An Adaptive Tenuring Policy for Generation Scavengers

DAVID UNGAR
Sun Microsystems
and
FRANK JACKSON
ParcPlace Systems

One of the more promising automatic storage reclamation techniques, generation scavenging, suffers poor performance if many objects live for a fairly long time and then die. We have investigated the severity of this problem by simulating a two-generation scavenger using traces taken from actual 4-h sessions There was a wide variation in the sample runs, with garbage-collection overhead ranging from insignificant, during three of the runs, to severe, during a single run. All runs demonstrated that performance could be improved with two techniques: segregating large bitmaps and strings, and adapting the scavenger's tenuring policy according to demographic feedback. We therefore incorporated these ideas into a commercial Smalltalk implementation. These two improvements deserve consideration for any storage reclamation strategy that utilizes a generation scavenger

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments—*interactive*; D.3.2 [**Programming Languages**]: Language Classifications—object-oriented languages, *Self, Smalltalk*; D.3.4 [**Programming Languages**]: Processors—*run-time environments*; D.4.2 [**Operating Systems**]: Storage Management—*allocation/deallocation strategies*

General Terms: Algorithms, Languages, Measurement, Performance, Theory

INTRODUCTION

Automatic storage reclamation is an important component of many modern interactive programming environments. In such systems, the runtime environment discovers when data are no longer needed and automatically reclaims their memory space. Automatic storage reclamation eliminates two

A preliminary version of this paper appeared in the *Proceedings of the 1988 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)* (San Diego, Calif., 1988).
This work was supported by Xerox Corporation and ParcPlace Systems.
Authors' addresses: D. Ungar, Sun Microsystems, 2550 Garcia Ave. MS 29-112, Mountain View, CA 94043; F. Jackson, ParcPlace Systems, 1550 Plymouth Street, Mountain View, CA 94043

types of pernicious bugs: either storage is not reclaimed and slowly leaks away, or storage is reclaimed before its time, resulting in total chaos. Despite these compelling advantages, several problems have forestalled the universal acceptance of automatic storage reclamation:

(1) distracting pauses while reclaiming garbage,
(2) failure to reclaim some types of data structures,
(3) high temporal and spatial overhead.

These problems can become even more apparent in interactive environments, given their more stringent response-time requirements.

One of the more promising approaches to automatic storage reclamation, generation-based storage reclamation, exploits an empirical property: young objects are likely to die and old ones are likely to continue to live. Generation-based systems attempt to exploit this property by segregating objects into separate generations according to the objects' ages and then concentrating the system's reclamation efforts on the younger generations. Assuming that young objects do, in fact, exhibit a higher death rate, such an approach should reclaim more garbage for a given amount of work than a system that distributes its reclamation efforts more evenly across all objects, young and old. In practice, this approach is so effective that certain implementations of one such technique, generation scavenging [23, 24], have succeeded in reducing pause times for reclamation to a fraction of a second at the seemingly modest cost of only 3 percent of the CPU cycles and 200 kb of main memory. The success of this and other generational automatic storage reclamation algorithms has led to their adoption in several commercial products, including all four commercial U.S. Smalltalk systems: from Apple [2], Digitalk [5], Tektronix [7], and ParcPlace Systems [25].

But there is a problem: if young objects do not exhibit a high death rate, generational algorithms fail to efficiently reclaim storage. For example, the original generation scavenging algorithm proposed by Ungar [23, 24] utilized only two generations, a young generation and an old generation. Objects are created in the young generation, and if they survive long enough they are eventually promoted to the old generation. We call this act of promoting an object to the oldest generation *tenuring*. Once ensconced in the old generation, these objects are no longer subject to reclamation via the scavenging mechanism; that is, they can only be reclaimed by invoking a secondary reclamation system, typically a global mark-and-sweep garbage collector. Thus, if an object lives long enough to attain tenure but then dies, its space goes unreclaimed until the next time the global garbage collector is run. We call such objects *tenured garbage*, and if there are too many of them, either much space will be wasted or a lot of time will be spent in global garbage collection.[1]

[1] While this paper was undergoing the review process, we undertook the construction of an *incremental* global garbage collector to further reduce the disruption caused by tenured garbage

Others have typically addressed these problems by forcing an object to pass through multiple generations (each of which is subject to periodic reclamation) before receiving tenure [7], by utilizing an incremental garbage collector to reclaim such objects [1], or by employing multiple generations that are reclaimed incrementally [16, 18]. In this paper, however, we investigate ways of mitigating this problem within the architectural confines of a simple, two-generation, nonincremental scavenger.

To this end, we conducted an experiment that recorded object birth and death times on a production system over hours of use by real users. From these data we first derived actuarial information on object lifetimes with a simple static analysis. The data then served as input for a simulation of generation scavenging. This dynamic analysis provided some insight into the time-varying demographics of objects. The results of the simulations led us to investigate the relationship between tenuring policies, wasted space, and pause times. This experiment helped shed some light on the design and feasibility of generation scavengers. We have incorporated the tenuring policies proposed in this paper into an experimental implementation of SELF [3, 15, 26] and into a commercial product, ParcPlace Systems' Objectworks®[2] \ Smalltalk (Versions 2.4 and later).

PREVIOUS WORK

Lifetime-Independent Automatic Storage Reclamation Algorithms

In 1960, Collins published a paper describing an automatic storage reclamation algorithm based on *counting references* [4]. Simplicity and short pause times helped this algorithm gain acceptance, and it was adopted for many systems, including the Dorado Smalltalk-80® implementation [8] and LOOM [14, 20, 21]. Deutsch and Bobrow halved the time cost, resulting in about 10 percent overhead [9, 10]. However, reference-counting automatic storage reclamation algorithms suffer from the inability to reclaim circular structures of objects [22], unless the algorithms incur the expense of additional recursive scanning [6]. Since circularity is a common characteristic of many data structures, this algorithm requires programmers to expend effort breaking circularities. This is the same sort of explicit freeing that automatic storage reclamation was supposed to eliminate in the first place.

Another automatic storage reclamation algorithm, *mark-and-sweep*, was also published in 1960 [17]. Unlike reference counting, which maintained a local approximation of whether an object was reachable, this algorithm relied on a global traversal of all live objects to determine which ones were eligible for reclamation. Reliance on a global traversal made it possible to reclaim all unreachable data, but introduced long pauses. The pauses got longer when memory sizes grew faster than processing speeds. By the early 1980s, some Lisp programs were spending 25 percent to 40 percent of their time marking and sweeping [11], and users were waiting an average of 4.5 s every 79 s [12].

[2] ® Objectworks is a registered trademark of ParcPlace Systems, Inc.
® Smalltalk-80 is a registered trademark of ParcPlace Systems, Inc.

Global, noninterruptible, mark-and-sweep storage reclamation does not seem to be practical on contemporary (1991) computers, especially for highly interactive tasks.

Baker found a way to smooth out the long pauses caused by the mark-and-sweep at the cost of memory space [1]. His algorithm divided up memory into two equal areas, or semispaces, and copied live objects from one to the other, a few at a time. In addition to its spatial overhead, the Baker algorithm incurred significant temporal overhead because of the need to copy every live object back and forth and the extra work required for programs to follow forwarding pointers. These defects have been remedied in the generation-based, stop-and-copy algorithms.

Generation-Based Automatic Storage Reclamation Algorithms

In order to build more effective automatic storage reclamation algorithms, language implementors have chosen to leave the realm of theory and exploit an empirical property of data: infant mortality. Generation scavenging takes an extreme position; it divides objects into two generations and expends all of its effort on the younger generation. Other reclamation algorithms are less extreme; they segregate objects into multiple generations and devote a portion of their time to reclaiming older generations as well as younger ones.

The rationale for having multiple generations goes roughly as follows:

(1) Having multiple generations permits the implementor to tailor the size of a given generation and its tenuring policy so that the objects that reside in that generation are handled with greater efficiency. In particular, objects that are in the youngest (and most frequently collected) generation can be promoted quickly without increasing the amount of permanent garbage that is created. Promoting new objects quickly keeps the average size of the youngest generation fairly small, thus reducing the pauses caused by collecting the youngest generation.

(2) Since the older generations are collected less often, objects will be held in these generations longer than they would be held in a system that had only one such generation, while still maintaining the same overall throughput.

(3) Forcing an object to pass through several generations before being granted tenure increases the likelihood that the garbage objects will expire in a generation where they can still be reclaimed quickly and efficiently.

These advantages are so compelling that Caudill [7], Lieberman [16], and Moon [18] all proposed or adopted reclamation algorithms that utilize multiple generations.[3] However, all three seem to employ fixed tenuring policies, rather than an adaptive policy such as the feedback-mediated policy proposed in this paper. Caudill [7] added a separate area for large objects, similar to

[3] Since this paper was submitted, more research in this area has been published by Shaw [19], Wilson [27], and Zorn [28]

the one that we report on in this paper, but did not publish a performance analysis. He did, however, claim that "large objects exhibited a tendency to have a longer life," but our data exhibited no such correlation.

METHODOLOGY

In order to understand the behavior of long-lived objects, we had to record when these objects lived and died. Long-lived objects are typically created in response to human interactions such as creating windows. So the experiment had to take place on a system that people were already using, as they were using it, without changing the way the system would be used. If the instrumented system ran too slowly, users might alter the way they used it and thus alter the data. The system we instrumented, ParcPlace Systems Smalltalk-80 Programming System Version 2.2, delivered good performance —at the time the fastest for Smalltalk on a MC680X0—and we were able to instrument it without too much overhead: only 11 percent. The instrumented version enjoyed a performance rating of 106 percent on a Sun 3/75 workstation[4] with 8 Mb of main memory, which was faster than many widely used Smalltalk systems.

There was another, equally important reason to use this particular Smalltalk implementation for our experiment: it is among the few high-performance systems that can accurately record an object's time of death. Birth is easy to detect, but many systems cannot efficiently detect death, because death occurs when the last pointer from a live object is destroyed. This rules out using any system that employs a mark-and-sweep garbage collector or a generation scavenger as its primary reclamation system, since these systems do not discover that an object is no longer referenced until some time after the actual dereferencing has occurred. In other words, to design a fast system based on scavenging, we had to instrument a fast system that was *not* based on scavenging.

On the other hand, systems that count references can detect deaths immediately. Unfortunately, the immediate detection of death exacts a price; most of the Smalltalk systems that employ simple reference-counting schemes have been too slow to support highly interactive applications. The system we instrumented achieved good performance by deferring reference-counting [9], which delayed the detection of deaths. Luckily, our system updated reference counts so frequently that it could still detect times of death to the nearest second. Reference counting systems also suffer from the inability to reclaim circular structures. This is not a major problem for the Smalltalk-80 system, since it was developed on a reference-counting virtual machine [13] and consequently takes care to explicitly break cycles when objects are no longer needed. Thus the fortuitous circumstance that made this work possible was

[4] This workstation uses an MC68020 processor running at 16 MHz. A performance rating of 106 percent means that the system ran the Smalltalk macrobenchmarks 6 percent faster than the canonical Smalltalk implementation on a Xerox Dorado.

the availability of a fast system with an automatic storage reclamation algorithm that rapidly reclaimed dead objects.

The Measurement Data

For this work, we gathered information on object births and deaths by writing time-stamped records to a file when an object was created or when its death was detected. In addition to the timestamp, each record included the object's size, its address, and a bit specifying whether or not the object contained pointers to other objects (see Figure 1). This last bit is important because objects without pointers can be scavenged faster since they need not be searched for references to additional objects.

Recording the birth and death of every object would have taken too much time and space. Instead, we truncated times down to an integral number of seconds and ignored every object that died in the same second it was born.[5] This corresponds to recording only those objects that would survive at least one scavenge in a system that scavenges exactly once per second. Since most objects die very quickly, this reduced the instrumentation overhead by an order of magnitude. How did this optimization change the resulting measurements? The generation scavenger described in [24] scavenged approximately once per second on this class of machine, and experience with Berkeley Smalltalk has shown that the interval between scavenges does not vary by too much. Thus, we believe that this optimization did not drastically affect our measured pause times or our measurements of tenured garbage.

The actual instrumentation was accomplished by storing birth times in a table containing an entry for each object. The Smalltalk virtual machine was modified to compare the current time against its birth time when freeing an object. If the two differed, it wrote a record for the object. In addition, more work was required to obtain data on objects that existed when the session started and objects that remained when the session was over.

(1) The object table was scanned at the beginning of each session and birth records were written out for all active objects, giving them a birth date that distinguished them as preexisting objects.
(2) The object table was also scanned at the end of each session and death records were written out for all active objects, placing distinguished values in their time-of-death slots indicating that the objects were still alive at the end of a session.

Nepotism

Our simulations must *underestimate* the amount of tenured garbage. In a real system, a new object may be kept alive by a reference from tenured garbage and eventually become tenured garbage itself (see Figure 2). In

[5] To minimize the system overhead, the clock was maintained by an interrupt-driven system call that only interrogated the operating system once a second

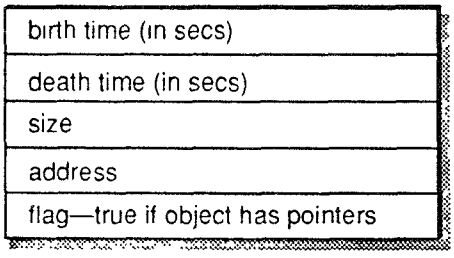


Fig. 1 Contents of records

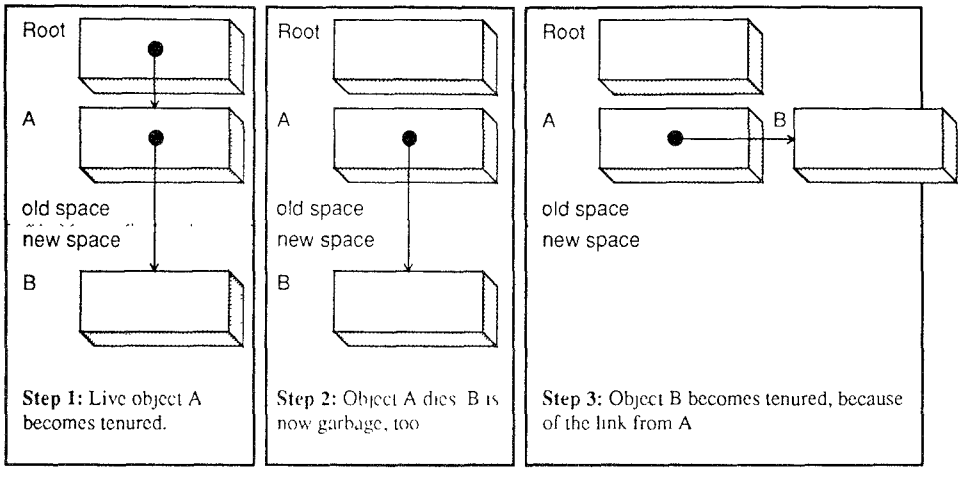


Fig. 2 Nepotism.

other words, an object which has been tenured, but is now dead, will eventually cause all of the young objects that it references to be tenured. These young objects will, in turn, cause the young objects that they reference to be tenured and so on. We describe this as *nepotism*. Our simulation could not take this effect into account. However, even if nepotism does contribute a large amount of tenured garbage, the relative merits of the tenuring policies discussed in this paper should still hold because neither of our optimizations would exacerbate nepotism.

The Sessions

We then collected data by having various users run the Smalltalk-80 system atop the instrumented virtual machine. Table I shows the samples we obtained.

This information allowed us to compile both static and dynamic results. Static results are based on histograms of object lifetimes and are simpler to derive. However, their predictive value is limited by the absence of dynamic information. Since work in other areas (e.g., virtual memory) has shown that

Table I. Sessions

Name	Duration (h:mm)	Interactive?	Activity
edit	6:40	Interactive	Active text editing
mail	6:53	Interactive	Longest of three runs, mostly reading mail
prog	4:36	Interactive	Writing Smalltalk programs
sim	5:15	Noninteractive	Running time-sharing and financial simulations

user activity is not stationary, but rather consists of phases, we simulated *dynamic* system behavior.

Four Kinds of Objects

The samples allowed us to group objects into the following classes (see Figure 3).

(1) *Transients* were those objects that came to life and then died within a given session. These objects could have become tenured garbage if they were promoted prematurely. For our purposes, transients were the most important class of objects.

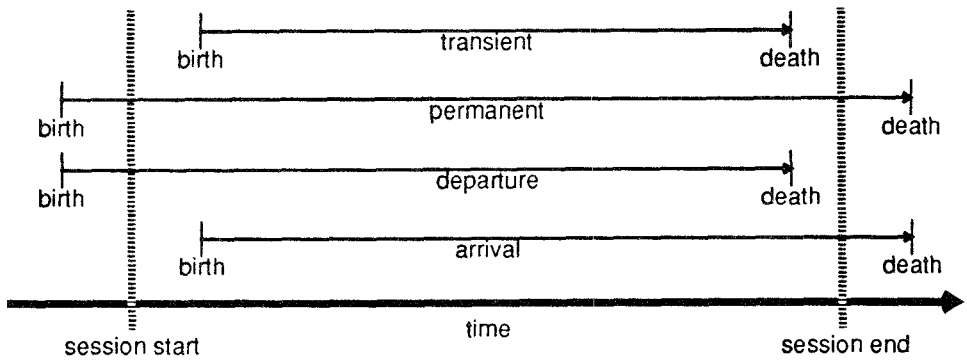
(2) *Permanent* objects were alive at both the beginning and the end of a session. The aggregate size of these objects should be quite large relative to the amount of tenured garbage present in the system; otherwise one runs the risk of performance problems caused by excess paging or, even worse, running short of address space. Since none of these objects died, we ignored them in this study of tenured garbage.

(3) *Departures* were those objects that were alive at the start of a session, but then died before the session was over. These objects could have also become tenured garbage, but they turned out to be inconsequential since there were so few of them—in all of our samples, the departures were less than 1 percent of the transients. Thus we ignored these objects as a source of tenured garbage.

(4) *Arrivals* were those objects that came to life, but did not die during that session. In a perfect system, these were the only objects that would be granted tenure. It turned out that, in our samples, these were also small in number. We therefore ignored arrivals as well.

Objects Great and Small

In Smalltalk, objects can contain either references to other objects or uninterpreted data such as the characters in a string or the bits in a bitmap. Large objects containing uninterpreted data are candidates for special treatment in a scavenging system: since they have no pointers, scanning their data during a scavenge is unnecessary; and since they are large, copying their data would be expensive. We therefore separated them out from other objects in our data (large objects are defined as containing at least 1024 bytes of uninterpreted



	died during session	died after session
born before session	departure	permanent
born during session	transient	arrival

Fig. 3. Classes of objects.

data). When it became clear that our system was using a significant number of these objects (primarily to cache the images of occluded windows), we decided to simulate a system that kept the data for these objects in a separate *large-object area*, scavenging only the headers of large objects. This dramatically improved performance, as described later in the paper.

On the other hand, we did not single out large pointer objects for special treatment in our simulations, since such treatment is not nearly so effective. That is, the scavenger still has to scan the data of any large pointer objects that it needs to preserve. Only the copying phase could possibly be omitted. For the purposes of our study, then, large pointer objects were treated the same as small pointer objects.

Counting Objects or Counting Bytes?

We also had to decide whether to measure the number of objects or the number of bytes consumed by the objects. We chose to count bytes, not objects, for three reasons:

(1) The time taken by published implementations of the generation scavenging algorithm have been shown to correlate well with the number of bytes scavenged [15, 24], and the per-object scavenge overhead is relatively low.

(2) The ill effects of tenured garbage, running out of address space and increasing the number of page faults, both depend on the amount of tenured garbage, not on the number of objects erroneously tenured.

(3) Finally, we constructed some scatter plots to see if there was any obvious relationship between an object's lifetime and its size. Such a relationship

would suggest that our results would be different for numbers of objects versus numbers of bytes. Surprisingly, we were unable to uncover any such relationship.

Accordingly, the results reported in the rest of the paper are in terms of the number of bytes consumed by the objects.

Six bytes of overhead were included for each object to account for the space taken by each object's header. The virtual machine we instrumented included an additional four bytes of per-object overhead for an object table entry. Each object-table entry contained a pointer to the object's data, which permits the virtual machine to relocate that data without updating any object references, since all object references pointed to the object-table entry rather than directly to the data. We excluded the object-table overhead from our calculation, because such overhead was clearly implementation specific, and because the technique of using an object table to provide a level of indirection for data references was not utilized by several existing implementations [7, 23, 24]. Nevertheless, it was important to include some form of overhead in our calculations, since all published Smalltalk implementations did have some per-object overhead. The exact amount of per-object overhead varied from implementation to implementation, of course, but the 6 bytes included in our calculations should yield a reasonable approximation.

Pause Time Statistics

Since we were interested in developing a reclamation system that does not disrupt interactive response, we needed to employ pause time metrics that measured maxima rather than central tendency. For example, neither the mean nor the median of a distribution convey any information about the shape of its tail, yet such distinctions are essential for the evaluation of pause times. We suspect that users of interactive systems might be willing to tolerate a few "glitches," but not too many. In other words, we believe that systems will be configured so that the vast majority of pauses are just small enough to avoid disruption. Therefore, we examined the 90th percentile pause times. In addition, we examined the maximum pause times in each session to estimate the severity of the worst glitches.

Pause Time Measurements

As tenuring policies become stricter, fewer objects receive tenure. Instead, these objects remain in the youngest generation, surviving from scavenge to scavenge and thus increasing pause time. So the limiting factor for a strict tenuring policy is the maximum acceptable pause time. Previous work [23, 24] has shown that pause time can be estimated by the number of bytes copied multiplied by the time to scavenge each byte. This multiplier was found to be 8 μs for Berkeley Smalltalk running on a 10-MHz MC68010 with a 400-ns instruction time, and 2.4 μs for SOAR running with a 400-ns cycle time. Since we were building a new Smalltalk-80 system (ParcPlace Systems' Objectworks\Smalltalk Version 2.4), we decided to instrument the new system to determine how long it takes to scavenge each byte. The measure-

Table II. Measured Pause Times for Generation Scavenging

Virtual machine	BS	SOAR ST	VM 2.4
Written in	C	Assembler	C
Workstation	Sun 2/100	(Simulator)	Sun 3/60
Processor	MC 68010	SOAR	MC 68020
Register add time	400 ns	400 ns	100 ns
Pause time	8 μs/byte	2.4 μs/byte	2 μs/byte

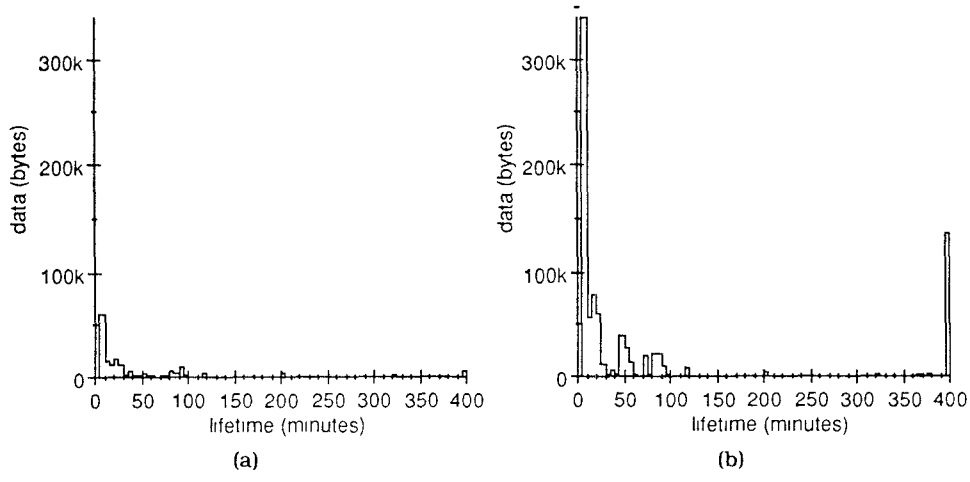


Fig 4. Lifetime density for "edit." (a) Small objects only. (b) All objects.

ments were performed on a 20-MHz MC68020-based SUN 3/60. A statistical analysis of the data lies beyond the scope of this paper. We can say that, at a 100-ms pause time, the system could scavenge between 33 and 83 kb. Therefore, in the rest of our analysis, we assumed a scavenge speed of 500,000 byte/s, or 2 μs/byte. Table II summarizes these results.

STATIC ANALYSIS

We first examined the core hypothesis of all generation-based storage reclamation algorithms: most objects die young. To do this, we wrote a static-analysis program which read the data collected for a given sample and classified the objects. In addition, this program produced two kinds of plots, lifetime densities and tenured garbage distributions.

Lifetime Densities

Figures 4 through 7 show the object-lifetime data for our four runs. In them, objects have been placed into buckets according to their lifetimes, and each bucket is 5 min wide. For example, all objects that lived at least 10 min and less than 15 would be in the same bucket. For the reasons noted above, we counted the total number of bytes of data in each bucket, instead of reporting

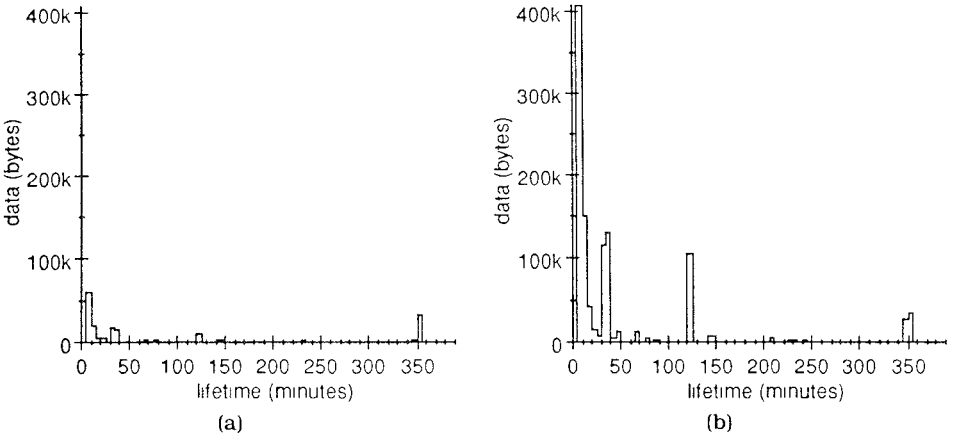


Fig 5. Lifetime density for "mail." (a) Small objects only. (b) All objects.

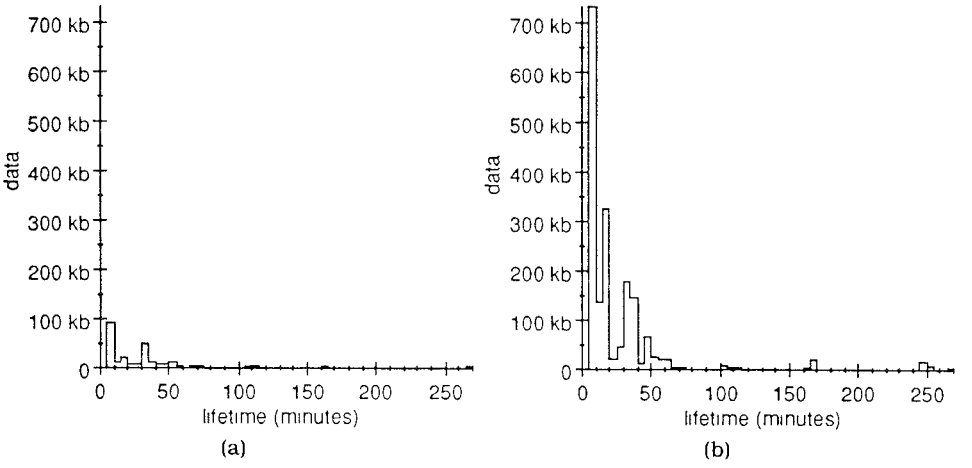


Fig 6. Lifetime density for "prog." (a) Small objects only. (b) All objects.

numbers of objects. Also, the first bucket had so many bytes in it that it would have thrown off the scale of the graphs, and we therefore omitted it. Finally, the distribution for small objects only is shown on the left, and the distribution for all objects is shown on the right.

These plots suggest some interesting observations about lifetimes:

(1) Most objects die young. On all of the plots, objects tail off around a lifetime of 40 min. This characteristic is important because it makes generation-based reclamation effective.

(2) The lifetime distributions do not fall off smoothly. Instead, each distribution contains idiosyncratic clumps of objects that live for more than one-half hour. These data suggest that, contrary to conventional wisdom,

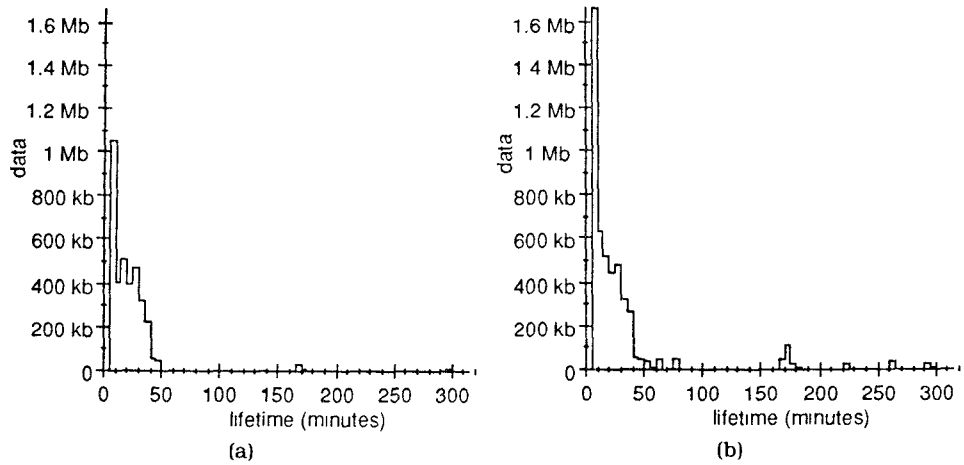


Fig. 7. Lifetime density for "sim." (a) Small objects only. (b) All objects.

systems must be designed to cope with irregular, time-varying distributions of object lifetimes.

(3) The data suggest a qualitative difference between the simulation run and the other runs. The "sim" lifetime density tails off to almost nothing, but the portion of the curve before the tail contains more objects which live longer than for any of the other runs. This difference may correspond to the fact that the tasks which made up the simulation run lasted longer than the tasks which made up the other runs, and hence the temporary data generated by the simulation tasks were more likely to be kept alive for extended periods of time. One might expect the effectiveness of a simple two-generation scavenger to be severely reduced for programs that hang onto large amounts of temporary data for extended periods of time. How common are such programs? The answer to this question requires further study with more diverse sample sets.

(4) Finally, there is another difference between the simulation run and the other runs relating to the size of the objects. The space in the other runs was mostly taken by *large* objects, whereas the space in the simulation run was mostly taken by *small* objects. This result probably reflects the higher use of bitmaps for window caching in the other runs—not surprising since each of these runs was heavily interactive. Since such large nonpointer objects are easily handled by a separate large-object area, this difference will be another reason why our reclamation algorithm will be more effective with the nonsimulation runs.

Tenured Garbage Distributions

The next set of plots, Figures 8(a)–(d), assume a *fixed-age* tenuring policy: every object that attained a certain age received tenure. The age at which an object would receive tenure is called the *tenuring threshold*. On these plots,

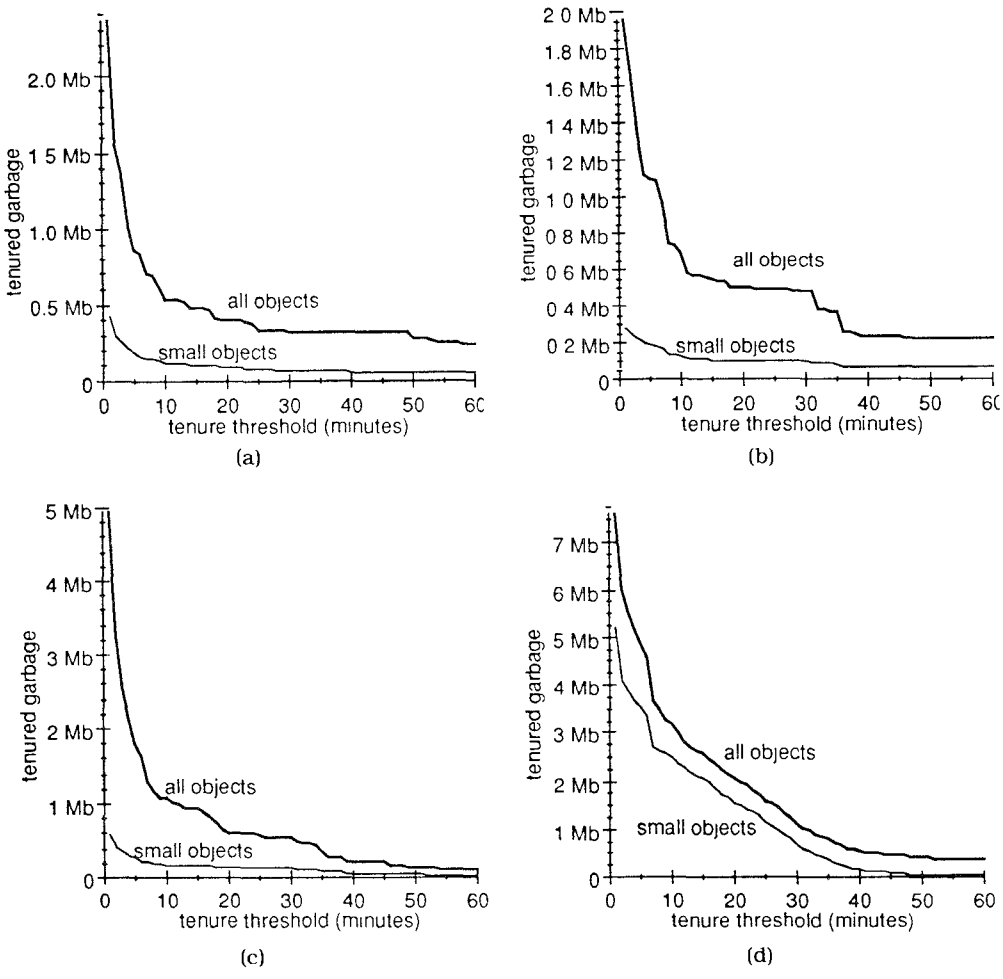


Fig. 8 Tenured garbage versus lifetime for various runs. (a) The edit run. (b) The mail run (c) The prog run. (d) The sim run.

the tenuring threshold is plotted against the amount of tenured garbage that would be created. For example, Figure 8(a) shows that about 500 kb of tenured garbage would result from a tenure threshold of 10 min. This means that there were 500 kb of transient data that survived for at least that long. The thin line gives the data just for small objects. Thus the thin line shows that only 100 kb out of the 500 kb of data living for over 10 min were the contents of small objects. In fact, for all three of the interactive runs these plots show that special treatment for large bitmaps and strings could drastically reduce the amount of tenured garbage.

The three interactive runs also share another characteristic: a knee at about 7 to 10 min. This may indicate an effective tenuring threshold for these runs. On the other hand, the simulation run (Figure 8(d) shows a deluge of

objects living up to 40 min. For example, suppose that over the course of a work session a total of 200 kb of tenured small-object garbage were deemed acceptable. (We show how to handle the large objects later.) This would have been attainable with a 5-min tenuring threshold for the edit run, a 4- to 5-min threshold for the mail run, and a 3-min threshold for the programming run, but would have required a 37-min threshold for the simulation run! This reinforces the suggestion that some programs need storage reclamation mechanisms above and beyond a simple two-generation scavenger.

DYNAMIC ANALYSIS OF TENURING POLICIES

Static analysis of object lifetime information can only hint at the amount of tenured garbage, because it neglects the following.

(1) *Pause-time constraints.* Since many scavenging algorithms, including generation scavenging, periodically halt execution to copy all live new objects, the amount of space taken by live new objects must be limited to avoid distracting pauses. This may result in tenuring extra objects.
(2) *Space limitations.* Since the amount of storage reserved for the youngest generation is finite, more objects may have to be tenured if it fills up.
(3) *Baby booms.* If many fairly long-lived objects are born at the same time, there will be a surge of live new objects. This will exacerbate extra tenuring caused by space and time constraints.

To understand these effects, we simulated a two-generation scavening system as described in Ungar [23], using as input the object-lifetime data that we had collected earlier. The results shed light on the relationship between tenured garbage and pause time and on how tenured garbage is created.

The Scavenging Simulator

Our scavenging simulator was driven by two files; one sorted by births and another sorted by deaths. In addition, a clock measured virtual time. This simulator worked by keeping track of the currently surviving new objects in a table (called "newSurvivorSet" in Figure 9). At each clock tick,

(1) All death events taking place at that time were processed. The dying objects were removed from the set of new survivors and tallied as either transients or departures.
(2) Next, births were processed by adding them to the set of new survivors.
(3) Last, the set of new survivors was scanned for objects to be tenured. These objects were removed from the set and tenured.

Statistics were gathered on survivor size, tenured live data, tenured garbage, births, and deaths. In order to compute the pause times, we kept the sum of survivor size, tenured garbage, and tenured live data for each second. This sum was the total number of bytes copied. So, to estimate how long a particular scavenge would take, we simply divided the number of bytes

```
read first birth record into birthRec
read first death record into deathRec
while . . .
        time = time + 1;

        while deathRec.deathTime <= time
                if newSurvivorSet include deathRec
                        remove deathRec from newSurvivorSet
                        survivorSize = survivorSize - deathRec.size
                else
                        tenuredGarbage = tenuredGarbage + deathRec.size
                read next death record into deathRec

        while birthRec.birthTime <= time
                add birthRec to newSurvivorSet
                survivorSize = survivorSize + birthRec.size
                read next birth record into birthRec

        for each survivorRec in newSurvivorSet
                if survivorRec age > tenureThreshold
                        remove survivorRec from newSurvivorSet
                        survivorSize = survivorSize - survivorRec.size
```

Fig 9 Summary of scavenging simulator.

copied during the scavenge by 500,000—the actual pause times vary according to processor speed.

Simulator Results Fixed-Age Tenuring

First, we ran the simulator with a fixed-age tenuring policy. In addition to this simplification, no special treatment was given to large bitmaps or strings. We ran the simulator many times, varying the tenure threshold from 1 to 15 min. From this, we computed tenured garbage, maximum pause time, and 90th percentile pause time as a function of the tenure threshold. Then we graphed the pause times against the tenured garbage, using the tenuring threshold as a parameter. The results are shown in Figures 10(a)–(d). Each graph has two traces: one showing the 90th percentile pause time as a function of tenured garbage and one showing the maximum pause time as a function of tenured garbage.

As expected, a low tenure threshold produced short pause times, as shown by a small survivor size, and more tenured garbage; the right-hand endpoint of each curve represents a threshold of 1 min. (Thereafter, each cross on the curve represents a 1-min increment.) Also as expected, increasing the threshold reduced tenured garbage at the expense of pause time; the left-hand endpoint represents a threshold of 15 min. Finally, there was much more

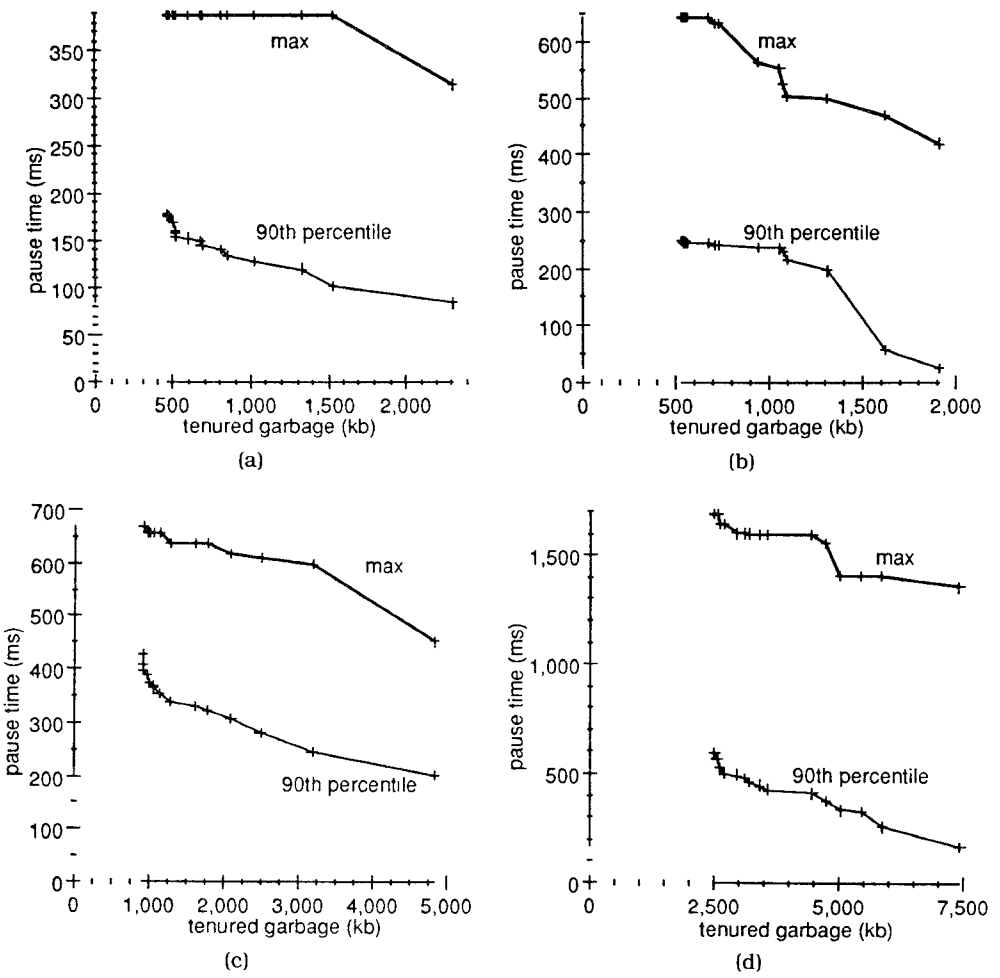


Fig 10 Pause time versus tenured garbage (fixed tenure threshold, without large-object area) for various runs. (a) The edit run. (b) The mail run. (c) The prog run. (d) The sim run.

data surviving in the simulation run than the other runs; this showed up as both more tenured garbage and longer pauses.

Segregating Large Bitmaps and Strings

Next, Figures 11(a)–(d) examine the benefits of special treatment for large bitmaps and strings. The simulator was changed to implement a *large-object area*. A large-object area kept the data of all large bitmaps and strings in a separate area, storing only their headers in the area housing the youngest generation. The headers were then scavenged (copied from one portion of the youngest generation to the other), but no time was spent copying the data in the large bitmaps and strings. Furthermore, these objects were never tenured

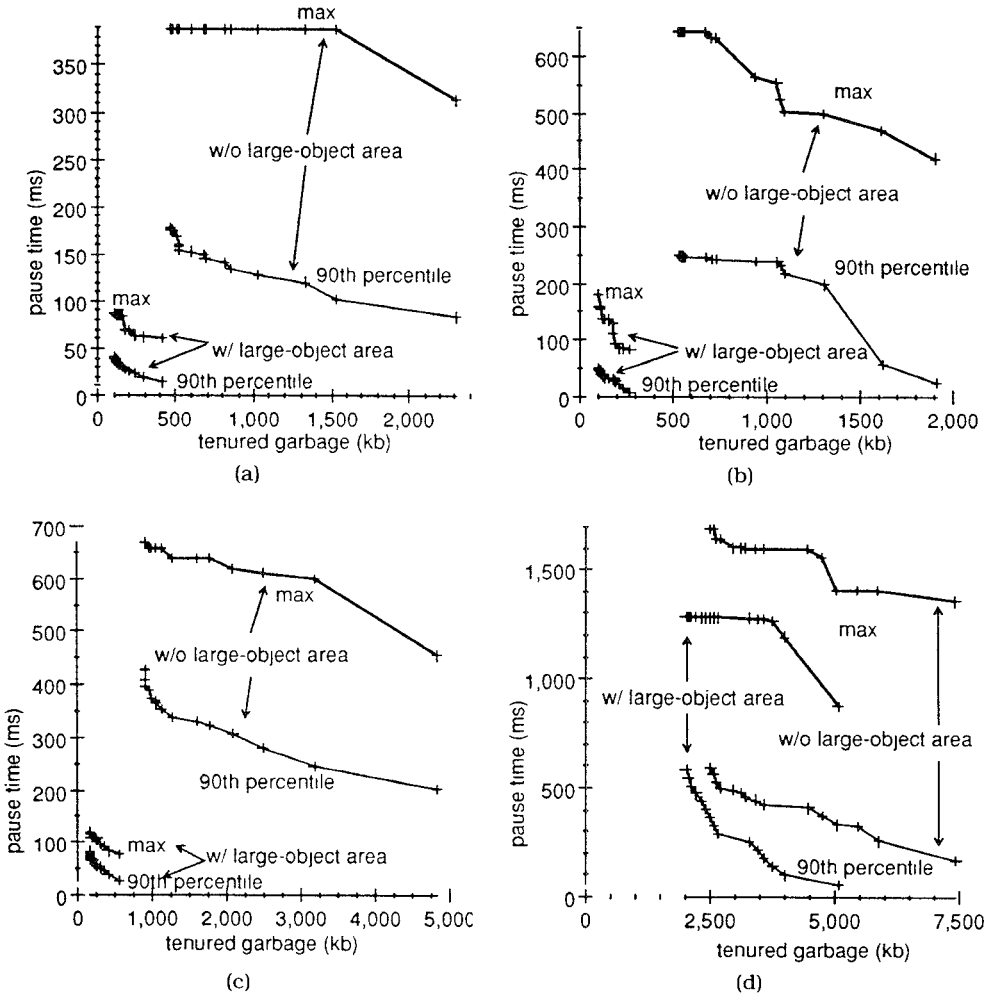


Fig. 11. Effect of large-object area (fixed tenure threshold) for various runs (a) The edit run (260 kb for large-object area). (b) The mail run (330 kb for large-object area). (c) The prog run (390 kb for large-object area) (d) The sim run (740 kb for large-object area)

(i.e., their headers were retained in the youngest generation until the object died), because the large-object area could be made big enough to hold the data of all large objects. So, for this run, we kept track of the maximum space taken by large bitmaps and strings (the capacity needed for the large object area) and excluded large bitmaps and strings from survivor size and tenured garbage calculations.

As in the previous graphs, the tenure threshold in each trace ranges from 1 to 15 min in 1-min increments from right to left. Once again, this low tenure threshold produced the most tenured garbage and the shortest pause times. Again, as expected, this high tenure threshold produced the least tenured

garbage and the longest pause times. Finally, as before, there was much more data surviving in the simulation run than the other three runs; this showed up as both more tenured garbage and longer pause times.

A separate area for large bitmaps and strings paid off, although the graphs look different for the first three runs when compared to the "sim" run. For example, in the "mail" run, dedicating 330 kb to the large-object area saved over a megabyte of tenured garbage and cut pause times by a factor of 4. On the other hand, in the sim run, dedicating 740 kb to the large-object area made a difference that did not look so dramatic, but the absolute reduction in tenured garbage, roughly 2 Mb, was still substantial. Since most current workstations have at least 4 Mb of main memory, the amount required for the large-object area seems reasonable. This idea was so effective that the remainder of our investigations assumed that large bitmaps and strings were segregated.

A Demographic Feedback-Mediated Tenuring Policy

The previous section suggested a mechanism for handling large bitmaps and strings. This section explores a better tenuring policy for the rest of the objects. A two-generation, fixed-age tenuring policy has three problems:

(1) *Occasional long pauses.* If there are many objects younger than the tenuring threshold, the pauses will be long.
(2) *Extravagant tenuring.* If there are very few objects being scavenged, there is no need to tenure any, but a fixed-age policy will tenure objects that are old enough anyway.
(3) *Random tenuring.* When the new area runs out of space in the midst of a scavenge, the remaining scavenged objects are tenured, regardless of age. (This problem does not occur for systems in which the youngest generation is organized as a pair of semispaces.)

None of these problems would be very severe if object demographics were stationary, but our traces show that this was not the case. Instead, long-lived objects seemed to be born in clumps, which diminished only slowly with time, like a pig that had been swallowed by a python.

We therefore devised a tenuring policy, illustrated in Figure 12, with two components:

(1) *Feedback mediation.* At the end of each scavenge, the system examined the amount of surviving data in the youngest generation. Recall that the pause time is proportional to the amount of data that has to be copied. Thus, the amount of surviving data provided an estimate of the danger that the next scavenge would exceed the maximum acceptable pause time. If, after a scavenge, the survivor size were small, the tenuring threshold would be set to infinity for the next scavenge; no objects would be tenured the next time. If, on the other hand, the survivor size were large, then the tenuring threshold would be set to a value designed to tenure the excess data on the next scavenge. How was this value chosen?

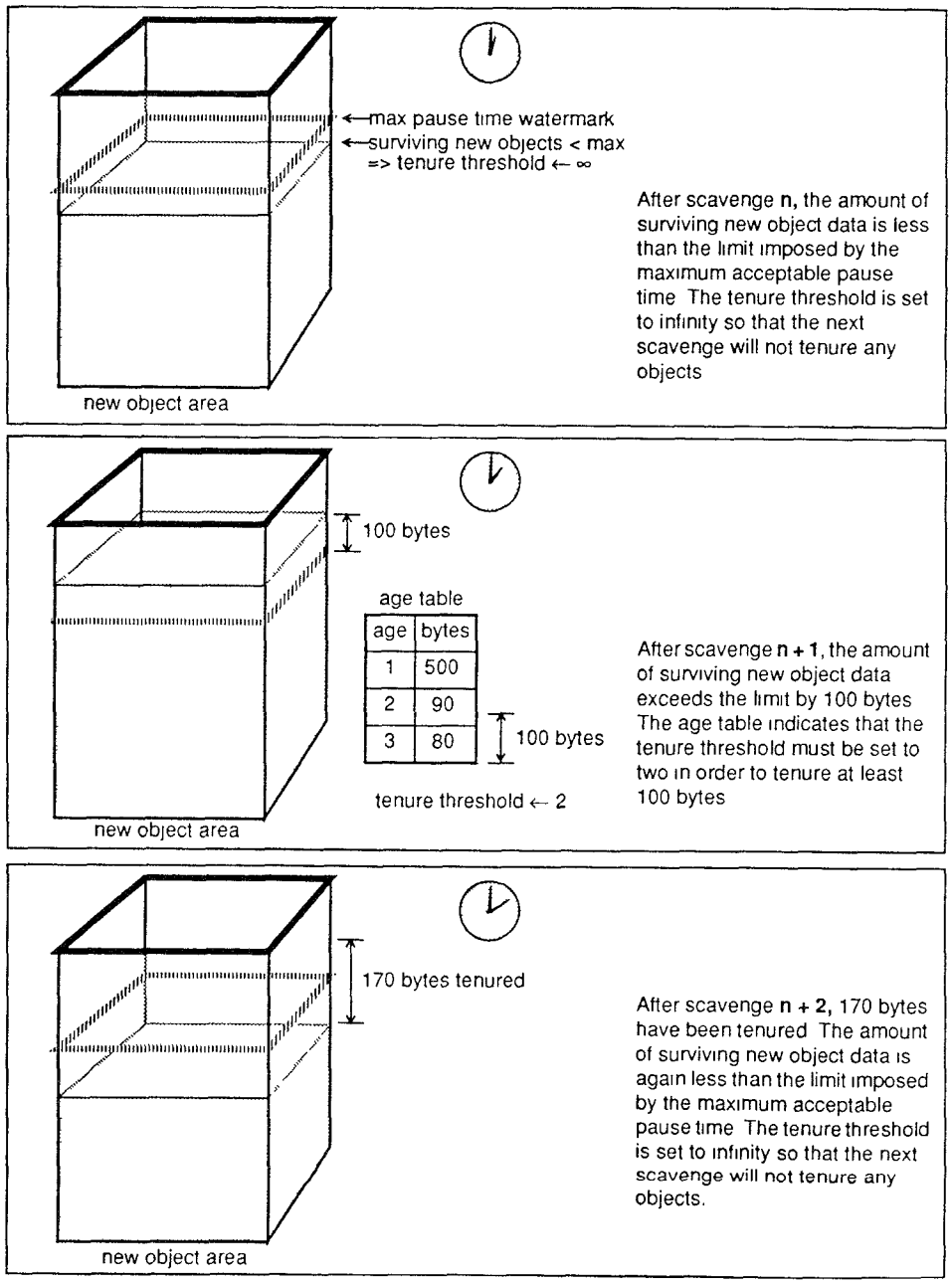


Fig. 12. Demographic feedback-mediated tenuring.

(2) *Demographic information.* After a scavenge, if the aggregate size of the surviving data exceeded the pause time threshold, a pass through the surviving young objects computed a table indexed by age containing the number of data bytes for each age. A backwards scan through this table yielded the appropriate setting for the tenuring threshold.

Both parts were important: the feedback mediation triggered tenuring only when needed, and the demographic information made it possible to tenure only as many objects as needed to be tenured, while using age as a tenuring criterion. Since age was the best predictor of lifetime, age was still the best determinant of which objects to tenure. The effect of this policy was to limit pauses when many objects survived and to eliminate tenuring when few objects survived.

To evaluate this policy, we modified our simulator and ran it for various pause time thresholds. Figures 13(a)–(d) show the results of demographic feedback-mediated tenuring. Each graph has four traces, all assuming a large-object area. Two traces show a fixed-age policy for comparison; as before, the crosses on these curves represent tenuring thresholds from 1 to 15 min in 1-min increments. The other two traces show the feedback-mediated policy; the crosses on these curves mark pause time thresholds from 20 ms to 200 ms in 20-ms increments.

In all sessions, the feedback-mediated algorithm succeeded in controlling pause times and narrowing the gap between the 90th percentile and the maximum pause times. This benefit was expected, but how did the overall performance of the new policy compare with the fixed-age tenuring policy? In the first three sessions ("edit," "mail," and "prog"), it was essentially the same for 90th percentile pause times and much better for maximum pause times. For example, in the mail session, for 150 kb of tenured garbage, the new policy reduced the maximum pause time from 140 ms to 66 ms. In the sim session, the results were different: the new algorithm substantially increased the 90th percentile pause times. For example, at 3.5 Mb of tenured garbage, it increased 90th percentile pause time from 240 ms to 400 ms. Even in this difficult session, the new algorithm improved the *maximum* pause times: at 3.5 Mb of tenured garbage it shrank the maximum pause time from 1.3 s to 440 ms. This decrease meant that the young object area could shrink proportionally, from 650 kb per semispace to 220 kb, without overflowing.

Feedback-mediated demographic tenuring offers more control over pause times and space needed for the survivors. The problem with a fixed-age tenuring policy is that the system implementor must set the tenure threshold in advance and hope that this fixed threshold avoids overflowing the portion of memory set aside for the young generation and yields good performance for the various programs that will be run on the system. Feedback, on the other hand, allows the implementor to set the parameter that the user sees, pause time, and forces the tenure threshold to adjust to different programs. Since the pauses are bounded, there is less risk of overflowing the youngest generation.

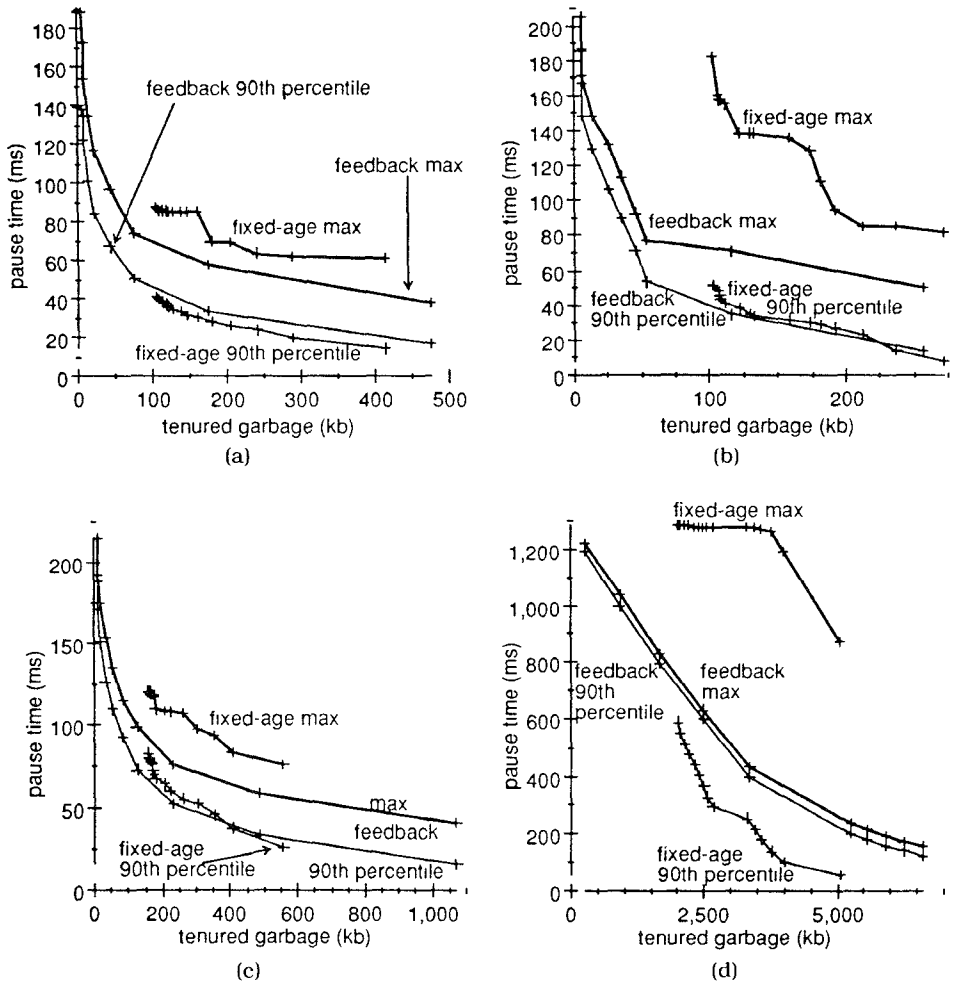


Fig. 13. Effect of demographic feedback-mediated tenuring for various runs. (a) The edit run (260 kb for large-object area) (b) The mail run (330 kb for large-object area). (c) The prog run (390 kb for large-object area). (d) The sim run (740 kb for large-object area).

To further illustrate the effect of demographic feedback, we combined the data from our runs to simulate a session composed of the various activities. For each tenure threshold we took the largest maximum pause time and the total tenured garbage. The first combined plot, Figure 14, combines the three interactive runs: edit, mail, and programming. The upper trace shows the results for a fixed-age tenuring policy and the lower trace shows the results for a demographic feedback-mediated policy. For example, over the course of this simulated 18-h run, a fixed-age threshold that limited pauses to 100 ms resulted in 700 kb of tenured garbage whereas a feedback-mediated threshold reduced this to 200 kb. Feedback saved half a megabyte of tenured garbage.

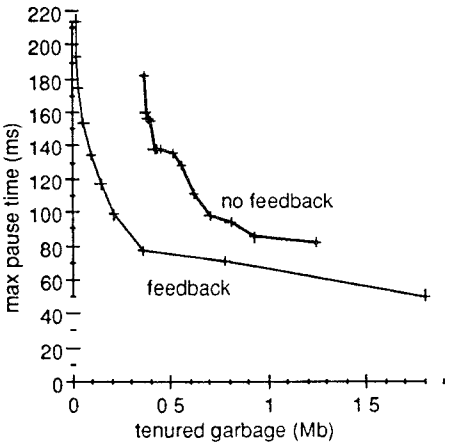


Fig 14. Totals for edit, mail, and prog runs (740 kb for large-object area).

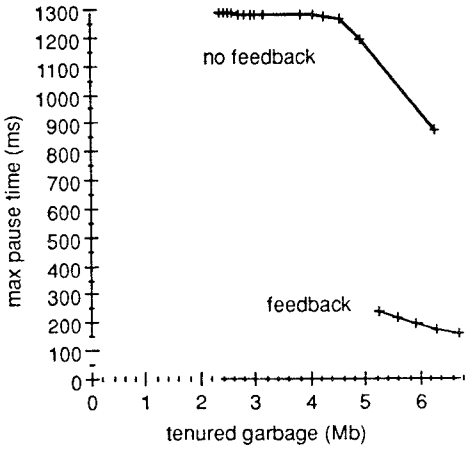


Fig. 15. Totals for edit, mail, prog, and sim runs (740 kb for large-object area).

The second plot, Figure 15, combines all of the runs: edit, mail, prog, and sim, for a total of 23 h. Since the simulation run generated so many intermediate-lifetime objects, it was very different from the other three runs. In fact, it was so different that we could only combine the feedback data at 6 data points instead of 10 (the range of thresholds we had simulated were overlapping but not identical). Nevertheless, one can see that feedback was essential for controlling pause times during this run. For example, a fixed-age tenuring policy that limited the total amount of tenured garbage to 5.2 Mb had a maximum pause time of over 1.1 s, whereas the feedback-mediated policy cut pauses to 240 ms. Feedback limited pauses by compensating for the differences between one activity and another.

ACTUAL IMPLEMENTATIONS

The tenuring policy proposed above has been implemented for the SELF programming language [15]. In addition, we incorporated both demo-

graphic feedback-mediated tenuring and a large-object area into a commercially available Smalltalk implementation—ParcPlace Systems' Objectworks\Smalltalk (Versions 2.4 and later) [25]. Although a statistical analysis of these two language implementations with respect to the above techniques is beyond the scope of this paper, the preliminary results seem encouraging.

SELF

Lee has analyzed the performance of a SELF implementation on a Sun 4/260 whose generation scavenger used a demographic feedback-mediated tenuring policy but did not include a large-object area. Lee measured a direct correlation between the amount of data scavenged and the pause time required to perform a scavenge, verifying one of the fundamental assumptions behind the demographic feedback-mediated tenuring policy. His system, which ran on a platform with a 60-ns register add time, took 2.2 μs to 2.8 μs to scavenge a byte compared with our system, which ran on a platform with a 100-ns register add time and took 2 μs to scavenge each byte. Unfortunately, Lee did not compare feedback to a fixed-age tenuring policy and did not measure long sessions of actual use; so for lifetime-dependent metrics, we cannot compare his results to ours.

ParcPlace Systems' Objectworks\Smalltalk

Demographic feedback-mediated tenuring has improved the performance of ParcPlace Systems' Objectworks\Smalltalk. For example, when the source code of a large application written in Smalltalk was compiled using an implementation whose generation scavenger employed a fixed-age tenuring policy (this particular implementation was a preproduction version of the 2.4 system), the compilation process produced approximately 2.7 Mb of tenured garbage. When this same application was compiled using a Smalltalk implementation that was identical in every respect except that its scavenger employed a demographic feedback-mediated tenuring policy, the compilation process produced 320 kb of tenured garbage—an improvement of almost an order of magnitude!

This difference was considerably greater than our simulations had led us to expect. The age threshold of the fixed-age scavenger had been set to 600 scavenges,[6] which had resulted in very little tenured garbage in prior tests. But further investigation revealed that most of the tenured objects had not reached the age threshold; instead they received tenure because their aggregate size far exceeded the capacity of the area used to house untenured objects (the total size of this area was 220 kb). In other words, the fixed-age scavenger was constantly forced to tenure objects because of overflow conditions. This overflow not only permitted objects that might otherwise have died after a scavenge or two to receive tenure, but, because of nepotism, caused all of their referents to eventually receive tenure as well. In this

[6] 600 scavenges corresponded roughly to 10 min of wall-clock time on a 16-MHz Sun 3 system

particular case, the feedback-mediated scavenger was also tenuring objects frequently, but it lowered its age threshold automatically as soon as the aggregate size of the surviving objects exceeded the amount that could be safely scavenged without incurring a disruptive pause (50 kb in this case). In doing so, the feedback-mediated scavenger was not only able to avoid chronic overflow conditions, but was able to preferentially tenure the older objects during each scavenge, resulting in significantly less tenured garbage.

For this workload (i.e., compiling this application), lowering the age threshold would probably have improved the performance of the fixed-age scavenger. It is unlikely, however, that any single age threshold would perform acceptably in all cases. In fact, this particular case demonstrates how a scavenger with a fixed-age threshold that has performed quite well on numerous other occasions can perform quite abysmally when its notion of the appropriate age threshold is suddenly at odds with the demographics of the objects produced by a particular computation. On the other hand, the feedback-mediated policy adapted quite well in this particular case.

Finally, we have gathered anecdotal evidence suggesting that a large-object area may have considerable practical value for certain applications. The developers of a file-system navigation tool written in Smalltalk-80 reported that the number of global garbage collections that the system performs whenever memory is filled with garbage could be significantly reduced by increasing the size of the large-object area. Although no attempt has been made to quantify these results, the developers report that without the large-object area, they were unable to give even a 3-min demonstration of their system without incurring a global garbage collection. By devoting 1 to 2 Mb of memory to the large-object area, it was possible to demonstrate the system without intrusive garbage collections.

CONCLUSIONS

Generation scavenging offers good garbage-collection performance if most objects die young, but it performs poorly if too many objects live for a while and then die. How well will it work in real systems? To answer this question, we have instrumented a production Smalltalk system to produce traces of objects that live for several minutes to several hours. The traces drove simulations of a simple two-generation scheme. Our sample runs lasted 4 to 6 h; this is considerably longer than the samples generally used in these analyses, which typically are only a few minutes long.

Our performance metrics were the pause time and total space consumed by unreclaimed garbage. Our two-generation reclamation algorithm had one parameter controlling when an object was promoted to the older generation. By varying this parameter, we were able to plot our space metric against our time metric. This gave us one curve for each strategy, and the strategies could be compared by comparing the curves.

The samples displayed a wide variation. With feedback-mediation and a separate large-object area, our simulated generation scavenger performed well on three of the sessions. For instance, it could simultaneously limit 90

percent of the pauses to 100 ms and total tenured garbage to 200 kb. On the other hand, even with our improvements, our simulated generation scavenger could not cope with the fourth session. Even allowing pauses to double, it could not reduce tenured garbage below 7 Mbytes. If our samples are representative, then reclamation systems must be prepared to cope with a wide variation in object demographics.

The lifetime distributions did not fall off smoothly. Instead, each distribution contained idiosyncratic clumps of objects that lived for more than one-half hour. These clumps suggest that, contrary to conventional wisdom, systems must be designed to cope with irregular, time-varying distributions of object lifetimes. For example, without feedback, clumps of middle-aged objects are likely to fill up the areas housing them and cause scavenges to overflow into the old object space. As a result, many objects will receive premature tenure and subsequently fill up old space. Feedback allows the system to adapt to varying object demographics.

Finally, the analysis suggested two important improvements to the basic generation scavenging algorithm: segregating large bitmaps and strings can cut pause times fourfold and reduce tenured garbage by megabytes, and demographic feedback-mediated tenuring can provide much more control over the maximum pause times, while reducing the amount of tenured garbage. We subsequently incorporated both these techniques into an actual language implementation, and the preliminary measurements seem to verify our analysis. These two improvements should be incorporated into systems using generation scavenging.

ACKNOWLEDGMENTS

Many people have helped to make this work possible. Ron Carter and Russ Pencin inserted the instrumentation into the virtual machine. Peter Deutsch, Adele Goldberg, Nanette Harter, David Leibs, Kenny Rubin, and Stephen Pope braved the instrumented Smalltalk-80 system to provide us with data. Finally, Allan Schiffman helped us both technically and editorially.

REFERENCES

1. BAKER, H. G. List processing in real time on a serial computer. A I Working Paper 139, MIT-AI Lab., Boston, Mass., Apr. 1977.
2. BECK, K. Private communication, Mar. 1988.
3. CHAMBERS, C., UNGAR, D., AND LEE, E. An efficient implementation of SELF, a Dynamically-typed object-oriented language based on prototypes In *Proceedings of the 1989 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)* (New Orleans, La.).
4. COLLINS, G. E. A method for overlapping and erasure of lists. *Commun ACM 3*, 12 (Dec 1960), 655–657.
5. BOSWORTH, G. Private communication, Mar. 1988.
6. BROWNBRIDGE, D R. Recursive structures in computer systems Ph D dissertation, University of Newcastle upon Tyne, Sept. 1984.
7. CAUDILL, P. J., AND WIRFS-BROCK, A. A third generation Smalltalk-80 implementation. *ACM SIGPLAN Notices 21*, 11 (Nov. 86).

8. DEUTSCH, L. P. The Dorado Smalltalk-80 implementation: Hardware architecture's impact on software architecture. In G. Krasner, Ed., *Smalltalk-80 Bits of History, Words of Advice*, Addison-Wesley, Reading, Mass., 1983, pp. 113–126.
9. DEUTSCH, L. P., AND BOBROW, D. G. An efficient incremental automatic garbage collector. *Commun. ACM 19*, 9 (Sept. 1976), 522–526.
10. DEUTSCH, L. P., AND SCHIFFMAN, A. M. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages* (Salt Lake City, Utah), Jan. 1984.
11. FATEMAN, R. J. Private communication, 1983.
12. FODERARO, J. K., AND FATEMAN, R. J. Characterization of VAX maxsyma In *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation* (Berkeley, Calif., 1981), pp. 14–19.
13 GOLDBERG, A , AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.
14. KAEHLER, T., AND KRASNER, G. LOOM—Large object-oriented memory for Smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner, Ed., Addison-Welsey, Reading, Mass., 1983, pp. 251–271.
15. LEE, E. Object storage and inheritance for SELF, a prototype-based object-oriented programming language. Engineer's thesis, Stanford Univerity, Stanford, Calif., Dec. 1988.
16. LIEBERMAN, H , AND HEWITT, C. A real-time garbage collection based on the lifetimes of objects. *Commun. ACM 26*, 6 (June 1983), 419–429.
17. MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, I. *Commun. ACM 3* (1960), 184–195.
18. MOON, D A. Architecture of the symbolics 3600. In *Proceedings of the 12th Annual International Symposium on Computer Architecture* (Boston, Mass., June 1985), pp. 76–83.
19. SHAW, R. A. Empirical analysis of a Lisp system. Tech. Rep. CSL-TR-88-351, Stanford University, Stanford, Calif., Feb. 1988.
20. STAMOS, J. W. A large object-oriented virtual memory: Grouping, measurements, and performance. Tech. Rep. SCG-82-2, Xerox, PARC, Palo Alto, Calif., May 1982.
21. STAMOS, J. W. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Trans. Comput. Syst. 2*, 3 (May 1984), 155–180.
22. STANDISH, T. A. *Data Structure Techniques.* Addison-Wesley, Reading, Mass , 1980, pp. 220–225.
23. UNGAR, D. Generation scavenging: A nondisruptive high performance storage reclamation algorithm. In *Proceedings of the ACM Symposium on Practical Software Development Environments* (Pittsburgh, Pa , April 1984), pp. 157–167 Also published as *ACM SIGPLAN Notices 19*, 5 (May 1984) and *ACM Software Engineering Notes 9*, 3 (May 1984)
24. UNGAR, D. The Design and Evaluation of a High Performance Smalltalk System, ACM 1986 Distinguished Dissertation, M.I.T. Press, Cambridge, 1987.
25. UNGAR, D., AND JACKSON, F. Tenuring policies for generation-based storage reclamation. In the *OOPSLA '88 Conference Proceedings* (San Diego, Calif., 1988), pp. 1–17.
26. UNGAR, D., AND SMITH, R. B. SELF: The power of simplicity. In *Proceedings of the 1987 ACM Conference Proceedings* (Orlando, Fla., 1987), pp. 227–242.
27. WILSON, P. R., AND MOHER, T G. Design of the opportunistic garbage collector. In the *OOPSLA '89 Conference Proceedings* (New Orleans, La., 1989), pp. 23–36.
28. ZORN, B. G Comparative performance evaluation of garbage collection algorithms. Ph.D. dissertation, Tech Rep. UCB/CSD 89/544, Univ. of Calif. at Berkeley, Mar. 1989.