

# Tenuring Policies for Generation-Based Storage Reclamation

*David Ungar and Frank Jackson*

David Ungar\*  
CIS, Room 209  
Stanford University  
Stanford, CA 94305  
ungar@amadeus.stanford.edu  
(415) 725-3713

Frank Jackson  
ParcPlace Systems  
2400 Geng Road  
Palo Alto, CA 94303  
jackson@ParcPlace.com  
(415) 859-1025

## Abstract

One of the most promising automatic storage reclamation techniques, generation-based storage reclamation, 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 Generation Scavenging automatic storage reclamation from traces of actual four-hour sessions. There was a wide variation in the sample runs, with garbage-collection overhead ranging from insignificant, during the interactive runs, to severe, during a single non-interactive run. All runs demonstrated that performance could be improved with two techniques: segregating large bitmaps and strings, and mediating tenuring with demographic feedback. These two improvements deserve consideration for any generation-based storage reclamation strategy.

---

\* This work was performed while consulting at ParcPlace Systems.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1988 ACM 0-89791-284-5/88/0009/0001 \$1.50

## Introduction

I'm fixing a hole where the rain gets in  
And stops my mind from wandering  
Where it will go  
I'm filling the cracks that ran through the  
door  
And kept my mind from wandering  
Where it will go  
[McCartney67a]

Automatic storage reclamation is an important component of 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 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:

- distracting pauses while reclaiming garbage,
- failure to reclaim some types of data structures, and
- high temporal and spatial overhead.

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

One of the most promising automatic storage reclamation techniques, generation-based storage reclamation, exploits an empirical property: young objects are likely to die and old ones are likely to continue to live. This behavior is so pronounced that one refinement of this technique, Generation Scavenging [Ungar86, Ungar84], can reduce pause times to a fraction of a second, while using only 3% of the CPU cycles and 200kb 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 [Beck], DigiTalk [Bosworth], Tektronix [Caudill], and ParcPlace Systems.

But there is a problem: these algorithms depend on a high rate of infant mortality, an empirically observed property. These algorithms cannot gracefully reclaim objects that live to a ripe old age before dying. For example, Generation Scavenging divides objects into two generations, young and old. Reclamation via scavenging is confined to young objects. When a young object has survived long enough, it is made immune such to reclamation by promoting it to the old generation. We call this tenuring. If an object lives long enough to attain tenure but then dies, its space goes unreclaimed until the next time the global garbage collector runs. Since the time required to perform a global garbage collection typically exceeds the response-time requirements of modern interactive systems, the amount of such tenured garbage, and thus the tenuring policy, can make or break systems using generation-based storage reclamation.

In this paper, we report on 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-based automatic storage reclamation algorithms in general and Generation Scavengers in particular.

## Previous Work

I've got to admit it's getting better  
A little better all the time.  
[McCartney67b]

## Lifetime-Independent Automatic Storage Reclamation Algorithms

In 1960, Collins published a paper describing an automatic storage reclamation algorithm based on *counting references* [Collins]. Simplicity and short pause times helped this algorithm gain acceptance, and it was adopted for many systems, including the Dorado Smalltalk-80\* implementation [Goldberg] and LOOM [Kaehler, Stamos82, Stamos84]. Deutsch and Bobrow halved the time cost, resulting in about 10% overhead [Deutsch76, Deutsch84]. However, reference-counting automatic storage reclamation algorithms suffer from the inability to reclaim circular structures of objects [Standish], unless the algorithms incur the expense of additional recursive scanning [Brownbridge]. Since circularity is a common characteristic of typical 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 [McCarthy]. 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 as memory sizes grew faster than processing speeds. By the early eighties, some Lisp programs were wasting 25% to 40% of their time marking and sweeping [Fateman], and users were waiting an average of 4.5 seconds every 79 seconds [Foderaro]. Mark-and-sweep automatic storage reclamation does not seem to be practical on contemporary (1988) computers.

\* Smalltalk-80 is a trademark of ParcPlace Systems.

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

## Generation-Based Automatic Storage Reclamation Algorithms

In order to build more effective automatic storage reclamation algorithms, we have been forced to leave the realm of theory and exploit an empirical property of data: infant mortality. For example, Generation Scavenging [Ungar86, Ungar84], which imposes only 3% time overhead, divides objects into generations according to their ages. Once objects have been segregated by age, attempts to reclaim storage can be directed at younger objects, which are more likely to be reclaimable. In fact, so many of the young objects die in the interval between collections (95% for Generation Scavenging), that more time can be saved by ignoring the dead ones and concentrating on saving the live ones. Other generation-based algorithms include *Generation Garbage Collection* [Lieberman], *Ephemeral Garbage Collection* [Moon], and the Tektronix Smalltalk reclamation algorithm [Caudill].

However, there is a catch: it takes a lot of time to reclaim the space occupied by old objects that die. In Generation Scavenging, the only way to reclaim the space occupied by the dead objects in the old area is a full mark-and-sweep garbage collection. This can take from half a second to several minutes of real time in Smalltalk-80 systems. We have described promoting an object from the new area to the old area as *tenuring*. If many *tenured* objects die, much space will be wasted. We call this the *tenuring problem*.

## Multiple Generations

The tenuring problem generally forces implementors to trade off pause time against wasted memory. A liberal tenuring policy promotes more objects, which reduces the pause time needed to copy the new objects but increases the space wasted by dead tenured objects. On the other hand, a conservative tenuring policy promotes fewer objects, which decreases wasted space but increases pause times. In an attempt to minimize both pause times and temporal overhead, some researchers have used more than two generations. The rationale for having multiple generations goes roughly as follows:

- 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 scavenged) 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 scavenging the youngest generation.
- Since the older generations are scavenged less often, objects can be held in these generations for a longer period of time than they could be held in a system that had only one scavenged generation, while still maintaining the same overall throughput.
- Forcing an object to pass through several generations before being granted permanent status increases the likelihood that the garbage objects will expire in a generation where they can be still be reclaimed quickly and efficiently.

These advantages are so compelling that [Lieberman], [Moon], and [Caudill] all proposed or adopted multiple generations. However, all three seem to employ fixed-age promotion policies, rather than a feedback-mediated policy as proposed in this paper. Berkeley Smalltalk [Ungar84, Ungar86] did include a primitive form of

feedback-mediated tenuring, but no analysis of tenuring was possible at the time. The other idea put forth in this paper, segregation of large bitmaps and strings, was included in Tektronix Smalltalk [Caudill] but also was not analyzed. As shown below, both segregation for large bitmaps and strings, and demographic feedback-mediated tenuring can dramatically improve garbage collection performance.

We have not yet adopted multiple generations because such systems have some drawbacks:

- The extra generations increase the code complexity, not only because the extra generations require additional code, but because of potential interactions with other runtime subsystems.
- It may be harder to tune a multi-generation system. Some of these systems suffer from distracting pauses when scavenging the intermediate generations.

Given these drawbacks, there should be good reason for paying the price of a multi-generation system. Unfortunately, there is a paucity of data in the literature about the true extent of the *tenuring problem*. This work is an empirical study of object lifetimes which should help researchers decide how many generations are really needed.

## Methodology

Send me a postcard, drop me a line,  
Stating point of view,  
Indicate precisely what you mean to say  
[McCartney67c]

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 altering 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, delivers good performance—the fastest for Smalltalk on a MC680X0—and we were able to instrument it without too much overhead: only 11%. The instrumented version enjoys a performance rating of 106% on a Sun 3/75 workstation\* with 8 Mb of main memory. Even with our instrumentation, this system runs faster than many widely used Smalltalk systems.

There was another, equally important reason to use this particular Smalltalk implementation (a.k.a. *virtual machine*) 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-sweep garbage collector or a scavenger as its primary reclamation system, since these systems do not discover that an object is no longer referenced until sometime after the actual de-referencing 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.

A reference-counting system would be appropriate for our purposes, since such systems reclaim de-referenced objects immediately; however, most of the Smalltalk systems that employ simple reference-counting schemes have been too slow to support highly interactive applications. The system we instrumented achieves good reclamation performance by employing Deutsch-Bobrow deferred reference-counting [Deutsch76]. Since our system updates reference counts more than once a second, we can obtain times of death to the nearest second. Thus, the fortuitous circumstance that made this work possible was the availability of a fast system with an automatic storage reclamation algorithm that rapidly reclaimed dead objects.

There is one problem with a reference counting system: it cannot detect circular structures of garbage. However, this is not a major problem because the Smalltalk-80 system was developed on a reference counting virtual machine [Goldberg] and consequently takes care to explicitly break cycles when objects are no longer needed.

\* This workstation uses an MC68020 processor running at 16 MHz.

## 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 destroyed. 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 (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.

Figure 1: Contents of Records

birth time (in secs)
death time (in secs)
size (in words)
address
true if object has pointers

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. This corresponds with 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. Truncating time to the second shrank the timestamp to 16 bits, which allowed for a maximum session duration of 65535 seconds, or about 18 hours; none of our users had longer sessions.

How will this optimization change the resulting measurements? As described in [Ungar86], Generation Scavenging systems on this class of machine scavenge approximately once per second, and experience with Berkeley Smalltalk has shown that the interval between scavenges does not vary by too much. Thus, we do not expect that this optimization will drastically affect our

measured pause times. Finally, since we are concerned with tenuring times that range from one to fifteen minutes, we do not expect this optimization to alter our measurements of tenured garbage.

The actual instrumentation was accomplished by storing birth times in the object table.\* The 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:

- 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 pre-existing objects.
- 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

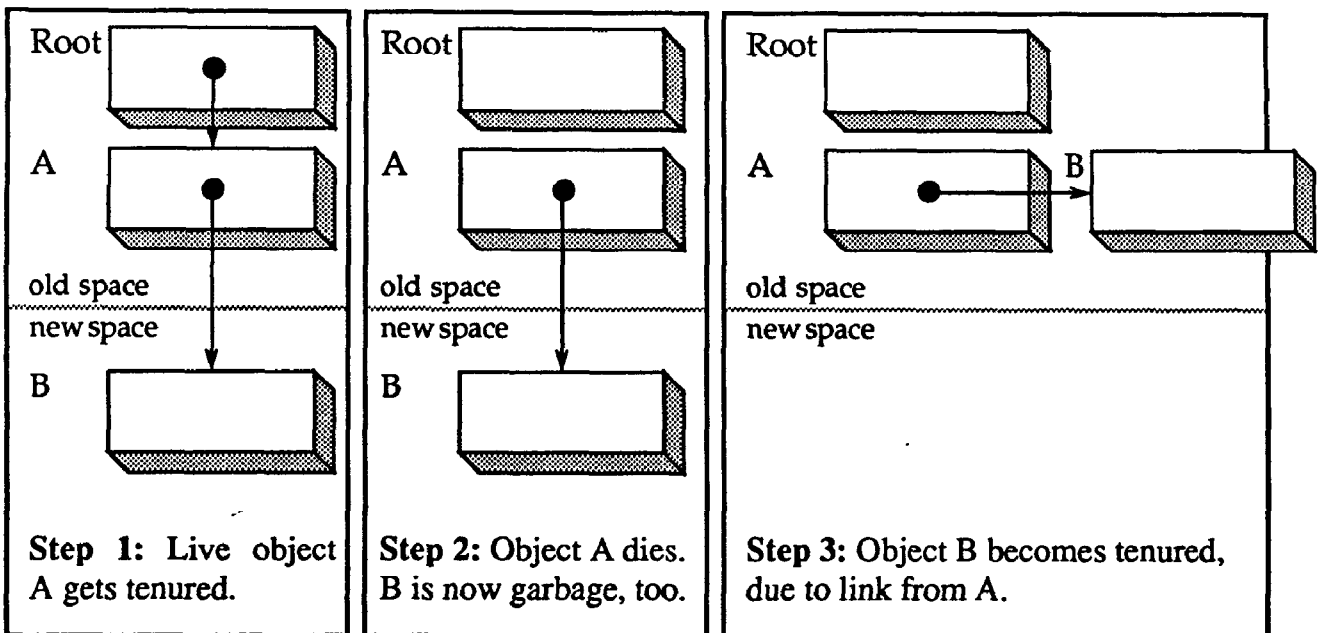
Our simulations must *underestimate* the amount of tenured garbage. In a real system, a new object may be kept alive by a reference from tenured garbage and eventually become tenured garbage itself (Figure 2). In other words, an object which has been tenured, but is now dead, will eventually cause all of the objects that it references to be promoted, live or dead. We therefore describe this as *nepotism*. Our simulation cannot take this effect into account.

There is some reason to believe that nepotism contributes little to tenured garbage. Other experiments have measured the number of references from old objects to young ones to be approximately 50 [Ungar86, Ungar84]. The relatively small size of this number compared to the number of tenured objects suggests that

---

\* The clock was maintained by an interrupt-driven system call that only interrogated the operating system once a second.

Figure 2: Nepotism



nepotism may not be a big problem. Even if it contributes 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 1 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 user activity is not stationary, but rather consists of phases, we simulated *dynamic* system behavior.

## Four Kinds of Objects

The traces allowed us to group objects into the following classes (Figure 3):

- **Transients** are those objects that come to life and then die within a given session. These objects can become tenured garbage if they are promoted prematurely. For our purposes, transients are the most important class of objects.
- **Permanent** objects are alive at both the beginning and the end of a session. They are read in from the disk when the user starts Smalltalk. 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 die, we ignore them in this study of tenured garbage.

**Table 1: Sessions**

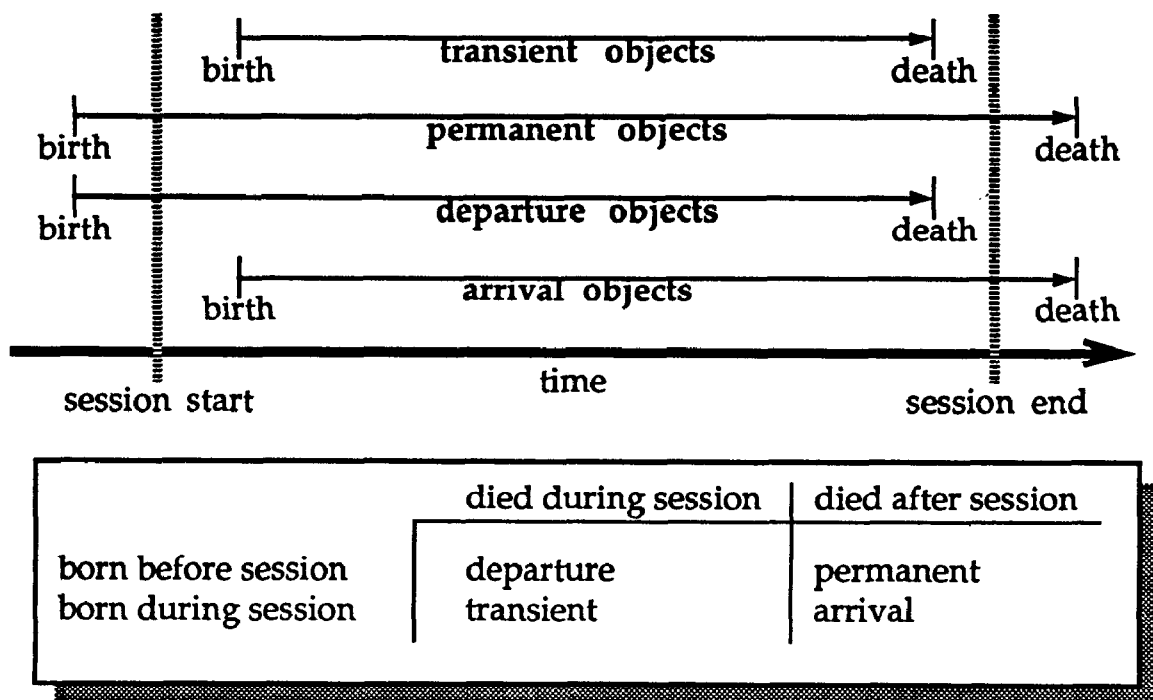
ID	Duration (hrs:mins)	Interactive?	Activity
edit	6:40	interactive	Active text editing
mail	6:53	interactive	Longest of three runs, mostly reading mail
prog	4:36	interactive	Writing Smalltalk programs
sim	5:15	noninteractive	Running financial simulations

- **Departures** are those objects that are alive at the start of a session, but then die before the session is over. These objects could also become tenured garbage, but in fact turn out to be inconsequential since there are so few of them: in all of our samples, the departures are less than one percent of the transients. Thus, we ignore these objects as a source of tenured garbage.
- **Arrivals** are those objects that come to life, but do not die during that session. In a perfect system, these are the only objects that would be granted tenure. It turns out that these are small in number compared to the tenured objects, also less than one percent. We therefore ignore arrivals as well as departures.

## Objects Great and Small

In Smalltalk, objects can contain either references to other objects, or uninterpreted data, such as characters in a string or bits in a bitmap. Large objects containing uninterpreted data are candidates for special treatment in a scavenging system: since they have no pointers, scanning is unnecessary; and since they are large, copying is expensive. We therefore decided to separate them out from other objects in our data. (Large objects are defined as containing at least 1024 bytes of uninterpreted data.) When it became clear that our system was using a significant number of these objects (to cache the images of occluded windows), we decided to simulate a system that kept the

**Figure 3: Classes of Objects**



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.

## 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:

- The time taken by Generation Scavenging has been shown to correlate well with the number of bytes scavenged [Ungar86]. The per-object overhead is relatively low.
- 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.
- Finally, we constructed some scatter plots to see if there were 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, there was no such relationship.

Accordingly, the results reported in the rest of the paper are bytes of space taken by the objects. Six bytes of overhead are included for each object.

## Dynamic Analysis of Tenuring Policies

Only the good die young  
[Joel]

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

- **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.

- **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.
- **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 scavenging system, 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 is driven by two files: a file sorted by births and another sorted by deaths. In addition, a clock ticks off virtual time. This simulator works by keeping track of the currently surviving new objects in a table (Figure 4). At each clock tick:

- All death events taking place at that time are processed. The dying objects are removed from the survivor table and tallied as either transients or departures.
- Next, births are processed by adding them to the survivor table.
- Lastly, the survivor table is scanned for objects to be tenured. These objects are removed from the table and tenured.

Statistics are 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 is the total number of bytes copied. Measurements on a Sun 3/60 (20 MHz 68020) have shown that about 500,000 bytes can be scavenged in a second. So, to convert bytes to seconds, we divided by 500,000—the pause times will vary according to processor speed.



## Figure 4: Summary of Scavenging Simulator

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

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

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

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

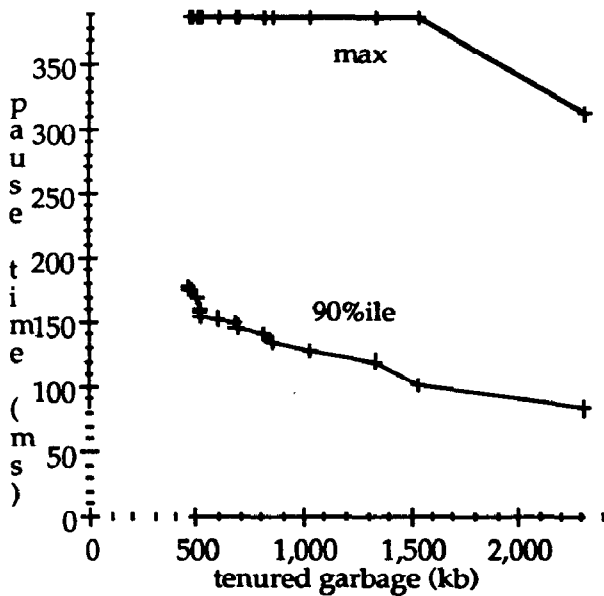
## Simulator Results: Fixed-Age Tenuring

First, we ran the simulator with a very simple tenuring policy: every object that attained a certain age, the *tenuring threshold*, was tenured. We call this 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 one to fifteen minutes. From this, we computed tenured garbage, maximum pause time, and 90% 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 5-8. Each graph has two

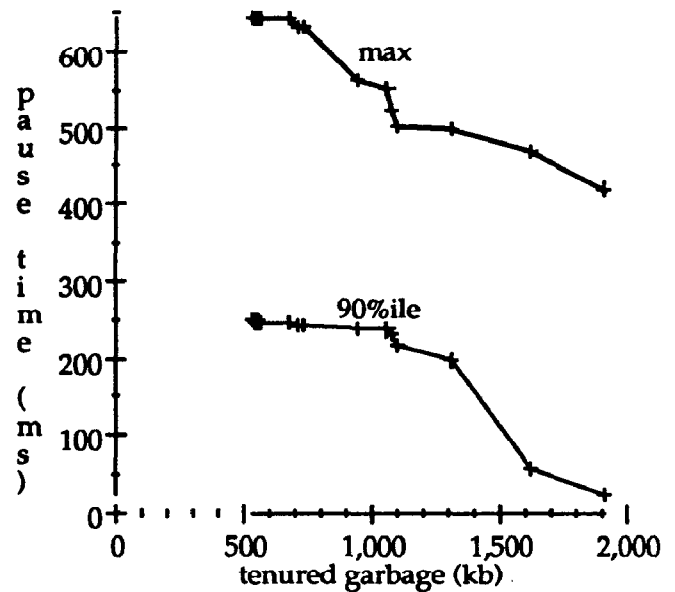
traces: one showing the 90th percentile pause time as a function of tenured garbage and one showing the maximum pause time as a function of tenured garbage.

The tenure threshold in each trace ranges from one to fifteen minutes in one minute increments. The right hand endpoint of each curve corresponds to a tenure threshold of one minute. As expected, this low tenure threshold produced the most tenured garbage and the shortest pause times. The left hand endpoint corresponds to a tenure threshold of fifteen minutes. As expected, this high tenure threshold produced the least tenured garbage and the longest pause times. Finally, there is much more data surviving in the "sim" run than the other (more interactive) runs; this shows up as both more tenured garbage and longer pause times.

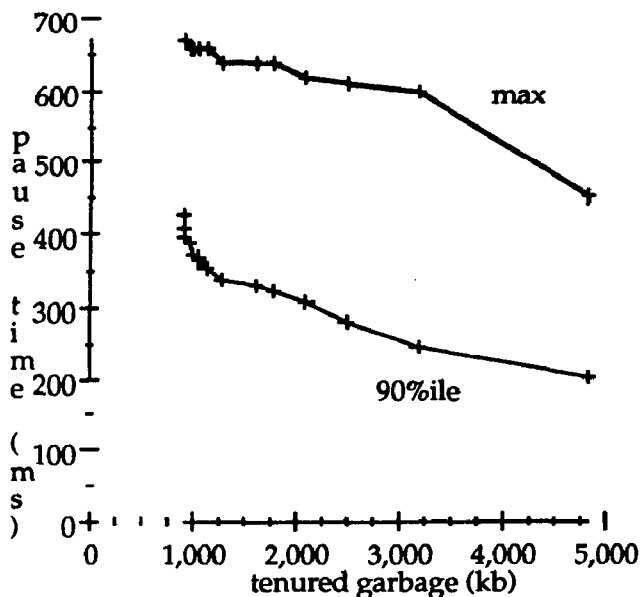
**Figure 5: "edit" run**  
**Pause Time vs. Tenured Garbage**  
**(Fixed Tenure Threshold)**  
**(Scavenging Large Bitmaps & Strings)**



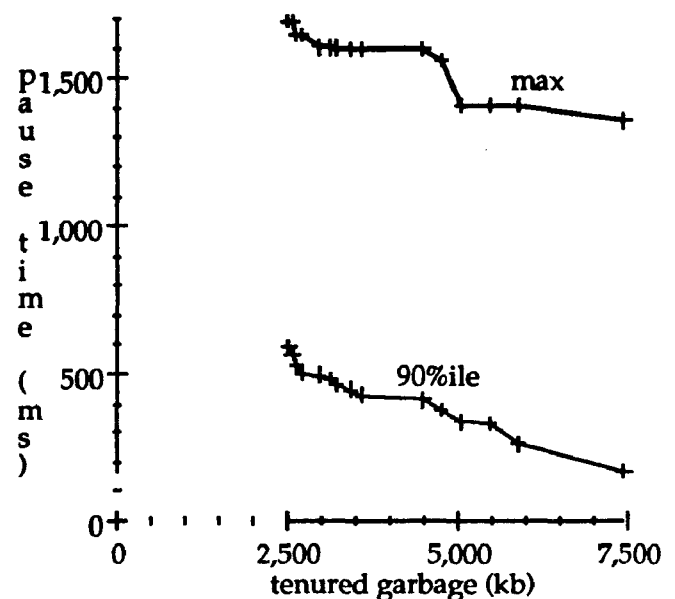
**Figure 6: "mail" run**  
**Pause Time vs. Tenured Garbage**  
**(Fixed Tenure Threshold)**  
**(Scavenging Large Bitmaps & Strings)**



**Figure 7: "prog" run**  
**Pause Time vs. Tenured Garbage**  
**(Fixed Tenure Threshold)**  
**(Scavenging Large Bitmaps & Strings)**



**Figure 8: "sim" run**  
**Pause Time vs. Tenured Garbage**  
**(Fixed Tenure Threshold)**  
**(Scavenging Large Bitmaps & Strings)**



## Segregating Large Bitmaps and Strings

Next, we examine the benefits of special treatment for large bitmaps and strings. The simulator was changed to implement a *large object area*. The idea is to put all large bitmaps and strings in a separate area, storing only their headers in the new area. The headers are then scavenged, but no time is spent copying the data in the large bitmaps and strings. Furthermore, these objects are never tenured, because the large object area can be made big enough to hold all of them. 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.

Segregating large bitmaps and strings saves time and space. Figures 9–12 compare the performance without a large object area to the performance with a large object area. Each graph has four traces: two are repeated from the previous graphs for comparison, and the other two give the 90th percentile and maximum pause times for a system with a large object area. On all curves the crosses mark tenuring thresholds from one to fifteen minutes in one-minute increments.

A separate area for large bitmaps and strings pays off, although the graphs look different for the interactive runs when compared to the noninteractive “sim” run: For example, in the “mail” run, dedicating 330 kilobytes to the large object area saves over a megabyte of tenured garbage and cuts pause times by a factor of four. On the other hand, in the “sim” run, dedicating 740 kilobytes to the large object area makes a difference that does not look so dramatic, but the absolute reduction in tenured garbage, roughly 2 Mb, is still substantial. Since most current workstations have at least four megabytes of main memory, the amount required for the large object area seems reasonable. This idea is 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 one explores a better tenuring policy for the rest of the objects. A fixed-age tenuring policy has three problems:

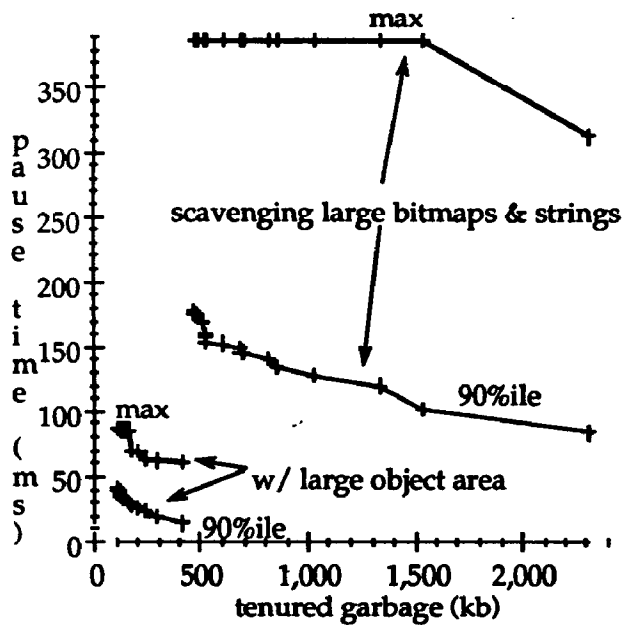
- **occasional long pauses:** If there are many objects just younger than the tenuring threshold, the pauses will be long.
- **extravagant tenuring:** If there are very few objects being scavenged, there is no need to tenure any, but a fixed-age policy will tenure objects that are old enough anyway.
- **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 the youngest generation when it 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 is not the case. Instead, long-lived objects seem to be born in clumps, which diminish only slowly with time, like a pig that has been swallowed by a python.

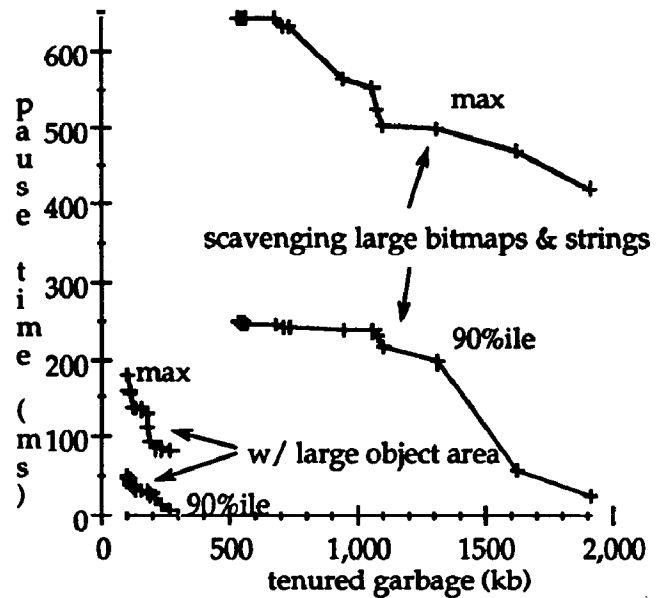
We therefore propose a tenuring policy—illustrated in Figure 13—with two components:

- **feedback mediation:** At the end of each scavenge, the system examines the amount of surviving data in the youngest generation. Recall that the pause time is proportional to the amount of data that must be copied. Thus, the amount of surviving data provides an estimate of the danger that the next scavenge will exceed the maximum acceptable pause time. If, after a scavenge the survivor size is small, the tenuring threshold is set at infinity for the next scavenge; no objects will be tenured the next time. If, on the other hand, the survivor size is large, the tenuring threshold is set to a value designed to tenure the excess data on the next scavenge. How is this value chosen?

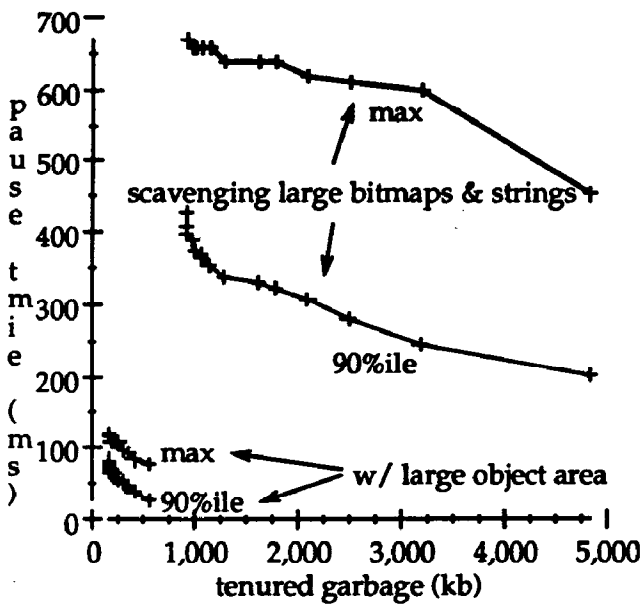
**Figure 9: "edit" run**  
**Effect of Large Object Area**  
**(Fixed Tenure Threshold)**  
**(260 kb for large object area)**



**Figure 10: "mail" run**  
**Effect of Large Object Area**  
**(Fixed Tenure Threshold)**  
**(330 kb for large object area)**



**Figure 11: "prog" run**  
**Effect of Large Object Area**  
**(Fixed Tenure Threshold)**  
**(390 kb for large object area)**



**Figure 12: "sim" run**  
**Effect of Large Object Area**  
**(Fixed Tenure Threshold)**  
**(740 kb for large object area)**

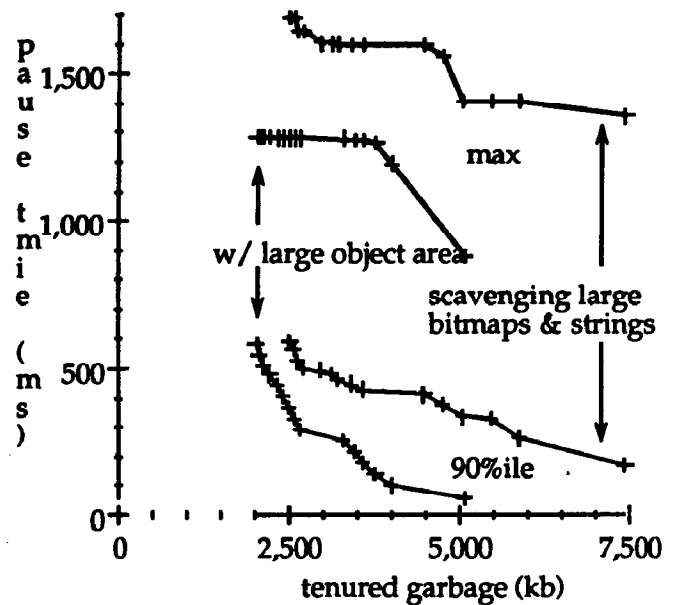
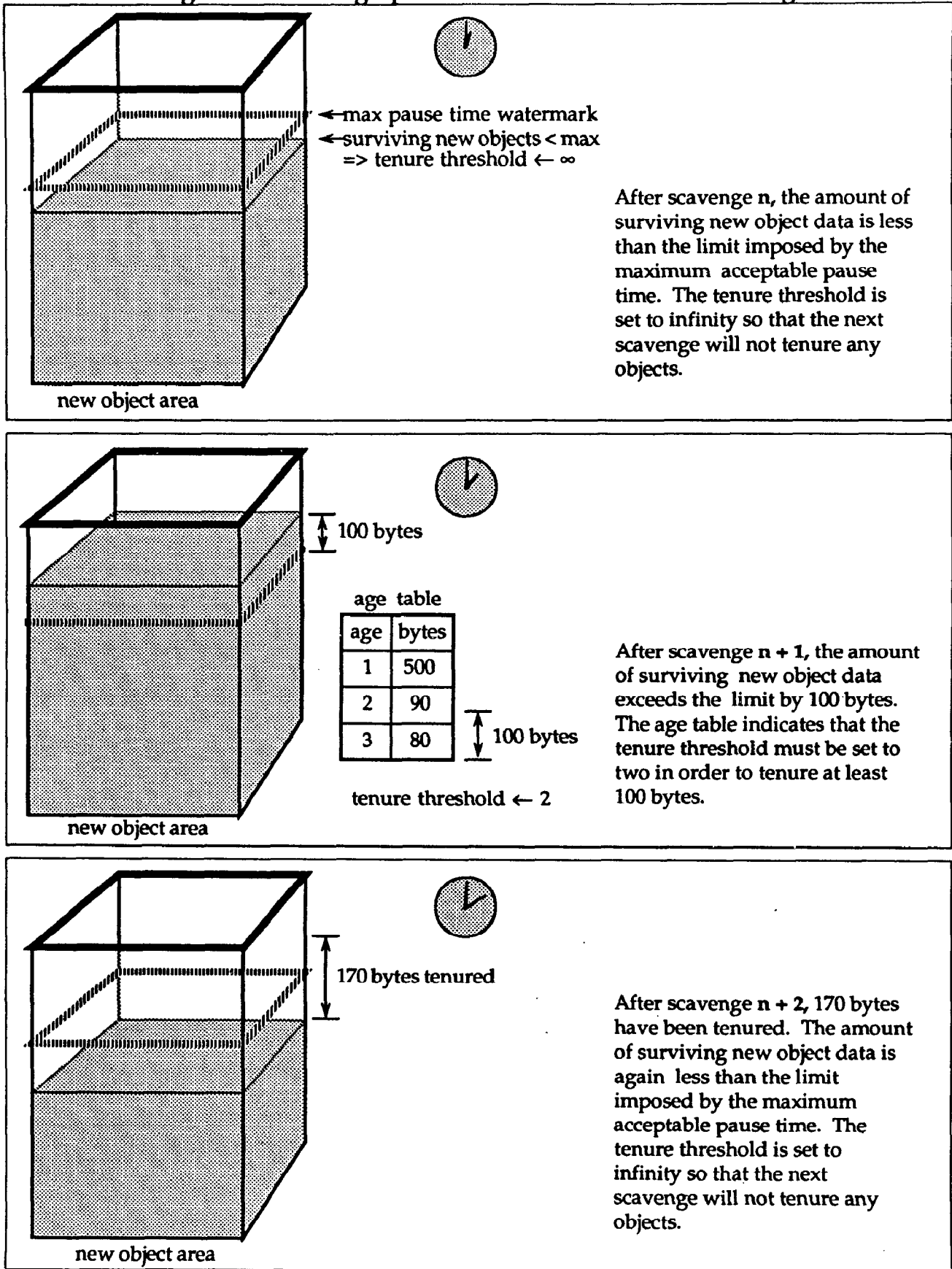


Figure 13: Demographic Feedback-Mediated Tenuring



- **demographic information:** After a scavenger, if the aggregate size of the surviving data exceeds the pause time threshold, a pass through the survivor area computes a table indexed by age containing the number of data bytes for each age. A backwards scan through this table yields the appropriate setting for the tenuring threshold.

Both parts are important: the feedback mediation triggers tenuring only when needed, and the demographic information makes it possible to tenure only as many objects as need to be tenured, while using age as a tenuring criterion. Recall that since age is the best predictor of lifetime, age is still the best determinant of which objects to tenure. The effect of this policy is to limit pauses when many objects survive and to eliminate tenuring when few objects survive.

To evaluate this policy, we modified our simulator and ran it for various pause time thresholds. Figures 14–17 show the results of demographic feedback-mediated tenuring. Each graph has four traces, all assuming a large object area. Two traces show the fixed-age policy for comparison; as before the crosses on these represent tenuring thresholds from one to fifteen minutes in one minute increments. The other two traces show the feedback-mediated policy; the crosses on these curves mark pause time thresholds from 20 milliseconds to 200 milliseconds in 20 millisecond 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 does the overall performance of the new policy compare with the fixed-age tenuring policy? In the interactive sessions (“edit,” “mail,” and “prog”), it is 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 reduces the maximum pause time from 140 ms to 66 ms. In the “sim” session, the results are slightly different: the new algorithm brings both the 90th percentile and maximum pause times to a

level that is a bit worse than the old 90th percentile pause time. Even in this difficult session, the new algorithm drastically improves the maximum pause times: at 3.5 Mb of tenured garbage, it increases 90th percentile pause time from 240 ms to 400 ms, but reduces maximum pause time from 1.3 secs to 440 ms. Since this session is noninteractive, minimizing tenured garbage may well be more important for it than limiting pause times. If so, the feedback tenuring policy can easily be set to limit tenured garbage. Feedback-mediated demographic tenuring offers more control over pause times and space needed for the survivors.

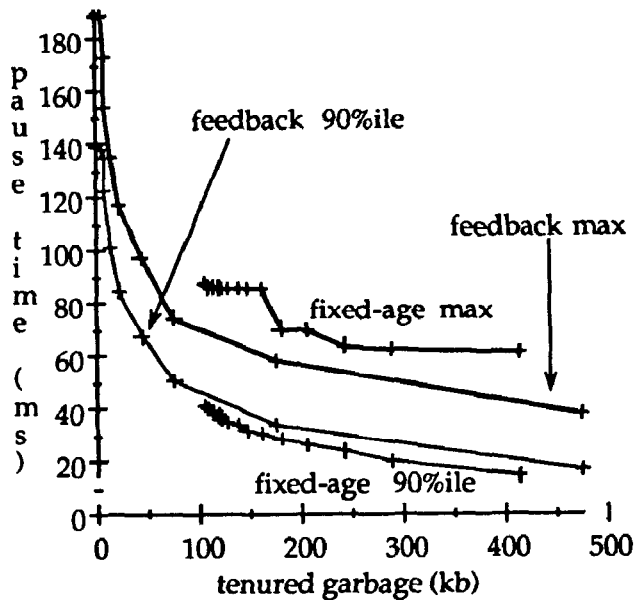
## Conclusions

Climb in the back with your head in the clouds.  
And you're gone  
[Lennon67c]

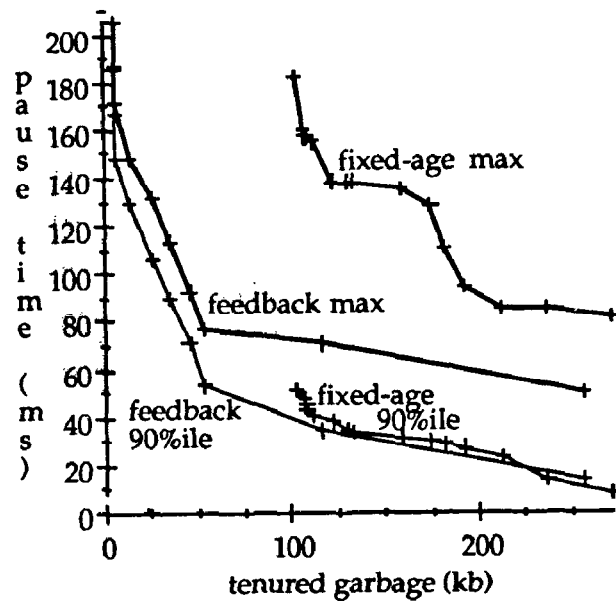
Generation-based garbage collection offers good performance if most objects die young, but it performs poorly if too many objects live for a while and then die. How well will it work in real systems? To answer this question, we have instrumented a production Smalltalk system to produce traces of objects that live for several minutes to several hours. The traces drove simulations of a simple two-generation scheme, Generation Scavenging. Our sample runs lasted four to six hours; 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 total space consumed by unreclaimed garbage, *tenured garbage*, and the *pause time*. 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.

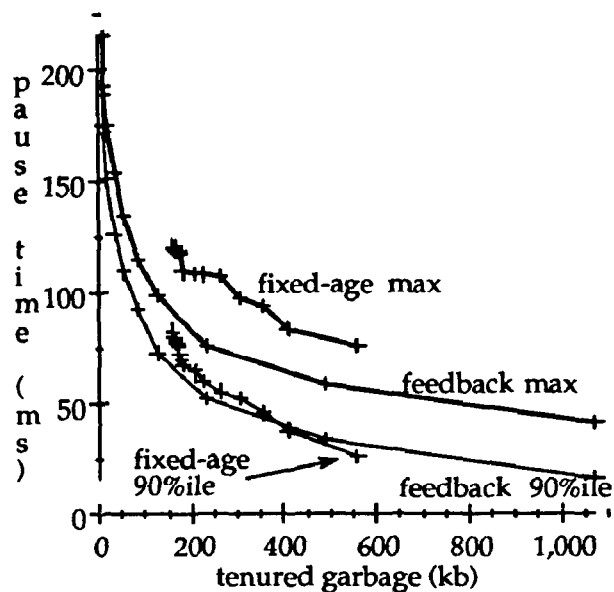
**Figure 14: "edit" run**  
**Effect of Demographic**  
**Feedback-Mediated Tenuring**  
**(260 kb for large object area)**



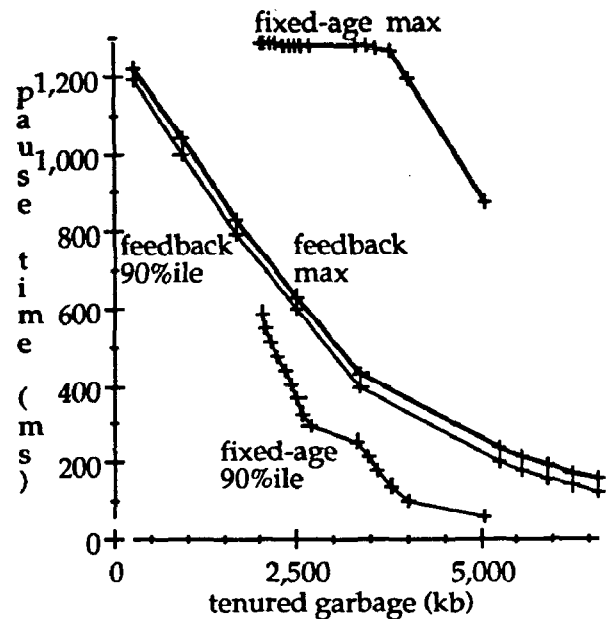
**Figure 15: "mail" run**  
**Effect of Demographic**  
**Feedback-Mediated Tenuring**  
**(330 kb for large object area)**



**Figure 16: "prog" run**  
**Effect of Demographic**  
**Feedback-Mediated Tenuring**  
**(390 kb for large object area)**



**Figure 17: "sim" run**  
**Effect of Demographic**  
**Feedback-Mediated Tenuring**  
**(740 kb for large object area)**



The samples displayed a wide variation—more than a factor of 80! Our two-generation storage reclamation algorithm performed well on the three interactive session. For instance, even if pauses were limited to 100 ms, no more than 200kb of garbage would be tenured during the four to six hour sessions. However, even if pause times were permitted to lengthen to 200 ms, seven megabytes of garbage would still be tenured during the five-hour noninteractive session. Our simple two-generation storage reclamation algorithm would not be sufficient for the noninteractive session. It may be that our algorithm works well for interactive runs but not for long-running programs. We expect to investigate this further.

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. These two improvements should be incorporated into systems using Generation Scavenging.

## Acknowledgments

What would you think if I sang out of tune,  
 Would you stand up and walk out on me.  
 Lend me your ears and I'll sing you a song,  
 And I'll try not to sing out of key.  
 I get by with a little help from my friends.  
 [Lennon67d]

This work has been supported by Xerox Corporation and ParcPlace Systems. Many people there have pitched in 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 test ideas by serving very ably as devil's advocate.

## References

- She said: "You don't understand what I said"  
 I said: "No, no, no, you're wrong."  
 [Lennon66]
- [Baker] H. G. Baker, "List Processing in Real Time on a Serial Computer," A.I. Working Paper 139, MIT-AI Lab, Boston, MA, April 1977.
- [Beck] Kent Beck, private communication, March, 1988.
- [Collins] G. E. Collins, "A Method for Overlapping and Erasure of Lists," *Communications of the ACM* 3, 12, December 1960, 655-657.
- [Bosworth] George Bosworth, private communication, March, 1988.
- [Brownbridge] D. R. Brownbridge, *Recursive Structures in Computer Systems*, doctoral dissertation, University of Newcastle upon Tyne, September, 1984.
- [Caudill] Patrick J. Caudill, Allen Wirfs-Brock "A Third Generation Smalltalk-80 Implementation," *OOPSLA '86 Conference Proceedings*, Portland OR, 1986, 119-130. Also published as *ACM SIGPLAN Notices* 21, 11, Nov. 86.
- [Deutsch76] L. Peter Deutsch and Daniel G. Bobrow, "An Efficient Incremental Automatic Garbage Collection," *Communications of the ACM* 19, 9, Sept. 1976, 522-526.
- [Deutsch84] L. Peter Deutsch and Allan M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, Salt Lake City, Utah, January 1984.
- [Fateman] Richard J. Fateman, private communication, 1983.
- [Foderaro] John K. Foderaro and Richard J. Fateman, "Characterization of VAX Maxsyma," *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, Berkeley, CA, 1981, pp. 14-19.
- [Goldberg] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.



- [Joel] Billy Joel, "Only the Good Die Young," in *Billy Joel Greatest Hits*, ASCAP, 1977.
- [Kaehler] Ted Kaehler, Glenn Krasner, "LOOM—Large Object-Oriented Memory for Smalltalk-80 Systems," in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (ed.), Addison-Wesley, Reading, MA, 1983, 251-271
- [Lennon66] John Lennon, Paul McCartney "She Said She Said," *Revolver*, BMI, 1966.
- [Lennon67a] John Lennon with Paul McCartney, "A Day in the Life," *Sgt. Pepper's Lonely Hearts Club Band*, BMI, 1967.
- [Lennon67b] John Lennon, "Good Morning," *Sgt. Pepper's Lonely Hearts Club Band*, BMI, 1967.
- [Lennon67c] John Lennon, "Lucy in the Sky with Diamonds," *Sgt. Pepper's Lonely Hearts Club Band*, BMI, 1967.
- [Lennon67d] John Lennon and Paul McCartney, "With a Little Help From My Friends," *Sgt. Pepper's Lonely Hearts Club Band*, BMI, 1967.
- [Lieberman] Henry Lieberman and Carl Hewitt, "A Real-Time Garbage Collection Based on the Lifetimes of Objects," *Communications of the ACM*, 26, 6, June 1983, pp. 419-429.
- [McCartney67a] Paul McCartney, "Fixing a Hole," *Sgt. Pepper's Lonely Hearts Club Band*, BMI, 1967.
- [McCartney67b] Paul McCartney, "Getting Better," *Sgt. Pepper's Lonely Hearts Club Band*, BMI, 1967.
- [McCartney67c] Paul McCartney, "When I'm Sixty-Four," *Sgt. Pepper's Lonely Hearts Club Band*, BMI, 1967.
- [McCarthy] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, I," *Communications of the ACM* 3, 1960, pp. 184-195.
- [Moon] David A. Moon, "Architecture of the Symbolics 3600," *Twelfth Annual International Symposium on Computer Architecture*, Boston, MA, June 1985, pp. 76-83.
- [Stamos82] James W. Stamos, "A Large Object-Oriented Virtual Memory: Grouping, Measurements, and Performance," Xerox technical report, SCG-82-2, Xerox, PARC, Palo Alto, CA, May 1982.
- [Stamos84] James W. Stamos, "Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory," *ACM Transactions on Computer Science* 2, 3, May 1984, 155-180.
- [Standish] Thomas A. Standish, *Data Structure Techniques*, Addison-Wesley, 1980, pp. 220-225.
- [Ungar84] David Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *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.
- [Ungar86] David Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, ACM 1986 Distinguished Dissertation, MIT Press, Cambridge, MA, 1987, 250 pages