KEY OBJECTS IN GARBAGE COLLECTION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Barry Hayes
March 1993

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Robert W Floyd
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Hans-Juergen Boehm

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

John L. Hennessy

Approved for the University Committee
on Graduate Studies:

Abstract

When the cost of global garbage collection in a system grows large, the system can be redesigned to use generational collection [28, 30, 40]. The newly-created objects usually have a much shorter half-life than average, and by concentrating the collector's efforts on them a large fraction of the garbage can be collected at a tiny fraction of the cost.

The objects that survive generational collection may still become garbage, and the current practice is to perform occasional global garbage collections to purge these objects from the system, and again, the cost of doing these collections may become prohibitive when the volume of memory increases. Previous research has noted that the objects that survive generational collection often are born, promoted, and collected in large clusters[41, 21].

In this dissertation I show that carefully selected semantically or structurally important *key objects* can be drawn from the clusters and collected separately; when a key object becomes unreachable, the collector can take this as a hint to collect the cluster from which the key was drawn.

To gauge the effectiveness of key objects, their use was simulated in ParcPlace's Objectworks\Smalltalk system[31]. The objects selected as keys were those that, as young objects, had pointers to them stored into old objects. The collector attempts to create a cluster for each key by gathering together all of the objects reachable from that key and from no previous key.

Using this simple heuristic for key objects, the collector finds between 41% and 92% of the clustered garbage in a suite of simple test programs. Except for one program in the suite, about 95% of the time these key objects direct the collector to

a cluster that is garbage. The exception should be heeded in improving the heuristics. In a replay of an interactive session, key object collection finds 59% of the clustered garbage and 66% of suggested targets are indeed garbage.

Acknowledgements

It is a rare event for more than one person to be listed as the author of a thesis, but I doubt that one person could write a thesis alone. To mention by name all of the contributors to the intellectual exploration that has lead me to write this would be a Herculean task, and I regret that I can't thank everyone personally.

The work presented here is the product of collaboration with many people at the Xerox Palo Alto Research Center, where I have been ensconced for several years now[1]. I was initially hired by Mark Weiser to work on a garbage collector for Cedar devised by Danny Bobrow and still haven't quite left yet.

There was widespread interest in the initial project and follow-up. The expertise about the internal workings of Cedar was well-distributed around the lab — I seem to recall most of my help coming from Russ Atkinson, Bob Hagman, Peter Kessler, Carl Hauser, Alan Demers, and Mike Spreitzer — able navigators all. Willie-Sue Orr was particularly brave — together we made the last Dorado microcode change.

The population of collector-heads in my environment has been changing through the years. Hans Boehm has been the continuing local presence, ready to hear my new ideas and share his. His questioning keeps me on my toes and makes me think at least a bit more than I talk. Frank Jackson found me a raft of guinea pigs at ParcPlace systems for the first experiment that supported key objects, arranged a source license of Objectworks for me, and has been my native guide in that territory. Paul Wilson generates more ideas over a beer than any three other people I know. I look forward to continuing our friendship.

[1]Special thanks to Lia Adams, who taught me never to subtract years from each other — you only learn things you'd rather not know.

Chapter 3 would not have been possible without the volunteers who allowed me to slow down their Dorados with my data-gathering code. Thanks go out to Rick Beach on Shangrila, Peter Kessler on Bennington, Subhana Menis on Saratoga, Steve Wallgren on Fremont and many others. Both Steve Wallgren and Polle Zellweger went far beyond the call of duty and endured unreliable and painfully slow beta-test versions of the code.

The continuing support from Xerox has been invaluable. Sharon Johnson managed to find disk space for the unreasonable volume of data I was generating. Lorna Fear and Anne Dobson did more support work on my behalf than I could ever repay. I am glad that John White, if he subtracts years, has put up with my continued presence for so long.

Continuing financial support for this work was supplied by the Northern California Chapter of ARCS, Inc. Some of the gaps were filled by the Dr. John Koza Fellowship, and the Schlumberger Foundation Collegiate Award Fellowship in Computer Science. I would have been unable to continue in graduate school if my patchwork funding had not been pieced together, year after year, by the Reverend Dr. Carolyn Tajnai.

My parents and sister have been supportive of my work and life in all ways, and without the sense of curiousity and willingness to risk failure that they gave me I would have neither started nor finished graduate school.

I would have been a quivering mass of ooze long ago if all I did was work on garbage collection. Thanks go out to The Diners — Yglennie, Ystewie, Yjohnny, Yrichie, and Ycraigie founding members — who saw to it that if I wasn't happy, at least I was fat. Brian and Victoria encouraged me to go to graduate school in the first place, and I don't hold it against them. Don talked me into staying when I was ready to chuck it, and I don't hold it against him, either. Marshall, Aaron, Penni, and Pavel have put up with more of my bad bidding and bad play than I can thank them for. If the four of them had been in the same place at the same time, I would have had to find other things to do at lunch. Miriam, Geoff, Kate, and a few hundred other people are a joy to sing with, and Bill kept us all sounding good. A collection of sisters, cousins, aunts, police, pirates, tars, daughters, dragoons, ancestors, and a fair number of electricians, carpenters, and painters also helped my sanity — Bonnie,

Paul, Jay, and Richard can, I hope, stand for the supernumeraries. Mike, Amy and David have been good friends to me for years, and I couldn't have finished without them and all my other friends.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Explicit resource management in a complex system can be a daunting task, even for a small number of resources. When the resources include hundreds of millions of bytes of memory partitioned into tens of millions of smaller resources, systems must follow a carefully thought-out discipline for explicitly managing memory. For every memory resource or *object* there must be a point in the system that is responsible for freeing that resource. Many programming languages require the programmer to explicitly release memory no longer in use, but leave it to the programmer to design the memory management discipline.

If this memory management discipline is not complete, correct, or followed precisely, two types of errors in memory management will occur: memory leaks and multiple allocations. A leak occurs when a resource is not released, even though it is no longer needed. A leak is particularly dangerous in a long-running program since even a small leak in a large resource — memory, for example — may eventually consume enough of the resource to cause program failure. For scarcer or less fungible resources — tape drives, for example — a leak can almost never be tolerated. Tools that find resource leaks can point out where the resource was allocated and the last time it is used, but they do not find the logical errors in resource management that lead to the leak.

Multiple allocation errors occur when the management discipline incorrectly releases a resource that is still in use. The resource is now in an inconsistent state: it

is in use, but not allocated. Often the user can continue using the resource without perceptible problems, and the system can advance well beyond the point where this error occurred. Serious trouble may become apparent only when the resource is allocated again. The resource is now in use for two different purposes, but this may not become apparent until well after the double allocation.

It is difficult to avoid these errors in resource management discipline, and resource management is a major development and maintenance burden. Explicit storage management was estimated to take 40% of the development time in the Mesa system [33]. The resource management problem needs global information for its solution, and modular decomposition of programs makes this information hard to collect in a coherent way that reflects the information hiding aspects of the modules.

1.1 Tracing Garbage Collection

Garbage collection is broadly used to mean any automatic method of reclaiming memory addresses that are no longer used. It is a memory management technique that avoids the problems inherent in resource management disciplines by allowing a set of processes — the *collector* — access to all the objects used by the other processes — the *mutator*[1]. Memory is modeled as a finite directed graph where the nodes are objects and there is an edge from one node to another if and only if the first object contains a pointer to the second. The mutator can reach a fixed set of *root* nodes and any objects reachable by following a chain of edges from a root.

Periodically, the collector partitions the memory resources into reachable and non-reachable. It does this by starting with the reachable roots and traversing the edges of the graph. All of the memory thus traversed may be reachable to the mutators[2].

[1] Interested readers should refer to Cohen's survey article [11] for more sources of information about early garbage collection techniques, and Wilson's survey for more recent techniques[45].

[2] If the collector had further information, it might be able to determine that the mutator would not access some of the traversed memory. For example, a pointer value in a register in the mutator might never actually be used in the future, and so some storage identified as reachable in the model can be freed without causing a double allocation bug. In fact, not freeing the storage causes a kind of memory leak, but presumably the collector will find the memory in the future, when the register value has been destroyed. The collector should always err on the side of leaking rather than double allocating.

The memory not traversed is the *garbage*. This memory can be released without affecting the mutator, since the mutator can not possibly reach it if operating within the model.

Within this framework there is a great variety of *tracing garbage collectors*. One major difference among these collectors is the implementation of the method used to keep track of the non-garbage objects that have already been visited as part of the transitive traversal of the storage.

1.1.1 Mark and Sweep Collection

The earliest collectors were *mark-and-sweep* collectors[29]. With these collectors, a bit is reserved either in the object itself or in a table, and as each object is reached in the traversal, the associated bit is set to indicate that the object is reachable. This ensures termination of the traversal, and when the traversal is finished the bits indicate which objects are garbage and which are not. In addition, the collector must have another data structure to distinguish the marked objects that may contain pointers to unmarked objects — the objects that are on the frontier of the traversal.

Typically the frontier is kept as a stack, leading to a depth-first traversal of the reachable objects, but the collector may benefit from being able to explore frontier elements based on other considerations. For example, a collector that favored tracing through objects already in fast memory would be expected to cause fewer page faults.

When all the reachable objects have been marked, the collector can find and release the unmarked garbage. This *sweep* phase is often combined with a compaction phase, allowing the collector to regain memory and at the same time combine all unreachable memory into a single block. Allocation in a compacted heap, with only a single block of free memory, is much simpler than allocation in a heap where the free memory is scattered throughout.

1.1.2 Copying Collection

Most modern collectors are *copying* collectors[3, 19, 9]. In these collectors, memory is divided into two *semispaces*. At any time when the garbage collector is not running,

one semispace is empty and the other contains all the objects in the system. The garbage collector copies all objects reached by a traversal of the *from-space* into the previously empty *to-space*. At the end of the collection, all reachable objects have been copied and compacted into the to-space, and any objects remaining in the from-space are garbage. The roles of the two spaces are now *flipped*, so that the previous to-space will be where all new allocations occur, and the previous from-space will be treated as empty at the next collection.

Unlike mark-and-sweep collection, the frontier of the collection need not be kept as a separate list. If the to-space is treated as a queue of objects, with the least-recently copied object always selected as the next to traverse, the collector need only maintain one additional pointer, dividing the traversed objects from the objects not yet traversed or *scanned*, and the collector can do a complete breadth-first traversal of the live objects. The basic copying collection algorithm is easy to implement, as is the basic mark-and-sweep algorithm.

1.2 Parallel, Incremental, and Real Time Collection

As heaps grow, the simple tracing schemes outlined begin to suffer from the increased scale of the problem. The simple implementations of the collectors assume that the mutator has been stopped — no allocations are occurring, and the structure of the graph of objects is constant throughout the collection. As collections begin to take more time, the users see longer pauses due to garbage collection, the system's responsiveness becomes an issue, and the algorithms need to be modified to allow the mutator to make progress even when a collection is running[3].

Copying garbage collectors can be altered to allow the mutator and the collector to run in parallel[3]. Rather than copying all of the heap at a garbage collection,

[3]Much of the vocabulary of parallel garbage collection is due to Edsger Dijkstra[17], who examined parallel garbage collection as an example of cooperation between seqential processes. His paper is valuable as an outline, and identifies the major concerns of parallel collection, but is concerned with collector correctness more than efficiency.

the collector presents to the mutator the illusion that the copy has been done by use of a *read barrier*. The collector guarantees that after the collection has begun, the mutator will never have a from-space addresses in its registers. The collector's first actions include making to-space copies of all of the objects directly reachable from the mutator's registers, and changing the registers to point to these unscanned copies. Making the copies also involves establishing a path between the old copy and the new — a *forwarding pointer* that will allow a pointer to the old copy to be ued to find the new copy. When the mutator loads a pointer from memory into a register and there is a garbage collection being done in parallel, the collector checks to see if the to-space object addressed has been scanned. If it hasn't been, the collector scans the object and the load returns the pointer contained in the now-scanned to-space copy, guaranteed to be a pointer into to-space. The collector continues to copy other objects as well, to ensure that the collection finishes.

Using stock hardware, the memory-protection facilities can provide the garbage collector with a way to monitor the mutator's loads, but the added expense of the protection violations on most operating systems is large enough to cripple the mutator's response time, and while the mutator can still make progress, the sluggishness due to collection is quite apparent[2, 18].

Using a read barrier in a marking, compacting collector — rather than in a copying collector — is complex[37]. The collector must, as with the copying collector, keep forwarding pointers to find the correct copy of the object if it has been moved for compaction. In addition, when a pointer to an unmarked object is stored into a marked object by the mutator, the collector must be told to visit the unmarked object, since the only pointer to it may be in a marked object. Thus, not only is a read barrier required, but a write barrier is required as well.

Managing parallelism in non-copying, non-compacting collectors is somewhat easier, since the collector and the mutator will never disagree on the identity of an object. Both can use the address of the object throughout the collection, but the connections between objects and the structure of the object-graph can change.

One way to construct a parallel trace-and-sweep collector is to take a snap-shot of the heap in a consistent state and garbage collect over the snap-shot[1, 42]. When

the collection is done the collector will have a list of objects that were free at the point in the past when the snap-shot was taken. A well-behaved mutator cannot access garbage from the past, and so that past garbage will still be garbage. If the allocator and collector are written to use the same data structures, the collector need only make one atomic write and the structure representing the collector's garbage can become the structure representing the allocator's free-list.

Taking the snap-shot need not be as expensive as copying the entire heap — any objects not changed by the mutator can be shared between the mutator and collector. It is only necessary that the collector be able to get the old copy of a changed object when tracing, and the mutator be able to get the new copy.

A slightly more complex way to parallelize a mark-and-sweep collector is to allow the marker to run in parallel as the mutator is changing the heap. When the marker is done, it has marked an approximation of the live objects — some live objects may be unmarked, and some marked objects may be no longer reachable. The live objects that are still unmarked, the *undiscovered objects*, must be found to ensure that the collector does not produce double allocations. For any undiscovered object, there must be a pointer to it from some other live object, since it is alive, and that object must be either undiscovered or be an object altered by the mutator after it was traced by the collector — a *dirty object.*

The collector can now begin trying to catch up with the mutator, starting another marking phase that uses the dirty objects as a new frontier, and marking objects altered by the mutator in the added phase as dirty. If at the end of some marking phase there are no dirty objects, the marking is complete and all reachable objects objects have been marked. It may require cooperation from the scheduler to guarantee that the collector catches up to the mutator eventually[7].

The two methods of making a mark-and-sweep collector parallel are roughly comparable. The interlocking between the mutator and the collector is minor — in both cases, objects changed by the mutator while the collector is running must be given special attention. For the snap-shot collector, these objects need to be copied, requiring extra memory for this collector, but for the catch-up collector these objects need only be noted or marked.

The objects marked by the snap-shot collector are exactly the objects live at the beginning of the collection, while the catch-up collector has marked all of the objects reachable at the end of the collection plus some of the objects that became garbage while the collector was running. But the extra marking done by the catch-up collector has an extra benefit — an object that is created and becomes garbage while the collector is running may be collected by the catch-up collector, but not by the snap-shot collector.

In fact, empirical measurements show that almost all objects created become garbage very soon after their creation, and almost all of the objects created while the collector is running would be garbage by the end of the collection. Fortunately, *generational garbage collection* provides a simple and efficient way to focus collection efforts on these objects. If the objects allocated during the full collection are in the province of a generational collector, the full collection's actions on them are unimportant.

1.3 Generational Garbage Collection

Generational garbage collectors grow from the observation that a large fraction of the memory allocated becomes unreachable garbage soon after it is allocated. If the collector is repeatedly doing *full collections*, naively tracing the entire live heap, it will be using much of its time examining objects that are old and unlikely to have become unreachable. If the collector can focus its efforts on the young objects, it can find almost all the garbage for a fraction of the cost. This is the basis of *generational collection.*

Data gathered on a variety of systems, languages, and applications supports the notion that most objects have short lifetimes. This observation was first made for systems that have a large volume of system- and language-based allocations. An interpreted system, for example, may create a large volume of objects to drive the mechanism of the interpreter, but a large number of these objects are garbage after a signle step of the interpreted program. Even some non-interpreted systems can have large overheads. Some Smalltalk systems create objects for every mouse-move event

and rely on cheap generational collection to clean up after them[4].

Other languages have no allocations but those explicitly included in the code by the programmer, but do not avoid the generational effect[48]. Part of this may be due to common programming techniques. For example, a procedure may produce a volume of related information packaged in a returned object — the individual callers of the procedure may immediately extract the information they desire and discard the returned structure.

A generational collector can be designed using any of the tracing techniques so far described. In the simplest implementations, the objects are partitioned into two classes by age, the young objects and the old objects. A full collection will behave as before, and any garbage in the heap will be collected. A generational collection treats the old objects as if they were part of the root set — the old objects are not transitively traversed, but any young objects reachable from them are.

Only the young objects are *threatened* in a generational collection, but not all the garbage in the threatened set of objects will actually be collected. There may be old, unreachable objects that point to new, unreachable objects. Since the old objects are all treated as reachable roots in the generational traversal of the young objects, some young objects will also falsely be seen as reachable. By carefully choosing the generation gap, the collector can ignore a large fraction of the objects — the old — and still recover almost all the garbage.

Along with the costs previously associated with garbage collection, a generational collector magnifies a cost that previous collectors could usually ignore: the cost of finding the pointers from the non-threatened objects to the threatened objects. In full collections, the entire heap is threatened, and only the global objects, the registers, and the stacks are in the root set. The generational collection may add megabytes of old heap objects to the root set. The first objects marked or copied are those objects pointed to by the roots, but finding those initial objects can now be a large portion of the cost of collection[23].

The search for initial objects is also expensive if there is a large volume of global

[4]Section 3.1 discusses just how quickly objects must become unreachable before they are useful as an identified subset.

and stack memory, even without a generational collector. Multi-threaded and highly recursive programs, for example, may generate large stacks and force finding the roots of the collection to be re-engineered for better real-time response[3].

The most generally applicable method to speed up the search for roots of a collection is to rely on locality effects in computer systems, and cache results of previous searches for roots. If the system has previously produced a list of old objects that point to new objects, there are only a two ways that this list can become invalid: some previously young objects can be re-classified as old, or some old objects can be changed. Both of these events are rare enough in most systems and the caching makes the collector faster and better able to meet real-time requirements. The initial root-finding phase of tracing collection becomes a "cache rebuilding" phase as well. Section 2.1 will detail some of the techniques used to keep the cost of the root-finding phase down.

1.3.1 Multiple Generations

While many implemented generational collectors maintain two classes of objects — young and old — the original generational collector partitioned the objects into many generations by age[28]. The caches kept track of all pointers across age partitions that point from an object to an object younger than itself. In a mostly-functional language like LISP, these pointers were relatively rare, since they are caused by destructive operations like RPLACA, and these operations were avoided in most LISP styles of the time.

The traversal of any individual partition can be started by using the caches to find all the pointers from older partitions, and explicitly scanning the younger partitions. If the younger partitions are being collected at the same time, they need not be explicitly scanned, since they will be traversed. The collector was able to cope with concurrent collections focusing on various regions — a collector unable to perform collections on the youngest objects while a full collection was running would allow a build-up of young garbage that might be unacceptable. The collection of partitions was based on a real-time variant of a two-space copying collector[3].

The mechanisms for partitioning the memory into regions and being able to have the collector work more or less independently in the various regions are orthogonal to generational collectors[5]. The innovation in generational collectors is to use the age of the regions to drive the policy for garbage collections. As the partitions proliferate the policies for deciding when to threaten any particular subset of the objects become complex.

In a multi-threaded system it may be an advantage to collect all of the memory allocated to a particular thread when that thread terminates. Again, the mechanisms in the collector should be robust enough to allow thread-by-thread collection to run without interfering with generational or full collection, and to allow other collection policies to be implemented safely. I will use "generational collection" to refer to collection policies where an age-based cut-off is used to define a threatened set of objects. In general, collectors that partition the heap and threaten some partitions will be called "partial collectors".

1.4 Reference Counting

In the typical taxonomy of memory management, *reference counting* is a sibling to tracing collection. In reference counting, each object has associated with it the number of pointers to that object that currently exist within the memory. When a reference is copied, the count must be incremented, and when a reference is destroyed, the count must be decremented. When the count becomes zero the object is garbage and the memory may be safely recovered[12].

Reference counting has many advantages: it is simple to modify it to get some real-time features[11, 3, 20], the memory and CPU overhead is predictable, and it is simple to implement and maintain. The main drawback usually attributed to reference counting is that it fails to recover garbage that is linked in cycles — if two objects point to one another and no other pointers to either exist, each will have a non-zero reference count and neither will be recovered.

This is exactly the same effect that occurs in partial collectors when a cycle of objects is split across partitions: collecting either partition alone does not collect

either member of the cycle, but threatening both of the partitions at the same time will collect them both. Reference counting can be seen as a degenerate case of partial collection, where each and every object is assigned to its own partition. Collecting any single partition is trivial: the threatened set contains only one object, and examining all of the caches, as embodied in the reference count, finds either that there is a pointer to the object or that there is not. In either of these cases, the traversal of the threatened objects is trivial.

While it did not grow from generational collection, it is useful to view reference counting as a method of keeping caches of information about references up to date. While a partial collector needs to know which objects in a region are reachable from the other other partitions, the reference counter has only one object in a partition, and so needs to know only the count. Many improvements to reference counting are applicable to partial collection as well.

For example, cycles in reference-counted structures can be reclaimed if the system maintains a *group reference count* for a collection of objects that counts references to objects in the group from objects not in the group. Even if no object in the collection has a zero reference, all of the objects in the group can be reclaimed when the group count is zero[6]. In partial collection terms, the collections are the partitions of the heap, and partitioning the heap into logical groups for a partial collector likely to improve performance. When a collection is done on a group of related objects and their reference count would be zero, the collector will find no pointers to the threatened group from outside of it, and the objects will all be collected.

Another improvement is *deferred reference counting*[16]. Many of the changes to objects' reference counts come about from pointers stored into or deleted from the stack and registers. These are much more common than stores and deletions in the heap. The reference counter can take advantage of this by keeping an accurate count of the heap references, but not counting references in the stacks, registers, and other frequently-changed locations. The mutator can be stopped occasionally, and the "hot locations" scanned. Any objects with both a zero reference count and no pointers to them in the hot locations are garbage.

Similar trade-offs are made in maintaining the caches for partial collectors. Some

caches may be easily kept up to date and accurate without great expense by adding code to pointer manipulations, just as there is code that would change the reference counts. Other caches are best re-built with each collection, and the collector may keep just one bit of information to let it know that the cached information is no longer valid.

1.5 Finalization

Sometimes the collection system must be concerned with valuable resources other than address space. For example, many operating systems limit the number of files a single process can keep open at a single time, and the file system itself may have a limit on the number of open files. Much as memory management can be a difficult chore for the application writer, so can management of these other resources. It may be difficult for an application to know that it has no further need for a particular open file.

Finalization is a part of the client interface to the collector that allows other resources to be managed exactly the same way as memory by associating the resource with some storage allocated by the user[22].

The simplest form of finalization simply notifies the user process when an object associated with a resource is collected. Finalization is associated with *weak pointers* — a special kind of pointer with added semantics. If an object is only reachable by paths involving weak pointers, the collection system recycles the object, and zeros the weak pointers to ensure that there are no pointers to deallocated space. To get finalization from weak pointers, we need only add notification. When the collection system zeros a weak pointer, it sends a message to a user object, and that object may deal with finalization. Since the user is notified when the finalizable object is collected, any data needed for finalization must be kept somewhere other than the finalizable object[31].

Figure 1.1 shows an example of this style of finalization. When the set of objects that implement a file become unreachable, the file manager would like to flush the file and then close it. Typically, this is done by adding a level of indirection. All

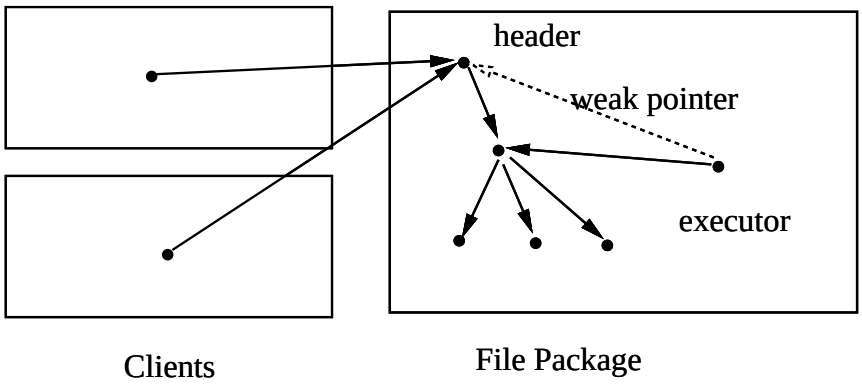

Figure 1.1: "Last Rites" Finalization

normal users of the file have a pointer to an open-file header, and from that header the rest of the file's functionality is available. However, the file manager maintains a pointer that bypasses the header and a weak pointer to the header. After all of the clients of the file manager have deleted their pointers to the file, the header will be collected. At that point, the weak pointer is disabled and the collector notifies the file manager. The file manager cannot dereference the disabled weak pointer, but can use the bypass pointer to flush and close the file. This style of finalization is called "executor finalization," since some process in the system is notified of an object's "death" and expected to execute its "will."

Another style of finalization involves notifying the user when an object has become "almost unreachable." This is sometimes called "resurrection semantics" because it can appear to some processes that an unreachable object is given back to the user by the collector. This is misleading, since the object isn't unreachable, just a bit out of the way. In this variation, the user identifies some pointers to be ignored by the collector. When every path to a finalizable object from a root includes an ignored pointer, the object is almost unreachable, and interested processes are informed[33]. Unlike weak pointers, these ignored pointers are not zeroed by the collector, nor is the object's memory recycled.

Figure 1.2 shows a file manager using these semantics. To get finalizable files with this semantics, the file manager would keep one or more pointers to the file, but would designate them as *package pointers* to the collection system. When all of the

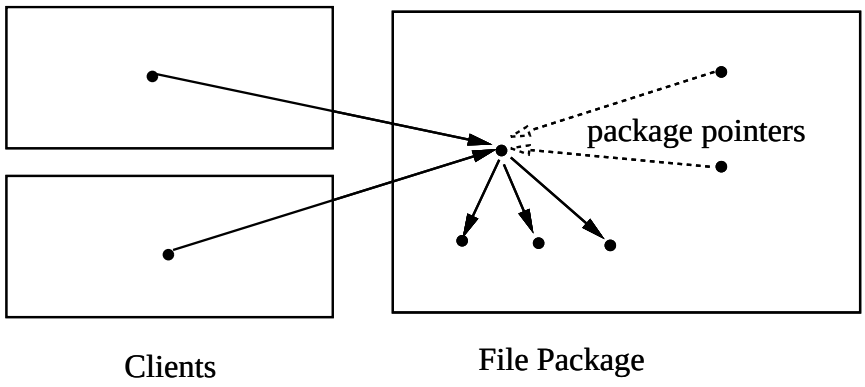

Figure 1.2: "Almost Unreachable" Finalization

clients of the file manager have deleted their pointers to a file, the package pointers remain. Since every path to the object is through a package pointer, the collection system will notify the file manager that the object is barely reachable. The manager may then flush and close the file.

Under either semantics the results are the same. The collection system for address space can help manage other resources, even user-defined resources. In Section 4.4 the use of "almost unreachable" objects will surface again as a solution to some problems with key objects.

1.5.1 Promptness

Managing user-defined structures through finalization brings the problem of *promptness* into sharp relief. When a resource is recycled soon after it is no longer needed, the management of that resource is said to be "prompt." For example, since full reference counting finds unreachable memory as soon as the last pointer to it is deleted, it is prompt for all collected objects[5].

Full tracing collectors only find unreachable storage when they are run, and so are less prompt than reference counters. Partial collectors have differing degrees of promptness for different objects. Objects that the collector focuses on often will be collected more promptly than objects examined less frequently. In addition, cycles of

[5]Of course, circular structures are never recovered, so it's only prompt when it works.

objects that cross partition boundaries might be collected less promptly than objects that do not, since the cycle can only be collected when all objects involved in the cycle are threatened at the same time.

Since memory is reasonably abundant and fungible, promptness of memory collection is less of an issue than promptness of finalization of other more scarce resources. Unfortunately, neither promptness nor finalization has been addressed adequately by current research, and there are no systems that allow users and applications to specify the importance of promptness across classes of objects.

1.6 Garbage Collector Design

There is no "best" garbage collection technique — like most system design problems, there is a large design space for garbage collectors and the choice of technique depends a great deal on the properties of the input. As the volume of storage grows, the design of the collection system may become more complex, with the collector choosing one technique for one sub-population of the objects and a different technique for another.

For example, when the volume of threatened objects is small and the collector expects that most of them are garbage, a copying collection seems the reasonable choice — the live objects are copied, and the garbage can be returned to the system without attending to each garbage object in turn. But copying techniques are less attractive when the volume of threatened objects is large and most of the threatened objects survive — this would require a large to-space, making the space use inefficient.

In addition, the collection with a large volume of survivors will take a long time, and many mutators would require parallel or incremental collection to meet response requirements. This in turn puts burdens on the mutator, and that alone may swing the choice on collection style.

The usual pattern is that a large fraction of the objects in the heap have unpredictable lifetimes, and the policy used to collect them involves occasional collections over the entire heap. These collections must be frequent enough to satisfy promptness constraints in the system, but as the heaps grow larger and larger these collections begin to take more and more time.

The best way to avoid the overhead of these collections is to look for subsets of the heap objects that have more predictable lifetimes. These objects can be segregated and a special-case memory management policy can be designed for these subsets. Among garbage-collected objects, the two most important subsets discovered to date are the permanent objects and the short-lived objects.

1.6.1 Special Cases in the Heap

An obvious improvement to full collections over all objects involves segregating all objects that will be required for the entire lifetime of the application, and will never be collected. Even if the permanent objects are just the system objects and no user objects are recognized as permanent, this can be a large improvement. The permanent objects can still be threatened at the user's request, but by not exposing them to the usual treatment the garbage collection overhead can drop significantly.

As noted in Section 1.3, generational collectors address the short-lived objects as a special case. Recently-created objects are collected more frequently than the rest of the heap. The cost of these collections is independent of the total size of the heap, and can still find a large fraction of the garbage soon after it becomes unreachable. When an object has survived the initial generational collections, it is reclassified.

As heaps grow, collectors must mature and find more special cases to carve out of the idiosyncratic storage patterns. The collection methods for these may be mixed, involving copying, marking, and reference counting when most appropriate, and collection mechanisms will incorporate more tools from the bag of tricks that has been developed in the past.

1.7 Contributions of the Thesis

Measurements of object life-time distribution in Cedar shows the rapid decrease in value of lifetime information. Objects that are newly-created have a much shorter expected lifetime than older objects, but even a short wait, allowing another half Mb of storage to be allocated, will usually tell if the object will become unreachable

quickly. There is little value in attempting generational collection past that point. The first contribution of this dissertation is establishing this decrease in value.

The central contribution of this thesis is the introduction of key object opportunism as a policy decision to address scheduling of collections. As more special cases are identified in a partitioned heap, a unified way of triggering the collection of the partitions is needed. If the heap is partitioned into loosely connected components, where all of the objects in a component have roughly the same lifetime, the components can be scheduled efficiently by waiting for a carefully selected subset of the component to become unreachable and only then collecting the whole component. This opens the heap to any number of system- and application-specific specializations.

1.8 Organization of this Document

Problems particular to collecting large heaps are introduced in Section 2. Mechanisms to address problems in large heaps are discussed in Bishop's dissertation [5], but policy is not addressed by that work. Section 3 presents measurements that suggest the limitations of generational collection. Section 4 introduces key objects and key object opportunism. Experimental evidence in support of key object opportunism is given in Section 5.

Chapter 2

Collecting Large Memories

One issue that soon comes up in design of a collection system is the scaling of the collector to larger and larger memories. There is a limit to how much memory can be traced while interactive mutators are paused — pauses over about 500 milliseconds are disruptive. Once the traced portion of the heap grows beyond that point, the collector must be able to access the heap in parallel with the mutator in order to maintain interactive response.

Beyond the simple matter of responsiveness, a tracing collector has to worry about the overhead costs it imposes on a large system. If it collects too frequently, it may trace large volumes of objects and recover very little each time — if it collects infrequently, large volumes of memory will become filled with unreachable objects.

If memory were uniform in access time and cost, a tracing collector could keep its total overhead limited by triggering full collections when the heap had grown some multiplicative fraction larger than the heap's total volume at the previous full garbage collection. For example, if in the steady state the heap contains 100 megabytes of live storage, and it allows the heap to grow to 125 megabytes before triggering a full collection, the collector will recover 25 megabytes having traced 100 megabytes — it had to trace four bytes for every byte recovered. If instead it allows the heap to double to 200 megabytes before collecting, it will recover 100 megabytes having traced 100 megabytes — a one-to-one ratio. By fixing the cost in bytes swept that the system would be willing to pay for each byte recovered, the cost of tracing collections can be

limited by fixing the collection triggering threshold.

Almost all of the assumptions involved in this calculation are false: memory is non-uniform in access time and cost, and the heap does not usually maintain a steady state. The collector is competing with the mutator for the right to use the fast main memory and cache, and the costs of moving objects up and down the memory hierarchy begins to dominate. Even if giving up 30% of the main memory for fast, effective, frequent, generational scavenges is a good idea, it doesn't mean that giving up 30% of the disks for long-running, infrequent, full collections is[1].

The most common suggestion for handling the scaling problems is to partition the allocated memory into parts that can be collected in a reasonable time given the hardware. This also avoids some of the problems introduced by having a memory hierarchy. The partitions can be sized so that each one fits comfortably into the appropriate cache, and the bottlenecks associated with moving data back and forth between different levels in the memory hierarchy can be avoided.

2.1 Finding Rescuing Pointers

As mentioned in Section 1.3, a generational collector pays an increased cost in finding the roots of its young partition. It cannot afford to scan all of the old space for pointers to new space, and needs to cache the results of previous searches and update them. This section will explore some of the methods for finding the inter-generational pointer information in a collector.

Recall that all objects not being collected — the older objects — are assumed to be reachable and not garbage, and these objects are treated as roots. The collector needs to find all of the *rescuing pointers* from the old objects to all of the young objects in order to begin a traversal of the young objects. If the rescuing pointers are in a small fraction of the non-threatened objects, caching the results of a scan of the non-threatened objects should be an effective way of finding the pointers from the

[1]White [42] goes to the extreme on this topic and suggests that garbage collections on large address spaces should occur as infrequently as possible — perhaps once each year — and be supported by enormous volumes of secondary and tertiary memory. Memory may be cheap, but it doesn't seem to be cheap enough for this scheme yet.

older objects to the younger objects.

The cache of rescuing pointers can become inaccurate only when a pointer is stored in an old object or a new object is promoted. New cross-area pointers may have been created, and old ones may have been deleted. The problems involved in keeping an accurate set of rescuing pointers are nearly the same as those involved in building reference counted systems, and the solutions proposed are nearly the same[23].

The system can interpret each store and keep the cross-area pointer cache always accurate. This is much like keeping fully accurate reference counts for just the young objects — note that if the collector had the inter-area reference counts for young objects, that's all it would need to start a generational garbage collection. It is possible to build into the system a *store-trap* so that when a pointer to a young object is stored into an old object, the collection system can gain control long enough to make a note of the interesting pointer. Since most old objects are not frequently changed, trapped stores would not be common, and special purpose hardware seems uncalled for — it would be unused almost all of the time. Likewise, a software check that is executed on every store may be expensive and would not often find changes to the cache.

Systems can use standard page protection mechanisms to trap all stores to the old objects. Any page that contains an old object can be write-protected, and a store to these pages can be interpreted to see if it changes the cross-area pointers. This has the benefits of the store-trap, but uses no special hardware, and does not delay the write if the fault is not taken. Unfortunately, most operating systems do not make user control of the protection mechanism cheap. To take a write-fault, note all the changes to the inter-area pointers, interpret the write, and re-set the protection can take thousands of instructions if that path in the operating system is unoptimized.

Another method is to take advantage of the locality of the writes, and simply invalidate the parts of the cache that pertain to a page when it is first written to, faulting only once for any given page. At some point the collection system will have to rebuild the cache of rescuing pointers, but it need only rebuild the parts of the cache that have been invalidated. The effort to rebuild may be smaller than would be needed to scan the entire heap, since only targets of stores to non-threatened objects

have been invalidated. In addition, the collector can attempt to gather the frequently written objects on a few pages, and not even bother with protecting them.

Finding an optimal bookkeeping system for the cross-generation pointers depends on many things including the cost of write-protecting pages, the cost of taking and recovering from write-protection violations, the frequency of these writes, and the cost of rebuilding the cache from a dirtied page.

The important commonality of these systems is that having determined a threatened set the collection system can construct the set of pointers from the old objects to the new objects.

2.2 Partial Collection

By noting the points that make it effective to focus collection efforts on young objects and looking for other classes of objects that meet these same criteria generational collection can be generalized :

- There are few pointers from objects outside the class to objects inside the class.
- The objects in the class have the same expected collection time.

The first ensures that the bookkeeping for the cross-class pointers is not too complex, and the second ensures that at some point the area can be collected with a large fraction of the storage getting recycled.

In general, a system can divide the memory into logical areas, and be able to collect these areas more or less independently. The areas need not be physical, in that objects from different areas may share disk pages, but many of the algorithms are easier to envision if we assume that the logical areas are physically realized, and that simple examination of a pointer value will tell what area the pointer's referent is in.

2.2.1 Restricted Partial Collections

If the system retains the flexibility to collect one or more areas at a time, it may be that collection of some area will only occur in conjunction with collection of another.

For example, several gradations of time-based areas might exist, with objects moving from the youngest to the oldest in their lifetime. Collections would be most frequent in the youngest generation and least frequent in the oldest, but collection of any area might only occur in conjunction with collection of all younger areas.

If the collection system will never collect area A without collecting area B as well, then it will never need to know if there are any pointers from objects in B to objects in A, and it need not make this information easy to recover. Conversely, if it would be difficult to find the pointers from objects in B to objects in A, the system need not be concerned if it is willing to always collect B when it collects A.

Two systems that explore partial collection are the ORSLA memory system [5] and the A/F collector [14]. The ORSLA system is more concerned with being able to collect relatively static areas even when an active process has pointers into those areas without keeping track of all of the cross-area links explicitly. It notes dynamicly when a *local computation area* of a process has pointers to a static area by creating a mono-directional *cable* from the LCA to the static area. The cable tells the system that it need not keep track of pointers from the LCA to the static area, and that when the static area is collected the LCAs cabled to it must be collected at the same time.

The A/F system formalizes what cross-area information a collection system must keep and what can be lost if collection of one area implies collection of another. The flavor of this system is more static. The areas are put in a partial order and collection of one area requires collection of all areas above it in the partial order. Cross-area pointers can be ignored, then, if the tail of the pointer is in an area below — in the sense of the partial order — the area containing the object at the head of the cross-area pointer.

The most valuable contribution of the A/F system is that it shows how various views of the objects can be combined and what cross-area information must be maintained when views are combined. For example, it is a good idea to be able to collect the young objects, all else being equal. It might turn out to be a good idea to be able to collect all of the objects created by a single process as well, regardless of age. Under the A/F system, these two collection schemes can be represented as two partial

orders, and a new collection scheme can be constructed which allows either collection by age or collection by creating process.

2.3 Full Collections

There is not enough experience with large persistent object stores to know how much of a problem full collections would be. When a large heap begins to be used as a file system, it may be that, like a file system, objects never are deallocated but just moved between different media. Most file systems include an ability to restore "unreachable" files by using archive tapes or write-once memory. A persistent object store, when used as a file system, would need to duplicate this feature. Once an object is part of a file, there may be no need to try to collect it if the semantics of the file system guarantees its continued existence. The memory manager should include a way to make these objects permanent.

In order to eliminate unreachable storage, there is no substitute for full collection. No set of partial collections will be able to guarantee recovery of all of the unreachable objects, since each partial collection must expand the root-set of the collection, and the expanded root-set may contain unreachable objects, which may in turn contain pointers to other unreachable objects. The partial collections may attempt to collapse unreachable loops into the same partition by moving objects from one partition to another[4, 24][2]. When all of the unreachable objects in a clique are in the same partition, then a partial collection can recover the storage.

Partial collections can be used as helpful stepping-stones in full collections. As soon as a full collection establishes that the set of true roots is a superset of some set of roots used for a partial collection, the results of that partial collection can be accepted, provided that none of the objects collected by the partial collection have been changed.

A simple example of this arises with multiple collections, both full and partial,

[2]This may be in conflict with other goals of the system. If there are multiple processors, moving related objects into the same partition may cause page-sharing to increase and shared pages to thrash. The collector is not the only system consideration in placing objects in memory.

over a file system. Most file systems are essentially tree-structured, with few access paths to a file. Each file is in one directory, or perhaps a small number of directories. The data in most files are not changed often. If a previous partial collection has found a subtree that is rooted at a node and no elements of the subtree have been altered since the partial collection, the full collection need not trace the entire subtree once it finds the node reachable. The full collection need only touch a small fraction of the objects.

2.4 Benefits of Areas

Areas can be optimized to meet goals other than ease of garbage collection. Groups of objects may have common intentional or extensional features that are of interest. For example, access to some objects by some users may be restricted, or objects may use a non-standard byte order or floating point format. Objects in an area may all have the same type, or may have all been allocated with the same language-specific allocator. Keeping those objects in isolated physical areas allows other system components to check for these object features with a simple pointer range check.

There are other reasons to gather certain objects together. One of the most compelling is that access patterns of objects tend to be grouped. Knowing that a process has read or written one object in a cluster of related objects suggests that other objects related either logically or by a chain of pointers might soon be accessed. The system has many opportunities to notice the formation of these working sets, and can try to treat them as connected units. All of the pages in such a unit can be swapped in when any one of them is requested, and if the collection of objects is well-constructed, this would decrease paging delays.

2.5 Areas in Programming Languages

Areas have been implemented as language-level structures, visible to the programmer, but different models for areas are implemented for different styles of languages. In PL/I, storage management is the responsibility of the user, with explicit freeing of

data rather than garbage collection[27, 25]. Areas are regions of memory that are specified when an object is allocated, and the memory for that object will be take from the named area or a default area. The programmer, rather than explicitly freeing each object in an area, can free all of the objects in an area in one operation, thus avoiding the cost of the individual freeing operations and the complexity of walking the data structures. Of course, the same dangling pointer problems are present here as in explicit storage management without areas.

In some Lisp dialects, areas are available with the same kind of allocation semantics as in PL/I[30]. Objects are allocated in an area if one is specified or a default area otherwise, but the deallocation is done by a garbage collection system rather than an explicit free. Areas are mostly a way to let the user ensure that pointers tend to be local for paging, and effectiveness of garbage collection is a secondary benefit.

2.6 Age-based Areas

While it might seem to be difficult to find a good division of the objects into areas, there is evidence that the information present in the program execution is useful for predicting locality [10] as well as lifetime[21, 41]. Objects allocated nearby in time tend to have pointers to one another and tend to have roughly the same lifetimes. This should come as no surprise, considering that programs are often designed to build a single interconnected data structure at a time, moving on only when the structure is complete.

Chapter 3

Effectiveness of Generations

As mentioned in Section 2.1, simple generational collection has proven to be a benefit in many situations. By segregating the young objects from the old objects, almost all of the unreachable storage can be reclaimed with almost none of the effort. The success of generational collection is due to two measurable facts:

- Most objects that become unreachable do so very soon after they are created.
- It is easy to keep track of the pointers from old objects to young objects.

These facts are usually verified indirectly, by building a generational collector and seeing the improvement in collector efficiency[28, 30, 40].

The young generations in a collection system offer flexibility to the system designer. Some static analysis at compile time might be able to lower the allocation volume of an application by changing dynamic allocations to stack or static allocations. Inter-procedural analysis would even allow the compiler to get rid of objects that are created to return composite values, only to be dismantled and discarded by the caller, but efficient generational collection makes this analysis less pressing.

The lifetimes of other objects will be dynamically dependent on the input data and not open to analysis. The young generations will pick up many of the objects that are not easily analyzed. This trade-off of static and dynamic analysis is analogous to the analysis involved in static register allocation, where the hot locations that perhaps

should be in registers end up in the cache instead. Things need not be that much slower, and the analysis can be limited.

The effect of youngest generations is strongly influenced by the language, style, and system used by the mutators. In languages where objects are created as part of the underlying computational model or of a hidden mechanism like a Lisp interpreter, the volume of objects unreachable soon after creation will be quite high for any mutator. Languages like C and Cedar have no computational allocation — every allocation is at the user's request. The allocation volumes are lower in these languages.

Since all of these languages offer roughly the same data model, most of the difference in allocation volume comes from the computational allocation. When the language-dependent creations are picked up by the young generations, the lifetime distribution of the remaining objects is based on the data structures of the task at hand, rather than the implementation language.

3.1 Multiple Generations

For any tenuring policy in a simple two-level generational collector, there will be some objects that live long enough to be tenured, and will become unreachable after the initial generational period. Collection policy among tenured objects has only recently begun to receive direct attention[43, 24].

Many collection systems have been built around a generalization of generational collection. As objects age, they go through a series of generations, each with an age-based tenuring threshold. The older generations are threatened less and less frequently compared to the younger generations.

This involves two tacit assumptions parallel to the two that make generational collection so successful:

- For any age break, relatively younger objects become unreachable at a rate faster that relatively older objects.
- It is easy to keep track of the pointers from relatively old objects to relatively young objects.

These two assumptions constitute the *strong generational hypotheses*, in contrast to the *weak generational hypotheses*, where these are restricted to the newly created objects. Neither of these necessary conditions for the success of multi-generational collectors has been well documented. Some studies have been published that are useful in estimating the applicability of the strong generational hypotheses[34, 15, 47].

To validate the strong generational hypotheses, it is necessary to measure object creation over a long period of time. Enough objects must be survive the initial generational collection, only to become unreachable later, to gather any meaningful statistics or make meaningful projections. Benchmarks traditionally used for garbage collection focus on generational collection, and are fairly short — collectors with large tenuring thresholds can run the entire suite without having any objects tenured.

3.1.1 Permanent Object Creation

Many mutators create some objects that are meant to survive for a long time relative to the span of time used for plotting the decay rates. These objects may be part of the internal history of the mutator, useful for undoing commands, recovering from errors, or just internal tracking. Other long-lived objects may be the results of requests from users, and will have life-times akin to the lifetimes of files.

These "permanent" objects limit the collection rates. When permanent objects make up a large fraction of a generation, then the survival rate for that generation cannot be smaller than the volume of these objects divided by the volume of the generation.

Some of the studies of object lifetimes have explicitly ignored objects that live longer than the sample time[15, 21, 41], and others use only benchmarks that create no permanent objects[49]. By its nature, it is hard to measure the creation rate of permanent objects, since any measurement is limited in duration, and perhaps the objects would be collected if the traces collected were only slightly longer. But for any non-zero creation rate of permanent objects, we cannot expect to get high collection rates generation after generation. Even four generations with a collection rate of 80% produce an overall survival rate of .16%. A permanent object creation rate above

that rules out the possibility of an four generation collector with only 20% survival rates at each generation.

3.2 Object Collection Rates and Half-lifes

Object collection and survival rate curves have been published for Lisp [34], Cedar [21], Modula-2 [15], and Smalltalk [41]. The curves are difficult to compare. One source of confusion is the different metrics used by the various sources to measure the age of objects. While some use wall-clock time, most use mutator allocation rate to age objects. The latter has many advantages over the former, perhaps foremost that it takes account of processor speed in aging objects. Allocation-bound mutators will have age curves independent of processor speed.

Another problem in comparing the curves is that most presentations compress the time-scale by showing the data logarithmically. While this is a natural way to compress the data, it interferes with the ability to compare the actual survival rates with the rates that would be seen if age had no predictive ability. If age does not predict lifetime the survival function of fraction surviving by age will be exponentially decaying, with some half-life, and the plot would be an exponential curve.

By plotting the data with a logarithmic time scale, the shape expected if age had no predictive value is not a simple line or exponential curve. The actual data shows a shape similar to that of a curve when lifetime has no predictive value, but it doesn't seem to be easy to visually compare curves of this shape. When data is presented this way, it is hard to tell if age has any predictive value whatever.

Presentations of the survival-rate data also sometimes show the data cumulatively as the fraction of total memory that survives past some threshold. This has the property of guaranteeing that the plot is monotonic, and the tendency to make the curves continuous, but has beguiled researchers into thinking of the half-life of objects as changing smoothly and monotonicly with object age.

The statistic of interest is not the cumulative surviving fraction, but the *instantaneous half-life*. When age is not a factor in predicting the lifetimes of objects, this is simply a flat line. It can be derived from same data as the cumulative surviving

fraction. For every age where objects are collected, T, assume that T is the tenuring threshold for the next, older generation, and that these objects are the last to be collected by the previous, younger generation. Now look for the first few objects to be collected in the older generation. They have survived in the old generation for additional time ΔT.

If the instantaneous half-life at time T is h_T, and the volume of objects that survive to be promoted is V_T, then the volume of objects that we should expect at time $T + \Delta T$ is $V_{T+\Delta T} = V_T(1/2)^{\Delta T/h_T}$. Taking the logarithm base $1/2$ and solving for h_T, the half-life at time T is seen to be $\Delta T/\Delta \log_{1/2} V$. This is the time we could expect to wait for half of the older generation to be collected if its decay rate is reflected by the early decays.

Generational collection shows us that for many cases, in fact, the instantaneous half-life curve must rise, and the fraction surviving must be steeper than exponential — generational collection is effective when the half-life rate for the young objects is shorter than that of the old objects. There is, among the young objects, a much higher decay rate than there is for the older objects, and by simply plotting the instantaneous half-life, any such patterns will be evident.

3.3 Collection Rates in CEDAR

Over a period of four months, the code in ten Dorados [32] at Xerox PARC was modified to record object allocation and deallocation. Not all machines were enabled at all times. The mutator code comprises a wide variety of applications including compilers, mail and network servers, editors for graphics and text, debuggers, file servers, and window managers [39]. All users incorporated a large number of applications in their daily lives, and storage traces covering more than a few hours usually include allocations from many different tools.

Nearly all of the mutator code is written in Cedar, a Modula-like language developed at Xerox. Dynamic storage is an integral part of the language, but it is not used as part of the primitive operation set. All object allocations are explicitly included in the Cedar code.

Deallocation is done by a deferred reference-counting collector [33]. Pointers to allocated objects can be unambiguously identified except for those pointers on the stacks and in the registers. These regions are treated conservatively, so a bit-pattern on the stack that looks like a pointer to an allocated object will keep the object from being deallocated[8].

Care was taken to ensure that the monitoring code would not interfere with the performance of the mutator. Some slowdown was inevitable, but most users reported little interference with their work. One user provided several long graphical editing sessions, but while watching him at work we decided that the response time of the editor had dropped below an acceptable level, and it would interfere with his work to ask for more samples. Other users may have biased their work load to avoid some tasks while the monitoring was installed.

Dorado users invoked monitoring code after the machine was booted and fundamental modules were already running. Objects created before the monitoring began did not have their creations recorded. These objects can produce deallocation records without having produced allocation records. Likewise, objects created while the monitoring code is running but that survive past system shut-down have allocation records, but no deallocation record. Some objects have neither an allocation record nor a deallocation record. These objects were created before the monitoring began and survived until the system was shut down.

To determine lifetimes of objects, a cohort of objects was taken from each sample. All objects except those in the last 5 Mb allocated in the sample are in the cohort. The age of any object in the cohort can be determined up to the age of 5 Mb, or kept as "greater than 5 Mb." Some samples did not contain 5 Mb of allocated data, and so yielded no results. Creations, deletions, and thus lifetimes are only accurate to the granularity of the garbage collections. Any objects that survive the same set of collections will be given the same age. This is probably accurate to within about 32 kilobytes.

To limit any particular user's effect, the two largest samples from each machine will be used in this section — a range in volume allocated of about 60 Mb to 1 Mb. Three of these twenty samples are smaller than 5 Mb, and so cannot be used to find

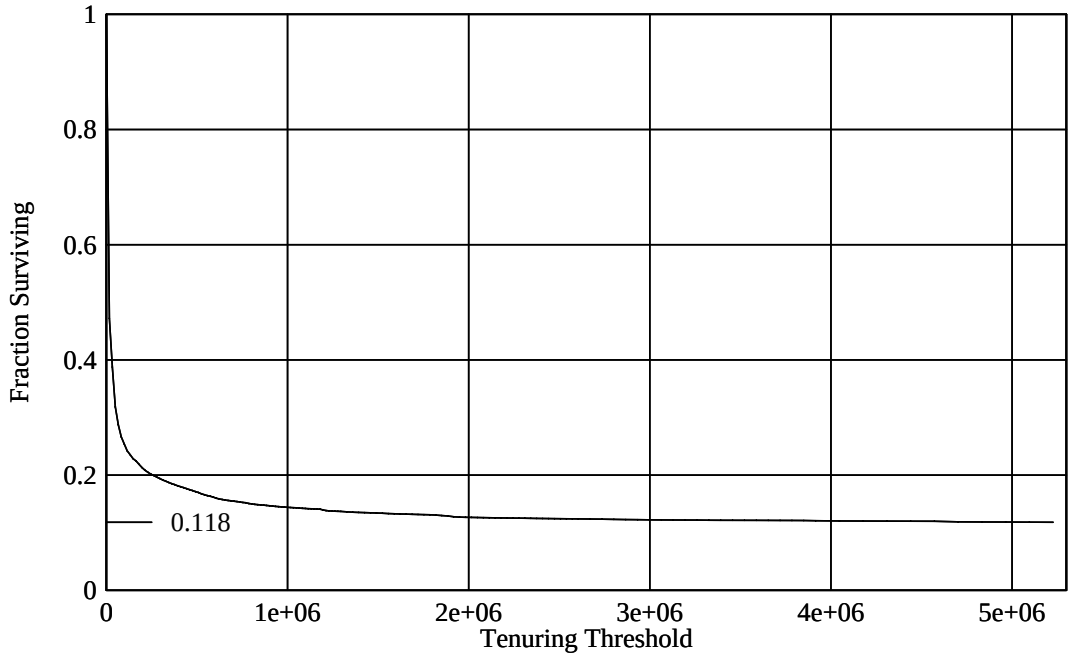

Figure 3.1: Cumulative Survival of Bytes

objects living longer than 5 Mb. The smallest acceptable sample is just 6 Mb, and so generates only about 1 Mb of allocations whose life-times can be tracked. The total is about 262 Mb of allocated objects. This section will present data that is a balanced sum of all the samples that produced more than 5 Mb of allocated storage – the 6 Mb sample will be given the same weight as the 60 Mb sample[1].

Figure 3.1 shows the cumulative decay of the allocated storage as a function of age. Any particular byte of allocated storage has about a 10% chance of surviving after 5 Mb more has been allocated. It is difficult to tell from a graph such as this how flat the curve is getting and it is hard to project what fraction of the objects would be permanent and uncollected if the applications were run for a very long time.

This curve is sometimes mistakenly called "exponential," since it has the fast drop-off and long tail often associated with an exponential curve, but even a quick examination shows that this curve is not exponential. Exponential curves display the same fractional drop across the same X-distance anywhere along the curve — the ratio of the values is the same if the difference in X is the same.

[1]Appendix A includes details about each sample.

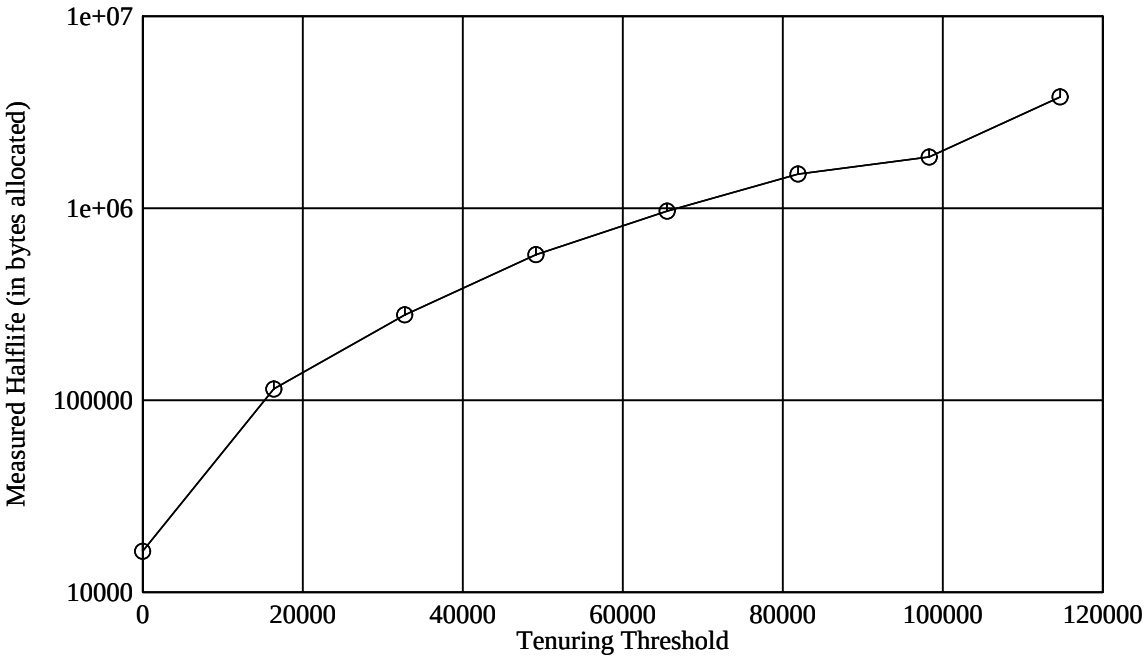

Figure 3.2: Half-life of Allocated Storage by Age

Figure 3.1 shows no constant half-life. The first 50% of the objects are collected in under 16 kb of allocation. The remaining 50% of the objects have a half-life of about 48 kb. Half of the hardy 25% of the objects, which have survived the first 16 kb and the second 48 kb, survive another 816 kb of allocations. This puts the data well on the flat section of the curve, and another 4.3 Mb of allocation brings us to the 5 Mb mark, where we can no longer gauge the age of objects. But the oldest 12.5% of the objects has only been reduced to 9.8% of the initial total. The last 4.3 Mb has only removed about 1 byte in 5.

Figure 3.2 shows the half-life of allocated objects as a function of age. Since fewer than $2 \times 9.8\% = 19.6\%$ of the allocated bytes survive 176 kb of allocation, the half-life of objects that have aged more than that can not be found in the data — more than half the bytes that survive 176 kb of allocations survive past the 5 Mb horizon.

However, the survival curve from Figure 3.1 can be directly translated into a curve showing instantaneous half-life, Figure 3.3. Since there are few objects that are collected after a very long time, ΔT is allowed to get larger over time, to smooth the plot.

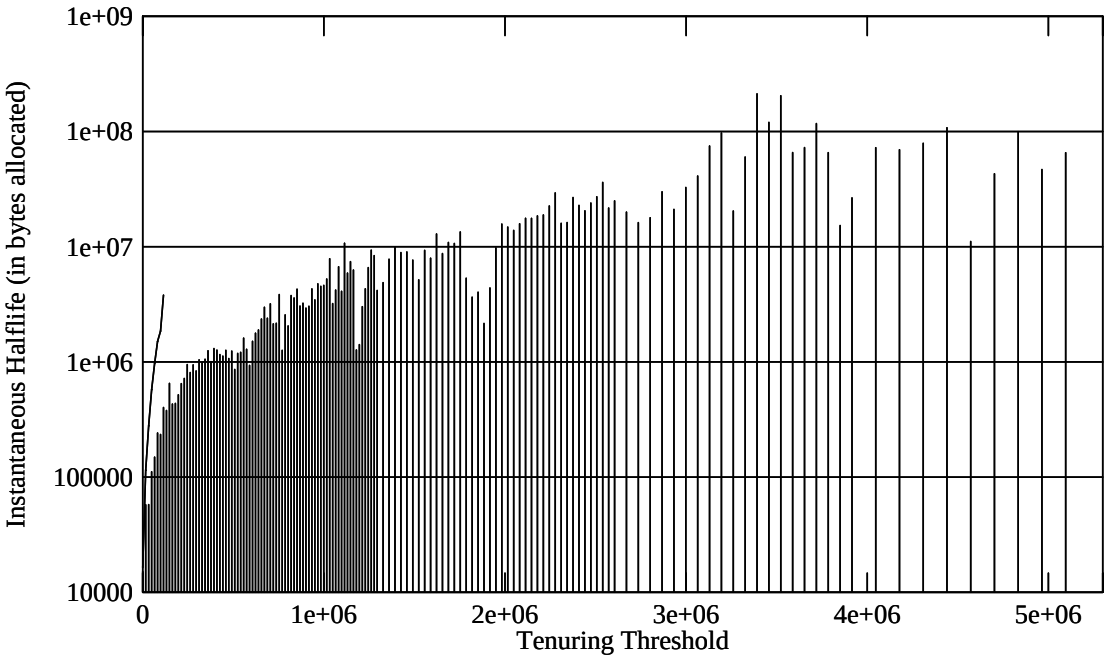

Figure 3.3: Instantaneous Half-life of Allocated Storage by Age

Figure 3.3 also incorporates Figure 3.2, the real half-life seen as the sharply rising line on the left. The real half-life is always greater than the calculated instantaneous half-life because the later assumes that the present half-life will continue into the future, but the half-life rises as objects age. When the future instantaneous half-life is larger than the present, as it must be if generational collection should be used, the instantaneous half-life is a low estimate of the true half-life.

The half-life can be scaled to find the waiting period for fractional decays other than 1/2. For example, if all of the bytes in a generation are 2 Mb old, the half-life is about 10 Mb of base level allocation, as taken from Figure 3.3. That means that after 30 Mb, 12.5% of the bytes can be expected to remain if the current decay rate continues to apply to those bytes.

The half-life is tied to the base-level allocation rate in the Cedar language and system, and differing results would be obtained for other languages or systems. But if the same application using the same data structures were written in two different languages, the language used would not effect the collection rates of the algorithm's data structures relative to one another — only the rates relative to the allocations

required by the language would change.

If almost all of the differences between the languages are in the short-lived objects, and the early generations of a multi-generational collector would remove the language-dependent differences in the allocations, then the later generations would contain only the data structures required by the algorithm. Of course, the early generations will also pick up short-lived objects that come from the algorithm. By looking farther and farther down the curves, the allocations related the language make up a smaller fraction of the remaining objects, and the allocation structures inherent in the algorithms dominates.

When the half-life is constant, the age of an object is not a good predictor of its remaining lifetime. When the half-life is increasing, as in the early parts of Figure 3.3, collecting the young objects and old objects at differing rates might be a good idea. If the half-life is decreasing, the older objects are being collected at a faster rate than the young, and generational collection would be a poor idea. But while the half-life shows a marked increasing tendency in the young objects, the tendency among the older objects is less clear. In addition, the half-life itself has increased substantially. The next section will develop generation size as a measure of the difference between the young and not-so-young objects.

3.4 Generation Size

By using allocation rate as a measure of time, the half-life curve is tied to the language. The dependency can be eliminated by correcting the collection rate at every age for the *arrival rate*, the rate at which bytes are promoted past the threshold age. The most convenient measure for expressing this is generation size.

Figure 3.4 shows a simple model of a cascading generational collector, centered on one generation. The previous generation has just been collected, and so is entirely empty. The current generation is full, but some of it is unreachable. If this generation has not been threatened by a collection recently, the unreachable objects have not yet been discovered. The next generation also contains a mixture of reachable and unreachable objects, but has not yet been filled to its capacity. The current generation,

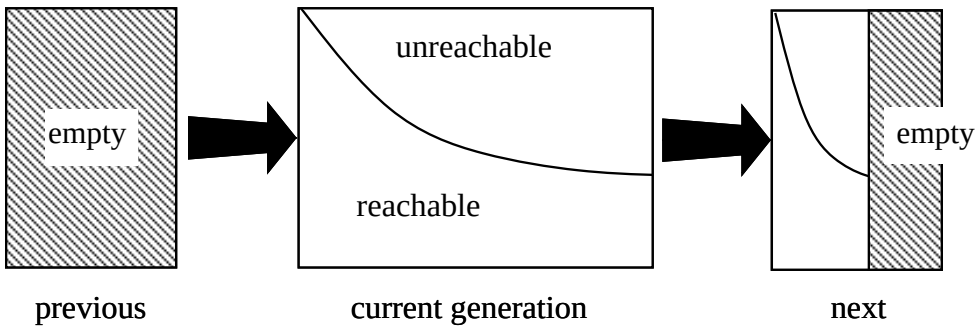

Figure 3.4: A Cascade of Generations

since it contains no empty space, will be collected soon.

The earlier generations, when collected, will promote to the next generation only objects that have survived T bytes of allocation. For every byte that is allocated, some fraction of a byte, a, survives T more allocations to arrive at the new generation.

Arrival from the earlier generation continues until the current generation is full. If its volume is V, the current generation will be full after $\frac{V}{a}$ bytes have been allocated at the top level. The objects in the generation have been aging for a range of time — the newest arrivals have been there for no time at all, and the oldest have been there for $\frac{V}{a}$ allocations.

Notice that we are taking some liberty by assuming that the survivors from the previous generation are promoted after they have survived some length of time, but the current generation is being collected when it is full. The two descriptions of the tenuring policy do not differ by much if each generation is collected many times before the next generation fills. If this were not the case and the two generations were collected at comparable rates, there would be no reason to have two generations.

The half-life h at time T can be translated into a retention rate, r by using the half-life formula, $r^h = 1/2$. The rate r is the fraction of a byte that will remain for each byte allocated at the top level. If this retention rate at time T continues to hold at later times, the fraction of bytes that will survive a collection of the older generation is

$$F = \frac{a}{V} \int_0^{\frac{V}{a}} r^t dt.$$

The charts indicate that, in fact, the half-life rises with time, and thus the retention

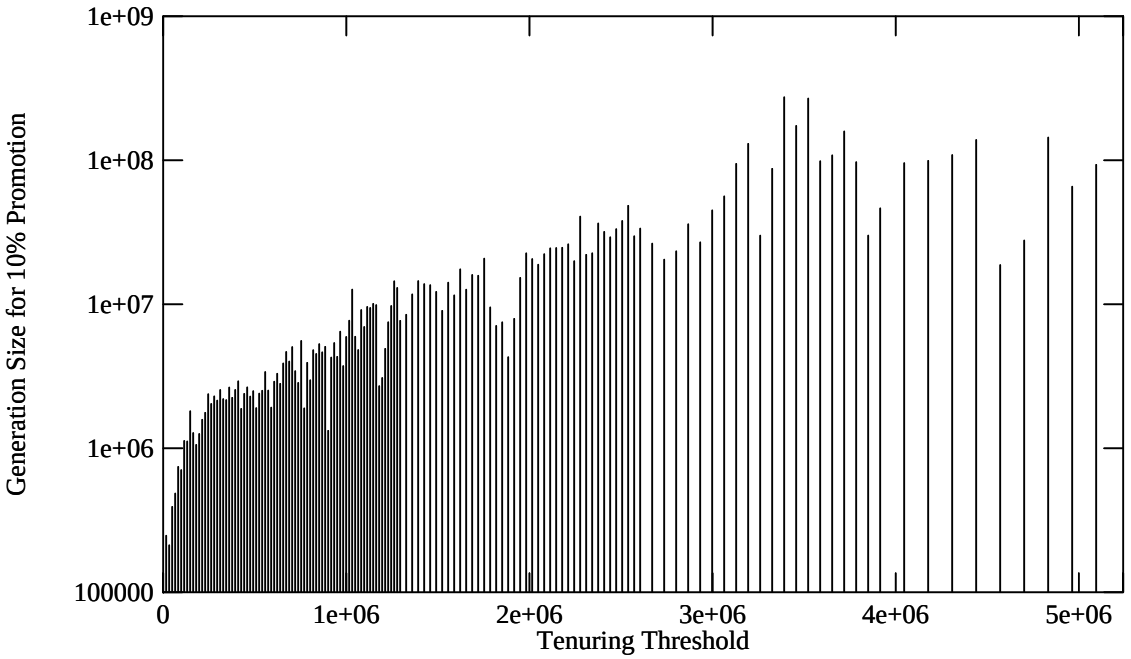

Figure 3.5: Generation Size Needed for 90% Collection Rate

rates in the future will be higher than the instantaneous retention rate — the integral underestimates the generation size.

Solving this integral gives the basic relation between the volume V of a generation and the fraction of storage that will be tenured from that generation, given a fixed promotion rate a and instantaneous retention rate r:

$$F = \frac{a}{V \ln r}(r^{\frac{V}{a}} - 1).$$

The measured data supplies values for a and r for all times up to 5 Mb allocated. Figure 3.5 shows the generation size that would be required to have a cascaded generation that contains only objects that have already survived T bytes of allocation, and a survival rate of 10%.

The volumes needed are small for the early generations, but to maintain a 10% survival rate for generations with longer tenuring thresholds, the volumes grow to hundreds of megabytes. Recall also that these calculations are based on the instantaneous retention rate, and the volume calculated is a conservative estimate since the half-life continues to rise.

The desire for a high collection rate in later generations is in conflict with the desire to keep the generations small. The model of a cascade of generations leads to either large heaps with infrequent collection and poor storage use or poor collection rates in later generations and large costs for tracing. This is a direct consequence of the long half-lives seen among the older objects. A system that makes promptness guarantees for finalizable objects and collects generations only when full can not afford to have large generations. Objects would be threatened infrequently, and the promptness guarantees would be hard to meet.

3.5 Age-based Clusters

While using age directly as a predictor of life-time seems unpromising, further analysis of lifetime data both in Cedar[21] and in Smalltalk[41] shows that for individual applications object life-time graphs are not smooth — there are clusters of objects that are created at roughly the same time and that become unreachable at the same time[2].

These clusters of objects do not necessarily get collected in a stack-like manner, as would befit multi-generation collection, but seem to be more idiosyncratic. While generational collection can collect the young objects, the clusters that are tenured from the generational collection need other methods to keep the costs of collection low. The next two chapters will show how these clusters can be detected and collected efficiently.

[2]These clusters can be inferred from many of the "Generation size" charts in Appendix A. The gaps in these charts show ages where no objects are collected, and the spikes show ages where objects are. In addition, near-by spikes are not the same height — the volumes needed for 10% promotion rates can change ten-fold when volumes of objects are just on the cusp of the tenuring threshold.

Chapter 4

Key Objects

Early garbage collectors scheduled collections only when memory ran out. For large memories this leads to long times between collections, but the collections themselves can be lengthy, and can occur at inconvenient times. In addition, if collections are scheduled only after no more memory can be allocated, the mutator will only be able to run in parallel with the collector as long as it tries to do no allocations, and much potential parallelism will be lost.

One simple allowance for parallelism between the collector and the mutator is to begin collection when the available space, used at a projected rate, will run out at the same time the collection is projected to be finished[26]. If these projections are correct, no mutator will ever be denied memory. The heap must still be guarded against simultaneous access by the mutator and collector to ensure that collections are correct, but this is a first step towards recognizing scheduling of collections as a major problem in complex collection systems.

The drive to consider garbage collection as a scheduling problem is often classed under the rubric of *opportunism*. When the heap is divided into areas, the collection system must have a policy to decide not only when to collect, but also a policy to select some subset of areas for collection. The policy must also be in agreement with the mechanisms available to the collector.

With a typical multi-generational collector, collection of a middle-aged generation can only be done in conjunction with a collection that threatens all objects younger

than middle-age as well. This restriction to the collection system's ability to choose areas to threaten independently is coupled with the decision to remember only pointers from older objects to younger objects. In general, if the collector does not have access to the list of pointers from objects in area A to objects in area B, area B cannot be threatened without threatening or scanning area A [14]. Limited by the availability of this cross-area pointer information, the collection system has a set of choices in selecting a set of areas to threaten with garbage collection and choosing times for performing these collections.

One form of opportunism looks for times when the garbage collector could be run without having the user notice[43]. For example, if the processor has been idle for an hour, it is likely that it will remain idle for a while longer, and a ten second garbage collection is unlikely to be noticed. On the other extreme, if CPU-bound processes are already making interactive response poor, running the garbage collector will not degrade the response much further. The goal is to make sure that the collection system does not get in the user's way.

A second form of opportunism looks for times when the garbage collector could be run with great effect and in minimal time. This involves trying to find areas rich in unreachable objects soon after they become rich, and avoiding running the collector in areas that are unlikely to contain unreachable objects. Generational collection is the exemplar of this form. The areas containing the youngest objects are almost always rich in garbage, and so are scheduled for collection frequently.

Unlike the first kind of opportunism, the second kind can increase the actual efficiency of the collection system. The two kinds of opportunism are orthogonal — a well designed collection system will have both. The emphasis in this dissertation will be on the second, efficiency-enhancing kind of opportunism, and will assume that designers of collection systems will see to the first.

In order for the collection system to do a good job of scheduling effective collections, it must be willing to gather information about the running system from different sources. Often the best way to view the input to the collection system is as a series of "hints" about the areas where garbage collection might be considered and the scheduling of the collections. This decouples the implementation of the collection

system from the scheduling policy.

4.1 Hints

Hints can take various forms, from demands for garbage collection to suggestions, and can include predictions of the amount of storage that would be freed by the suggested course of action as well as the projected costs. The collector can then determine what areas to collect, if any, by integrating the hints with global system information — current and projected real-time demands, CPU load, and even time of day.

Generational garbage collection can be viewed as opportunistic collection where the hints tell the collector what areas of memory have been used for new objects. By concentrating its collection efforts in just those areas and ignoring areas where no allocations have occurred, the collector finds all of the young objects that have become unreachable. Of course, a generational collector is designed around this particular hint. There are few areas where new objects are allocated, and the hint is always applicable to these areas.

Other suggestions for the sources of hints include looking for sections of the dynamic trace of the program where the allocation is essentially stack-like. At the beginning of the section and at the end of the section, the heap is in more or less the same state, but between many objects have been allocated (pushed) and become unreachable (popped) [35]. These sections of code could be marked by the programmer, discovered by compiler analysis or profiling, or noticed dynamically by the allocation system [1].

Such a section of the program will define a group of objects that can be placed together in an area. When the program execution reaches the beginning of the section, the new area is set up and all allocations are directed to that area. When the program execution reaches the end of the section the area is closed off to allocations and garbage collected. Depending on the particular collection system involved and the length of

[1] Programmers are notoriously bad at identifying CPU hot spots in programs when hand optimizing. It would be unreasonable to expect that they would be any better at generating good garbage collection hints without the aid of profiling tools.

time the new area is active, generational collections might be run on the area as well. Because the allocation is stack-like, most of the objects that have been pushed after the beginning have been popped by the end, and a collection of the area will find few objects reachable.

In an application where much of the allocation is stack-like, the state of the heap is reflected in the state of the activation stack. If the collection system monitored the invocation stack, it could schedule collections for times when the stack has recently shrunk, and avoid times when the stack is large. By doing this, it is likely that the collections will occur when the active heap has recently shrunk.

One simple way to monitor the stack is to alter the activation records so that the return address of some chosen frame is changed to call into the collection system interface. The system can note that the return occurred and return to the original caller [43]. This approach aims at invoking collections on all of the objects allocated by an application when the application terminates, and high-level semantic information can give the collection system good places to mark the stack.

Sections 4.2 and 4.3 define key objects and show how key objects can be used to offer hints to the collection system, both to find areas to be collected and to find good times to collect them. Section 4.4 shows how finalization, a mechanism already in place in some collection systems, can be used to build keys and hints. Section 4.5 offers some suggestions on how keys might be discovered.

4.2 Key Objects

When the entire object space is partitioned into areas, the collection system must decide if a particular area is ripe for collection. One possibility would be to take a "core sample" of an area, consisting of a randomly chosen set of objects from that area. If the collection of the core sample proved to be effective, then the collection system could schedule a collection of the entire area.

Core sampling allows the system to expend a small effort to predict the effectiveness of a collection on an area. Unfortunately, choosing a random subset as a core sample seems unlikely to produce good results. Imagine an area where the objects in

the area are all unreachable from the roots, and thus collectible, and are connected in a tree. Any subset that does not include the root of the tree will have no unreachable objects in it, since the root of the tree will be a root of the collection.

One goal of key object opportunism is to choose these representative subsets in a way that will allow generation of useful hints. A good key object is an allocated memory object that the collector can choose to threaten in order to get statistical information on the distribution of reachable and unreachable objects.

4.3 An Example

Figure 4.1 shows how a sample cluster and its key interact with the collection system. This example will be couched in terms of a copying collector to simplify the bookkeeping, but a collector can do the bookkeeping without doing the copying [14].

a) A cluster, in this case a tree, is created in the area reserved for new and young storage. The objects in the "new" area tend to die quickly, and tend to be written more than old objects. In a pure form, access to the cluster would be through the key objects only, in this case the root of the tree. The only access to the root is through a global pointer held in an old object or global variable.

b) After the cluster has passed some age or survived some number of collections, it is copied out of the young area. But rather than being copied to an age-segregated area, each cluster is moved to an area of its own. The time-based generational collector will try less frequently to collect the cluster. In a standard time-based generational collector, this is the "promotion" step.

   The key objects for the cluster are copied to a separate area. The collector will continue to try to collect the key objects, but the remainder of the cluster will be promoted and faced with fewer or no collection attempts.

c) It may be that not all the objects will be copied from the young area at the same time, as they may have a range of ages, and objects created later may be hung from various branches of the tree.

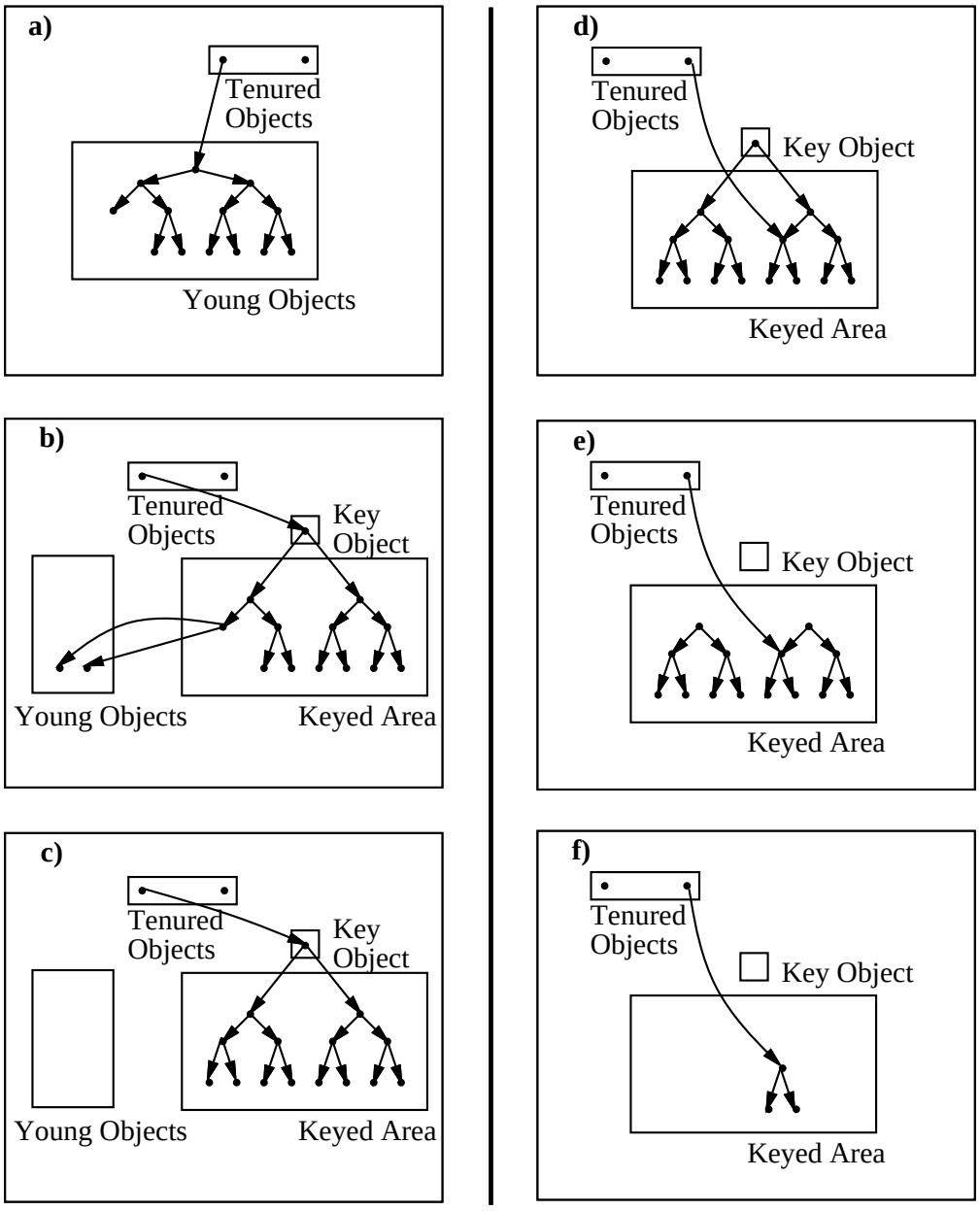

Figure 4.1: An Example of Key Object Opportunism

After looking at the pre-existing keyed areas that point to new objects, copying can be used to promote new objects to areas that already pin the objects rather than to a general "old" area[5], or the objects can be self-identifying, indicating which cluster they would like to belong to when they are promoted.

d) When the mutator is done with the cluster it writes the variable leading to the key object. In a cyclic program, this will happen when the next cycle stores a pointer to the new root. A program aware of the garbage collection strategy might explicitly zero the pointer, just as cycles are broken for reference-counting collectors.

Pointers to other parts of the cluster may continue to be active, either because a part of the cluster is being actively saved, or a pointer was created which was never zeroed.

e) Since the collection system has not promoted the key objects, they are not out of harm's way. Eventually, the un-referenced key object will be reclaimed. At that time, the collector can take this as a strong "hint" at an opportunity to collect the associated keyed area.

Note that the collector can not simply deallocate all of the objects in the cluster, since some of them may still be alive through pointers that bypass the key objects of the cluster. The bookkeeping information will show all of these cross-area pointers.

f) A collection over the keyed area is, in this case, fruitful. Most of the objects in the cluster are not saved by outside pointers and can be reclaimed.

The collection system must employ a policy to handle those objects that live longer than their keys. A simple solution would be to copy those objects to a permanent tenured area to wait for the next full collection. Instead, they could be combined with a cluster they are rooted in, or even analyzed to find a new key among them[5]. Further research and measurement would be needed to find good policies for objects that escape both generational collection and key object collection in some particular collection system.

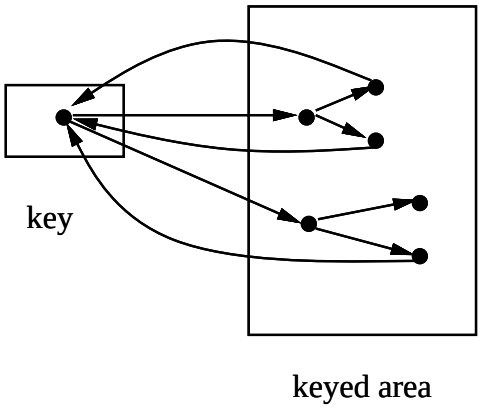

Figure 4.2: Backpointers to a Key

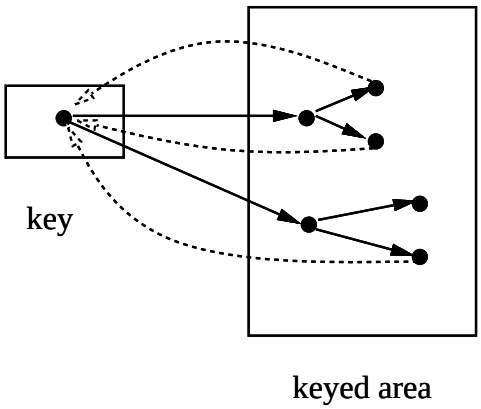

Figure 4.3: Backpointers to a Key, Ignored by Finalization

4.4 Using Finalization

In some systems with finalization, the mechanism used for finalization notification can be used to issue garbage collection hints. Finalization, as mentioned in Section 1.5, allows an object to have code associated with it that is run when the object becomes unreachable or nearly unreachable. This is exactly what key object opportunism demands — when the key object is nearly unreachable, it can use the finalization to file its hint with the garbage collector.

Figure 4.2 shows one of the problems that can occur with otherwise good choices

for key objects — some objects in the key's cluster contains a pointer to the key object. Using the inaccessibility of the key to trigger a collection of the objects in the cluster will not work if objects in the cluster have pointers to the key. With back pointers, attempting to collect the key more often than the cluster is doomed to failure. The cycle involving the key object and the objects in the keyed areas will keep either from being collected until they are both threatened by the same collection. The repeated attempts to collect the key alone are futile.

However, if the semantics for finalization allow the key object's finalization code to run when the object is "nearly unreachable," then the root can be used as a key, even in the presence of back pointers. All that is needed is to register the back pointers as irrelevant to the finalization, as in Figure 4.3, using the same mechanism that allows package pointers to be ignored. Whatever mechanism is used to determine that a certain object should be used as a key for a certain cluster will also automatically define the set of back pointers that would have to be ignored — the pointers to ignore are just those pointers from a keyed area back to a key that controls collections of that area.

When the root becomes unreachable except by a path that uses these back pointers, the finalization code — now being used to issue a garbage collection hint — will run and register a hint to collect the cluster keyed from the root.

This use of designated pointers to break cycles resembles weak pointers, as discussed in Section 1.5. One of the chief uses of weak pointers is to break cycles in data structures for reference counting collection systems[38]. In such systems, the weak pointers are not counted in the reference counts. Reference counting systems cannot easily collect loops, but if at least one pointer in a simple closed loop is weak, some object will have a reference count of zero rather than one. That object can be collected, and the cycle will be broken. But the collection system cannot change a strong pointer to a weak pointer without risking a change in the semantics of the program. The programmer has to consider what data structures would produce loops and where weak pointers could be placed to allow the collection system to deal with the loop, and must be very careful that the intended semantics of the program are not effected by the change.

The package pointers used by finalization and the designated back pointers used for key objects are not weak pointers. If they were, then key objects would be collected when the remaining non-weak pointers were eliminated, even if some objects in the cluster were still reachable. The key object would be collected even though it is still reachable from a reachable object in the cluster.

If the system provides only "executor" semantics for finalization, there is no simple way to use finalization to register hints. Under these semantics, the finalization can only be run when the object becomes thoroughly unreachable. In the presence of back pointers, this could only be established by collecting the key object along with the cluster where the back pointers come from, rending moot the hint to collect that cluster.

4.5 Finding Key Objects

Some key objects are objects through which accesses to a cluster are usually made, for example the root of a tree or the head of a list. The life-times of these objects reflect the lifetimes of a larger set of objects, and often data structures are designed so that there are few external pointers to the list or tree other than the root. A generational garbage collector may traverse the young objects many times before they are handed off to the key object collector, and so has many opportunities to discover which objects are the roots of large structures. But other objects may reflect the lifetime of a larger set, and so could be used to signal an opportunity for collection.

4.5.1 Random Selection

A collector could promote half of the objects that had lived past the tenuring threshold and use the rest for keys [2]. There will be some correlation between the fraction of garbage in the half not promoted and the fraction in the half promoted. When the collector notices an unusually large number of unpromoted objects of roughly the same age being collected, it could then examine the objects promoted at the same time.

[2]Paul Wilson suggested this at an informal discussion at the OOPSLA '90 garbage collection workshop.

Since objects that are created at roughly the same time tend to become unreachable at the same time, this approach might be expected to work well.

Many objects among the unpromoted would be kept alive only by pointers from other objects in the cluster, but no pointers from outside the cluster. There would be many pointers from the promoted half to the unpromoted half. This would increase the bookkeeping costs by expanding the set of pointers from the promoted objects back to the others. If the cluster has already survived generational collection, it is unlikely that its structure will be changing radically, and these bookkeeping costs would have to be carried for the life of the cluster.

For most clusters, being split randomly in half wouldn't be likely to produce good indicators. Many objects in the unpromoted half will have pointers to them from other unpromoted objects. The objects pointed to will not become unreachable unless the structure of the cluster changes or the objects containing the pointers become unreachable. When the accessibility of one key object depends directly on another's, one key alone will usually suffice.

The only way that random selection would work well is if the system were fortunate enough to not promote the roots of structures and promoted most of the rest. If the head of a list is promoted, the unpromoted fraction will be worthless as an indicator for the list since nothing in the unpromoted fraction can be collected before the head is. Keeping a large random fraction of the old objects unpromoted would put a burden on the collection system, but a small random fraction seems unlikely to be a good indicator for collections.

4.5.2 Key Discovery

The collector could gather information in the generational phases that would help to uncover some of the keys without any human intervention. When the time-based collector copies a group of objects from one generation to the next, it could keep track of which objects seem to be the root objects, like the heads of lists and the roots of trees.

Each generational collection starts from a set of pointers contained in the global

state of the system and the set of cross-generation pointers from the old objects to the new objects. Any young object, A, which is not pointed to by one of these initial pointers is only being kept alive by some other live, young object, B. If this situation persists across several generational collections, it would seem that objects like B — the roots in the young generation — would make better keys than objects like A, since A will only become collectible when object B is collected or changed.

The only candidates remaining are those objects directly pointed to by one of the initial pointers into the young area. The generational traces can also gauge the volume of storage that a cluster reachable from some such object contains, eliminating objects as potential keys if they are not responsible for large volumes. For some collection styles including Cheney copying[9], it would not be difficult to also examine the potential clusters for modularity.

Pure Chaney copying traverses objects in a breadth-first order. The collector can make the traversal in an order that attempts to keep clusters of objects together, and the cost is small — only a few extra scan pointers and fill pointers need be kept[30, 36, 44, 46, 13]. When constructing a cluster, the collector can easily discover all of the pointers from the cluster to objects outside it. If two clusters would have a large number of pointers between them, they could be combined into a single cluster.

It is also tempting to look for keys by finding pointers from old objects and global objects to much younger objects. The intuition is that these pointers will indicate important features of the mutator's data structures, pointing out old objects that have their contents altered again and again, and the current tenant residing there may be evicted in the future.

Of course, some ancient objects, for example hash tables, might be quite libertine and point at a large number of objects [3]. Ideally, a key object would be indicated by a pointer from an ancient object to a new object, where the ancient object had a history of changes in the set of objects it pins. This would give some credence to the presumption that a future change to the old objects might free the suspect key.

[3]Due to other considerations, it is a good idea to identify hash tables to the collection system [44, 46].

4.5.3 Stack-Based Key Objects

A key object will only be collectible when all the objects pointing to it either become collectible or are altered to remove the pointer to the key. A pointer from an object that does not change would tend to indicate that the object pointed to is not a key object, but an object pinned only by volatile pointers would be an excellent candidate. In a stack-based language implementation, the pointers from the stacks can be treated as volatile.

The collection system can use information about the stacks to help it find opportunities to collect mature objects. Consider an object that is pointed to only from a stack frame. When the routine associated with that stack frame returns, the object will be collectible unless the routine makes a special effort to save that pointer. Saving the pointer involves either returning it as a result and having it stored into another stack frame or explicitly storing it in a global or the heap. Most pointers are not saved upon return, and most objects become unreachable quickly.

This suggests a heuristic to monitor the stack activity and schedule collections of mature objects. Previous suggestions included altering some of the active stack frames to generate hints for collection upon return[43, 44] and changing the call/return protocol to count the number of pops from the stack[28].

A collector can find stack-based keys as a side effect of tracing objects during a full collection. First, it would want to mark all of the objects reachable from the globals, and only then trace from the more volatile registers and stack. The objects reachable from the globals can be divided into areas and keyed by whatever heuristics are found to be appropriate.

The remaining untraced objects are reachable only via volatile pointers. The first level of objects reachable from the registers and stack would be likely to make excellent keys for clusters of objects reachable from only the stack and perhaps even other objects reachable from the globals.

A good way to divide the remaining objects into keyed areas would be to begin with the coldest, oldest stack frame. Objects reachable from this frame will not become unreachable until the frame is popped or the pointers are overwritten. An object reachable from this frame and another warmer frame will not be free when the

warmer frame returns, but must wait for both frames to be popped.

The collector can trace all previously untraced objects reachable from objects in the deepest frame, and project that they will all become free when the frame is popped. To know when the frame is popped, the collector can use the objects that are immediately reachable from the frame as keys. The collector can continue in this way along the stack from the coldest frame to the warmest, partitioning the remainder of the heap into temperatures based on the stack, with key objects for each partition. Figure 4.4 shows the result of such a partitioning.

The stack has a hot end — where the pushes and pops occur, and a cold end. The pointers from the stack to the heap are used to infer a similar temperature gradient on the heap, and objects will be collected in roughly a hottest-first order if the keys behave as expected. At the hot end of the stack, there may be pointers to objects still in the young generation. It seems prudent to allow generational collection to have its way with these objects before subjecting them to stack-based keying. At the cold end, the use of the stack frames may have more in common with global storage than stack-based storage, depending on the language, implementation and style of programming. The collection system will have to trade off among heuristics to find the best applicable range for stack-based keys.

After the partitions have been established and the mutators have been resumed the situation may resemble Figure 4.5. Some of the warm frames have been popped, some values have been stored into cold frames on the stack, some new frames have been pushed. The collector can get a useful summary of this activity by threatening the stack-based keys.

A collection on just these keys will result in the situation seen in Figure 4.6. Most of the keys associated with popped frames will be collected, and few of the keys associated with unpopped frames will have been. The collector can use this information to make a temperature cut in the stack-based partitions, and threaten all objects in partitions warmer than the cut-off.

The accuracy of stack-based keys is related to the degree that the mutators are following a functional paradigm. In a strictly functional language, the only violations of the stack allowed are in function returns. The success of generational collectors

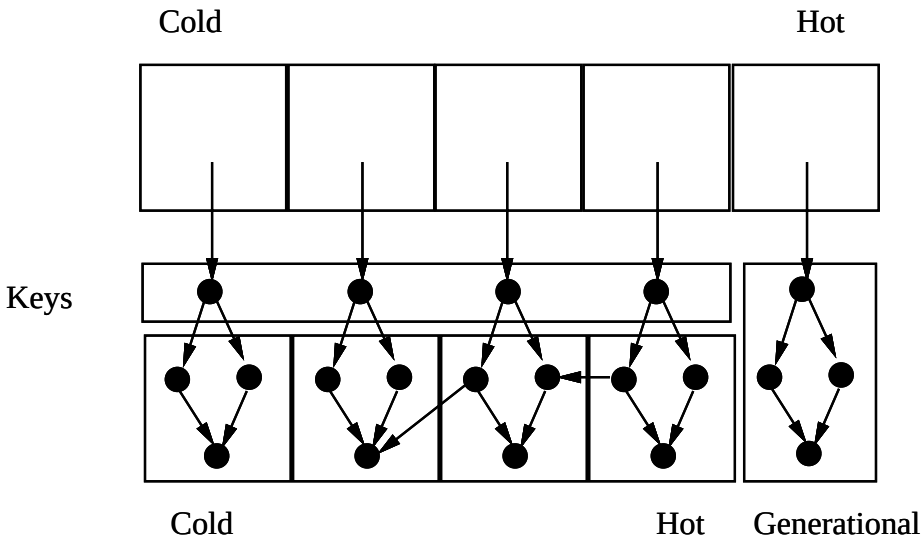

Figure 4.4: Keys Found by Stack Traversal

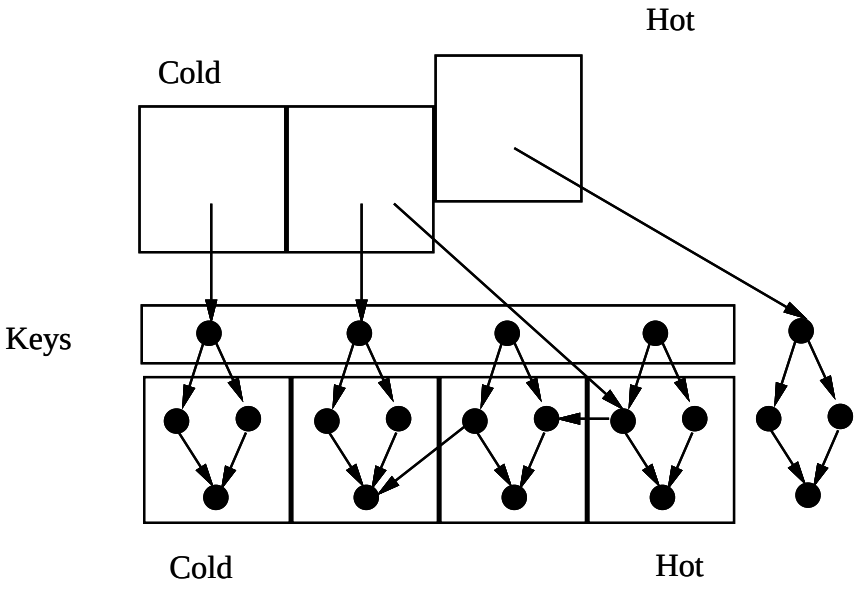

Figure 4.5: Unreachable Stack Keys

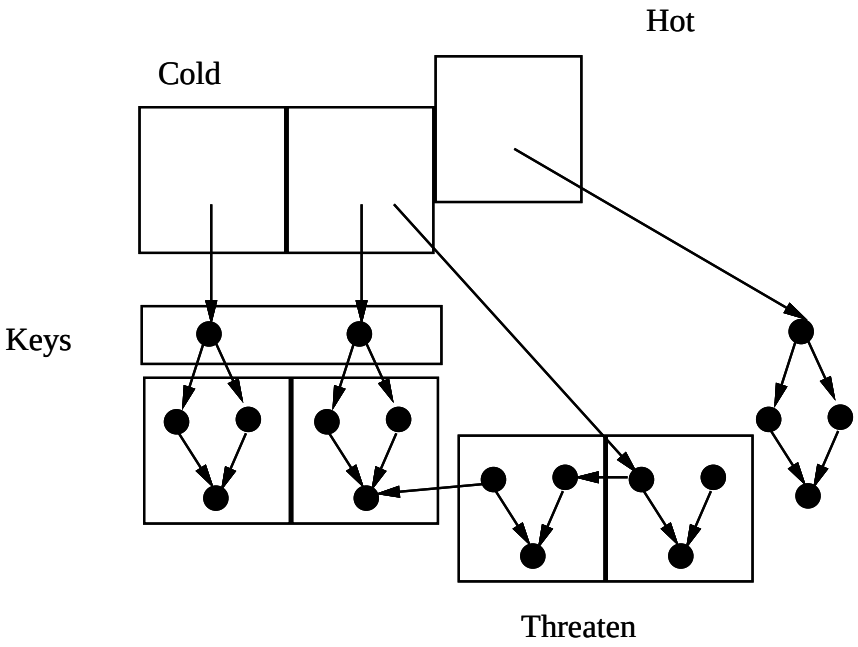

Figure 4.6: After Key Collection

implies that even languages that do not enforce functionality, such as Cedar and Smalltalk, are used in a functional way, and benefit from collection mechanisms derived from a functional viewpoint.

4.5.4 Serendipity

Key objects need not even point to objects in the clusters they monitor. The design of the system may include an object or small cluster to hold supplementary data about a large cluster. The keys are not actually pinning objects in the cluster, but have a lifetime similar to it. The semantics of the application that creates a cluster can imply that an object is a good candidate.

If the semantics of the application are not available to the collector, then the collector is limited in its ability to find these keys, but profiling of the collection system might be used to find them. An examination and profile of the objects promoted by the generational collector and deallocated by a collection over the older generations can show blind spots in the generational collector. If a particular application is

allocating a large volume of storage that is promoted and then later becomes free, but there is no obvious root or serendipitous key, the application can be changed to allocate a single object whose sole purpose is to serve as a key. A coincidence can be planned.

4.5.5 User Supplied Hints

Other key objects can be indicated by cycles of human use. There is nothing in the semantics of the editor, compiler, and debugger to suggest that people often use these tools cyclically, but the cycle arises out of human work patterns. This means that when a program developer begins to debug, there may be key objects in the debugger that trigger a collection of the compiler structures — use of the debugger is a good indicator that the compiler has finished. The two applications are connected not in their formal semantics, but only in their use by people. The profiling used to find keys for one application might uncover them in a different application.

It is hard to imagine that a connection through human use could be found easily and automatically. This suggests that a collector would benefit by being able to take hints about collections, including suggestions from applications about which objects would be good key objects for which clusters. Without this ability, the collection system is limited to discovering hints on its own.

Chapter 5

Simulation of Key Objects

In order to test key objects in a real system, I modified the ParcPlace Objectworks\Smalltalk memory management system[31][1]

Objectworks uses two garbage collectors, a two-space copying generational collector and a non-copying full collector[2]. The generational collector used by the system has been extensively tuned and studied, and has a promotion strategy designed to keep promotion at a minimum[41]. Rather than promoting objects that have reached a given age, the strategy is to allow object to remain in the young generation until they are too voluminous to collect without noticeable pauses. At this point, enough storage is promoted to bring the young generation back under the size imposed by the real-time limit.

In a previous incarnation, when the generational collector needed to promote objects it would promote the oldest objects currently under its control, choosing an age cut-off that would guarantee that enough objects would either be reachable and promoted, or be unreachable and collected. The present collector promotes those objects nearest the bottom of the semi-space.

Objects are created in a separate "birthing area", and if they thrive are copied to a semi-space. Objects are only promoted from the semi-spaces so promoted objects have

[1]Special thanks are due to Frank Jackson who arranged a source license for this project.

[2]It also contains an incremental collector and permanent area, but for these experiments they were disabled. In a production system, I would expect some kind of parallel or incremental full collections to be present.

survived at least two generational collections and may have survived many more[3].

Since promotions out of new-space are based on the relative addresses of objects, the generational collector also controls which objects will be promoted more aggressively and which will be kept in the new-space as long as possible. The first objects copied into the to-space will be the first promoted at the next generational collection, and the generational collector can control promotion order by using the copy order. The tweaks in the copy order to support key object collections will be discussed in Section 5.2.

5.1 Inter-generational Pointers

Previous studies have shown that most pointers connect objects of similar age[10, 15]. It is a rare event when a pointer to a new object is stored into an old object, and it marks both objects as being somewhat special[4]. The old object, where the pointer is stored, has shown itself to be the target of a store after it was promoted, and thus fairly long after its creation. Assuming that such an object once stored into is more likely to be stored into in the future, the object in new-space that the old-space object points to is a good candidate for being a key object; a future store that overwrites the pointer would free the key object in new-space unless it is explicitly retained.

Objectworks was modified to set a bit in any new-space object that had its address stored in any tenured object. The generational collector already maintains a "remembered set" of old objects that may contain pointers to new objects — it is used to find the roots for the generational collections. Since the generational collector must already trap these pointer stores to maintain the remembered set, setting the object's bit does not degrade performance to any large degree.

[3]When the survival rate of new objects is so high that the survivors in the birthing area will not fit in the to-space, they are tenured immediately. This is an unusual circumstance, and the methods used here to make key objects will not work in those conditions. For some of the tests presented here the birthing area was reduced in size to ensure that objects would take the expected path.

[4]Most system designs contain a few objects containing pointers to a large number of objects, and a few objects that are pointed to by a large number of objects. Examples of these are, respectively, an object table and the NIL object. These objects can be exceptions to the rule that objects connected by a pointer are of similar age, and might need special attention in a memory management system assuming this rule.

5.2 Marshaling Objects For Promotion

Since the collection system will be using these designated objects as key objects, it also needs to cluster together those objects that it believes will become free if any particular key object becomes free. The generational collector is used to gather together the keys and their clusters. Since the order of tracing of objects by the generational collector is also the order of promotion, it can control which clusters will be promoted aggressively, and which will be retained for future generational collections.

The first objects copied are those pointed to from old-space, but that have never been the target of an inter-generational store. The object in old-space must have been pointing to the object in new-space when it was tenured, and the pointer has not changed. These are the "nepotism" pointers — a structure was built in new-space, but only part of it has been promoted so far and part of it remains in new-space. These new-space objects will be the first promoted when the generational collector needs to lighten its load[5].

Next the collector tries to place large structures, even if made up of many small objects. As part of every scavenge, it notes when a single object and all the new objects reachable from that object have a volume of more than 250 bytes. The objects heading these clusters are marked, and, at the following scavenge, these clusters are the next in line to be promoted.

Finally, any remaining objects that are reachable from old-space are copied. It is only after all the objects reachable from old-space are copied that the objects reachable only from the stack are copied. These form the large majority of the un-keyed objects, and it is assumed that other methods would be used to find key objects among these and the objects they point to. They will only become free when the stacks that pin them are popped or altered[6].

[5]When the scavenger finds that the cut-off point to bring the volume of retained objects is in the middle of a cluster, it advances the cut-off to the next new key object. This helps eliminate nepotism pointers.

[6]The collector as shipped by ParcPlace has quite a different order. If the incremental collector is running when a scavenge occurs, the collector first copies all new objects reachable from the marked objects in the remembered set and then transitively copies those objects. This ensures that new

The Objectworks system has a level of indirection between pointers to object headers and the data they contain[7]. When a key object is tenured, the header is placed in a reserved portion of memory, and the key's data is placed with the data from the cluster. Key objects in new-space are marked with a bit, but in old-space their headers are segregated. The headers for the key objects contain pointers into the object data, and thus partition the object data into clusters. All collections of the old space maintain the order after collection by compacting the data. If a key object heading one cluster is collected, but some of the objects in the cluster survive, these objects will be merged into the preceding cluster.

5.3 Simulation of Key Object Collection

A collector using key objects would first determine which key objects indicate clusters that should be collected, and then use that information to collect those clusters and avoid the other clusters. Being able to selectively attack clusters is difficult since it requires finding all of the pointers in old-space that point from one cluster to another. The same effect can be achieved by scanning the old-space re-building at each key object collection the caches that would be kept in a real system. Efficiently keeping these caches and partitioning of heaps into nearly-independent clusters is a target of current research[5, 36, 46, 13].

---

objects promoted by the scavenger will be marked if they should be. If the incremental collector is not running, this step is omited. Next it searches the stacks for pointers to new objects and copies just those objects directly reachable from the stacks. Finally, it traces the rest of the remembered set for pointers to objects in new space, and copies the objects directly pointed to. These roots from the remembered set together with the roots from the stack are then copied transitively using the usual breadth-first two-space copying algorithm. Since the incremental collector is not usually running, the effect of this tracing order is to promote aggressively the objects pinned by the stacks.

As a simple experiment, the order was changed to first copy all objects transitively reachable from old space, and then copy those objects reachable only from the stack. The expected effect was that the generational collector would retain those objects more suited to generational collection — those pinned only by stack pointers — and promote other objects. Using the "session playback" data set, 30% fewer objects were promoted. The changed order relieves the burden on the next collector that would be responsible for these objects. This is only a single data point, but it suggests that generational collectors could benefit from considering exactly what objects to promote.

[7]This split allows for the Smalltalk "become" operation where two objects exchange identities. The pointers from the headers to data are exchanged in the two objects, and any pointer to the header of either of the objects will now find the data previously associated with the other.

The key objects in the old space are divided into "assumed live" and "assumed dead" by doing a mark and sweep on the clusters. The un-keyed and new objects are the root clusters, and pointers to key objects from there cause the collector to assume that those keys are live. For each key that is assumed live, its cluster is scanned for pointers to other key objects, and those are likewise assumed live. This continues until all clusters associated with keys assumed live have been scanned. Any key scanned is assumed live, and its cluster will be immune from collection. This scheme will not collect unreachable cycles of clusters where some clusters are old and some new. Only when all the clusters in a cycle have been promoted will they all be collected.

To simulate the collection that would occur by using the key objects, a full trace of the objects is first performed. This identifies all dead and live objects. Then each immune cluster is scanned and each un-marked object, representing an object that is free but would be undetected by a key-object collection, is marked. Objects reachable from these newly marked objects are transitively marked, and the old space is swept and compacted. Doing the marking in the reverse order — first marking all immune objects and then doing a full collection — would result in exactly the same objects being retained, but would prohibit evaluating the projections made by using key objects.

The Smalltalk system as shipped by ParcPlace triggers scavenges when the birthing area — "eden" — fills to 184 kb. The semi-spaces are each 40 kb, and promotions to old-space are triggered whenever a semi-space contains more than 25 kb. Simulated key object collections are run every fourth scavenge. The marshaling of objects is done when objects are copied between semi-spaces, and when the fraction of objects surviving becomes large, object may be promoted without being marshaled. For the "replay" example there are enough short-lived objects to guarantee marshaling, but for the other benchmarks the triggering threshold in eden is reduced to 18.4 kb — scavenges in these benchmarks occur ten times more frequently than in the default system. After every 128 scavenges, that is, every 32 key object collections, a full collection is run immediately after the key object collection.

5.4 Allocation Examples

Unfortunately, there is not a suite of well-accepted benchmarks for memory management systems. Some common tests are system- and language-specific, others are designed to test the effectiveness of generational collectors, without being effective for measuring the quality of the system once objects are promoted.

To gauge the effectiveness of key object collection, I selected four sample programs. The first allocates 10000 linked lists of length 100 and places them in random locations in an array of size 200. This is an example of the pattern that key objects would be expected to be effective on. As soon as the array is promoted, the head of each new list should be identified as a key object, and the remainder of the list is the cluster it controls. When an element of the array is replaced, the key object will be noted to be unreachable, and the cluster collected.

Two other examples are drawn from the Stanford benchmark suite. The suite consists of a series of small programs designed to measure the speed of language systems on various architectures. Only two of these programs cause objects to be promoted in Objectworks: the single-precision, floating-point, forty-by-forty matrix multiply and the five thousand element tree sort. To increase the volume of objects promoted, the tree sort is run fifty times for each session and the matrix multiply is run twenty times.

In both of the Stanford benchmark programs the objects promoted have a short and predictable lifetime since they do not survive the next run of the program. If the generation sizes were carefully chosen to contain the volume promoted by the benchmark, the generational collector could reduce the promotion volume to zero. Again, these benchmarks were not designed as garbage collection benchmarks but system performance benchmarks.

The original matrix multiply benchmark did not perform as well as the other tests. After examining the running benchmark, I made a few "minor" changes that produced significant changes in the benchmark results. These changes, the motives for them, and the effects on the key object scheme will be discussed later. Code for all of the examples is in Appendix B.

The last program is a replay of an interactive session using the Objectworks system. For two days each keystroke, mouse motion, and key click was recorded while I used Objectworks to build a sliding-15 puzzle[8]. The Objectworks system was altered so that playback would be possible, and so the style of use was oddly stilted, since various scrolling and time-dependent portions of the user interface were disabled. This example makes extensive use of the X windowing system, and time-dependencies present there cause the timings of allocations and collections to differ from run to run, and thus cause a larger spread in the statistics for this example than is present in the other examples.

At start-up, many objects are promoted directly out of eden without being marshaled, and many objects that were only reachable from the stacks become unreachable. For each test session, the Smalltalk system was started and an initial full collection performed. Some objects were created directly in the mature space, without passing through the earlier generations. The key objects explored here are not designed to cover objects pinned from the stack or objects created in the mature space.

5.5 Session Data

Table 5.1 gives an idea of how key object collection fares in the environment described. I ran twenty sessions of each of the four test cases and the modified version of the matrix multiply benchmark. The table records the size of the mature heap in megabytes for each benchmark sampled at each scavenge, giving the median, 10th and 90th percentile. The percentiles are taken from the combination of all twenty sessions.

Following the heap size are the volumes allocated at the youngest level[9]. Next is the volume tenured to and collected from the mature space. Some objects at start-up are created in the mature area, rather than gaining their place through tenure,

[8]Or more exactly attempted to build such a puzzle. I'm at best a novice Smalltalk user and never managed to get the mouse-clicks to move the puzzle pieces.

[9]Objects all have an eight-byte header. These headers are included in the total allocation, but no other statistic will include the header data.

Test	%ile	heap sz	alloc	tenured	total coll	available	keys coll	key coll
	10	1.28	26.6	3.87	3.93	3.79	3.48	91.6%
Array	50	1.29	26.6	3.89	3.95	3.80	3.49	91.7%
	90	1.30	26.6	3.91	3.96	3.81	3.50	91.9%
	10	1.25	10.5	1.74	1.77	1.61	1.29	80.4%
Tree	50	1.37	10.5	1.74	1.77	1.61	1.29	80.5%
	90	1.47	10.5	1.74	1.77	1.61	1.29	80.5%
	10	1.24	59.8	1.06	1.11	.649	.264	40.7%
Matrix	50	1.25	59.8	1.06	1.11	.649	.264	40.7%
	90	1.27	59.8	1.06	1.11	.649	.264	40.7%
Hacked	10	1.23	58.6	1.04	1.10	.943	.745	79.0%
Matrix	50	1.24	58.7	1.04	1.10	.944	.745	79.0%
	90	1.25	58.7	1.04	1.10	.946	.745	79.0%
	10	1.19	182.	.609	.635	.348	.191	52.5%
Replay	50	1.21	182.	.631	.653	.360	.214	58.8%
	90	1.24	182.	.644	.671	.378	.244	65.8%

Table 5.1: Key Object Sessions

and so the volume collected is sometimes larger than the volume tenured. In these and following columns, each session generates a single value for the statistic, and the numbers given are the second, tenth, and eighteenth ranked of the twenty sessions.

Recall that some mature objects are not in memory that is under the control of key objects. The "available" column records the volume in megabytes of the memory that was collected from the keyed areas alone, excluding the volume unkeyed. The next columns are the volume of objects actually collected by the key object collector, as a megabyte volume and as a percentage of the "available" bytes[10].

Table 5.2 shows some of the costs associated with key objects. The number and volume of keys in bytes is shown in the first two columns. The next two columns show the number and volume of objects that point to key objects. Various implementations of keys would have costs based on these values. For example, the keys might be segregated, bringing about a cost for each byte of key-data. As in the model implementation, just headers of the keys might be segregated, so there is a cost related

[10]Note that the available, keys collected, and percentages are the second, tenth, and eighteenth of their type, and are not necessarily all from the same session.

Test	%ile	Keys		Key Sources		Remembered	
		count	volume	count	volume	count	volume
Array	10	227	2824	41	2392	9	900
	50	337	3700	133	3084	14	1148
	90	526	5196	322	4248	22	1344
Tree	10	0	0	0	0	22	1796
	50	1537	55504	1259	16516	496	8212
	90	4561	97264	3684	46480	912	12040
Matrix	10	430	2668	40	3304	10	380
	50	1763	7796	75	8860	24	2800
	90	2789	12104	101	13140	44	6104
Hacked Matrix	10	636	1688	17	1764	26	1284
	50	957	4116	32	4340	35	2900
	90	1173	5168	40	5188	42	3908
Replay	10	44	1732	51	5712	62	3728
	50	114	5428	116	13956	91	8240
	90	212	7971	200	16428	156	11000

Table 5.2: Key Object Costs

to the number of key objects.

Depending on the implementation, determining which key objects are reachable might involve costs related to not just the number or size of the keys, but the number and volume of objects that actually contain pointers to key objects. Finding dead key objects can be expected to be much like finding dead objects in a generation scavenger.

For example, the Objectworks system uses a table of "remembered objects" that include all objects that point to new objects. A similar implementation of key objects might keep a table of objects that point to key objects. The third and fourth column show the number and aggregate size of objects containing pointers to key objects. The fifth and sixth columns give the size of the Objectworks remembered table for comparison.

The values for all columns in Table 5.2 are sampled at every scavenge, and the values given are the median and percentiles across all sessions of that test.

The final table, 5.3, quantifies the quality of the key object collections. The first

Test	%ile	Obj Size	Cluster Size	Obj/threat	Quality
Array	10	4.35	231.1	76.0	%98.6
	50	4.36	349.2	80.6	%98.8
	90	4.36	352.3	81.0	%99.0
Tree	10	16.5	44.7	2.71	%94.6
	50	16.5	44.7	2.71	%94.9
	90	16.5	44.8	2.71	%94.9
Matrix	10	9.55	23.0	2.40	%34.1
	50	9.57	23.3	2.43	%35.3
	90	9.57	23.5	2.46	%36.0
Hacked Matrix	10	9.57	29.5	3.08	%98.4
	50	9.57	30.2	3.15	%98.4
	90	9.60	30.3	3.16	%99.97
Replay	10	16.6	309.1	18.5	%61.3
	50	16.7	398.6	23.9	%68.0
	90	16.8	522.8	31.2	%75.6

Table 5.3: Key Object Quality

column shows the average promoted object size, in bytes. The next two columns shows the average volume and number of objects in a cluster for each threatened key. This is a measure of the leverage of the key objects — only the key object is a subject of the collection, but it has indicated that a cluster of objects is now unreachable. The cost paid collecting the key benefits all these objects.

If each key object were a perfect indicator for its cluster, 100% of the objects threatened by the key object collector would be found to be unreachable. But the key object analysis may indicate that some objects should be collected when they are not free. A good measure of the quality of the key objects analysis is the fraction of bytes indicated as free by the key object collector that is actually collectible. This is the last column in Table 5.3.

5.6 Analysis

The "array" test is a simple test, meant to demonstrate the allocation patterns that key objects are particularly good at tracking. About sixty lists of four-byte cells[11] of length 100 can be held in the 25 kb available in the semi-space. The remaining 140 lists in the array are promoted to the old space, and few other objects are promoted. The generational collector still pays the cost of copying and recopying the lists retained in new-space. The key-space collector monitors the heads of the promoted lists, and can collect the unreachable clusters easily, since there are no intercluster pointers other than to the head. The system might benefit, in this case, if it promoted clusters more aggressively, since the cost of repeatedly recopying the list is much greater than the cost of monitoring the single key object that heads the list.

The "tree" test and the "matrix" should be similar from the view of the memory management system: both build a large structure and process it to build another large structure. For the tree, it builds an array of random values and processes it by inserting the values into a sorted tree. The matrix multiply builds three arrays-of-arrays, initializes two with random values and calculates the product of these matrices in a straight-forward way to fill in values in the third.

Both tests create a large number of objects that are identified as keys, but would be better ignored. The only important key objects in the tree sort are the array of random values and the root of the tree. In the matrix multiply, just the three arrays themselves need to be keys to find all the garbage. Cross-generational stores happen for a large number of the tree nodes in the sort and almost every floating-point object in the product of the matrix multiply, but the locations in old-space where these objects are anchored, the key sources, are only written once by the tests. Monitoring the tree nodes and floating-point numbers, waiting for them to be freed by another write to the key sources, is a waste of effort.

The matrix test tweaks several shortcomings in the key object scheme used for these tests, and I prepared a modified version of the matrix test to work around these inadequacies. First, I was surprised to find an earlier decision coming back to

[11] The "Link" class is a CDR-cell with no CAR cell. Objects of this type are not usually created, but sub-classes of Link will have additional fields.

haunt me: when an object is pinned only from the stack it is promoted to a special segment, out of the address space that key objects and their clusters occupy. The method that builds the random-valued arrays, `initmatrix`, holds the intermediate values in stack-based local variables. Only when the entire array has been built is it returned and stored in a heap-based variable.

Since the arrays are each slightly larger than the tenuring threshold of the semi-spaces, it is almost certain that some small part of the array will be tenured before it is stored in an old objects and seen as a key[12]. Unfortunately, the part promoted will almost always include the root object, `m`, and it will be tenured to the region for objects pinned from the stack. The rest of the matrix, when tenured, will be pinned from that root or its children, and so will be tenured to the key-using segment. Unfortunately, this means that the two random-valued arrays have no useful keys, and must wait for a full collection to be reclaimed.

The portions of the arrays in the keyed segment will be part of some cluster, depending on what key object happens to be in memory before them. Some small key object in the heap will inherit, by proximity, the cluster that contains the arrays. When this key object is free, the collector will attempt to reclaim the array. The key object that it needs to insulate itself from the collection attempt is languishing, unrecognized, in the stack-only segment. This is the reason for the low "Quality" number in Table 5.3 for the matrix multiply.

To get the matrix test back on track, several small changes were made to the matrix source code with full knowledge of the collection style. First, the random-valued matrices can be built in two parts, each smaller than the tenuring threshold, and put in old objects before they are tenured. This lets the system see them as key objects. That done, the order of the matrices' creation is changed so that the data values for each matrix immediately follow the skeleton of the matrix. In the original

[12]Each Smalltalk object includes, in addition to any data, an eight-byte header that includes a four-byte pointer to the data. In eden and the semi-spaces this header usually is placed immediately before the object. When the object is promoted, the header and data are split off. The matrix comprises 41 arrays of size 40 and 1600 floating-point numbers. Each array contains a "size" field as well as slots for its data, and is of size $40 \times 4 + 4 + 8 = 172$ bytes. Each floating-point number is $4 + 8 = 12$ bytes. The total for the matrix is 26252 bytes. The tenuring threshold is a frustrating 25600 bytes.

routine, the bones of the result were created first, then the two multiplicands, then the result was fleshed out. The last change clobbers all three values at the same time to ensure that if the objects get out of order in promotion, it won't matter — everything will be free at that time. As expected, the changes raise both the fraction of bytes collected by key objects [Table 5.1, last column] and the fraction of the time that the key objects are correct when they indicate a collection opportunity [Table 5.3, last column].

The "session replay" sees a more varying environment. There are large swings in the key count as many related clusters are all found unreachable at the same time. Even this example, showing two days of slow systems use, does not promote a large volume of objects. Generational collection has promoted only 0.35% of the allocated objects, and tests on longer and more diverse user sessions are lacking.

As another measure of the effectiveness of key objects, I re-ran the "session replay" test, varying the size of the semi-spaces, and looking for the size that would promote roughly the same volume of objects that the combination of the current 25 kb semi-spaces and key object collector recovers. This would find a generation size at which the burden on the full collections is the same as the 25 kb generational collector plus key object collection. This size turns out to be about 100 kb — at that size the generational collector picks up about another quarter-megabyte, the volume collected by the key objects.

This could be achieved by adding a second generation of 75 kb rather than expanding the first generation, to attempt to keep the real-time response guarantee of the smaller two-space collections. A more complete test would be needed to gauge the trade-offs between a larger generation size and the addition of key objects.

The size of the remembered set compared to the number of objects that point to key objects provides a rough yardstick to one of the major costs of implementing key objects — finding dead keys. Just as the remembered set can be used to find roots for collections in the new objects, the set of objects that point to keys — the key-pins — must be examined to see what keys are reachable. Both of these sets may need to be updated when an old object is changed, and if new keys are excluded, no other stores need change the key-pin list. The number of stores into old objects tends to

be low, and using a key-pins list to find dead keys seems attractive[23].

Chapter 6

Conclusions

The synthetic tests used in Chapter 5 are quite simplistic, and the two-day session replay promotes few objects. Projections from those tests to system-wide collection improvements would have to be filtered through a detailed analysis of a realistic implementation of key objects and clusters, which can only be designed through the usual systems-building loops of trial and error, which would itself produce better *in vivo* tests.

The decisions made in the mock-up key object collector presented in Chapter 5 prove surprisingly effective at locating clusters, but have some short-comings. The heuristics for choosing key objects are too simple, and may be too closely aligned with the specific programs in the suite or the style used in those programs.

Not every inter-generational store should be read as indicating a key object. Some of the objects are clearly not likely to provide any leverage into a larger cluster — the floating-point numbers will not be the roots of any more objects. Many inter-generational stores are still initializing stores. If these can be detected — perhaps simply by checking to see if the value overwritten was a valid pointer — the first key placed in any cell would not be detected, but subsequent ones would be, and the initializing stores would not spoof the collector.

Treating objects that don't contain pointers as key objects is roughly the same as making them reference-counted — they will be collected when all the pointers to them are gone. If we allow these objects to be keys, then we must ensure that ill-chosen

key objects gracefully degrade into reference counting.

In reference counting, the costs of recovering memory are not related to volume of objects, as with tracing collection, but to the number of writes to memory. If the older objects are more stable, as is usually assumed, a variant of reference counting — as is used for generational collection — could easily be more efficient than any kind of tracing collection. As heaps grow to contain more and more live objects this gap will become more apparent — collectors in the future may never be able to afford to trace the entire heap to recover a small fraction of the memory.

The decision to make a key's cluster be the promoted objects between this key and the next key has many flaws. It requires the collector to compact memory when it wants to reuse internal fragments. Without compaction, new objects might be introduced into clusters that were not related to them. Another problem shows up when a key and only part of its cluster are collected: the remainder of the cluster is given to whatever key object happened to come before it in memory.

The inflexibility of the keys and clusters once they are promoted can also cause problems when the key was poorly chosen, most notably when the key is involved in reference cycles from its own cluster. When the last external reference is deleted, the key will be triggered and the cluster threatened with collection. If the cycle survives the collection, as will happen if there is an external reference to any member of it, the key will continue to trigger collection of the cluster at every key object collection. The mock-up scheme does not have the ability to let the object that is the target of the external reference take over the role of key object for the cluster, as it probably should.

No direct measurement was attempted of inter-area pointers that would rescue objects from a cluster that appeared collectible. These pointers need to be accounted for in calculating the cost of partitioning the heap into clusters — they are required to allow the collector to attack partitions individually. Some of the false hits in the clusters, where the key indicates garbage but the garbage isn't present, may be due to these pointers. The low volumes of falsely-indicated garbage is encouraging — not many cross-cluster keys are interfering with the collector. Some of the further developments in good partitioning criteria will go hand in glove with criteria for

finding good key objects.

If a collector is too aggressive in looking for unused memory, it will threaten memory that is not garbage — this would show up as a low "Quality" number in Table 5.3. If a collector is not aggressive enough, it will fail to threaten memory that is garbage — this would show up as a low "Key Collection" percentage in Table 5.1. Ideally, both of these figures would be 100%. This would indicate that the collector is threating only collectible objects, and is threatening all unreachable objects.

Despite the short-comings of the mock-up, it is surprisingly effective. In the "matrix," "array," and "tree" examples it rarely says an object is garbage when it is not, and approaches perfection in some cases. The modified matrix example has a similar false-positive rate. The "replay" example is the most honest test in the lot, and it manages to find targets for garbage collection clusters that are, in total, about 2/3 garbage, finding two hundred kilobytes of garbage in a 1.2 megabyte heap.

Likewise, the mock-up manages to collect a large fraction of the objects, even with its simple and general method of finding keys and clusters. For the "replay" example, up to 2/3 of the total garbage was collected using key objects. This would allow the system to eliminate 2/3 of the global collections and still keep about the same heap size.

6.1 Future Work

The major implementation barrier to key objects is reconstructing the memory system to allow a small degree of control over object promptness. The key objects must be, in some way, "denied tenure" and collected more often than the entire heap if they are to guide scheduling of collections of the larger heap. In the mock-up, this was done by promoting them and later pretending that they were being collected more frequently.

I believe that the best way to collect key objects is some kind of reference counting — either a version of deferred reference counting or a modification of the schemes used to find roots of collections in young generations. This will allow the collector designer to have some control over promptness in the keys, and from there the clusters. The

scheme will be made more complex by the requirement that references to a key from its own cluster be ignored, and loops involving several clusters and keys will be a particular challenge.

Collecting the key objects will probably require a technique similar to those used to find the roots for a scavenge of the young generation. But the size of the young generation is often limited while the number of keys will grow with larger heaps. As heaps grow, the techniques may require modifications that are more suited to larger sets of roots, and these may, in turn, benefit the remembered set. The links between the key objects will probably be more long-lived than those in the remembered set — the remembered set turns over at least as quickly as the new space. An efficient collector might cache key object reachability results and recalculate those parts of the cache that were invalidated rather than recalculating the entire reachability result from scratch.

This result can be turned around to benefit the remembered set: when the remembered set has not changed from the previous scavenge, the collector could avoid accessing the old objects — the roots of the scavenge are the same as the previous roots. When the mutator is in a phase where the old objects are not being altered, they need not be fetched into memory for the scavenge if the caching information can yield the roots, and the scavenges can be more efficient and better memory citizens[1].

The full value of key objects can not be realized until the heap is split into clusters. These clusters need to have few pointers between them, to keep the bookkeeping costs down and value of key objects up. There is currently work in the memory management community to try to isolate these clusters, mostly to improve paging and cache performance, and this work will also benefit key objects.

I suspect that some key objects will always be idiosyncratic, and that there should always be room for users to tell the system what key objects would do well for their

[1]Monitoring of inter-generational stores seems to be a rich source of information for finding keys, but there are many other tantalizing sources still unexplored. Some Smalltalk experiments suggest that type information may yield good results. Clustering was ignored in these experiments, and the keys were used to control the timing of full collections. Good correlations were found in Smalltalk between those collections that freed objects of certain system types and those collections that freed larger than average volumes of the heap. The best types are process and window related. The best of the best are UnixDiskFileAccessor, ScheduledWindow, and UnixProcess.

particular application. To discover key objects in particular applications, users will need easy-to-use profiling tools and an interface to the memory management system that will allow an application to communicate the key information to the system.

A key object interface to garbage collection would be built more easily on top of a more flexible finalization scheme, where the user could ask to be notified when the key object indicated collection for its cluster, and then to request a collection of the cluster. With these interfaces, more policy decisions could be removed from the garbage collector leaving only mechanism at the core.

Bibliography

[1] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 243–246, 1987.

[2] Andrew W. Appel, John R. Ellis, and Li Kai. Real-time concurrent collection on stock multiprocessors. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7):11–20, July 1988.

[3] Henry G. Baker, Jr. List-processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[4] Henry G. Baker, Jr. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3):66–70, March 1992.

[5] Peter B. Bishop. Computer systems with a very large address space and garbage collection. Technical Report TR-178, Laboratory for Computer Science, MIT, May 1977.

[6] Daniel G. Bobrow. Managing reentrant structures using reference counts. *ACM TOPLAS*, 2(3):269–273, 1980.

[7] Hans-J. Boehm, Alan Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, pages 157–164, June 1991.

[8] Hans-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[9] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[10] Douglas W. Clark and C. Cordell Green. On-the-fly carbage collection: an exercise in cooperation. *Communications of the ACM*, 20(2):78–86, February 1977.

[11] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, 1981.

[12] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[13] Vincent Delacour. Allocation regions and implementations. In *International Workshop on Memory Management, 1992*, September 1992.

[14] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *ACM Symposium on Principles of Programming Langauges*, pages 261–269, January 1990.

[15] John DeTreville. Heap usage in the Topaz environment. Technical Report 63, Digital Systems research Center, August 1990.

[16] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[17] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly carbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[18] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report 25, Digital Systems Reseaerch Center, February 1988.

[19] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[20] H. W. Glaser and P. Thompson. Lazy garbage collection. *Software Practice and Experience*, 17(1):1–4, January 1987.

[21] Barry Hayes. Using key object opportunism to collect old objects. In *Object-Oriented programming: systems languages and applications*, October 1991.

[22] Barry Hayes. Finalization in the collector interface. In *International Workshop on Memory Management, 1992*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[23] Antony L. Hosking and Darko Moss, J. Eliot B. Stefanović. A comparative performance evaluation of write barrier implementations. In *Object-Oriented programming: systems languages and applications*, pages 92–109, October 1992.

[24] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *International Workshop on Memory Management, 1992*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[25] IBM Corporation. OS PL/I Checkout and Optimizing Compilers: Language Reference Manual. Program Product, October 1976. Order Number GC33-0009-4.

[26] Donald E. Knuth. *Fundamental algorithms*, volume 1 of *The art of computer programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[27] Charles Philip Lecht. *The Programmer's PL/I*. McGraw Hill, New York, 1968.

[28] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[29] John McCarthy. Recursive functions of symbolic expressions and their computations by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[30] David A. Moon. Garbage collection in a large lisp system. In *ACM Symposium on Lisp and Functional Languages*, pages 235–246, August 1984.

[31] ParcPlace Systems. *ObjectWorks \ Smalltalk User's Guide, Release 4.1.* ParcPlace Systems, Inc, Mountain View, CA, 1992.

[32] Ken Pier. A retrospective on the Dorado, a high-performance personal workstation. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983.

[33] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, staticly-checked, concurrent language. Technical Report CSL-84-7, Xerox Corporation, July 1985.

[34] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Computer Science Laboratory, Stanford University, March 1987.

[35] James W. Stamos. Programmer-invoked, local garbage collections: A design. Technical Report — Draft —, MIT Laboratory for Computer Science, 1986.

[36] James William Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Programming Languages and Systems*, 2(2):155–180, May 1984.

[37] Guy L. Steele Jr. Multiprocessing compactifying garbage collecting. *Communications of the ACM*, 18(9):495–508, September 1975.

[38] Sun Microsystems. *NeWS 2.1 Programmer's Guide.* Sun Microsystems, Inc, Mountain View, CA, 1990.

[39] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. Technical Report CSL-86-1, Xerox Corporation, 1986.

[40] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Practical Programming Conference*, pages 157–167, April 1984.

[41] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[42] Jon L. White. Address/memory management for a gigantic lisp environment or, GC considered harmful. In *Conference Record of the 1980 LISP Conference*, pages 119–127, Redwood Estates, CA, June 1980.

[43] Paul R. Wilson. Design of the opportunistic grabge collector. In *Object-Oriented programming: systems languages and applications*, pages 23–35, October 1989.

[44] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. *SIGPLAN Notices*, 26(3):45–52, March 1991.

[45] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management, 1992*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[46] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 177–191, June 1991. Toronto, Canada.

[47] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, June 1990.

[48] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. Technical Report CU-CS-604-92, University of Colorado at Boulder, July 1992.

[49] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, March 1989.

Appendix A

Cedar Data

All of the samples for Cedar were drawn from June to October, 1990, from ten Dorados at Xerox's Palo Alto Research Center. The changes to the garbage collector and allocator recorded every allocation and deallocation. Some circular storage may have been created by the mutators, and would not have been recorded as unreachable by the reference counting collector. All lifetimes are calculated as the difference in bytes allocated between the creation of the object and the point when the garbage collector discovers that the object is unreachable. The collector runs about every 16 kb allocated, and for the purposes of this data, all lifetimes are rounded down to the nearest 16 kb.

Table A.1 lists the machines by name, and gives a rough idea of the work being done on each machine at the time of the samples. All the machines are mixed use — almost all users read mail, develop tools, write papers, and use other tools on the same machines. Bennington was a public machine and the two samples taken from it are shown individually.

Table A.2 shows the two largest sessions for each machine. Shown is the total number of bytes in the sample, the size of the cohort chosen from before the last 5 Mb, and the fraction of the cohort that survives longer than 5 Mb. Three of the samples are smaller than 5 Mb, and so yield no cohort. The table also shows the total real-time length of the session recorded. The allocation rates vary a great deal — one session lasting three days allocated under four megabytes, another lasting four hours

Baobab Systems programming — continuing work on a hypertext system.

Bennington Public Machine [session 1] — two different programming development efforts, and remote debugging of another machine.

Bennington Public Machine [session 2] — producing two drawings from the circuit simulation package. Not enough data for 5 Mb lifetimes.

Bluebell Systems programming — development of X-windows tools, and continuing debugging and maintenance of a Cedar-to-C translator.

Fairmont Graphic Arts — production of several complex drawings using Gargoyle.

Leyte Systems programming — work in support of color documents.

Queenfish Systems programming — continuing work on multi-media documents, mostly concerning audio.

Saratoga Support — support work for the user of Shangrila.

Sea-wolf Systems programming / Administration — mixed use machine used by lab manager. No sessions long enough for 5 Mb lifetimes.

Shangrila Systems programming / Administration — mixed use machine used by lab manager.

Skipjack Systems programming —

Table A.1: Dorados Monitored

allocated more than 47 megabytes. The former is a trace of a mostly-idle machine, and the later is a user of a memory-intensive graphics editor.

For the seventeen sessions that contribute to the sample set, Figures A.1 through A.34 show the cumulative population curve and the instantaneous generation volume needed to cause collections to retain only 10% of the storage promoted to that volume. These volume charts will have gaps where no objects of that lifetime exist.

Machine	Total Size (bytes)	Cohort Size (bytes)	5 Mb Surv	Span
Baobab	8875822	3145726	.198	25 hrs
Baobab	23065392	16773306	.0860	44 hrs
Bennington	13741523	8388591	.0832	29 hrs
Bennington	6373003	1048568	.0578	8.5 hrs
Bluebell	8819380	3145712	.268	3.5 days
Bluebell	20276620	14680062	.138	26 hrs
Fairmont	63119236	57667794	.0459	9 hrs
Fairmont	47280936	41939886	.0547	4 hrs
Leyte	3856601	0	3 days	
Leyte	11776523	6291454	.0845	11 hrs
Queenfish	63496066	57671554	.0301	28 hrs
Queenfish	22928352	16774116	.0844	57 hrs
Saratoga	23491975	17825708	.0685	4.5 days
Saratoga	19196479	13631485	.0718	4 days
Sea-wolf	1681036	0		1 hr
Sea-wolf	4252980	0		8 hrs
Shangrila	17590613	11534332	.0540	35 hrs
Shangrila	20319530	14676961	.0350	3 days
Skipjack	11511730	5242880	.144	2 hrs
Skipjack	9419842	3134651	.168	6 hrs
total:	401073639	293572786	.0984	30 days

Table A.2: Session Sizes for each Dorado

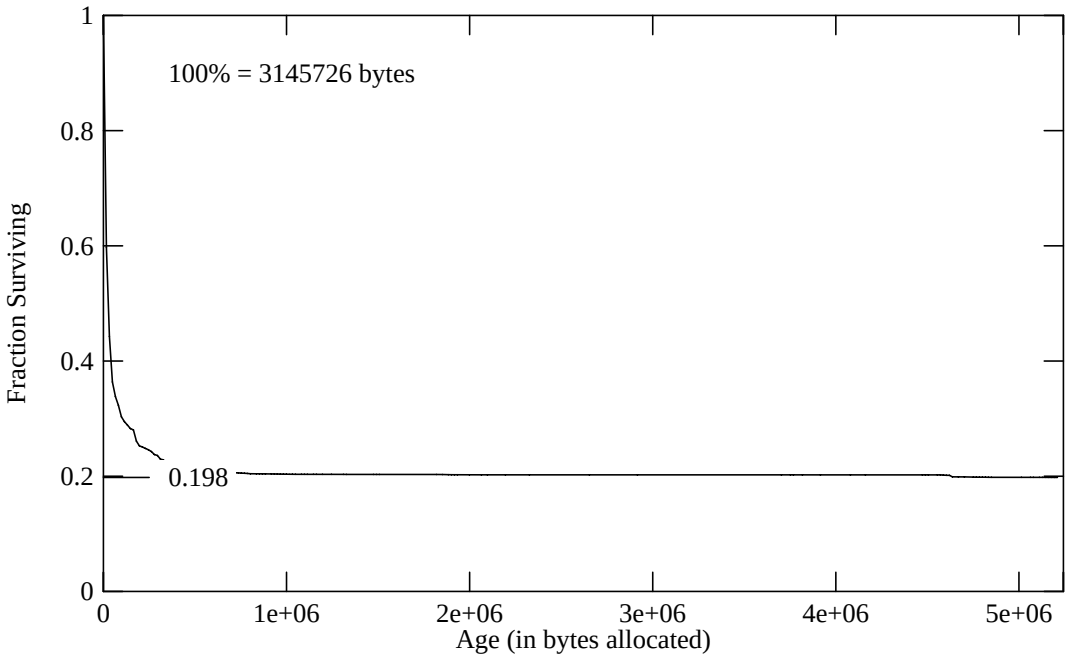

Figure A.1: Baobab 1, Decay of volume with time

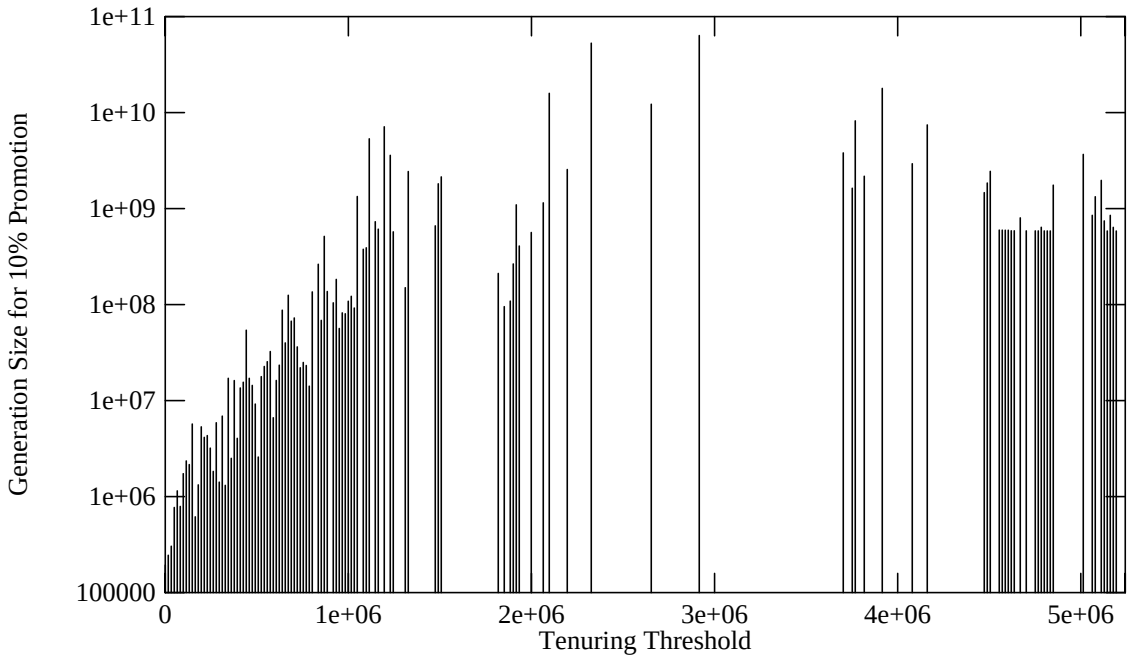

Figure A.2: Baobab 1, Generation size needed for 90% collection rate

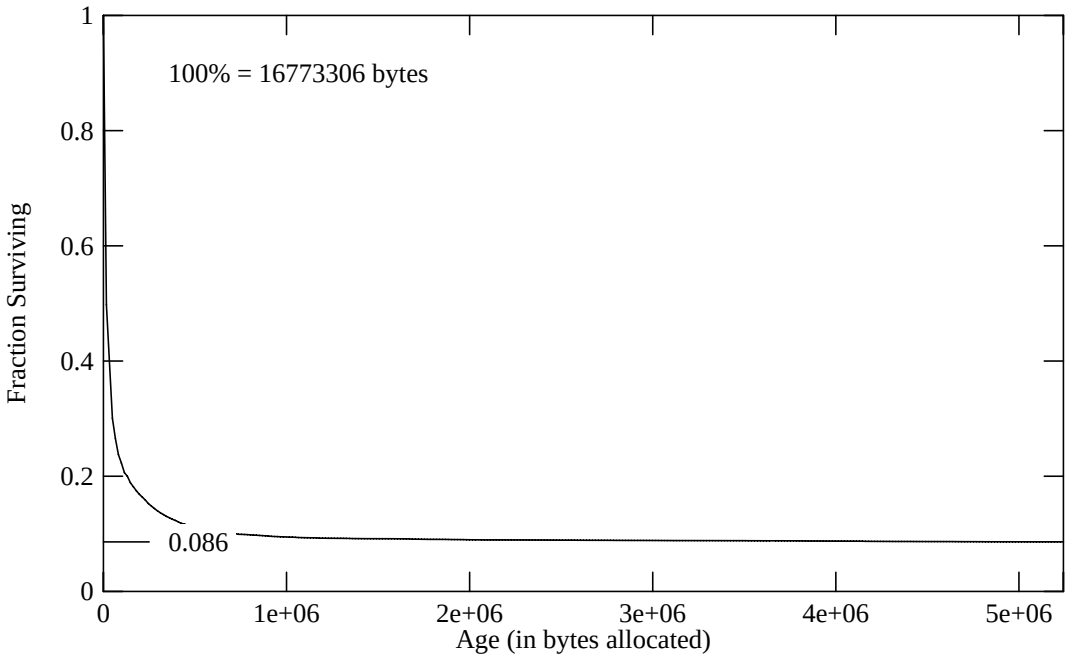

Figure A.3: Baobab 2, Decay of volume with time

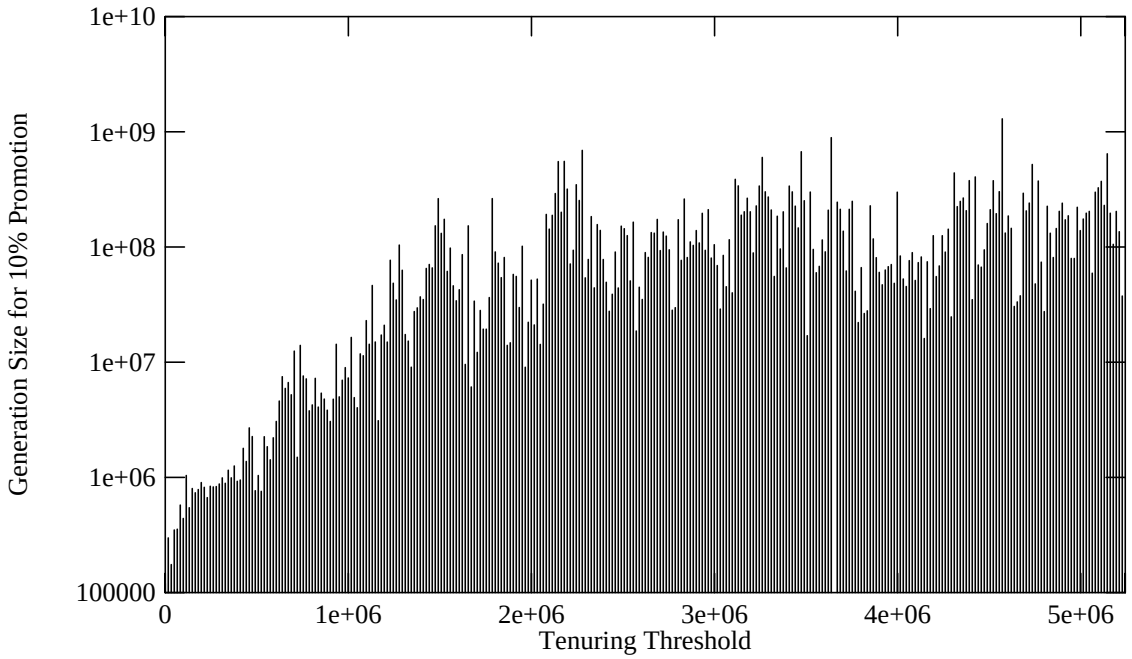

Figure A.4: Baobab 2, Generation size needed for 90% collection rate

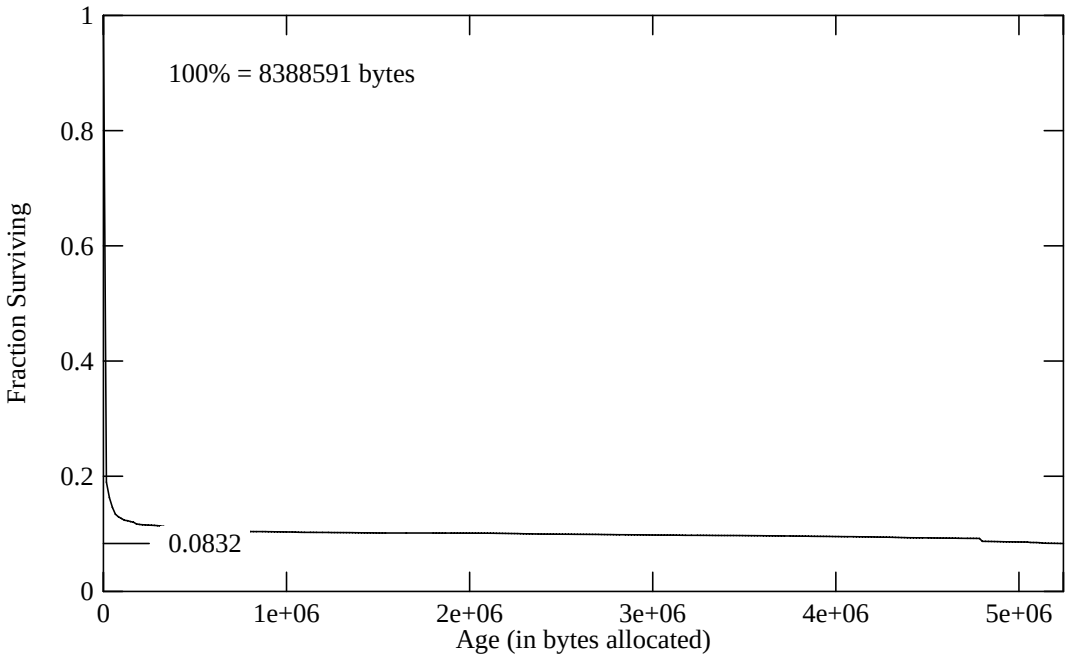

Figure A.5: Bennington 1, Decay of volume with time

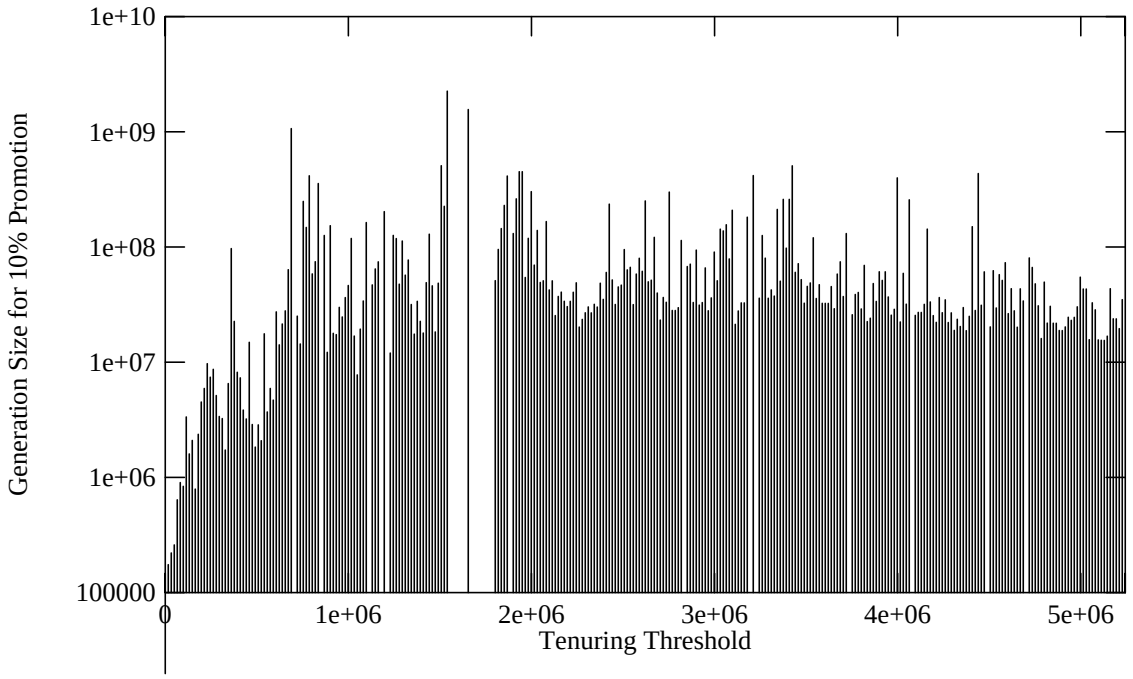

Figure A.6: Bennington 1, Generation size needed for 90% collection rate

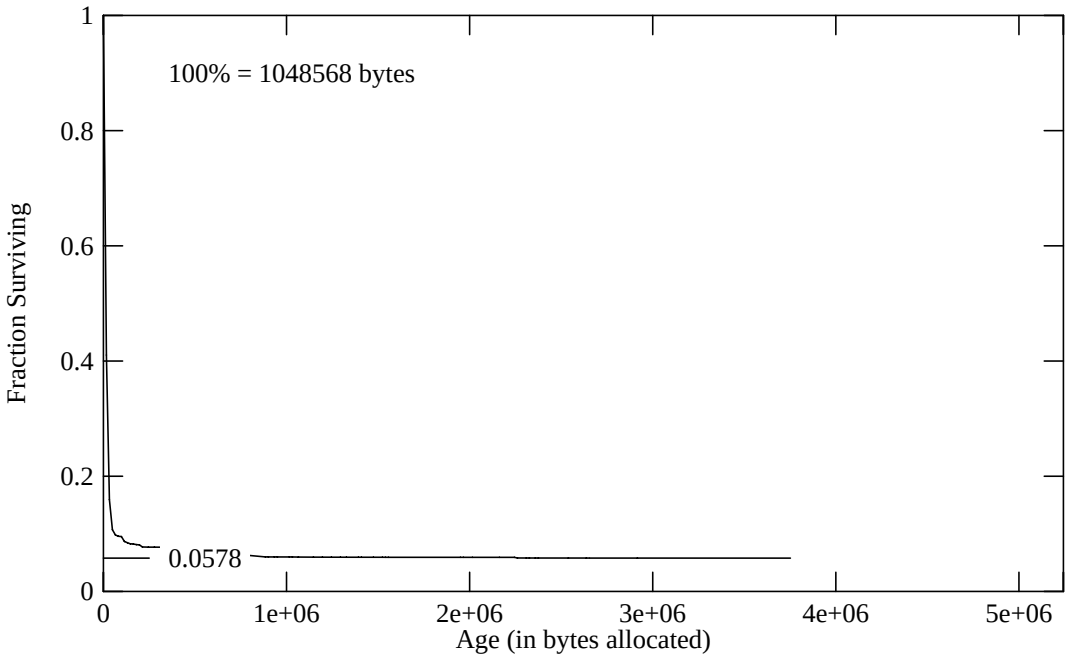

Figure A.7: Bennington 2, Decay of volume with time

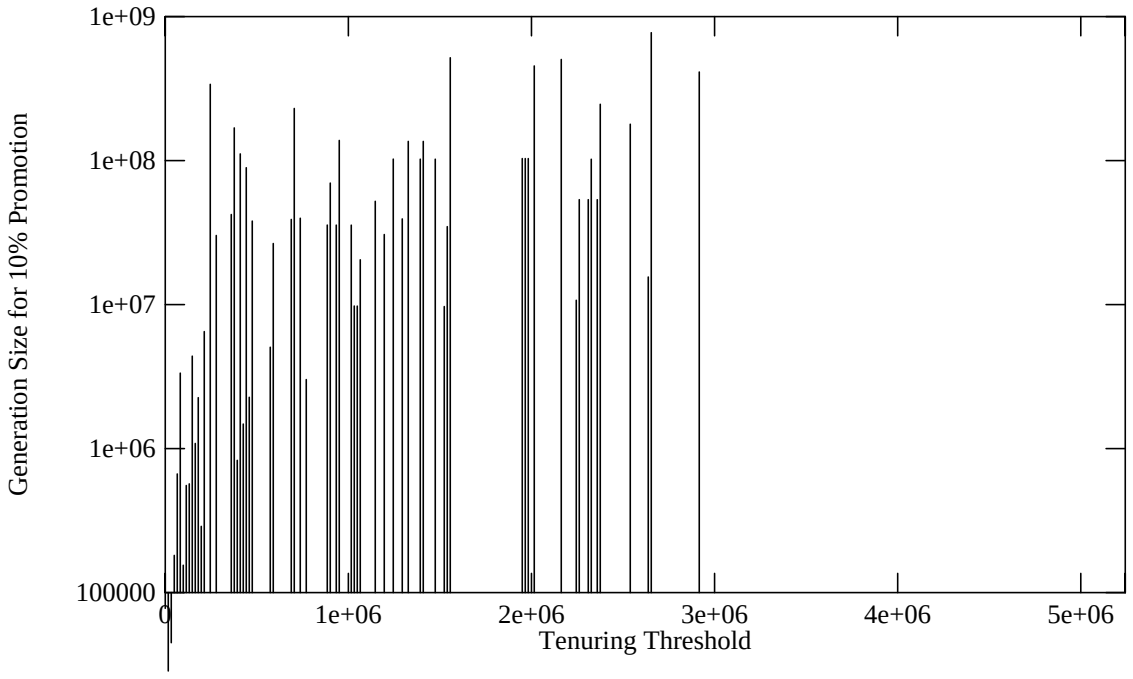

Figure A.8: Bennington 2, Generation size needed for 90% collection rate

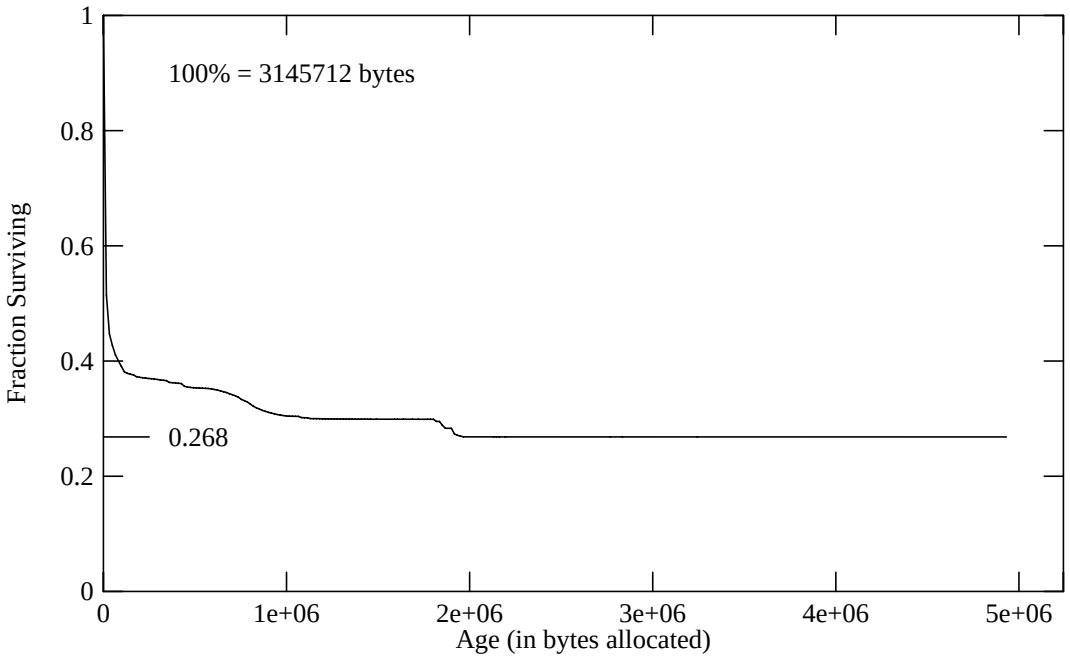

Figure A.9: Bluebell 1, Decay of volume with time

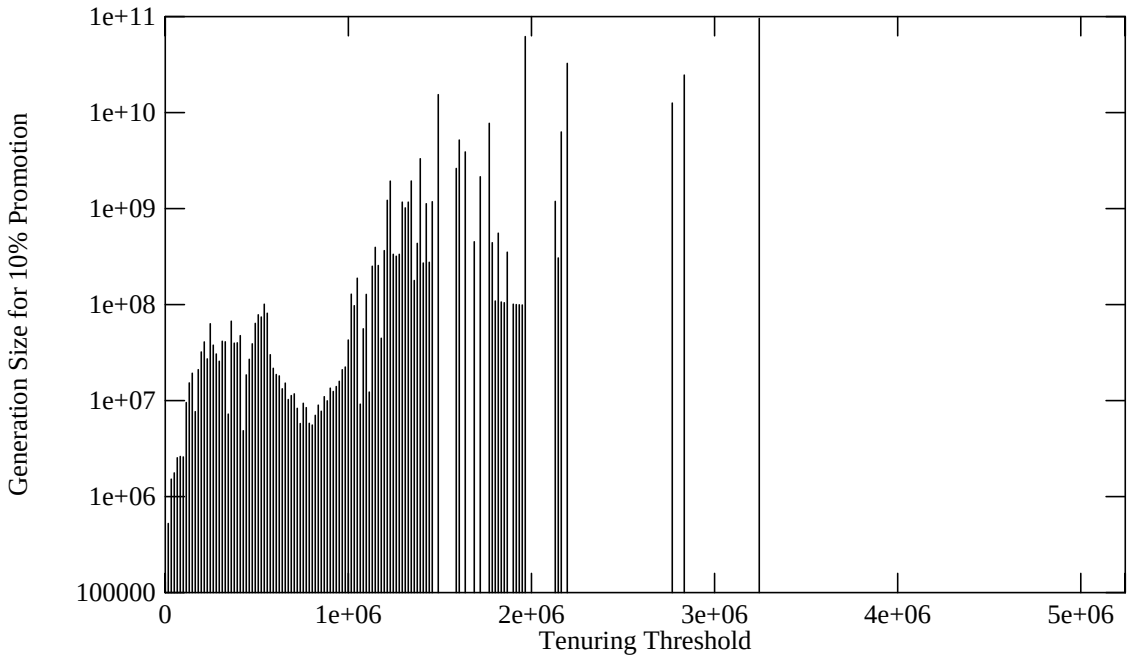

Figure A.10: Bluebell 1, Generation size needed for 90% collection rate

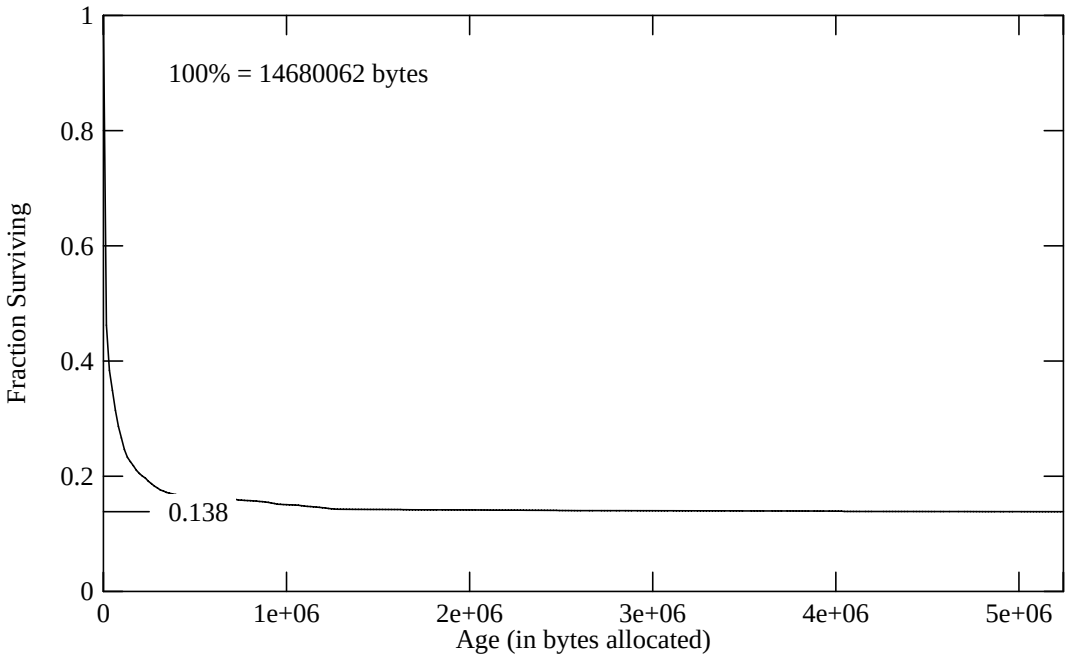

Figure A.11: Bluebell 2, Decay of volume with time

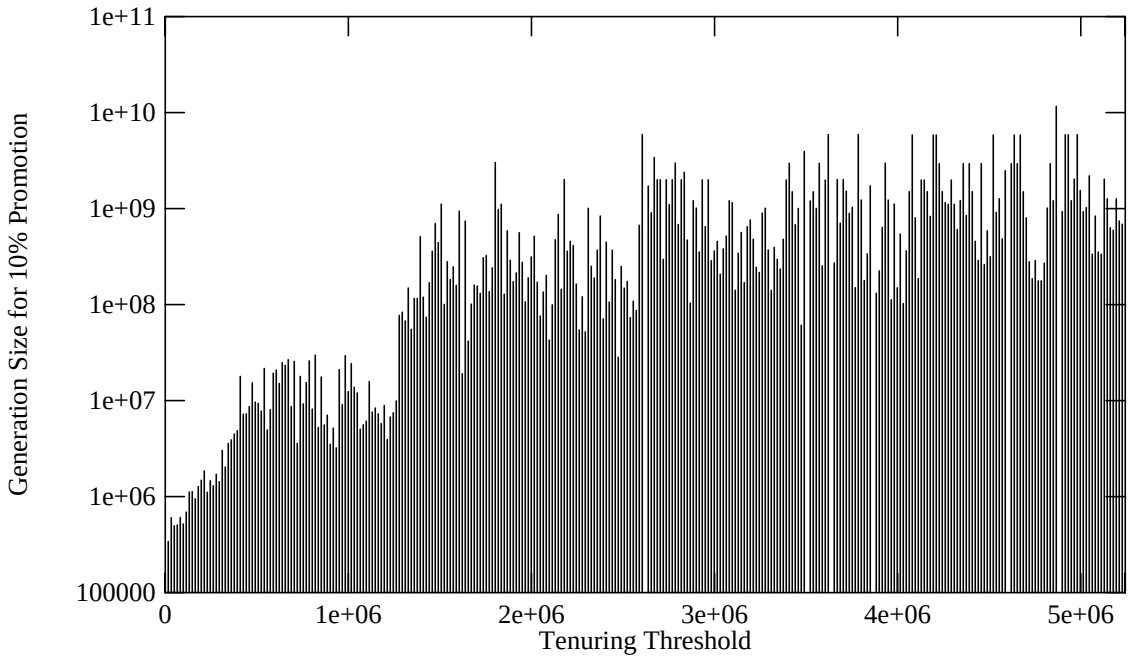

Figure A.12: Bluebell 2, Generation size needed for 90% collection rate

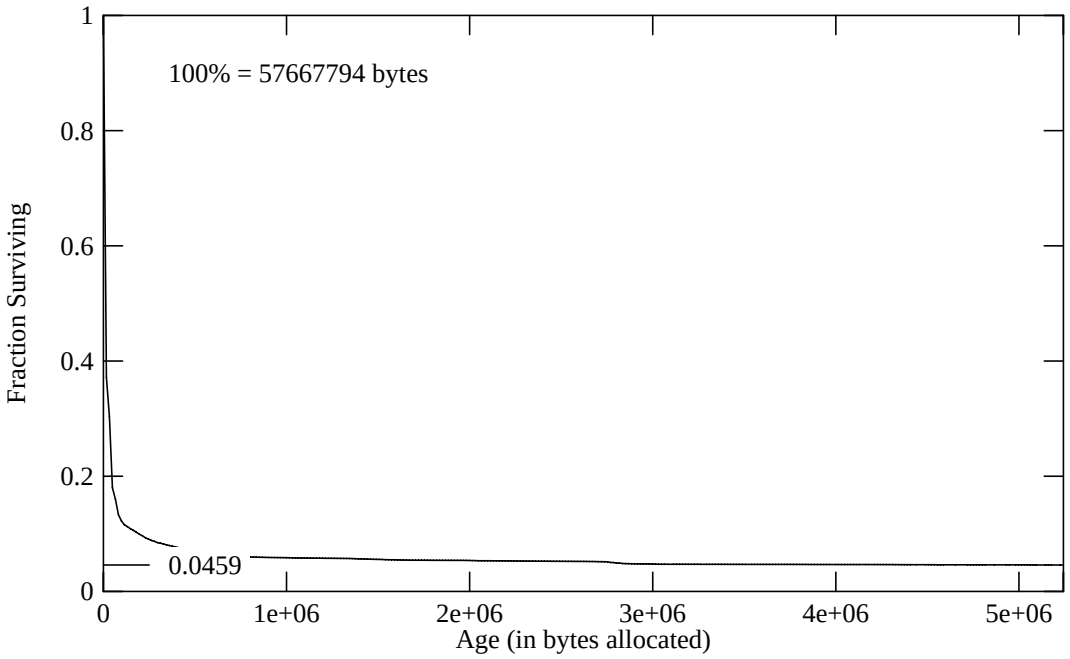

Figure A.13: Fairmont 1, Decay of volume with time

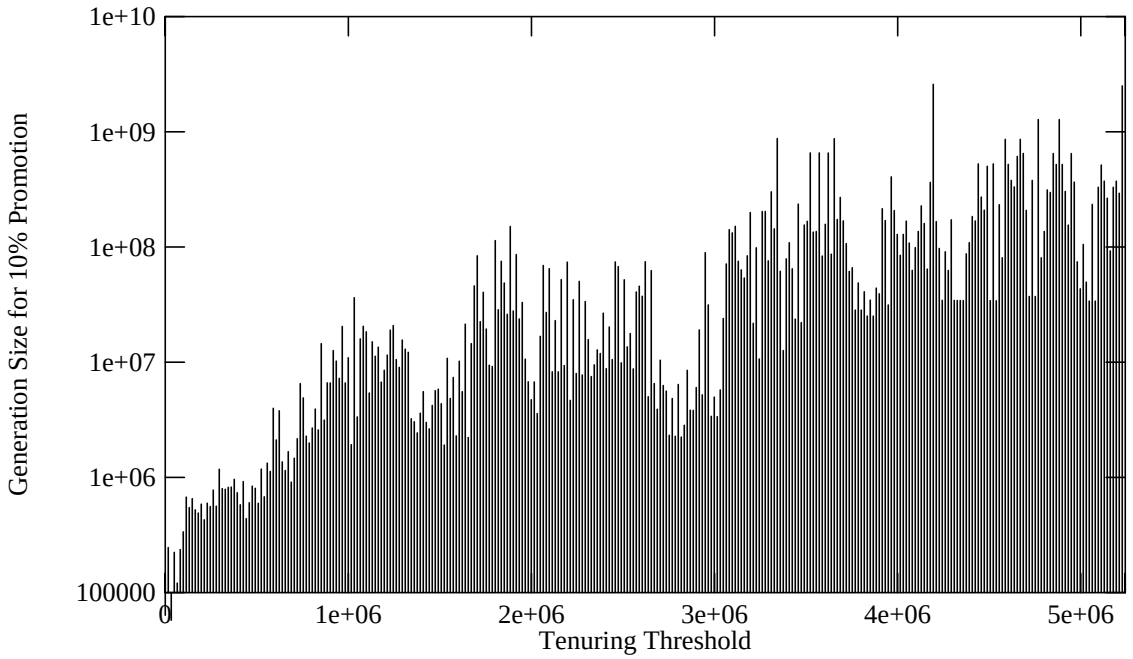

Figure A.14: Fairmont 1, Generation size needed for 90% collection rate

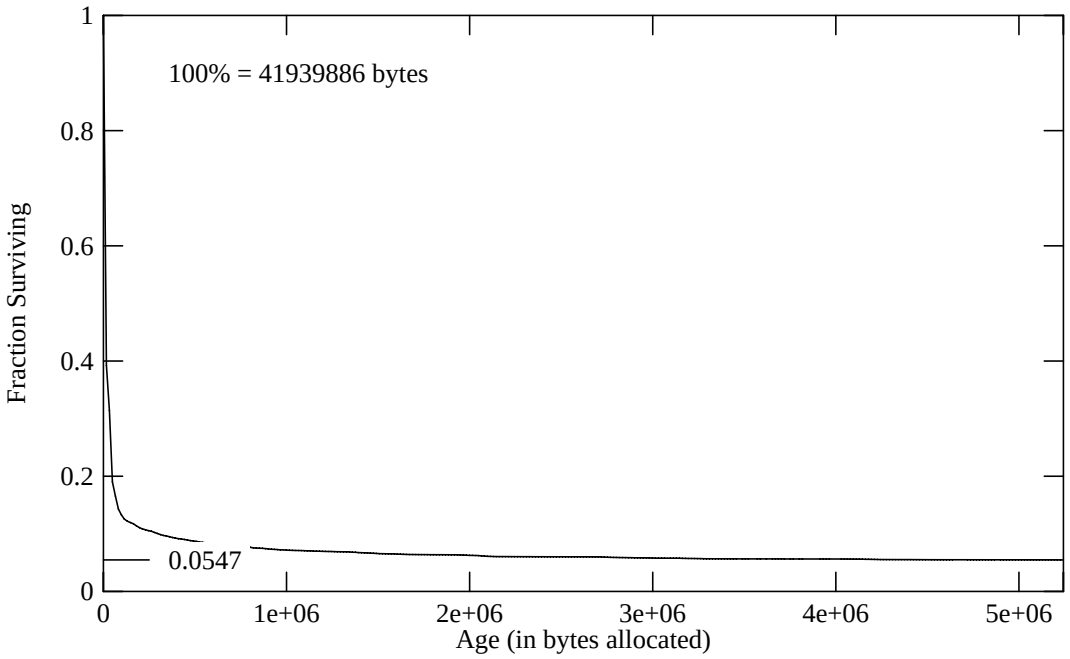

Figure A.15: Fairmont 2, Decay of volume with time

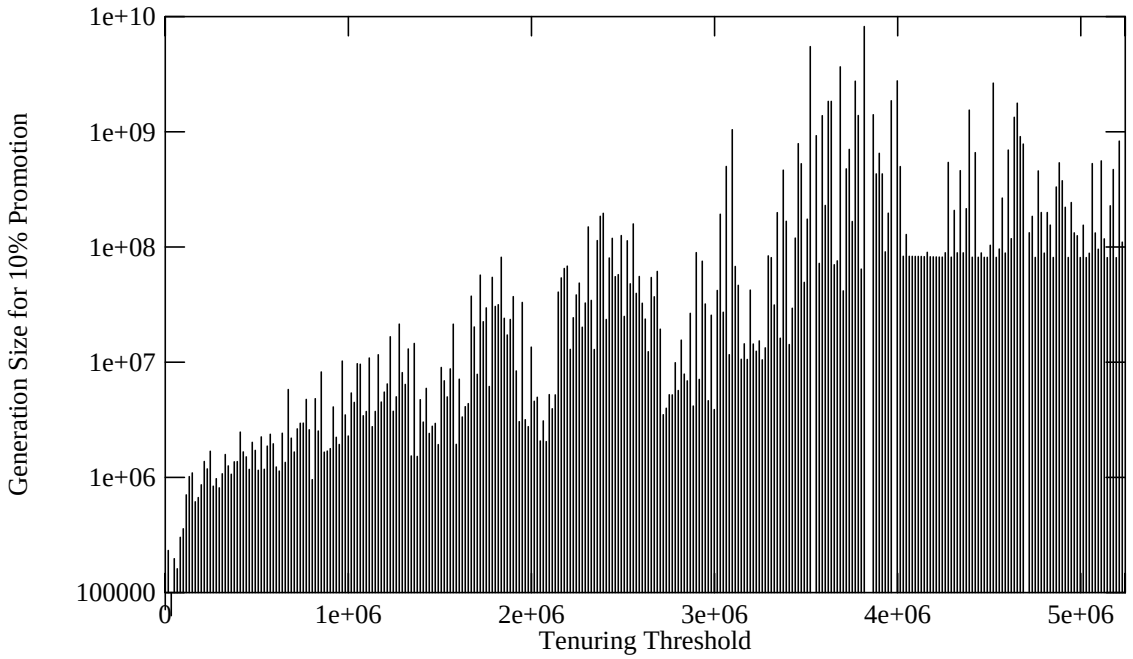

Figure A.16: Fairmont 2, Generation size needed for 90% collection rate

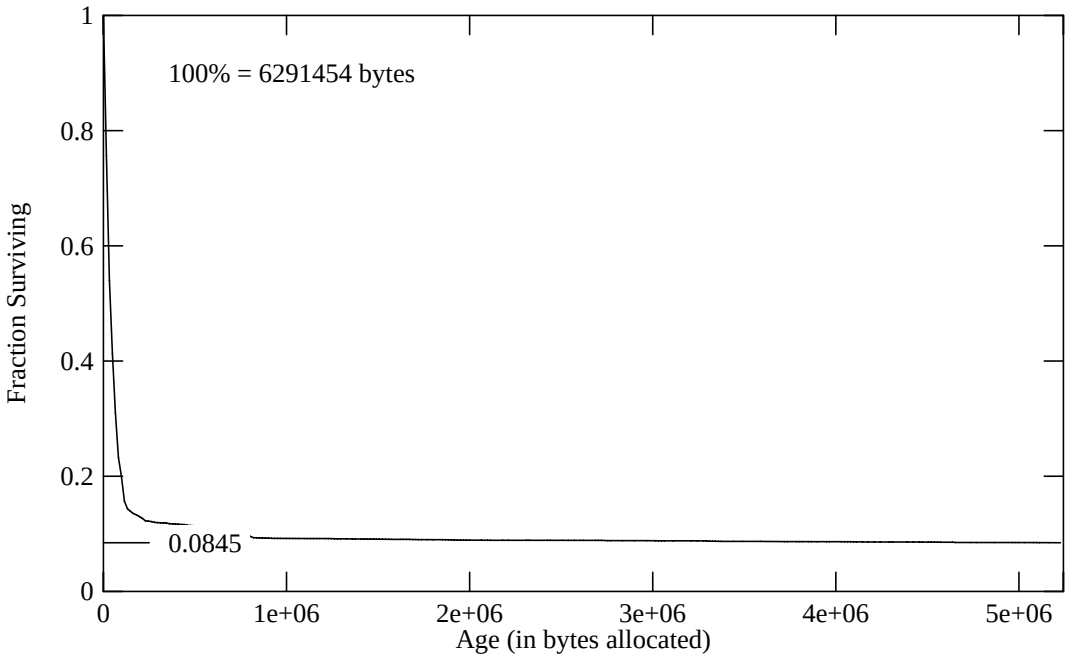

Figure A.17: Leyte 1, Decay of volume with time

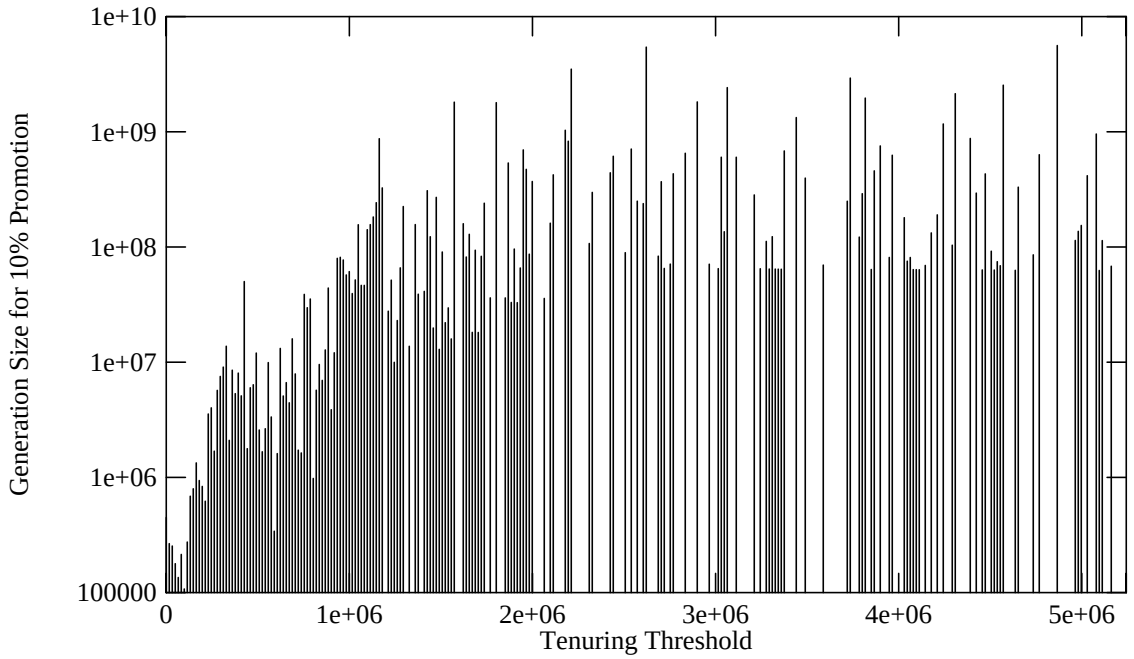

Figure A.18: Leyte 1, Generation size needed for 90% collection rate

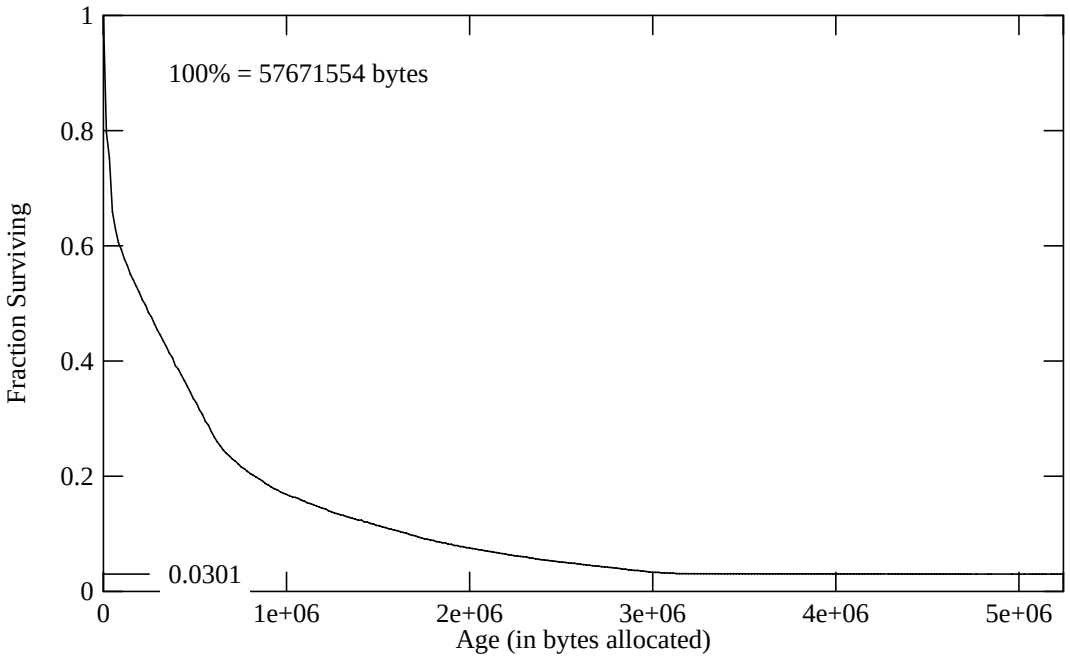

Figure A.19: Queenfish 1, Decay of volume with time

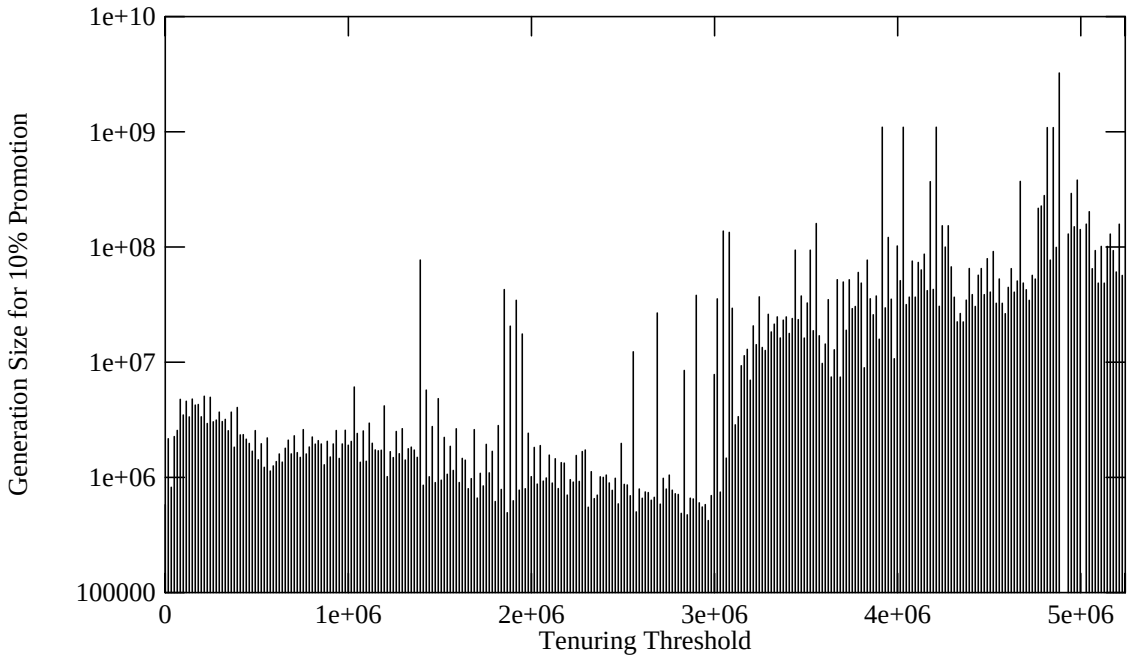

Figure A.20: Queenfish 1, Generation size needed for 90% collection rate

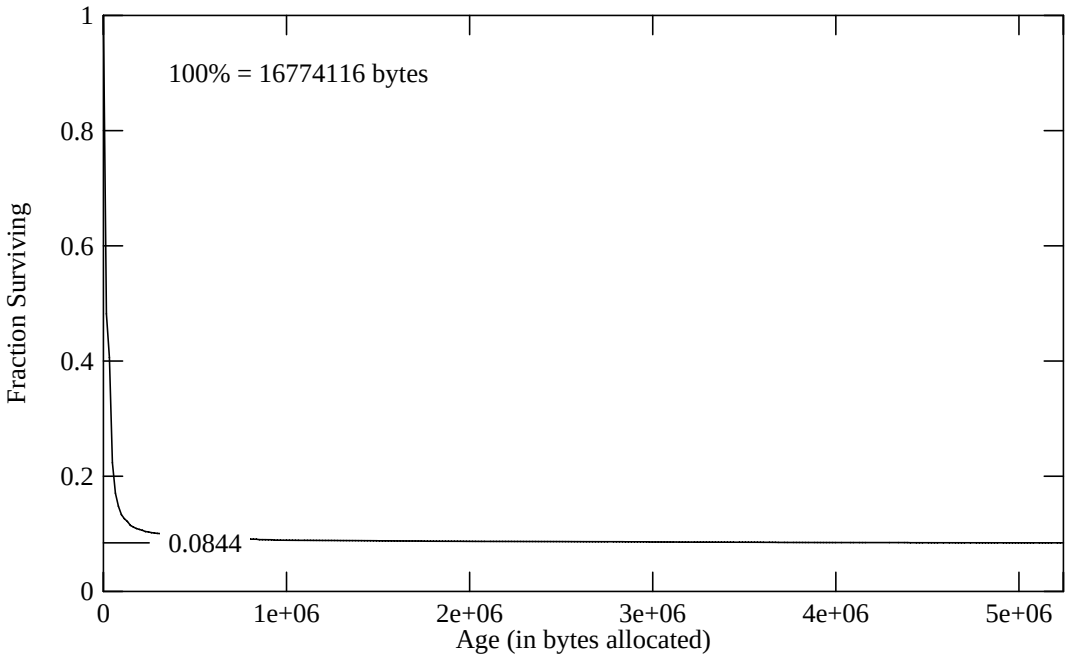

Figure A.21: Queenfish 2, Decay of volume with time

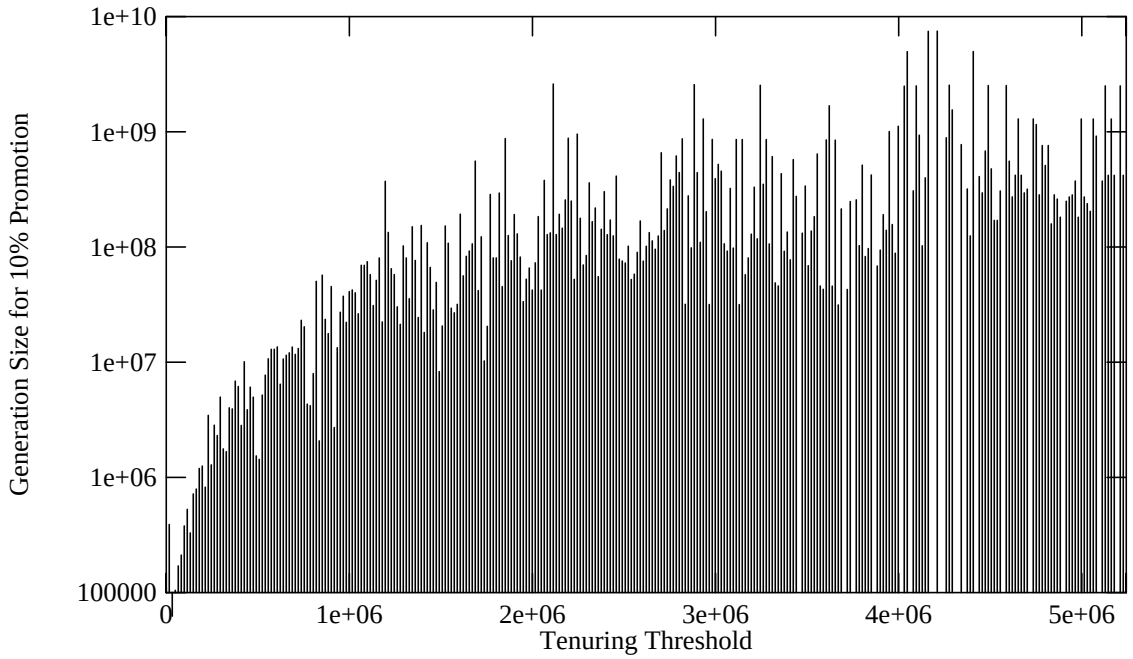

Figure A.22: Queenfish 2, Generation size needed for 90% collection rate

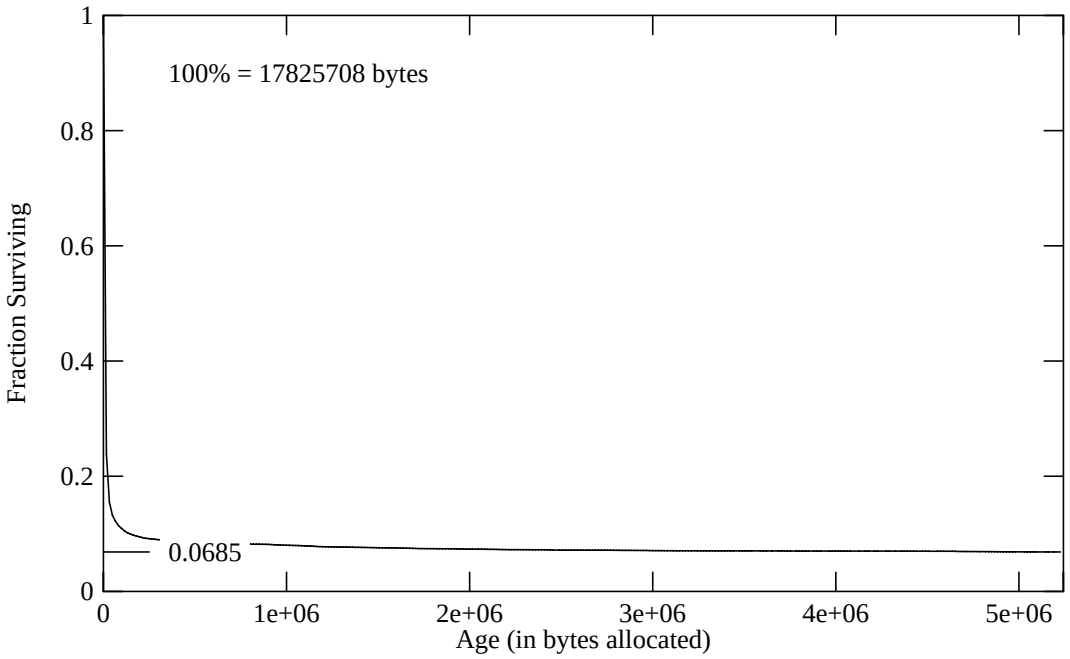

Figure A.23: Saratoga 1, Decay of volume with time

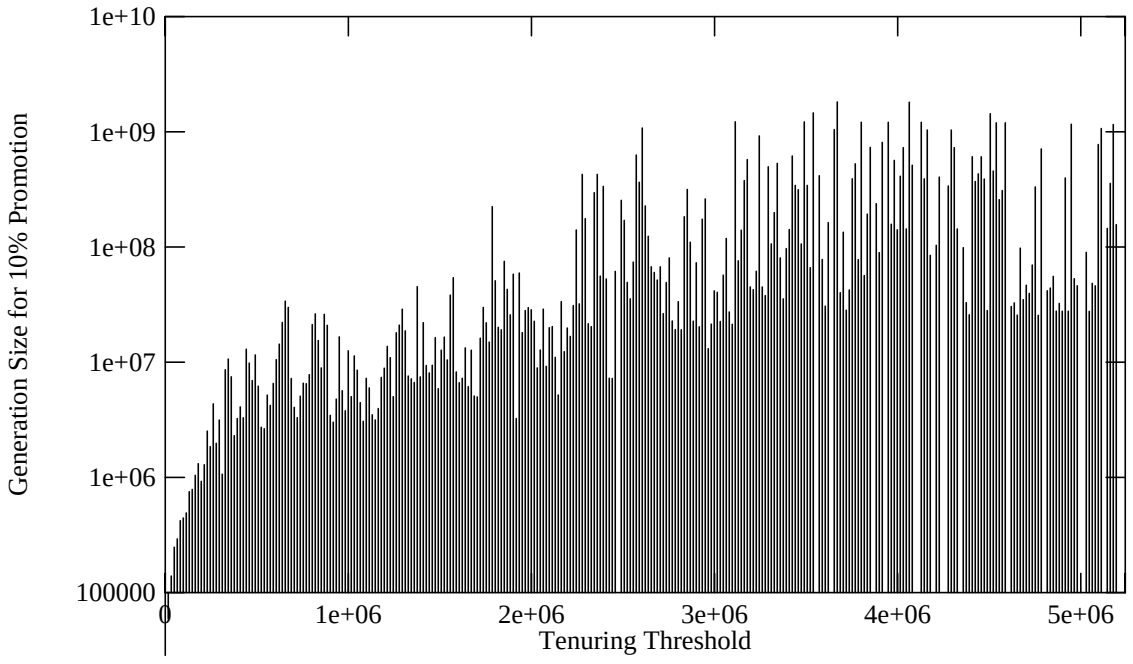

Figure A.24: Saratoga 1, Generation size needed for 90% collection rate

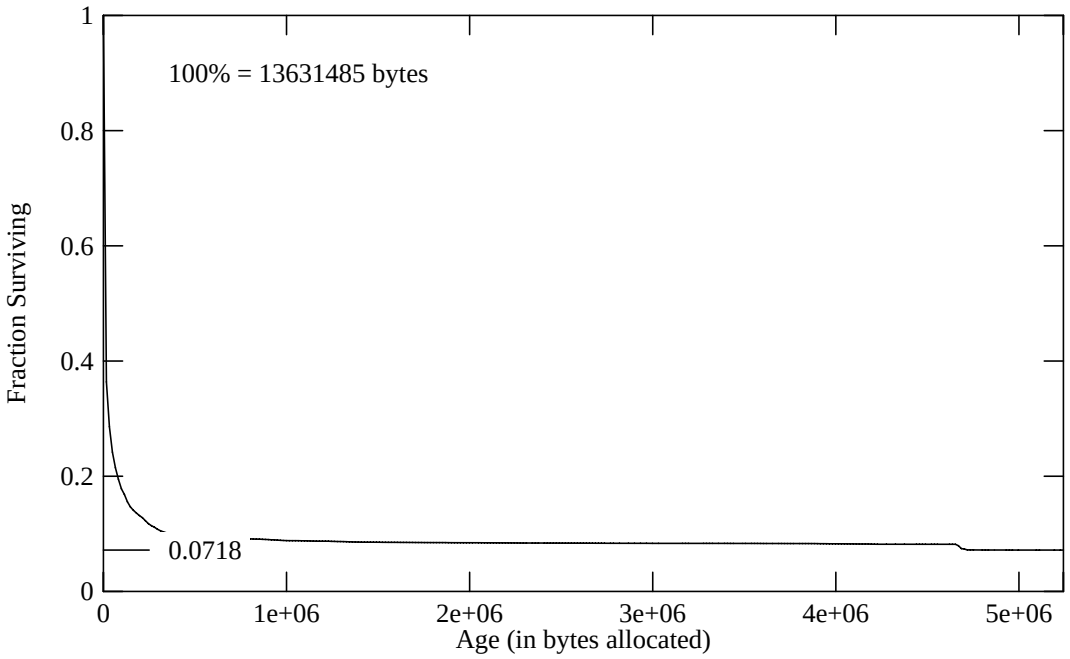

Figure A.25: Saratoga 2, Decay of volume with time

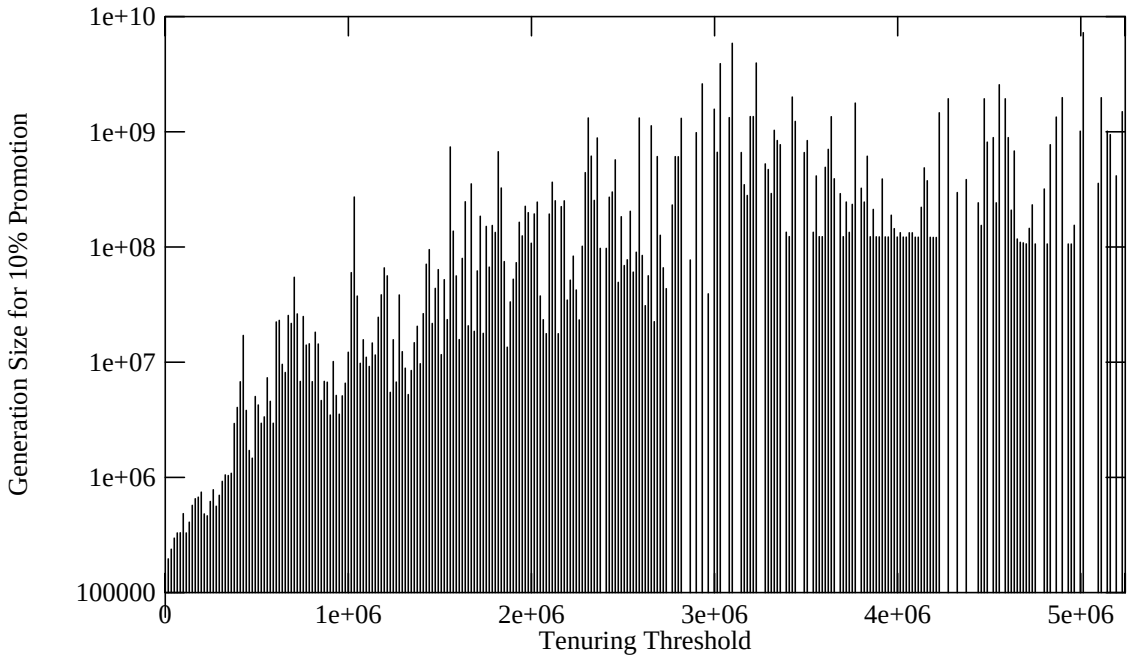

Figure A.26: Saratoga 2, Generation size needed for 90% collection rate

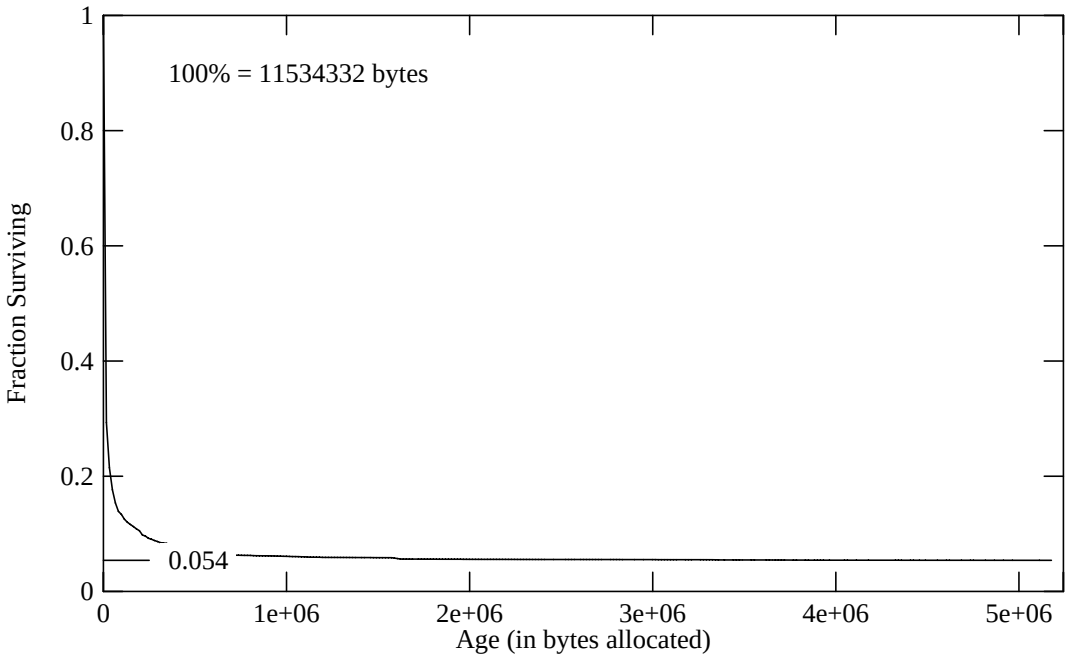

Figure A.27: Shangrila 1, Decay of volume with time

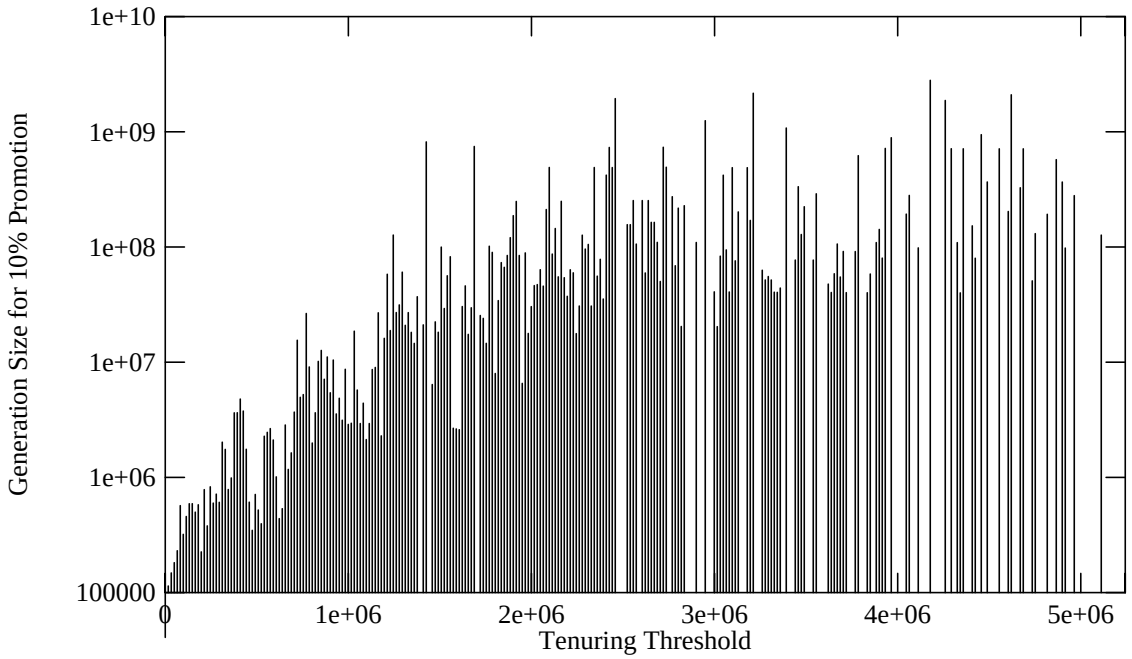

Figure A.28: Shangrila 1, Generation size needed for 90% collection rate

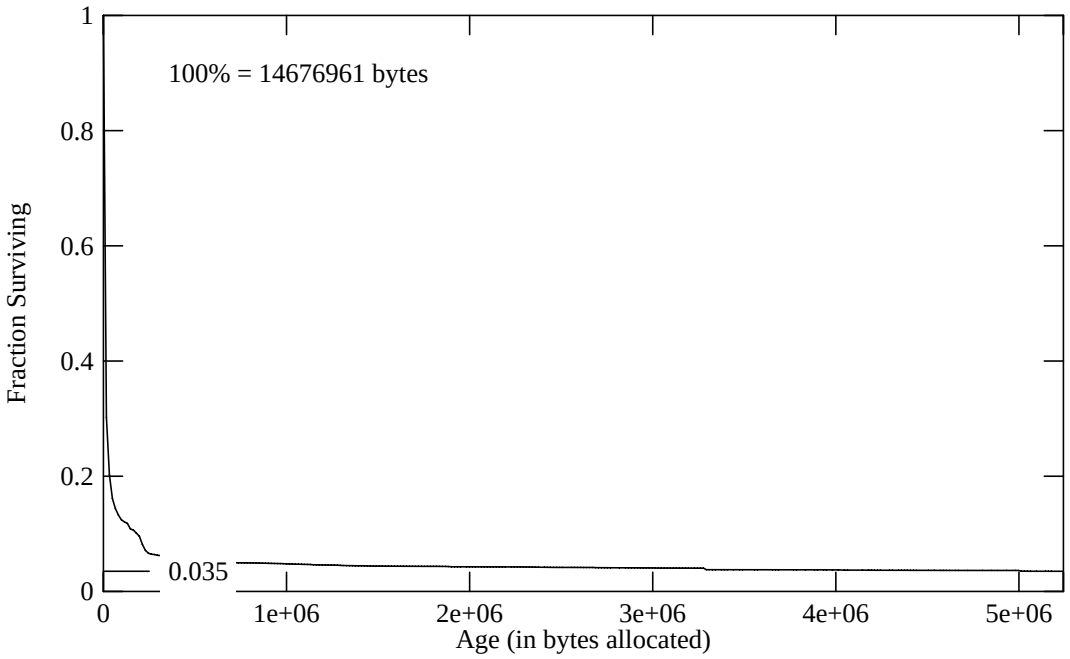

Figure A.29: Shangrila 2, Decay of volume with time

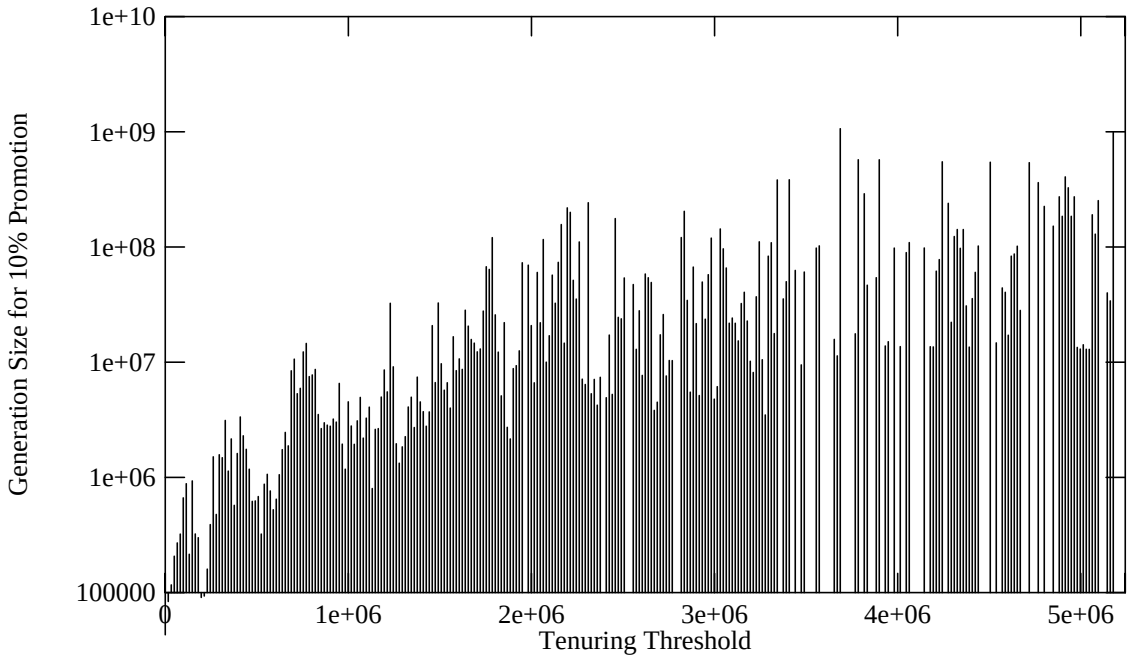

Figure A.30: Shangrila 2, Generation size needed for 90% collection rate

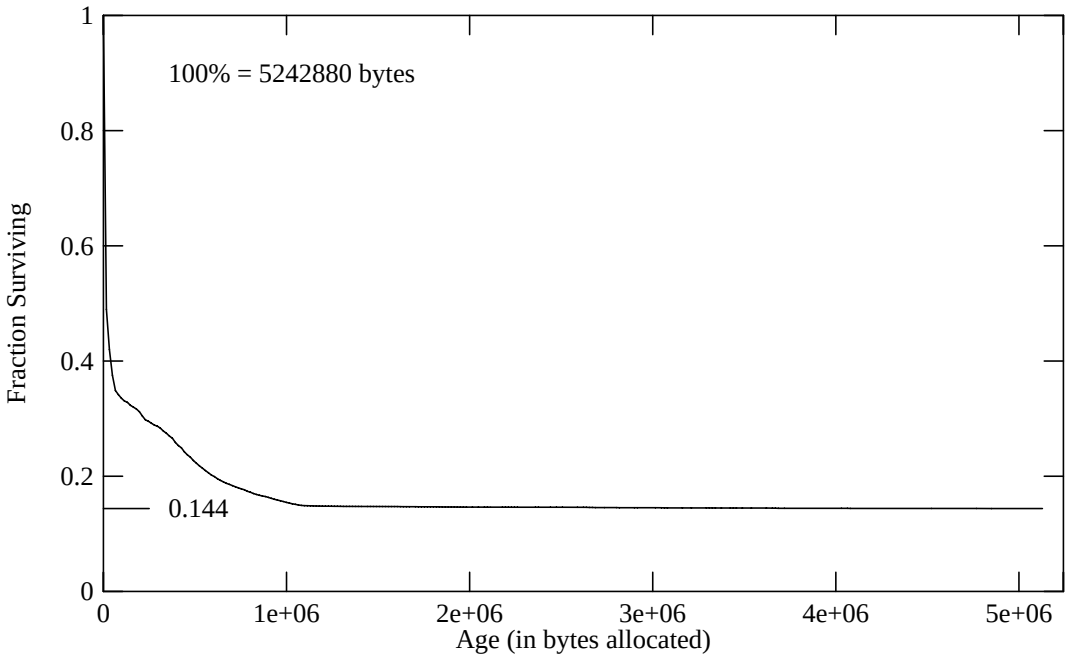

Figure A.31: Skipjack 1, Decay of volume with time

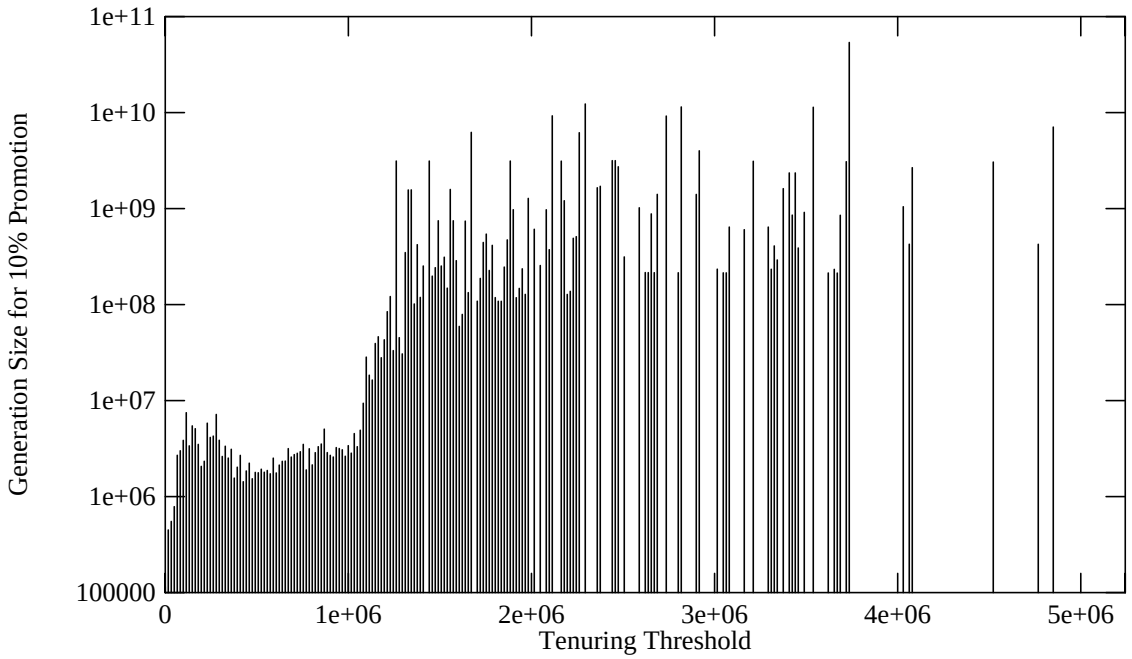

Figure A.32: Skipjack 1, Generation size needed for 90% collection rate

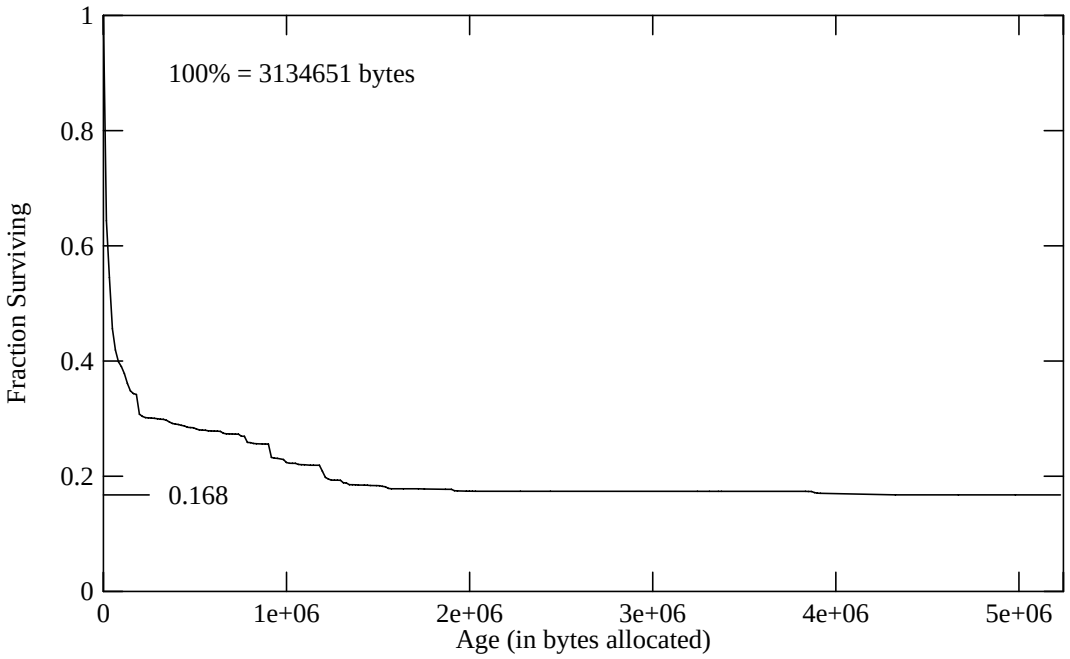

Figure A.33: Skipjack 2, Decay of volume with time

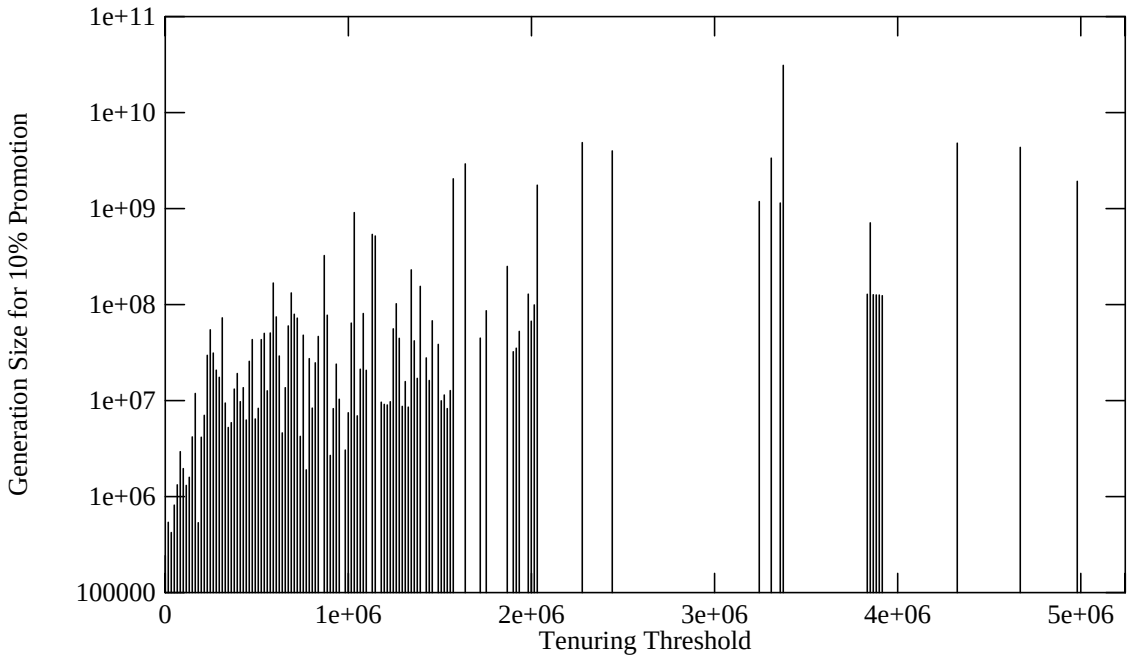

Figure A.34: Skipjack 2, Generation size needed for 90% collection rate

Appendix B

Smalltalk Benchmark Sources

The Stanford benchmarks were designed by John Hennessy as CPU and compiler tests, not garbage collector tests, and so may not represent a good benchmark suite. In the Objectworks system, only the tree sort benchmark and the floating point matrix multiply created garbage that was not collected by the generation scavenger. The floating point matrix multiply is a sub-class of the integer matrix multiply — the relevant portions of both classes are presented here. These Smalltalk versions were written by David Ungar, and are used with permission.

```
KeyTestArray class methodsFor: 'testing'

testKeys: depth by: width cycles: cycles
        "Make an array of size width, and cycles times replace
        elements with linked lists of length depth."
        "KeyTestArray testKeys: 100 by: 200 cycles: 10000"

        | base random |
        base := Array new: width.
        random := Random new.
        (1 to: cycles)
                do:
                        [:i |
                        | slot list |
                        slot := (random next * width) floor + 1.
                        list := LinkedList new.
                        (1 to: depth)
                                do: [:j | list add: Link new].
                        base at: slot put: list]
```

Figure B.1: The "Array" Test

```
run
        self initarr.
        tree := TreeSortNodeBenchmark new val: (sortlist at: 1).
        2 to: QSortelements do: [:i | self insert: (sortlist at: i)
                        Into: tree].
        (self checkTree: tree)
                ifFalse: [self error: ' Error in Tree.']

initarr
        | temp |
        self initrand.
        biggest := -1000000.
        littlest := 1000000.
        sortlist := Array new: QSortelements.
        1 to: QSortelements do:
                [:i |
                temp := self rand.
                sortlist at: i put: temp
                        - (temp // 100000 * 100000) - 50000.
                (sortlist at: i) > biggest ifTrue:
                        [biggest := sortlist at: i].
                (sortlist at: i) < littlest ifTrue:
                        [littlest := sortlist at: i]]

initrand
seed := 74755

initialize
QSortelements := 5000

rand
seed := seed * 1309 + 13849 bitAnd: 65535.
^seed
```

Figure B.2: The "TreeSort" Benchmark

```
insert: n Into: t
        n > t val
                ifTrue: [t left isNil
                        ifTrue: [t left: (self createNode: n)]
                        ifFalse: [self insert: n Into: t left]]
                ifFalse: [n < t val ifTrue:
                            [t right isNil
                            ifTrue: [t right: (self createNode: n)]
                            ifFalse: [self insert: n Into: t right]]]

createNode: n
        ^TreeSortNodeBenchmark new val: n

checkTree: p
        | result |
        result := true.
        p left notNil ifTrue: [p left val <= p val
                    ifTrue: [result := false]
                    ifFalse: [result := (self checkTree: p left)
                                                and: [result]]].
        p right notNil ifTrue: [p right val >= p val
                    ifTrue: [result := false]
                    ifFalse: [result := (self checkTree: p right)
                                                and: [result]]].
        ^result
```

Figure B.3: The "TreeSort" benchmark [cont.]

```
left
        ^left

left: l
        left := l

right
        ^right

right:r
        right := r

val
       ^val

val: v
       val := v
```

Figure B.4: The "TreeSort" benchmark [TreeSortNodeBenchmark]

```
run
        self initialize.
        mr := self mmMatrix.
        ma := self initmatrix.
        mb := self initmatrix.
        self initrand.
        1 to: IRowsize do: [:i | 1 to: IRowsize do: [:j | (mr at: i)
                                at: j put: (self
                                        innerproductOf: ma
                                        and: mb
                                        row: i
                                        column: j)]]

initialize
seed := 0

mmMatrix
        ^self makeMatrix: IRowsize + 1 by: IRowsize + 1

makeMatrix: n by: m
        | r |
        r := Array new: n.
        1 to: r size do: [:i | r at: i put: (Array new: m)].
        ^r
```

Figure B.5: The "IntMM" Benchmark

```
initmatrix
        | m temp |
        m := self mmMatrix.
        1 to: IRowsize do: [:i | 1 to: IRowsize do:
                        [:j |
                        temp := self newValue.
                        (m at: i) at: j put:
                                temp - (temp // 120 * 120) - 60]].
        ^m

innerproductOf: a and: b row: row column: column
        | result |
        result := 0.
        1 to: IRowsize do:
                [:i | result := result +
                        ((a at: row) at: i) * ((b at: i) at: column)].
        ^result

newValue
        ^self rand

initialize
        "IntMMBenchmark initialize"
        IRowsize := 40
```

Figure B.6: The "IntMM" Benchmark [cont.]

```
initmatrix
        | m temp |
        m := self mmMatrix.
        1 to: Rowsize do: [:i | 1 to: Rowsize do:
                        [:j |
                        temp := self newValue.
                        (m at: i) at: j put:
                                temp - (temp / 120 * 120) - 60]].
        ^m

newValue
        ^super newValue asFloat

initialize
        "MMBenchmark initialize"
        Rowsize := 40
```

Figure B.7: The "MM" benchmark

```
run
        self initialize.
        ma := mb := mr := 0.
        ma := self mmMatrix.
        self initMatrixValues: ma.
        mb := self mmMatrix.
        self initMatrixValues: mb.
        mr := self mmMatrix.
        self initrand.
        1 to: IRowsize do: [:i | 1 to: IRowsize do: [:j | (mr at: i)
                                at: j put: (self
                                        innerproductOf: ma
                                        and: mb
                                        row: i
                                        column: j)]]

MMBenchmark methodsFor: 'private'

initMatrixValues: m
        | temp |
        1 to: Rowsize do: [:i | 1 to: Rowsize do:
                        [:j |
                        temp := self newValue.
                        (m at: i) at: j put:
                                temp - (temp / 120 * 120) - 60]].
        ^m
```

Figure B.8: The Key-Friendly "MM" Benchmark