Partitioned Garbage Collection of a Large Object Store

Umesh Maheshwari Barbara Liskov

Technical Report MIT/LCS/TR–699
MIT Laboratory for Computer Science, Cambridge, MA 02139
October 1996

Abstract

This paper describes a new garbage collection scheme for large persistent object stores that makes efficient use of the disk and main memory. The heap is divided into partitions that are collected independently using information about inter-partition references. We present efficient techniques to maintain this information stably using auxiliary data structures in memory and the log. The result is a scheme that truly preserves the localized and scalable nature of partitioned collection.

Remembering inter-partition references does not collect garbage cycles that span partitions. We describe a new global marking scheme that collects such garbage. We believe that it is the first scheme that piggybacks global marking on partitioned collection, does not delay the collection of acyclic garbage, and is guaranteed to terminate correctly in the presence of concurrent mutations. Further, it preserves the disk-efficient nature of our collector.

We have implemented the part of garbage collection responsible for maintaining information about inter-partition references. We present a benchmark to evaluate this work and give performance results to show the advantages of our scheme.

Keywords: garbage collection, partitions, cyclic garbage, object database

1 Introduction

We present a new technique to collect garbage in large persistent object stores. Such storage, also known as a stable heap, is found in many object databases, persistent programming language environments, and distributed shared memory systems. In these systems, the heap resides on the disk because it is much larger than the main memory and must be recoverable after a crash. Applications access the objects through a memory cache and log updates for crash recovery.

Schemes that trace the entire heap (e.g., [Bak78, KW93, ONG93]) do not scale to very large heaps because the non-local nature of tracing would cause random disk accesses. Therefore, some systems partition the heap into independently collectible areas [Bis77, HM92, YNY94, AGF95, MMH96, CKWZ96]. This is also the approach taken in many distributed system [LQP92, LL92, ML94, FS96]. Generational collectors are a variant of partitioned collection that use the ages of objects to optimize the collection of younger, smaller partitions [LH83]; however, the age-based heuristics are not applicable to persistent stores [Bak93].

A major problem with partitioned collection is the efficient maintenance of information about inter-partition references, which is needed to trace partitions independently. For a large heap with many partitions, and also for fast crash recovery, the information must reside on disk, and care is needed in reading and updating it without degrading performance. Increasing the partition size helps reduce inter-partition references, but tracing very large partitions slows both the garbage collector and the applications due to increased contention for the cache and disk [AGF95].

We present a new partitioned scheme that uses a log and in-memory data structures to provide the following benefits:

1. Disk accesses for reading and updating information about inter-partition references are deferred and batched.
2. Reading objects from the disk and evicting them from the cache do not require processing garbage collection information or reading it from disk.
3. The in-memory data structures are compact yet available in an efficient form.
4. The scheme is fault-tolerant; collection information is recovered quickly after a crash.

The overall effect is that inter-partition references are handled efficiently, which makes it possible to use partitions that fit in a small fraction (say, a tenth) of the primary memory.

Our work on using the log to maintain inlists and outlists builds upon previous work by Ng [Ng96]. One other scheme, PMOS [MMH96], makes use of a log to defer and batch processing of information about inter-partition references. However, PMOS processes garbage collection information when objects are fetched and evicted, which would slow down applications.

Partitioned collection does not collect garbage cycles

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136.

that span partitions. We describe a new global marking scheme that collects such garbage. Our scheme piggybacks marking on partitioned collection such that it adds little overhead to the base scheme. It also preserves the localized and disk-efficient nature of our collector, and does not delay the collection of acyclic garbage. It can take a long time for a marking phase to terminate, but that is acceptable assuming cyclic garbage spanning partitions is generated slowly. We prove that the scheme is correct and that a marking phase is sure to terminate in the presence of application modifications. Previous proposals for using global marking on top of partitioned collection either delay the collection of acyclic garbage [Hug85], need to run separate traces for global marking and partitioned collection [JJ92], or are not guaranteed to terminate in the presence of modifications [LQP92].

We have implemented the part of garbage collection responsible for maintaining inter-partition references in the context of Thor [LAC+96]. We present a benchmark to evaluate the maintenance of inter-partition references, and we give performance results to show the advantages of our scheme.

The remainder of the paper is organized as follows. Section 2 describes the system model. Section 3 describes partitioned collection, and Section 4 describes the global marking scheme for collecting cyclic garbage. Section 5 describes our implementation, benchmark, and performance results. Related work is discussed in Section 6. We close in Section 7 with a summary of our contributions.

2 The Model

We assume a system with the architecture shown in Figure 1. The stable heap resides on disk, while applications access objects in the heap through a main-memory cache. Modifications to objects are recorded in a write-ahead log that is forced to stable storage as needed; the log allows the heap to be recovered in a consistent state after a crash. We assume that the head of the log is cached in primary memory even if it has been forced to disk; it is truncated when it grows too big. Similarly, when the stable log gets too big, it is truncated after ensuring that the modifications have been installed into the stable heap.

Objects are clustered in *segments* on disk. Like a page, a segment is stored contiguously on disk and is the unit of disk access. A segment also provides opaque references for its objects so that objects can be moved within their segment without having to update references to them stored in other objects (as in [AGF95]). References need not be completely opaque; for example, in Thor, each reference contains the segment number of the referenced object so that objects can be located efficiently without a global object table. Our scheme could also be used in systems that do not have opaque names within segments (e.g., systems that store virtual memory pointers in objects), but in that case it would not be possible to compact objects.

Objects in the heap may contain references to other objects. Applications navigate by starting at some *persistent root* object and may read or modify the objects they reach. They may also store references to objects in local variables. There could be a single application thread accessing the cache directly, as in persistent programming language environments. Alternatively, there could be multiple application threads, as in a client-server system, where clients access the server cache through a higher level interface and may have caches of their own.

The job of the collector is to reclaim storage allocated to objects that are useless because they are not reachable from the persistent root or any application variables.

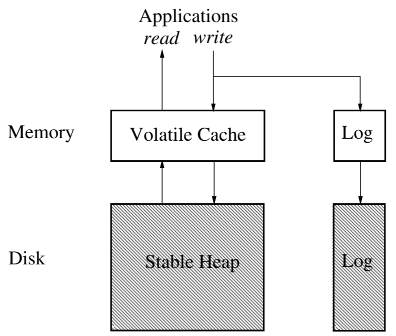


Figure 1: A generic architecture with large heap.

3 Partitioned Collection

This section describes our scheme. The first few subsections largely ignore fault tolerance; fault tolerance is discussed in Section 3.5.

The heap is divided into partitions, each of which can be collected independently. A partition is chosen to be an efficient unit of tracing. There is a tradeoff here: Small partitions mean more inter-partition references and also more inter-partition cyclic garbage. Big partitions mean more cache space used by the collector and possibly disk accesses during tracing. Our scheme provides mechanisms to handle inter-partition references so that partitions that fit in a small fraction of primary memory (say, 10%) may be used efficiently.

Partitions contain several segments, possibly non-adjacent. Decoupling partitions from segments has important advantages. First, a partition can be much bigger than a segment. For example, a partition could be several megabytes while segments could be tens of kilobytes. Segment size is chosen to allow efficient fetching and caching; to service an application cache miss, only the required segment is read in. Second, it is possible to configure a partition by selecting a group of segments so as to minimize inter-partition references—without reclustering objects on disk. Furthermore, partitions can vary in size, while segments provide a fixed-size unit of disk access. For example, a partition can represent the set of objects used by some application, and the size of the partition can be chosen to match the set.

A reference contained in an object to an object in the same partition is said to be intra-partition or *internal*; others are said be inter-partition or *external*. We assume that, given a reference to an object, its partition can be computed efficiently. We do this by keeping a map from segments to partitions and vice versa.

3.1 Inlists and Outlists

To collect a partition independently of the rest, the collector remembers the objects in the partition that are referenced from other partitions, and uses them as roots. We call this information the *inlist* for the partition. An inlist contains a list of references with associated reference counts. The reference count is the number of other partitions that contain one or more copies of the reference. When a reference counts drops to zero, the entry is removed from the inlist.

To efficiently update inlists as inter-partition references are created and deleted, we also maintain an *outlist* for each partition. An outlist is a list of external references contained in the partition. It provides an efficient means of detecting when a new external reference is created or when one disappears.

The following invariants guarantee that only unreachable objects are collected (safety):

---

1. All external references contained in a partition are included in the outlist.
2. The count of a reference in an inlist is equal to the number of outlists containing the reference.

Inlists and outlists are kept on stable storage because otherwise it would take a long time to recompute them after a crash. They can be maintained as regular heap objects, possibly outside the partition's segments.

When an inter-partition reference is created from partition p to q due to a modification, the outlist for p and the inlist for q need to be updated to preserve the invariants. This can be done lazily by scanning modified objects in the log; the constraints are that the log must be fully scanned before collecting a partition, and that modified objects must be scanned before they are truncated from the cached head of the log. Thus, invariant 1 holds when the log is fully processed.

Updating inlists and outlists has the problem that either these lists must be kept in the cache or they have to be fetched from the disk and later flushed back. This is undesirable because both the cache and the disk are precious resources for application performance. Note that in a large object store, the aggregate size of inlists and outlists may be large. (For example, in a 10 giga-word database, if one out of thousand words is an inter-partition reference, there would be 10 million inter-partition references. Assuming that an outlist entry uses a word and an inlist entry uses two words, the aggregate size would be 30 mega-words.) Our scheme saves cache space and defers reading or writing the disk to access these lists by using small, *potential* inlists and outlists in memory.

3.2 Potential Inlists and Outlists

When an object in partition p is scanned, we record any external references in the *potential outlist* for p, and we also update the *potential inlists* of the target partitions. Each entry in a potential inlist contains a reference count that counts the number of potential outlists that contain the reference. To distinguish inlists and outlists from their potential counterparts, we refer to the former as the *basic* lists. The following revised invariants still guarantee safety:

---

1. All external references contained in a partition are included in the basic or potential outlist (or both).
2. (a) The count of a reference in a basic inlist is equal to the number of basic outlists containing the reference.
   (b) The count of a reference in a potential inlist is equal to the number of potential outlists containing the reference.

The potential lists grow slowly because there are expected to be relatively few inter-partition references in modified objects. Further, references already present in the old values need not be added to the potential lists. For example, in transactional systems old copies of modified objects are retained in case the transaction aborts; if the old copy is cached, this information can be used to avoid unnecessary additions to potential lists. However, there may still be overlap between the potential and basic outlists. When potential lists grow too big, we merge them into the basic lists.

3.3 Merging Potential and Basic Lists

We move entries from the potential lists to basic lists in batches. This is a two step process: we merge the outlists first and merge the inlists later.

When the total size of the potential outlists grows beyond a certain limit, we select a few partitions with the largest potential outlists, and read in their basic outlists. References in a potential outlist that are not already present in the basic outlist are added to the basic outlist. The potential outlist is then discarded and corresponding potential inlist counts are decremented to maintain Invariant 2b.

Updating a single basic outlist can require increments to entries in several different basic inlists. Reading in these inlists at this point would result in several disk accesses. Therefore, we record the increments to the basic inlists in yet another data structure called the *delta* inlist.

A delta inlist contains a set of references with associated counts. Unlike potential inlists, which contain potential increments, delta inlists contain *definite* increments to the basic inlists. Invariant 2a is then revised to the following:

(2a) The count of a reference in a basic inlist plus that in the delta inlist is equal to the number of basic outlists containing the reference.

When the total size of delta inlists grows beyond a certain limit, we merge some of them into the basic inlists. We select a few partitions with a largest delta inlists, read in their basic inlists, add the counts in the delta inlists to the basic inlists, and discard the delta inlists. The generation of the various lists is shown in Figure 2.

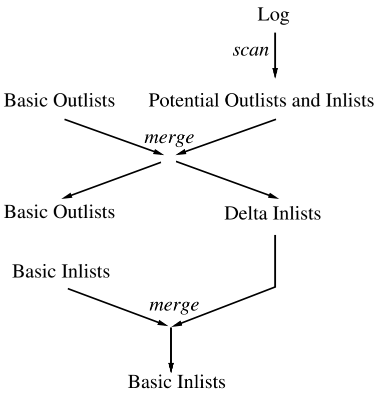


Figure 2: Data structures to batch disk accesses

3.4 Collecting a Partition

Any policy may be used to select partitions for collection. (Cook *et al.* showed that it is desirable to be flexible in selecting partitions [CWZ94].) To trace a selected partition, we load all of its segments into the cache and process the log completely to generate the partition's full potential inlist. Since a partition is a fraction (say, 10%) of the cache, it does not disturb the cache much when loaded.

The Roots

We include the following in the root set of a partition:

1. The persistent root of the heap.
2. Roots from applications, such as variables.
3. The basic inlist for the partition, which is read in from the disk. If there is a delta inlist, it is merged with the basic inlist and discarded.
4. The potential inlist for the partition, if any.

We call the first two *global roots*; the others are a consequence of partitioned collection. Obtaining application roots depends on the specific system model; special care is needed in systems where applications have caches of their own, but we ignore this issue in this paper and assume that application roots are readily available.

Compaction

If the collector compacted storage by moving objects such that there names changed, it would be necessary to fix up references to moved objects—including those that are stored in other partitions and in application variables. Most generational schemes solve this problem by remembering the exact objects or locations in other partitions that refer to a given object [Ung84], but this is too much information to maintain in a large heap. Therefore, we compact objects within each segment and thus preserve object names, as in [AGF95]. Note that objects that are not referenced from other partitions or applications (as given by the root set) could indeed be moved to other segments within the partition.

Our scheme could also be used in systems that do not have opaque names within segments (e.g., systems that store virtual memory pointers in objects), but in that case it would not be possible to do compaction.

Tracing Scheme

Our approach can be used in combination with various tracing schemes. For example, we could use a replicating collector like that described in [NOPH92], in which applications access the old copies of segments while the collector is generating the new ones with the aid of the modification log. Such a scheme requires little synchronization with applications, but needs space for two partitions in primary memory.

A mark-and-sweep scheme that rescans modified objects in the log can be used as well. Such an approach requires less space than copying. No synchronization is needed during the mark phase. The sweep phase can compact one segment at a time, either by locking the segment from applications and sliding objects or by making a new copy of the segment to replace the old one. No work is needed for segments with no garbage objects; this may be a significant advantage over copying collection in persistent stores where little garbage is created.

Updating the Lists

As the partition is traced, the collector generates a new basic outlist for it. After collection, it compares the old and the new outlists, increments the delta inlist entry for every new reference, and decrements the delta inlist entry for every missing reference. Thus there may be negative counts in delta inlists. These steps ensure that Invariant 2a is maintained.

The new basic outlist then replaces the old outlist. The potential outlist, if present, is discarded and the corresponding counts in the potential inlists are decremented.

3.5 Fault Tolerance

Crash recovery must preserve the invariants mentioned above. It would be lots of work to recompute inlists and outlists after a crash. Therefore, we store basic inlists and outlists as persistent objects. Their modifications are logged (e.g., at the end of collection and after merging lists), and are installed on disk later as with normal objects.

Potential outlist information for modifications stored in the log need not be logged separately: this information is recovered by reprocessing the log after a crash. However, when the stable log is truncated, the potential outlists of the affected partitions must be made stable to preserve invariant 1. This can be done by logging the part of each potential outlist that was not logged before. An efficient way to do this is to divide each potential outlist into stable and volatile parts. When the log is truncated, the volatile parts of potential outlists are logged and marked stable. Potential inlists are never logged; they can be recomputed from potential outlists on recovery, thus preserving Invariant 2b.

When a potential outlist is merged with a basic outlist, any log records for the potential outlist are deleted (by writing a deletion record). Any updates to the basic outlist and the delta inlist are logged atomically to preserve Invariant 2a. When a delta inlist is merged with a basic inlist, the log records for the delta inlist are deleted and the updates to the basic inlist are logged, which also preserves Invariant 2a.

The following summarizes the disk accesses involved in maintaining inlists and outlists stably:

1. Truncating the log:
   (a) log parts of potential outlists not logged before.
2. Merging potential outlist:
   (a) fetch basic outlist.
   (b) log updates to basic outlist and delta inlists, atomically.
   (c) remove potential outlist records from log.
3. Merging delta inlist:
   (a) fetch basic inlist.
   (b) log updates to basic inlist and remove delta inlist records, atomically.

Note that the log updates listed above need not be forced to the disk until the log is truncated. Thus, a crash might lose unforced updates to delta inlists and basic lists, but that is acceptable because the potential outlist information that generated those updates will be regenerated from the log, which still preserves Invariant 1.

The segments of a garbage-collected partition can be independently flushed to the disk. Thus all the collection state that was available before a failure can be quickly recovered afterwards and the log is used to reduce disk I/O's associated with storing the needed information.

3.6 Safety and Liveness

Invariants 1, 2a, and 2b guarantee that objects reachable from other partitions will not be collected: If there is a reference r from partition p to q, Invariant 1 implies that r must be in the basic or potential outlist for p. If r is in the basic outlist for p, Invariant 2a implies that the sum of the counts for r in the delta and basic inlists for q is at least one. If r is in the potential outlist for p, Invariant 2b implies that the count for r in the potential inlist for q is at least one. In either case, r will be included in the root set for q.

Further, our scheme is guaranteed to collect all garbage. An unnecessary entry in a basic or potential outlist will be removed when its partition is next collected and the corresponding inlist count will be decremented. This guarantees that objects not reachable from the roots are collected. From this it can be shown inductively that if the partitions are collected periodically, all garbage except for inter-partition cyclic garbage will be collected.

4 Collecting Cyclic Garbage

We collect inter-partition cyclic garbage using a global but incremental marking scheme. We mark all objects reachable from the global roots and then collect unmarked objects. The marks are propagated incrementally through partition traces. At the beginning of a marking *phase*, only the global roots are marked. Each partition trace propagates marks from the root set of the partition to the outlist. The marking phase terminates when marks are known to have propagated fully through all partitions.

Similar schemes have been used in some distributed systems [Ali85, Hug85], although these systems rely on global marking to collect both acyclic and cyclic inter-partition garbage. Other schemes either propagate global marks separately from regular partition traces [JJ92], or are not guaranteed to terminate correctly in the presence of concurrent mutations [LQP92]. We show that our scheme is correct and that it terminates in the presence of mutations. We found that a more sophisticated scheme is needed to meet these requirements than apparent at first.

Our scheme collects inter-partition cyclic garbage with little time and space overhead and does not delay the collection of other garbage. This is important since inter-partition cyclic garbage is generated relatively slowly, so the overheads must not be disproportionately high. As in partitioned collection, we employ data structures that use the disk efficiently.

4.1 Data Structures

Each partition has a basic *markmap*, which contains a mark bit per object. The markmap is implemented as a set of bitmaps, one per segment in the partition; each bitmap contains a bit per potential object name in the segment. Markmaps may be too large to keep in the cache, but they are a small overhead on disk. For example, a 1 MByte partition might contain up to 64K objects (allowing an average object size of 16 bytes). Thus we have 8 Kbyte markmap, which represents only 0.8% overhead on disk. Certain other schemes require mark bits for only inter-partition references [Hug85, LQP92]; we discuss the need for a mark bit per object in Section 4.5.

The mark of a basic inlist entry is defined to be the mark of the object it references, as stored in the basic markmap. A partition may also have an in-memory *delta markmap*, which stores updates to the basic markmap just as a delta inlist stores updates to the basic inlist. The delta markmap may be dynamically implemented as a list or a bitmap depending on the number of references in it. A delta inlist entry is said to be marked if the reference is contained in the delta markmap. Furthermore, each entry in basic outlists has an explicit mark bit.

Each partition also has a mark bit to denote whether the marks of its objects have been propagated to the outlist. As described later, tracing a partition causes it to become marked, but may cause other partitions to become unmarked. Mark bits for partitions are persistent, but are also cached in memory.

Note that we use the term "marked" for objects reached by global marking and "traced" for objects reached while collecting a partition. If partitions are traced using mark-and-sweep, separate bits would be needed for global marking and partition traces.

4.2 Invariants

For a marking phase to terminate, all partitions must be marked. We give the precise conditions for termination and sketch a proof of safety and liveness in Section 4.6. Here we give the invariants:

---

3. In a marked partition, for any external reference that is locally reachable from a marked object, there exists a marked basic or potential outlist entry.
4. (a) If a basic outlist entry is marked, the associated basic or delta inlist entry is also marked.

   (b) If a potential outlist entry is marked, the associated potential inlist entry is also marked.

These invariants ensure that, when marking terminates, all objects reachable from the global roots are marked. We use the following rules to guarantee termination:

---

5. A marked object is never unmarked during a phase.
6. Objects created during the current phase are marked.
7. Every time we unmark a partition, we mark at least one of its unmarked objects.

4.3 Starting a Phase

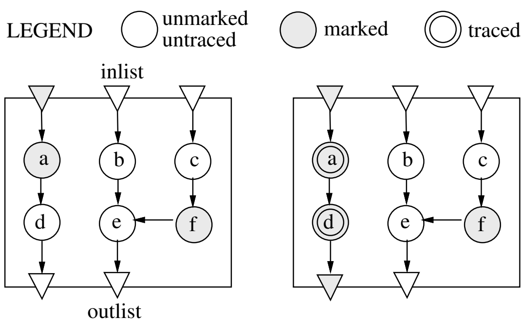
At the beginning of a phase, only the global roots are marked. All partitions referenced by the global roots are unmarked and the rest are marked. All outlist entries are unmarked. This satisfies Invariant 3.

We do not accomplish these actions by reading and writing all markmaps and lists at once. Instead, we perform them incrementally. We keep a persistent *global phase counter* that is incremented at the end of each phase. In addition, we store a phase counter with each inlist, which tells the phase during which the partition was last traced. When an inlist is fetched to be traced or merged, if its phase counter is one less than the global phase counter, this must be the first access to the inlist during the current phase, so we initialize the partition's outlist and markmap. (In the unlikely event that the partition's phase counter is even smaller, it was not visited during the previous phase; therefore the whole partition is garbage and can be discarded.)

Inlist entries that were unmarked in the previous phase are known to be garbage. However, we cannot simply remove these inlist entries because that would break Invariant 2a and cause incorrect execution. Instead, we remove all references in the objects pointed by these entries, as in [RJ96]. This breaks garbage cycles and ultimately causes associated inlist entries to be removed.

4.4 Tracing a Partition

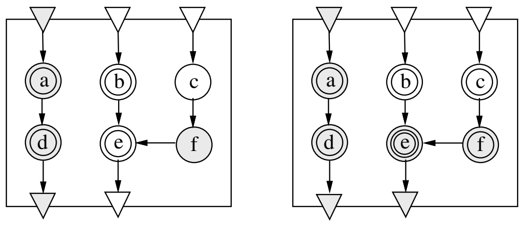
Global marking is piggybacked on regular tracing of partitions. We set the mark bit of a partition after tracing it. The global roots and marked inlist entries are traced first; objects reached and outlist entries created during this trace are marked. We call this the *marked trace*. Figure 3 shows the effect of this trace.



(i) Before tracing. (ii) After tracing from a.

Figure 3: Marked trace of a partition.

The unmarked inlist entries are traced next, which constitutes the *unmarked trace* and is illustrated in Figure 4. It is possible for this trace to reach marked objects that were not traced during the marked trace, such as f in the figure. These are objects that were marked in a previous collection, but modifications by applications have made them locally unreachable from the current set of marked inlist entries; we call them *marked orphans*. Such objects pose a problem that has not been considered in previous schemes: Invariant 3 requires that objects and outlist entries reachable from marked objects should be marked. However, marked orphans may point to objects that have already been traced but were not marked, such as e in the figure. We could preserve Invariant 3 by *un*marking the marked orphans, but this would violate Rule 5 needed for termination. Therefore, we need to trace all unmarked objects reachable from marked orphans, even if they have been traced before, and mark them. This rescanning is the small part of global marking not piggybacked on regular partition traces.



(i) After tracing from b. (ii) After tracing from c.

Figure 4: Unmarked trace.

We do not expect a large number of objects to be retraced because they comprise only unmarked objects reachable from marked orphans. In the worst case, since retracing marks unmarked objects, each object can be retraced at most once over an entire phase of global marking. In practice, retracing would be even less, since marked orphans are expected to be relatively uncommon.

At the end of collection, we compare the new and old basic outlists as before to generate entries in the delta inlists of target partitions. In addition, if an outlist entry is now marked and was unmarked or nonexistent earlier, we mark the reference in the delta markmap to preserve Invariant 4a.

Note that global marking does not cause any object to survive that would otherwise be removed by tracing partitions:

- Marking does not add any reference to any inlist.
- Untraced objects are removed even if they are marked, for these objects must be garbage.

4.5 Processing the Log and Merging Lists

We preserve Invariant 3 lazily by processing the references contained in modified and new objects in the log.

We summarize inter-partition references in potential outlists as before. We preserve Invariants 3 and 4b by assuming that all references in potential outlists and inlists are implicitly marked.

However, now we need to consider intra-partition (internal) references as well, since their creation may break Invariant 3. Specifically, the invariant would break if a reference is created from a marked object to an unmarked object. We ignore such references if the containing partition is unmarked. Otherwise, we insert the reference in the delta markmap. When a delta markmap grows big, we merge it with the basic markmap. If this causes any unmarked object to be marked, we unmark the partition to reinstate Invariant 3. Thus, Invariant 3 is known to hold only when the delta markmap has been merged. Further, note that we unmark a partition only if we discover an unmarked object that should be marked (Rule 7 needed for termination). This is why we need a mark bit per object, while some previous schemes used mark bits in inlists and outlists only [LQP92].

All new objects must be marked; we accomplish this by storing them in the delta markmap. Note that this does not imply that these objects cannot be collected during this phase. As mentioned, untraced objects are collected even if they are not marked. However, if new objects form an inter-partition garbage cycle, they will be collected in the next phase.

When a potential outlist is merged with the basic outlist, if an entry in the potential outlist was unmarked or nonexistent in the basic outlist, we mark the entry in the resultant basic outlist. This preserves Invariant 3. Further, we mark the reference in the delta markmap to preserve Invariant 4a.

4.6 Termination

Marking is *guaranteed* to terminate when the following conditions hold:

1. All partitions are marked.
2. All references in the log have been processed.
3. All potential lists, delta lists, and delta markmaps have been merged with their basic counterparts.
4. All application roots point to marked objects.

We use the following policy to test for termination. We wait until all partitions are marked. Then we process any unprocessed references in the log by generating entries in potential lists and delta markmaps, and merge them as usual. Then we check application roots; if any points to an unmarked object, we mark it and unmark its partition.

If all partitions are still marked, marking is complete. Otherwise, we wait until all partitions are marked again and repeat the procedure. Since we process the complete log and merge all lists to test termination, the work done in later tests would be smaller.

Safety

Processing all references in the log ensures that Invariant 3 holds. Therefore, given that all partitions are marked and all delta markmaps merged, if a partition p has an inter-partition reference to object x that is reachable from a global root or a marked object in p, then x must have a marked entry in the basic or potential outlist of p. From Invariant 4 there must be a marked entry for x in the basic, delta, or potential inlist for its partition. Since the mark of a basic inlist entry is just the mark of the referenced object, and since all delta and potential lists have been merged, x must be marked. Since all global roots are known to be marked, all objects reachable from them must be marked at termination.

Liveness

Marking is sure to terminate because a partition can be unmarked only a finite number of times during a phase. This is true because every time we unmark a partition, we mark at least one of its unmarked objects (Rule 7). Further, such an object must have been created before the current phase because objects created during this phase are always marked. Since the partition has a finite number of objects created before this phase, and marked objects are never unmarked, the partition can be unmarked only a finite number of times during the phase—even if applications are continually creating and modifying objects.

Termination does require that any unmarked partition be traced eventually, but the relative frequency of tracing various partitions can still be governed by an independent policy.

Although global marking is guaranteed to terminate, it is difficult to estimate a practical bound on the number of traces it would take in the presence of concurrent mutations. We can estimate the length of a marking phase by assuming that applications are quiescent, that is, not modifying objects. In this case, a partition is unmarked only as a result of tracing another partition. Suppose that there are n partitions and the maximum inter-partition *distance* of any object from the global roots is l. The distance of an object is the smallest number of interpartition references in any path from a global root to the object [ML95]. We make another simplifying assumption that partitions are uniformly selected for tracing, for example, in round-robin order. Then, marks will propagate fully in l rounds, or $n \times l$ partition traces. Note that this is the worst case bound given the round-robin order. With a thousand partitions and a maximum distance of ten, a marking phase would take at most ten thousand partition traces.

4.7 Crash Recovery

Since global marking takes relatively long to finish, it is desirable to resume it after a crash instead of restarting it. We maintain basic markmaps stably: they are updated after tracing a partition and also after merging delta markmaps. Updates to delta markmaps due to references in the log must be made stable before that part of the log is truncated; updates since the last log truncation are conservatively recovered by rescanning the log after a crash. The mark bits in basic outlists are persistent like other information in them.

The mark bits of partitions are stable too. After a partition is traced, its mark bit is stably and atomically updated along with other information. When the delta markmap is merged with the basic markmap, the mark bit of the partition is stably updated.

The global phase counter is stably updated when a phase terminates. The phase counters of partitions are stably updated when they are first traced in a new phase.

5 Implementation

We are implementing partitioned garbage collection in the context of Thor, an object-oriented database. We have implemented the part that is the focus of this paper: maintaining information about inter-partition references. The performance of this part is important because it is a steady-state activity that must be carried out as objects are modified—regardless of whether a partition is being traced. Tracing of individual partitions is largely orthogonal to our scheme; as mentioned in Section 3.4, any of several existing techniques for concurrent collection could be used. Our implementation does not yet include support for collection of inter-partition cyclic garbage.

5.1 The Context

Thor is a client-server object-oriented database [LAC+96]. Servers provide persistent storage for objects, while applications running on client machines read and modify objects. Object accesses are grouped into transactions to tolerate concurrency and failures.

Servers store objects on disk in fixed-sized segments, currently 32 Kbytes, which are the units of disk access. Servers also maintain a cache of recently fetched segments in memory. Applications fetch objects from servers into the client cache and access them locally. At commit, copies of modified objects are sent back to the servers. The modified objects are stored in an in-memory log that is intended to be stable through replication [Ghe95]. However, we simulate delays for log forces as if it were stored on a logging disk, separate from the database disk.

Persistence of objects at the servers is governed by reachability from persistent roots. The task of garbage collection in Thor is distributed across servers and clients [ML94, ML95], but this paper pertains to garbage collection within a single server.

5.2 Partitioned Collection

We perform garbage collection related work in a *collector* thread, which is run at low priority to avoid delaying application requests. The operating system ensures that the collector is not starved forever. The collector scans modified and new objects in the log for inter-partition references. It computes the partition for a reference by using a map from segments to partitions.

The implementation maintains only potential outlists since the potential inlist of a partition is needed only when tracing that partition, at which point it can be computed from the potential outlists. (There is a tradeoff here: the time cost of generating a potential inlist before tracing versus the space and time costs of maintaining *all* potential inlists. Potential inlists are still useful conceptually to reason about correctness.)

A potential outlist is implemented as a table hashed on references. When the aggregate size of potential outlists exceeds a certain threshold, the biggest lists are merged back until the size drops below a low watermark. Delta inlists are implemented as hashed tables mapping references to counts. The hash tables use about 20 bytes per reference.

Basic lists are stored as regular database objects. A basic list has an indirect block containing references to data blocks so that the list may grow or shrink easily and without external fragmentation. This structure has another advantage: when updating a basic list, we need to modify and log only the affected blocks. In our implementation, basic list blocks were 1 Kbyte each. An outlist data block uses 4 bytes per reference and an inlist block uses 8 bytes per reference. References are not stored in any particular order. Modified basic list blocks are committed using transactions, except that we bypass the concurrency control mechanism since only the collector accesses them.

The collector is given a fixed amount of memory to store potential, delta, and basic lists. We fixed this space at 2 Mbyte, while the fraction allocated to the various lists varied. Potential and delta lists are fully memory-resident structures, while the space for basic lists governs how many basic list objects can be cached. Table 1 summarizes the parameters employed.

Parameter	Value(s)
Segment size	32 Kbyte
Collector Cache	2 Mbyte
Potential lists size	10–80% of cache
Delta lists size	10–80% of cache
Basic lists cache	10–80% of cache
Basic list block	1 Kbyte

Table 1: GC parameters at the server

We have not yet implemented crash recovery and the actions needed on truncating the stable log. While these require care in implementing them correctly, we do not expect them to introduce a significant performance cost for the reasons given in Section 3.5.

5.3 Workload

Amsaleg et al. pointed out the lack of a standard benchmark for garbage collectors [AFFS95]. They identified certain metrics to evaluate collectors; these include isolated costs and benefits such as bookkeeping overheads and the rate of collection, as well as the effect on the overall application performance. They also identified certain parameters to control the measurement of these metrics, e.g., clustering and partition selection policies. Unfortunately, a standard benchmark for garbage collection remains elusive today.

We have designed a synthetic benchmark specifically for evaluating the overhead of tracking inter-partition references. The benchmark allows us to vary the locality of references and thus control the frequency of interpartition references systematically. The benchmark database consists of a homogenous collection of small objects, each of which has a single reference and some data fields. This is similar to the benchmark suggested by Amsaleg et al., except that the objects are not linked into a list as in their case. Instead, each object refers to another object selected randomly using some chosen probability distribution.

We chose a simple probability distribution: Objects are numbered sequentially, and object i refers to a random object in the range between $(i-r)$ and $(i+r)$ with uniform probability. (Object numbers wrap around when they overflow or underflow the bounds of the database.) The range of references, r, is a measure of locality. If each partition contains p objects, the expected fraction of references that cross partitions is analytically found to be:

$$\begin{array}{ll} r/2p, & \text{for } r \leq p \\ 1-p/2r, & \text{for } r \geq p \end{array}$$

which is an increasing function of the reference range over partition size (r/p). Figure 5 shows the percentage of references scanned by the collector that were found to cross partitions given a range over partition ratio. The results match the values expected analytically.

In practice, the range is highly application dependent. At one extreme, a linked list will have only one external reference regardless of the partition size. At the other extreme, a partition storing any n internal nodes of a k-ary tree will have $(k-1)n$ external references, regardless of how the internal nodes are packed. In most databases, we expect the range to be less than a few segments, so the range over partition ratio is likely to be small.

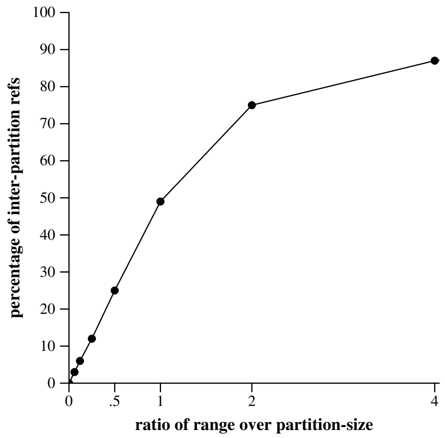


Figure 5: Inter-partition references vs. locality

We set the database size to 4K segments. Each segment contains 1K objects. Each object has a reference and 5 integers, which uses about 30 bytes in Thor. The range of references varied from 0 to 8 segments, and the size of partitions varied from 8 to 32 segments (up to 1 Mbyte). Table 2 summarizes the parameters employed for the workload.

Parameter	Value(s)
Database size	4K segments
Objects per segment	1K
Reference fields per object	1
Data fields per object	5
Reference range	0–8 segments
Partition size	8–32 segments

Table 2: Workload parameters

We evaluate the collector when the database is being created, which results in the creation of inter-partition references. Segments in the database are created in a random order to simulate the effect of concurrent applications.

In practice, the collector would fill some idle time on the server so that only some of its execution time is visible to the applications. Such idle time may result from think time in the applications, applications and server running on different machines, I/O in the server, etc. However, the overall effect is highly sensitive to the client-server setup and the mix of applications running. Therefore, our experiments measure the isolated overhead of maintaining inlists and outlists. To compute the overhead correctly, care is needed so that the collector's work is not hidden in idle periods such as disk accesses due to application fetches and commits. We ensured this by avoiding an external application and generating work for the collector within the server.

5.4 Performance Results

We ran experiments on a DEC Alpha 3000/400, 133 MHz, workstation running DEC/OSF1. The database disk has a bandwidth of 3.3 Mbyte/s and an average access latency of 15.1 ms. The model for the log disk has a bandwidth of 5 Mbyte/s and average rotational latency of 5 ms (we ignore seek time because the log is written sequentially).

5.4.1 Reference Range and Partition Size

Figure 6 shows the results of running the benchmark with various partition sizes and reference ranges. Each curve represents a constant partition size as the reference range varied. As expected, the overhead increases with reference range and decreases with partition size. The figure illustrates the danger of using very small partitions when the locality of references is poor.

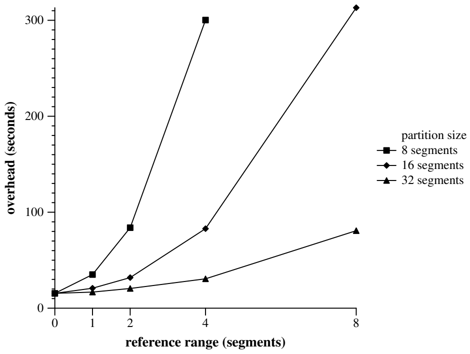


Figure 6: Effect of varying partition sizes.

Figure 7 shows the breakdown of the overhead for a partition size of 32 segments. The processor overhead has been divided into scanning overhead, which is a constant cost due to scanning objects, and list processing, which is mostly due to building up and merging potential and delta lists. The disk reads are due to reading in segments containing basic list objects, and the disk writes are due to flushing dirty segments back to the disk. The log forces are due to committing transactions containing modified basic list objects. The overhead increases steeply with the number of inter-partition references because the the database disk becomes an increasing bottleneck: Potential and delta lists are merged more often, causing increased accesses to the basic lists.

These results may also be used practically to determine a suitable partition size given the locality of references in some part of the database.

5.4.2 Memory Allocation

A major thesis of this paper is that primary memory is better spent on specific data structures such as potential

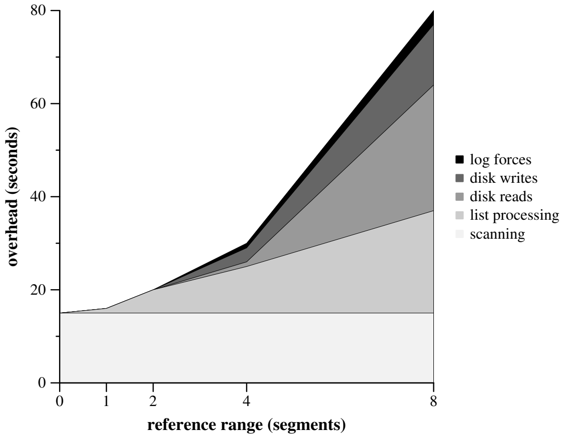


Figure 7: Breakdown of overheads.

and delta lists rather than on caching basic lists. Previous partitioning schemes can be viewed as the extreme case where memory is allocated only for caching basic lists. In this section, we present experiments to determine the best allocation of memory between the various data structures.

For these experiments, we fixed the partition size at 32 segments (1 Mbyte) and the reference range at 8 segments, so that about one-eighth of the references cross partitions. The collector was given 2 Mbytes of memory. Figure 8 shows the overheads for different allocation strategies. There are two independent variables: percentage allocation to the potential lists and percentage allocation to the delta lists, the remaining being used by cached basic lists. Each curve represents a fixed allocation to potential lists, while the allocation to delta lists varied. For example, the top curve represents 10% allocation to potential lists, while the allocation to delta lists varied from 10 to 80%.

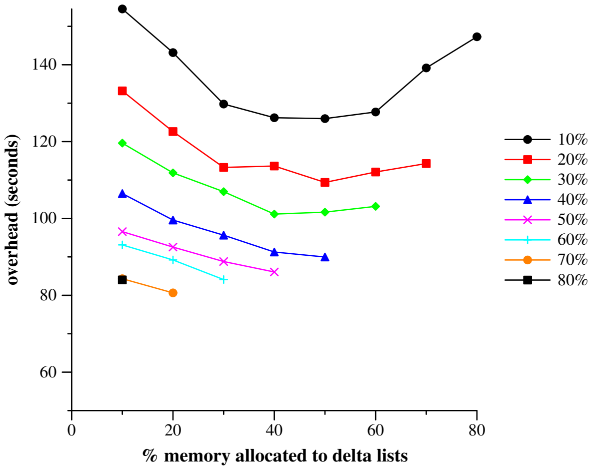


Figure 8: Memory allocation to various lists.

These results establish the advantage of using potential and delta lists over caching basic lists. The figure shows that the overhead decreases with greater allocation to potential lists. Further, when the allocation of potential lists is small (in the top few curves), overheads first decrease and then increase with greater allocation to delta lists. This is because a large allocation to delta lists is ineffective in preventing merges due to small potential lists. When the allocation of potential lists is large (in the bottom few curves), overhead decreases with greater allocation to delta lists. The overhead is minimum with 70% allocation to potential lists and 20% allocation to delta lists. We used this allocation strategy in the experiments performed in Figure 6.

We approximated the case when potential and delta lists are not used by allocating them only 0.1%, i.e., 2 Kbytes (not shown in Figure 8). The measured overhead was 436 seconds; the overhead if potential and delta lists were absent entirely would be even higher.

Figure 9 illustrates the benefit of allocating some memory to delta lists. It shows the overheads when the allocation to basic lists is fixed at 10% and the remaining 90% is shared between potential and delta lists. The overhead increases if the share of the delta lists is reduced below 20%.

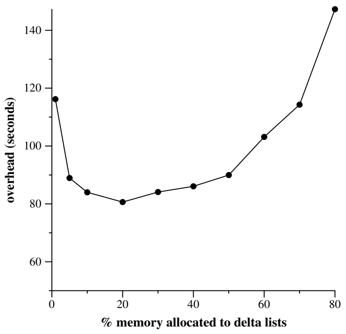


Figure 9: Memory allocation to potential and delta lists.

6 Related Work

Partitioned collection has much in common with independent collection in distributed systems. Therefore, we relate our work to systems with large heaps as well as large distributed systems.

Bishop proposed dividing a large address space into independently collectible partitions in 1977 [Bis77]. He proposed collecting cyclic garbage by migrating objects to partitions that reference them. Migration is also used in some distributed systems because it is fault tolerant and decentralized [SGP90, ML95]. The cost of migration is copying objects and patching up the references to the moved objects. Often, migration also requires inlists to track the identities of the source partitions for each incoming reference.

Hughes's algorithm, also designed for distributed systems, propagates *timestamps* from inlists to outlists and collects objects timestamped below a certain global threshold. This scheme collects cyclic garbage because the timestamps of only the global roots are advanced. (The scheme used timestamps to collect *all* garbage, which is slow.) An advantage of using timestamps over mark bits is that, in effect, multiple marking phases can proceed concurrently in a staggered manner, each identifying some garbage on termination. It is unclear whether such a scheme has any advantage over global marking when partitions are collected sequentially.

Juul and Jul designed a distributed collector with both partition traces and global marking [JJ92]. Inlist entries do not contain reference counts and are removed only by global marking. Thus, the system relies on global marking to collect both cyclic and acyclic inter-partition garbage. Global marking uses a mark bit per object; it is not piggybacked on partition traces and is conducted separately.

Lang *et al.* proposed marking within a selected *group* of partitions to collect inter-partition cyclic garbage contained in the group [LQP92]. Marking is piggybacked on partition traces as in our scheme and comprises a marked trace followed by an unmarked trace. Only inlist and outlist entries need mark bits. However, the scheme does not elaborate on concurrent execution with the mutator, and we believe that it would fail to terminate correctly in the presence of concurrent mutations.

Kolodner proposed recoverable collection of a large heap using unpartitioned but incremental copying [KW93]. Like other unpartitioned schemes, this collector must make random accesses in the old space.

O'Toole *et al.* proposed concurrent copying of a persistent heap by letting applications access the old space while the collector copies it to new space [NOPH92, ONG93]. The collector picks up the modifications made by the applications by using an update log. The scheme was designed for an unpartitioned heap that fit in the main memory.

Yong *et al.* compared unpartitioned incremental copying, reference counting, and partitioned collection in a client-server object store and found partitioned collection to perform the best [YNY94].

Ruffin pointed out similarities between log compaction and garbage collection [Ruf95]. In particular, cleaning log segments, as in Sprite LFS, is similar to partitioned collection. However, inter-segment references need not be tracked because the liveness of a log record can be verified quickly given the file system structure.

Amsaleg *et al.* designed a partitioned collector for a transactional, client-server database [AGF95]. The work focuses on supporting transactional mechanisms such as rollback. The collector uses the log to pick up updates, and the authors point out the need for efficient maintenance of stable inlists.

Maheshwari and Liskov proposed migrating objects suspected to be cyclic garbage [ML95]. Suspects are found using the *distance* heuristic: The distance of an object is the minimum number of inter-partition references on any path from a global root to that object. An object with a large distance is highly likely to be cyclic garbage. Although this reduces migration to the bare minimum, it still requires patching up references to the moved objects.

Rodrigues and Jones collect cyclic garbage in a distributed system using a heuristic to group partitions [RJ96]. A group comprises partitions reached transitively from objects suspected to be cyclic garbage. Multiple sites on the same cycle may initiate separate groups simultaneously, which would fail to collect the cycle. Conversely, a group may include many more partitions than necessary because garbage objects can point to live objects.

Ferreira and Shapiro designed garbage collection for a distributed shared memory system with persistence [FS96]. Segments may be cached at multiple sites and collectors may work on them concurrently. The work focuses on avoiding costly synchronization between the collectors. Each segment has an inlist, but segments cached at a site can be dynamically grouped to form a partition that is traced as a unit. This collects inter-segment cycles contained in the cache. However, note that if an object occurs in multiple garbage cycles (as in a doubly linked list or a tree whose leaves point back to the root), all of those cycles must be cached for any to be collected.

None of the above works addresses efficient maintenance of inter-partition references. The only previous work that addresses this issue is PMOS by Moss *et al.* [HM92, MMH96]. PMOS collects one segment at a time, which is the unit of both fetching and tracing. Each segment contains an inlist that identifies the source segments for incoming references. Outlists are not stored on disk; instead, whenever a segment is read into the cache, it is scanned to compute its outlist. When a modified segment is evicted, it is scanned again to compute differences from the old outlist; the differences are stored in an object equivalent to our delta inlists to avoid disk access. Our use of basic and potential outlists avoids scanning segments when they are fetched or evicted.

PMOS compacts objects across segments. This provides better compaction, but it changes the names of moved objects. Segments containing references to a moved object are scanned and updated. (Cached segments are updated immediately, while those on disk are updated when fetched.)

PMOS collects inter-segment cyclic garbage by grouping segments into *trains*. Reachable objects in a train are gradually migrated to other trains such that the train contains only cyclic garbage at the end and can be discarded. While collecting a segment, objects are moved to the newest segments in the trains that refer to them. Thus, collecting a segment might involve accessing multiple target segments.

Our work on using the log to maintain inlists and outlists builds upon previous work by Ng [Ng96]. His scheme scanned modified objects when they are truncated from the log, and put inter-partition references back into the log. The log was scanned before tracing any partition to find new roots, but that information was not summarized, so the log may be scanned multiple times on different traces.

7 Conclusion

This paper describes a garbage collection scheme for large persistent object stores that makes efficient use of the disk and main memory. The heap is divided into partitions that are collected independently using inlists and outlists, which are maintained stably for fast crash recovery. We use novel techniques to maintain these lists using auxiliary data structures that store conservative updates. The disk is accessed occasionally to reconcile these data structures with the basic inlists and outlists. These techniques also use less memory than caching basic inlists and outlists.

Tracking inter-partition references does not eliminate garbage cycles that span partitions and therefore we augment it with a new global marking scheme. A marking phase propagates marks from the global roots to all reachable objects; at the end of the phase all inter-partition garbage cycles are unmarked and can be collected. Our scheme piggybacks global marking on partitioned collection, does not delay the collection of acyclic garbage, and is guaranteed to terminate correctly in the presence of concurrent mutations. It can take a long time for a phase to terminate, but that is acceptable assuming cyclic garbage spanning partitions is generated slowly.

We have implemented the maintenance of inlists and outlists. We designed a benchmark to evaluate this work that allows to control the crucial parameters determining performance. We presented performance results that show the advantage of using our techniques to avoid disk accesses.

Acknowledgements

We are grateful to Liuba Shrira, Atul Adya, and Miguel Castro for their comments.

References

[AFFS95] Laurent Amsaleg, P. Ferreira, Michael Franklin, and Marc Shapiro. Evaluating garbage collection for large persistent stores. In *Addendum to Proc. 1995 OOPSLA Workshop on Object Database Behavior*. ACM Press, 1995.

[AGF95] Laurent Amsaleg, Olivier Gruber, and Michael Franklin. Efficient incremental garbage collection for workstation–server database systems. In *Proc. 21st VLDB*. ACM Press, 1995.

[Ali85] K. A. M. Ali. Garbage collection schemes for distributed storage systems. In *Proc. of Workshop on Implementation of Functional Languages*, pages 422–428, 1985.

[Bak78] Henry G. Baker. List processing in real-time on a serial computer. *CACM*, 21(4):280–94, 1978.

[Bak93] Henry G. Baker. 'Infant mortality' and generational garbage collection. *ACM SIGPLAN Notices*, 28(4), 1993.

[Bis77] Peter B. Bishop. Computer systems with a very large address space and garbage collection. Technical Report MIT/LCS/TR–178, MIT, 1977.

[CKWZ96] Jonathan E. Cook, Artur W. Klauser, Alexander L. Wolf, and Benjamin G. Zorn. Semi-automatic, self-adaptive control of garbage collection rates in object databases. In *Proc. 1996 SIGMOD*. ACM Press, 1996.

[CWZ94] Jonathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn. Partition selection policies in object databases garbage collection. In *Proc. 1994 SIGMOD*. ACM Press, 1994.

[FS96] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proc. 16th ICDCS*, 1996.

[Ghe95] S. Ghemawat. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 1995.

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In *Proc. IWMM*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[Hug85] R. John M. Hughes. A distributed garbage collection algorithm. In *Proc. 1985 FPCA*, volume 201 of *Lecture Notes in Computer Science*, pages 256–272. Springer-Verlag, 1985.

[JJ92] Neils-Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *Proc. IWMM*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[KW93] Elliot K. Kolodner and William E. Weihl. Atomic incremental garbage collection and recovery for large stable heap. In *Proc. 1993 SIGMOD*, pages 177–186, 1993.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. 1996 SIGMOD*, pages 318–329. ACM Press, 1996.

[LH83] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, 26(6):419–29, 1983.

[LL92] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *Proc. International Conference on Distributed Computing Systems*. IEEE Press, 1992.

[LQP92] Bernard Lang, Christian Queinniec, and José Piquer. Garbage collecting the world. In *Proc. POPL '92*, pages 39–50. ACM Press, 1992.

[ML94] Umesh Maheshwari and Barbara Liskov. Fault-tolerant distributed garbage collection in a client-server object-oriented database. In *Proc. 3rd Parallel and Distributed Information Systems*. IEEE Press, 1994.

[ML95] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of PODC'95 Principles of Distributed Computing*, pages 57–63, 1995.

[MMH96] J. Eliot B. Moss, David S. Munro, and Richard L. Hudson. Pmos: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proc. 7th Workshop on Persistent Object Systems*, 1996.

[Ng96] Tony Ng. Efficient garbage collection for large object-oriented databases. Technical Report MIT/LCS/TR–692, MIT LCS, 1996.

[NOPH92] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In *Proc. IWMM*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[ONG93] James W. O'Toole, Scott M. Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proc. 14th SOSP*, pages 161–174, 1993.

[RJ96] Helena Rodrigues and Richard Jones. A cyclic distributed garbage collector for network objects. In *Proc. 10th Workshop on Distributed Algorithms*, 1996.

[Ruf95] Michel Ruffin. Log compaction and garbage collection: What could they bring to each other? In *Proc. IWOOOS*. IEEE Press, 1995.

[SGP90] Marc Shapiro, O. Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapports de Recherche 1320, INRIA-Rocquencourt, 1990.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.

[YNY94] V. Yong, J. Naughton, and J. Yu. Storage reclamation and reorganization in clinet–server persistent object stores. In *Proc. Data Engineering*, pages 120–133. IEEE Press, 1994.