collector. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001* (Snowbird, UT, May 2001), vol. 36(5) of *ACM SIGPLAN Notices*, ACM Press, pp. 125–136.

[8] DIECKMAN, S., AND HÖLZLE, U. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings* (1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 92–115.

[9] LIEBERMAN, H., AND HEWITT, C. E. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26(6)* (1983), 419–429. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[10] MAY, C., SILHA, E., SIMPSON, R., AND WARREN, H., Eds. *The PowerPC Archiecture: A Specification for a New Family of RISC Processors*, second ed. Morgan Kaufmann, San Francisco, California, 1994.

[11] STANDARD PERFORMANCE EVALUATION CORPORATION. *SPECjvm98 Documentation*, release 1.03 ed., March 1999.

[12] STANDARD PERFORMANCE EVALUATION CORPORATION. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 ed., 2001.

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

[14] STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. Age-based garbage collection. In *Proceedings of SIGPLAN 1999 Conference on Object-Oriented Programming, Languages, & Applications* (Denver, CO, Oct. 1999), vol. 34(10) of *ACM SIGPLAN Notices*, ACM Press, pp. 379–381.

[15] 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.