Garbage Collecting Persistent Object Stores*

J. Eliot B. Moss
moss@cs.umass.edu

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Presented at the OOPSLA/ECOOP '90
Workshop on Garbage Collection

October 1990

Abstract

Many techniques have been devised for garbage collecting main memory heaps for programming languages, and there has more recently been progress on distributed garbage collection, for sets of long-lived processes in distributed systems. Little attention has been paid to garbage collecting large persistent stores, however. A persistent store is essentially a shared heap that outlives the execution of any given process; one can think of it as similar to a database, with object retention based on reachability (garbage collection) as opposed to explicit deletion. The Mneme[1] project is an effort to build a large distributed persistent store. Clearly, garbage collection is an issue Mneme must address. Here I consider some of the problems and solution approaches that might be taken. If invited to the workshop, I hope to convey the importance of the problem, how it is and is not related to single process and distributed garbage collection, to develop interest in the problem among other researchers, and to exchange ideas and results. I can make either a long or short presentation as the workshop organizers feel appropriate.

1 The Problem

First, consider the basic abstraction offered by a distributed persistent store. I will base the discussion on Mneme [Moss and Sinofsky, 1988; Moss, 1989a; Moss, 1989b], though most of the ideas are quite general. For our purposes, a *persistent object store* retains a *directed graph* of *objects*. In this graph, each object is a node, and each pointer or reference is a directed edge. Objects have additional structure, i.e., they may contain non-pointer data, the edges are usually ordered, etc., but those details are not especially relevant here. Also, the detailed representation of an edge is not directly relevant, except that we assume that an edge from object x to object y is recorded in object x and usually not anywhere else. This corresponds to the usual approach taken to representing objects in main memory heaps. Further, we assume that edge information is adequate to find the target object, but are not concerned with whether it is an actual address, an object id, etc.[2]

*This project is supported by National Science Foundation Grants CCR-8658074 and DCR-8500332, by Digital Equipment Corporation, and by GTE Laboratories.

[1] Pronounced NEE-mee, it is the Greek word for *memory*.

[2] For further discussion of address representation, see [Moss, 1989a].

The object graph alone is not enough to describe a persistent store; there is also a set of *persistent roots*, such that the only part of the graph that may be retrieved and modified is that part reachable from the roots. The roots map some kind of external name (of unspecified format) to internal pointers. The store must provide the ability to create, delete, and modify roots. If smoothly integrated with a persistent programming language, the persistent roots might be bound to global variables of programs, with new roots created in the process of compilation and linking.[3]

Finally, a *distributed* persistent object store permits distributed access to a distributed object graph via a distributed collection of roots. While one can conceive of single-user persistent stores, users will almost certainly wish to access distributed persistent stores concurrently. However, we will not delve deeply into concurrency control issues, or crash recovery for that matter, here.

The problem to be solved is to reclaim, with reasonable efficiency, the storage consumed by parts of the object graph that become unreachable. There are two aspects of the problem that make it especially different from main memory garbage collection. First, a persistent store is expected to be very large. We must assume it is larger than the real memory of any node in the system, or even than all their real memory combined. We can also expect it to exceed the virtual memory of any single processor, under current technology. The second aspect is distribution, which, because of the desire to allow individual nodes of a distributed system to have a high degree of autonomy, means that logically centralized algorithms are likely to be inappropriate. That is, we assume the system is decentralized and that there may be some degree of suspicion among member nodes. This means that distributed virtual memory, for example, is inappropriate, because it cedes too much control to other nodes. The primary implication is that we must view data as being owned, of having value, and having cost. While I will not attempt to address security rigorously here, I am arguing that (for example) we must not force data owned by one user to be retained merely because another user has a reference to it. On the other hand, if the users are friends or have struck some economic bargain, reachability may be a reasonable retention criterion. To reiterate, the problem is to manage distributed persistent memory resources efficiently, while offering a substantial degree of autonomy to participants.

2 Helpful Mneme Features

There are some aspects of the Mneme design that contribute to solving the problem. In particular, Mneme structures the space of objects into *files*. A file has a set of persistent roots and contains a collection of objects that can refer to each other using short object ids.[4] Objects in one file can also refer to objects in other files via a device we call a *forwarder*. A forwarder is a local stand-in or proxy for an object in another file. Thus, to refer to an object in another file, one refers to a local object marked as a forwarder; the forwarder can contain arbitrary information about how to locate the object at the other end.

Our first step toward garbage collection of the distributed persistent store then is to permit garbage collection of individual files. This serves the goal of autonomy, since files will be the unit of ownership. So that the addresses of objects within a file can be changed without affecting external references to the objects, we require that forwarders name objects in other files by naming the file and providing an *external* name for the target object. This external name is then looked up in the target file's *incoming reference table* (IRT). The IRT is similar in function to the persistent roots. As previously noted, though, it is not absolutely necessary to retain objects reachable only from the IRT. For example, one may replace an IRT binding with a null reference and cause later uses to fail, or move an object and leave a forwarding note. This allows the owner of the file to control object retention. The persistent roots of a file are used to indicate those items that the owner desires retained; all objects reachable from those roots will be kept when a local garbage collection is performed.

[3] This has been implemented, e.g., for the E database programming language [Richardson and Carey, 1987].

[4] The size argument is important overall, but not of direct relevance here; see [Moss, 1989a].

Note that the idea of splitting a large address space into smaller ones to aid garbage collection is not new; we got it from Bishop [Bishop, 1977]. There are some differences between our approach and his, since he was considering a shared virtual memory. What we add is distribution, which implies more variation and flexibility in the creation and manipulation of cross-file links via forwarders and IRTs. For example, a user need not permit certain other users to create new IRT entries or use existing ones, etc. Bishop's approach to collecting global cycles of garbage can be applied in our context, though. This would work as follows. User A determines that a certain set of objects in file A are reachable only from outside, and in fact has a record that user B created and uses the IRT entry from which the set of objects is reached. User A may then request user B to take ownership of the objects by copying them into file B, or else user A will delete the IRT entry and user B will lose access to the data. If B accepts the data, then any global cycle is shrunk; if B declines ownership, then any global cycle will be broken when A deletes the data. Clearly it is helpful if users/applications notify each other as forwarders are created/deleted, so that unneeded IRT entries can be purged with reasonable dispatch. Note, though, that under the assumption of autonomy, we permit IRT entries to be deleted, and previous users may have objects disappear if they have not made proper retention agreements. The point is that we do not need strict accuracy of information in all cases, which may simplify programming robust algorithms.

3 Dealing with Large Files

The preceding techniques do reduce the collection problem from a global one, involving multiple files, to a local one involving a single file at a time. (I do not intend to minimize the problems of managing forwarders, IRTs, and communication about them, but desire to keep the discussion at a high level and to get on to other problems.) The single file garbage collection problem is not necessarily amenable to traditional techniques, though, since a single file may still correspond to a large database, outstripping a single processor's real memory, or possibly even its virtual memory. Clearly we need to be able to take advantage of any locality a file may offer. This locality may be considerable, since objects used together are frequently used together because they refer to each other, and allocation will attempt to cluster together objects frequently used together. In fact, in addition to reclaiming storage, an important goal of garbage collection is to improve clustering so as to boost future performance. The same point has been made with respect to virtual memory: it may be more important to garbage collect to improve locality of reference than it is to reclaim virtual memory space. The tradeoff is not quite so obvious in the case of a persistent store, though, since in the virtual memory space the entire heap is reclaimed when the process terminates, while we must work to reclaim space in a persistent store.

3.1 Choice of General Approach

Here are some questions that arise almost immediately concerning how to approach garbage collecting a large file. First, should we use mark-sweep or copying as the basic technique? Second, should we do a whole file at a time or can we do pieces? Third, should we garbage collect offline (batch) or online (incremental)? The first question can perhaps be answered easily: since part of the point is to improve locality (clustering), we should use copying rather than mark-sweep as the basic technique. Going on to the other questions, online collection is likely to be more difficult and to impose some additional overhead of use during collection, but is necessary for continuously running applications. Doing a whole file at a time may also be simpler, and in some ways may cost less (e.g., if we do only a fraction of a file at a time, it will take many passes to collect the whole file), but will require less additional space (i.e., we may get away with less than double the space of the original file). For these reasons, I will confine further discussion to copying collection that focuses attention on parts of a file at a time. Whether such piecemeal collection is

done in a batch or incremental style is less relevant, since I will be describing my approach at a high level of abstraction.

We could reasonable split a files into multiple areas, keep track of cross area references, collect individual areas, and deal with global cycles of garbage all as Bishop does, and that approach may be appropriate for some files. It is not clear that files necessarily have that kind of locality, though. Suppose, for example, that a considerable part of a large file consists of a collection of records and several independent indexes over those records. The records might be grouped into areas based on their most desirable clustering (whatever that may be), but most of the indexes cannot be so clustered, with the result that there will be a large number of inter-area links. In short, indexes do not necessarily enjoy locality of reference (from their leaves to the records indexed), yet they are very important to achieving good performance.

Thus, I am not convinced that Bishop style areas solve the problem entirely, because not all collections of data have the assumed degree of locality of reference. This leaves us with the question of how to garbage collect piecemeal, which is, I believe, essentially the same question as how to garbage collect files large enough that the garbage collection tables for straightforward algorithms would themselves start to thrash.

3.2 Sketch of an Algorithm

Going back to basics, a copying collector maintains a mapping of old addresses to new addresses (for objects that have been moved), and a set of moved objects that have not yet been processed. (The latter set is often implicit in that it is the set of objects lying between two pointers in a contiguous region of memory.) The mapping can be quite large—near the end of garbage collecting a file the mapping will contain an entry for every surviving object. The set of objects yet to be processed can also be quite large, though in typical cases it may not be.

My notion is to proceed in passes over the old file, as follows. At any given time we have a set of *cluster root objects*, initially those objects referred to by persistent roots. This set may be represented as some kind of queue or list, or we may need a priority queue or B-tree (I am not sure how best to organize it). At any rate, these objects are ones that we will be retaining and that should form the roots of clusters. At any time we also have some partial clusters being formed, and a set of unfulfilled references from those clusters (objects that should be clustered but that we have not moved yet).

To make a pass, we select and remove some members from the cluster root set. Each of these establishes a new cluster to be worked on in this pass. We add the names (addresses) of these objects to the unfulfilled references set, but marked to be roots of new clusters. The unfulfilled references set is kept sorted in disk order. We make a pass over the file, in disk order, picking up each object named by an unfulfilled reference. It is either added to an existing cluster as requested, or starts a new cluster if marked to do so. As each object is processed, it may cause more references to be added to the unfulfilled reference set. Some of these may be "behind" us in the scan, some "ahead", and some within the current scanning window. (If the file was reasonably well clustered before, then many will be in the current window.) If the unfulfilled reference set is itself quite large, then the additions can be buffered, sorted, and added in periodic scans over that data structure, which might be kept as a B-tree.

As each object is moved into a new cluster, its new address is noted in the old object (for continued online use) and entered into the old-to-new object map. As new clusters fill up, they can be flushed and new ones started; this may be deferred to the end of a pass, but may increase the number of passes. Unfulfilled references out of a full cluster become new cluster roots. As we proceed through a pass we also update references to moved objects so that they indicate the new home. Thus, by the end of the pass after the one that moved an object, all references to the old location are gone. The following pass can mark the old copy of the object as deleted. When all the objects of a cluster are marked deleted, we can immediately reclaim the entire cluster. However, garbage objects will not be recognized until we have processed the whole *file*, which is recognized by the unfulfilled object set and cluster root set both becoming empty. At that point we

can reclaim any remaining old clusters.

3.3 Variations and Refinements

It is easy to come up with some interesting variations on this approach. First, the filling of clusters will typically not refer to all that many blocks of the file. Therefore, rather than scanning the whole file on every pass, it might make more sense to do several cluster filling passes, accessing only those blocks necessary, building up a moved object map. When that map reaches a size close to the real memory available, we should then scan the file, updating all objects, so that we can discard the map. The point here is to avoid I/O to forward references from old object homes to new ones. This is also nice if there is concurrent access by users, since (for example) clusters can be scanned and updated before giving them to the user. This will reduce I/Os needed to find old objects moved to new homes.

To handle update by users during operation of the algorithm, we need to detect updates that cause an object in a new cluster to refer to an old object not in the unfulfilled reference set. If the garbage collection algorithm is implemented in a server that manages the fetching and storing of clusters for users, then it will not be too hard to arrange the detection, though we might care to do it only periodically if the unfulfilled reference set is too large to be memory resident.

An additional issue that should be addressed is that not all references from an object are followed equally often, so not all descendants are as important to cluster with the referring object. This means that the unfulfilled reference set mechanism should include some priority. Whether that priority should be absolute, with lower priority items deferred to later passes, or whether it is better to accept slightly worse clustering to reduce the number of passes in the reclustering algorithm is hard to say. Of course, if the order does not matter too much, one can always set the priorities the same, or the algorithm could be designed to retrieve and cluster all items with priority within a certain amount of the current highest priority. I note also that I am talking mostly about mechanism here and ignoring the difficult issues of determining and expressing clustering *policy*.

3.4 An Important Short Cut

It is likely that applications can sometimes guarantee, or at least provide a strong hint, that specific objects have become garbage. For example, in a B-tree if all the items in a node are removed from the data structure, the node itself can be freed. For such cases, we can provide an explicit deletion operator.

Some data structures may be amenable to reference counting, though maintaining the counts is expensive, and count adjustments should definitely be buffered and applied opportunistically to buffers of objects as they are loaded into memory. It will probably be necessary to make occasional passes through a file to adjust reference counts and empty the deferred reference count buffers. Having reference count adjustment proceed as a background operation seems worthy of further investigation. It could also be combined with passes made by the garbage collector if desired.

Sometimes an application cannot make an absolute guarantee, e.g., because someone may have created an IRT entry to an object, but it can give a strong hint that an object is probably free. This can be helpful even in reclaiming cyclic garbage if the application "knows" the pattern of pointers in the data. For example, suppose a doubly linked list is used so that there is only one reference to the whole list, and that reference is to the head item. If the reference is deleted, then all items of the list can be marked as likely garbage. Compilers may be able to discover some of the interesting cases, and it is clear that strong typing and abstract data types help us in applying these techniques.

In any case, when we have a significant set of probable garbage objects (PGOs), we can perform a pass over the file checking to see if there are in fact any references to the PGOs. In so doing, we should of course ignore references from one PGO to another. At the end of a pass, we may find that some PGOs were

not really garbage, and we need to check to see if they refer to other PGOs. In effect, we must perform recursive tracing through the PGOs to remove those objects that need to retained. After this adjustment, the remaining PGOs are true garbage and may be deleted. Deleting some objects as we go frees up space in clusters and may permit new items to be linked into data structures without undermining locality as badly as always placing new objects in a separate area, as would be necessary if clusters were always full. Note that this argues for leaving some slack space in clusters, based on how frequently they are modified and how rapidly we can reclaim space.

4 Final Notes

Finally, I would like to note that while there is undoubtedly some relevant work on clustering and reclustering in databases, most such work in database and operating systems is not concerning with garbage collection, but with structures that are essentially trees, i.e., there may be back pointers and other redundancies for various reasons, but unreachable objects, as opposed to space explicitly marked *free*, are usually considered to be errors. Hence the problem is different, though to what extent is difficult to characterize.

In summary, I have described the problem of garbage collecting large distributed persistent object stores and reduced it to garbage collecting large local stores. I then sketched an approach to collecting large local stores that uses repeated sweeps across the disk to reduce disk arm movement, and that explicitly takes into account the amount of real memory available, similar to buffer management in a database system. The goal is to support database size heaps with object retention based on reachability.

References

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

[Moss and Sinofsky, 1988] J. Eliot B. Moss and Steven Sinofsky. Managing persistent data with Mneme: Designing a reliable, shared object interface. In *Advances in Object-Oriented Database Systems* (Sept. 1988), vol. 334 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 298–316.

[Moss, 1989a] J. Eliot B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. In *Second International Workshop on Database Programming Languages* (Gleneden Beach, OR, June 1989), pp. 269–285. Also available as University of Massachusetts, Department of Computer and Information Science Technical Report 89-68.

[Moss, 1989b] J. Eliot B. Moss. The Mneme persistent object store. COINS Technical Report 89-107, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, Oct. 1989. Submitted for publication as "Design of the Mneme Persistent Object Store".

[Richardson and Carey, 1987] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementations in EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, CA, May 1987), vol. 16, no. 3 of *ACM SIGMOD Record*, ACM, pp. 208–219.