Partitioned Garbage Collection of a Large Object Store

Umesh Maheshwari
Lab for Computer Science at MIT
umesh@lcs.mit.edu

Barbara Liskov
Lab for Computer Science at MIT
liskov@lcs.mit.edu

Abstract

We present new techniques for efficient garbage collection in a large persistent object store. The store is divided into partitions that are collected independently using information about inter-partition references. This information is maintained on disk so that it can be recovered after a crash. We use new techniques to organize and update this information while avoiding disk accesses. We also present a new global marking scheme to collect cyclic garbage across partitions. Global marking is piggybacked on partitioned collection; the result is an efficient scheme that preserves the localized nature of partitioned collection, yet is able to collect all garbage.

We have implemented the part of garbage collection responsible for maintaining information about inter-partition references. We present a performance study to evaluate this work; the results show that our techniques result in substantial savings in the usage of disk and memory.

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 primary 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 [Bak78, KW93, ONG93] do not scale to very large heaps because the non-local nature of tracing causes random disk accesses. Therefore, large systems partition the heap into independently collectible areas [Bis77, YNY94, AGF95, MMH96, CKWZ96]. This is also the approach taken in many distributed systems [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 [Ung84]; however, the age-based heuristics are not useful in persistent stores [Bak93].

To trace a partition independently of the others, one must remember references to its objects from other partitions and use them as roots. This introduces two problems. One is performance: maintaining information about inter-partition references has a space and time overhead. The other is completeness: tracing from inter-partition references does not collect garbage cycles that span partitions. This paper presents a partitioned collection scheme that solves both problems.

The information about inter-partition references must reside on disk for two reasons. First, a large heap may have enough inter-partition references that recording them in memory would take up substantial cache space. While increasing the partition size reduces the number of inter-partition references, previous studies have shown that tracing very large partitions slows down the collector and the applications due to increased contention for the cache and disk [AGF95]. Second, if the information about inter-partition references is not persistent, recomputing it after a crash would take a very long time.

Maintaining the information persistently requires care in keeping the disk utilization low, both for the garbage collector to perform well and, more importantly, to avoid degrading application performance. In particular, it is important to devise efficient techniques to record inter-partition references arising from the use of small partitions that fit in a fraction of primary memory. We present new techniques to organize and update this information with the following benefits:

1. Information about inter-partition references is recovered quickly after a crash.
2. Disk accesses for updating this information are deferred and batched.
3. Reading objects from the disk and evicting them from the cache do not require processing this information.
4. The information is compact yet efficiently usable.

One other scheme, PMOS [MMH96], batches disk accesses for inter-partition information; however, PMOS processes information and scans objects whenever they are fetched or evicted, which would slow down applications.

We also describe a new global marking scheme that collects cyclic garbage across partitions. Our scheme piggybacks marking on partitioned collection in a way that it adds little overhead to the base scheme. It does not delay the

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.

collection of acyclic garbage, and it preserves the localized and disk-efficient nature of our collector. We prove that the scheme is correct and that it is sure to terminate in the presence of application modifications. As in previous schemes, global marking can take a long time to terminate, but that is acceptable assuming cyclic garbage spanning partitions is generated slowly.

Previous proposals for using global marking with partitioned collection either delay the collection of acyclic garbage [Hug85], or need to run separate traces for global marking and partitioned collection [JJ92], or are not guaranteed to terminate correctly in the presence of modifications [LQP92].

We have implemented the part of garbage collection responsible for maintaining information about inter-partition references in the context of Thor, an object database [LAC+96]. We present a performance study to evaluate this work; the results show that our techniques result in substantial savings in the usage of disk and memory.

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 architecture shown in Figure 1. The stable heap resides on disk, while applications access objects in the heap through a 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. Recent log records are cached in primary memory even after they have been forced to disk. When the log on disk gets too big, it is truncated after ensuring that the modifications have been installed into the stable heap.

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.

We assume logical names are used to refer to objects. Such names are common in objects stores; for example, in Thor and EXODUS, each page provides logical names for its objects by mapping an object's index to the object's location in the page [AGF95, LAC+96]. Our scheme could also be used in systems that store memory addresses in objects, but in that case it would need to store more information to allow compaction.

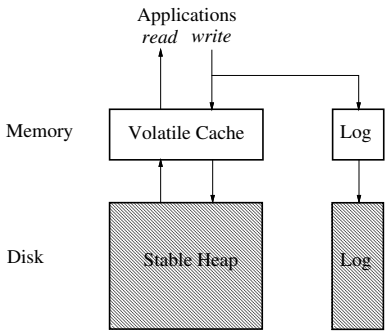


Figure 1: A generic architecture with large heap.

3 Partitioned Collection

The garbage collector divides the heap into partitions for separate collection. A partition contains several pages, possibly non-adjacent. This approach has important advantages. First, the page size is chosen to allow efficient fetching and caching, while a partition is chosen to be an efficient unit of tracing. For example, a page could be tens of kilobytes, while a partition could be several megabytes. Second, it is possible to configure a partition by selecting a group of pages so as to minimize inter-partition references—without reclustering objects on disk. 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.

There is a tradeoff in configuring partitions: 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 disk accesses during tracing. We provide efficient techniques to record inter-partition references so that partitions that fit in a small fraction of primary memory (say, a tenth) can be used.

We assume that, given a reference to an object, its partition can be computed efficiently. We do this by keeping a map from pages to partitions and vice versa.

3.1 Inlists, Outlists, and Translists

To collect partitions independently, we remember the objects in the partition that are referenced from other partitions in an *inlist*, and use the inlist as a root. We also record information about outgoing references from a partition in an *outlist*. The outlist isn't necessary but it allows efficient removal of references from inlists. The scheme by Amsaleg et al. has no outlists [AGF95]; after collecting a partition P, untraced inter-partition references from P must be removed by scanning inlists of other partitions. In PMOS, by Moss et al., the

outlist is computed whenever a page is fetched and also when a modified page is evicted, and the differences are applied to the inlists [HM92, MMH96].

We share information between inlists and outlists in *translists*: a translist from P to Q records the set of references contained in partition P to objects in another partition Q. The inlist of P points to the set of translists from other partitions to P, and its outlist points to the set of translists from P to other partitions. Figure 2 illustrates the various lists.

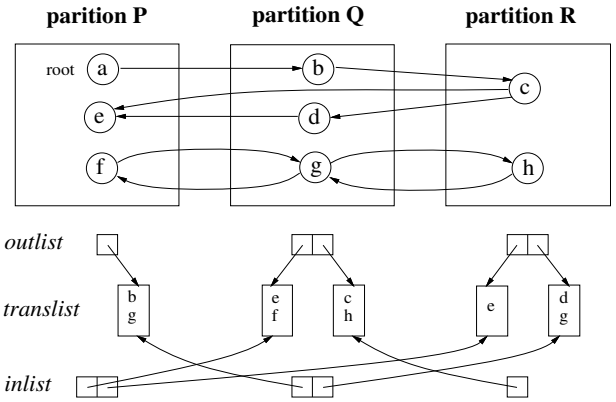


Figure 2: Inlists, outlists and translists.

Sharing translists between inlists and outlists provides significant advantages. Without translists, inlists and outlist would store redundant information, resulting in higher overheads and complexity. Our previous scheme did not use translists and as a result the system invariants were more complex than they are now [Ng96, LMN96]. Additionally, the new scheme provides compact storage. Inlists and outlists use two words per translist, and translists use one word per ⟨source-partition, reference⟩ pair.

The *remembered sets* used in some generational collectors record the locations—at the level of a word, object, or page—that may contain inter-partition references [Ung84, Sob88]. This scheme is not suitable for disk-based heaps because tracing a partition requires examining locations in other partitions. Further, storing locations at a fine granularity results in more information if multiple locations contain the same reference.

Inlists, outlists, and translists are kept on disk because otherwise they would occupy space in primary memory and it would take a long time to recompute them after a crash. We maintain them as regular heap objects. Translists to the same partition are clustered together, since this speeds up garbage collection.

3.2 Updating Translists

When an application creates an inter-partition reference from partition P to Q, the translist from P to Q needs to be updated. This can be done lazily by scanning modified objects in the log; for good performance scanning is done before that portion of the log is un-cached from primary memory.

Updating translists requires either keeping translists cached in primary memory or reading them from disk. We save cache space and defer reading or writing the disk as follows. When a modified object in partition P is scanned, we record references to another partition Q in an in-memory *delta list* from P to Q. Only new references need be added to the delta list. In many transactional systems, old copies of modified objects are retained in case the transaction aborts; if the old copy is cached, it can be used to avoid unnecessary additions to delta lists. However, there may still be overlap between delta lists and translists.

When the aggregate size of delta lists grows above a certain threshold, we merge the largest delta list into the corresponding translist. Fetching the translist is likely to prefetch other translists to the same partition; therefore we merge all other delta lists for that partition at this point. Merging is repeated until the aggregate size falls below a low watermark. The modified translists are written to the log; in addition if a new translist is created, we enter it in the appropriate inlist and outlist and write them to the log.

3.3 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 partition, we load its pages into the cache; this is possible since a partition is a small fraction of the cache. We also process the log completely to find all new references to the partition.

Roots. We include the following in the root set:

1. The persistent root of the heap.
2. Roots from applications.
3. The translists in the inlist of the partition.
4. Any delta list to the partition.

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.

As the collector traces a partition, it records all inter-partition references it reaches. At the end of collection, it uses this information to update the translists in the outlist of the partition. This effectively updates the root set for other partitions.

Compaction. If the collector compacted storage by moving objects such that their names changed, it would be necessary to update references to moved objects—including those that are stored in other partitions and in application variables. We avoid these updates by compacting objects within their pages, which preserves their logical names as described in Section 2. Additionally, objects that are not referenced from other partitions or applications can be moved to other pages within the partition.

Compaction that changes object names could be supported by recording locations containing inter-partition references, as in some generational collectors. However, such mechanisms add substantial cost and complexity in a disk-based heap.

Tracing Scheme. Our approach can be used in combination with various concurrent collectors. For example, we could use a replicating collector [ONG93]. Such a scheme requires little synchronization with applications, but needs space for two partitions in primary memory. A mark-and-sweep collector can be used as well, as in [AGF95]. The sweep phase can compact one page at a time, either by locking the page from applications and sliding objects, or by making a new copy of the page and updating the page table to point to it. No work is needed for pages with no garbage objects; this may be a significant advantage over copying collectors in persistent stores, where often little garbage is created.

Marking (or copying) in partition P can terminate when all objects logged before the collector scanned the last object in P have been scanned; it may be necessary to stop mutations briefly to ensure termination.

3.4 Fault Tolerance

We guarantee that inter-partition information survives crashes by storing translists, inlists, and outlists as regular persistent objects. Thus, their modifications are logged and installed on disk later.

Information in delta lists for modifications stored in the log need not be logged; this information is recovered by reprocessing the log after a crash. However, when the stable log is truncated, the affected delta lists must be made stable. One way to do this is to divide each delta list into stable and volatile parts. New references are first added to the volatile part. When the log is truncated, the volatile parts of the affected delta lists are logged and marked stable. A log record for a delta list remains in the log until the delta list is merged with the translist; logging the translist effectively deletes log records for the associated delta list.

3.5 Disk Accesses

The following summarizes the disk accesses involved in the scheme:

1. Truncating the log:
   - log volatile parts of affected delta lists.
2. Merging delta lists:
   - fetch inlist and translists, if necessary.
   - log updates to the translists and, if a translist is new, to the containing inlist and outlist.
3. Tracing partition P:
   - fetch inlist and outlist of P and all translists in the inlist.
   - fetch pages of P.
   - flush pages of P modified by the collector.
   - log updates to translists; if a translist is created or deleted, log outlist of P and affected inlists.

Note that the log updates in parts 2 and 3 above need not be forced to the disk until the log is truncated. A crash might lose unforced updates, but that is acceptable because the information that generated those updates will be regenerated later.

4 Collecting Cyclic Garbage

We collect inter-partition cyclic garbage using a global but incremental marking scheme. At the beginning of a global mark phase, only the persistent root is marked. Each partition trace propagates marks from the root set of the partition to the references in its outlist. The mark phase terminates when marks are known to have propagated fully through all partitions. We show that our scheme is correct and that it terminates in the presence of mutations.

The scheme has little time and space overhead and does not delay the collection of acyclic garbage; as in partitioned collection, we organize the collector information to use the disk efficiently. Previous marking schemes either propagate global marks separately from regular partition traces [JJ92], or delay the collection of acyclic inter-partition garbage [Hug85], or are not guaranteed to terminate correctly in the presence of concurrent mutations [LQP92].

4.1 Data Structures

Each partition has a *markmap*, which contains a mark bit per object. The markmap is implemented as a set of bitmaps, one per page in the partition; each bitmap contains a bit per potential object name in the page. For example, in Thor, a 32 Kbyte page provides a name space for 2K objects; thus each bitmap is 2K bits, which represents a 0.8% overhead. Nevertheless, the aggregate size of the markmaps is large, e.g., 80 Mbytes for a 10 Gbyte database. Therefore, we maintain markmaps on disk.

A partition may also have an in-memory *delta markmap*, which stores updates to the markmap, thus deferring and batching disk accesses for the markmap. Delta markmaps are created on processing entries in the log and also on tracing a partition. We say that an object is marked if it is marked in the markmap, and is delta marked if it is marked in the delta markmap; a reference is marked (delta marked) if the object it refers to is marked (delta marked).

Each partition also has a mark bit to denote whether the marks of its objects have been propagated. As described later, tracing a partition causes it to become marked, but may cause other partitions to become unmarked.

4.2 Invariants and Rules

For a global mark 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 invariant used in the proof:

Invariant 1. In a marked partition, all references contained in marked objects are marked or delta marked.

The invariant is true of both inter- and intra-partition references in marked objects. It ensures safety: when marking terminates, all objects reachable from the persistent root are marked. We use the following rules to guarantee termination:

Rule 1. A marked object is never unmarked during a phase.
Rule 2. Objects created during the current phase are marked.
Rule 3. Every time we unmark a partition, we mark at least one of its unmarked objects.

In fact, we keep a mark bit per object expressly to guarantee termination by enforcing these rules. Otherwise, mark bits for just the inter-partition references would suffice, as in [Hug85, LQP92].

4.3 Starting a Phase

When a phase starts, only the persistent root is marked. The partition containing the persistent root is unmarked and the rest are marked. This satisfies Invariant 1. We perform these actions incrementally as follows.

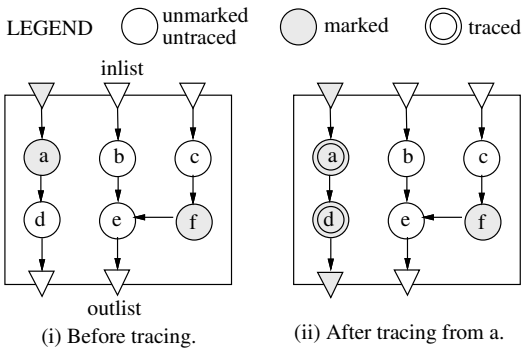
We keep a *global phase counter* that is incremented with each phase. In addition, a local phase counter for each partition tells the phase during which it was last traced. When a partition is traced, if its local counter is one less than the global counter[1], this must be its first trace in the current phase. Objects that were unmarked in the previous phase are known to be garbage and are deleted. Then, the mark bits of all remaining objects in the partition are cleared, while those of non-existing objects are set to prepare for their future creation (Rule 2).

4.4 Tracing a Partition

Global marking is piggybacked on regular tracing of partitions. Before tracing, we merge the partition's delta markmap into its markmap. At the end of collection, we mark the partition and log its markmap.

We use the term "marked" for objects reached by global marking and "traced" for objects reached while tracing the partition. The persistent root, application roots, and marked inlist references are traced first; we call this the *marked trace*. Objects reached during this trace are marked, while inter-partition references reached are marked in the delta markmap of the target partition. Figure 3 shows the effect of the marked trace.

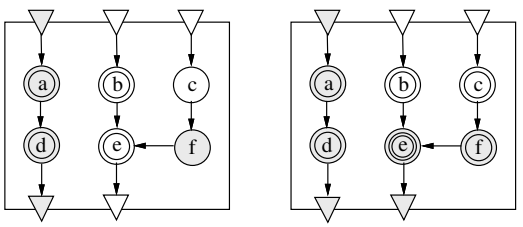
[1] If the partition's local counter is even smaller, it was not visited during the previous phase; therefore the whole partition is garbage and can be discarded.



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

Figure 3: Marked trace of a partition.

Unmarked inlist references are traced next; this constitutes the *unmarked trace* 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 or delta marked earlier, but subsequent modifications by applications have made them locally unreachable from the current set of marked inlist references; we call them *marked orphans*. Orphans pose a problem that has not been handled in previous schemes: Invariant 1 requires that references in marked objects be marked or delta marked. However, marked orphans may point to objects that have already been traced but were not marked, such as e in the figure.



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

Figure 4: Unmarked trace.

Some solutions that seem simple at first are in fact not acceptable. We could preserve Invariant 1 by *un*marking the marked orphans, but this violates Rule 1 needed for termination. Alternatively, we could trace from all marked objects first, whether or not they are in the inlist. However, this can cause many objects that would otherwise be collected to survive the entire global phase.

An acceptable solution is to delta mark any unmarked reference e contained in an orphan. However, this will make e an orphan in the next trace. We expedite the propagation of marks by recursively tracing any unmarked reference in an orphan as another orphan—even if it has been traced before. This expedites termination of global marking at the cost of possible retracing of unmarked objects reachable from marked orphans. However, even in the worst case, each object can be retraced at most once over an entire global phase.

This is true because retracing marks an object that was unmarked. In practice, retracing would be much less frequent, since marked orphans are likely to be rare.

Note that global marking does not cause any object to survive longer than the object would otherwise:

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

4.5 Processing the Log

We preserve Invariant 1 lazily by processing references contained in modified and new objects in the log. We process a reference y contained in object x as follows. If the partition containing x is unmarked, we ignore the reference. Otherwise, we enter y in the delta markmap of y's partition, which may be different from x's partition.

All new objects must be marked; we accomplish this by keeping mark bits set for unused object names as discussed in Section 4.3. Marks do not keep new objects from being collected in the current phase: as mentioned, untraced objects are collected even if they are marked. However, if marked objects form an inter-partition garbage cycle, they will be collected in the next phase.

When the aggregate size of delta markmaps grows above a certain limit, we select the ones with the most marks, read their markmaps, and merge them into the markmaps. If this causes any unmarked object to be marked, we unmark its partition to preserve Invariant 1. This also follows Rule 3 needed for termination.

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 delta markmaps have been merged.
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 delta markmaps. Then we merge all delta markmaps. 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 and repeat the procedure.

Safety. Processing all references in the log ensures that Invariant 1 holds. Merging all delta markmaps ensures that there is no reference that is delta marked but not marked. Therefore, by Invariant 1, all objects reachable from marked objects must be marked. In particular, since the persistent root and applications roots are known to be marked, all objects reachable from them are marked.

Liveness. Marking is sure to terminate because a partition can be unmarked only a finite number of times during a phase. Every time we unmark a partition, we mark at least one of its unmarked objects (Rule 3). Further, such an object must have been created before the current phase because objects created during this phase are always marked (Rule 2). Finally, a partition has a finite number of objects created before this phase, and marked objects are never unmarked. Therefore termination is guaranteed 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, as recommended by Cook el al. [CWZ94].

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 persistent root is l. The distance of an object is the smallest number of inter-partition references in any path from the persistent root to the object. 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 ten thousand partition traces.

4.7 Fault Tolerance

Since we maintain markmaps on disk to save primary memory, it takes little more to make them recoverable after crashes. This allows global marking to be resumed after a crash, which is desirable because global marking takes relatively long to finish. We maintain markmaps and mark bits of partitions as regular persistent objects. They are updated after tracing a partition and also after merging delta markmaps.

Updates to delta markmaps due to references in the log are 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. Updates to delta markmaps due to tracing a partition are logged before marking the partition stably. When a delta markmap is merged into its markmap, logging the markmap effectively removes any log records for the delta markmap.

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 Performance

We are implementing partitioned garbage collection in Thor, an object database [LAC+96]. This section describes some details of the implementation and then presents some experiments to evaluate our technique for maintaining information about inter-partition references. The performance of maintaining this information is particularly important because it is a steady-state activity that must be carried out as objects are modified—unlike tracing partitions, which can be scheduled occasionally.

5.1 The Context

Thor is a client-server object database. Servers store objects on disk in fixed-sized pages, currently 32 Kbytes, which are the units of disk access. Servers also maintain a cache of recently fetched pages in memory. Applications running on client machines fetch objects from servers into the client cache and access the objects locally.

Objects are accessed within transactions that run at the clients; at commit, copies of modified objects are sent back to the servers. The server stores modified objects in a memory-resident log and installs them into disk pages in the background [Ghe95]. Log stability is intended to be accomplished through replication [LGG+91], but in our experiments we simulate delays for log forces as if the log were stored on a logging disk, separate from the database disk.

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 Implementation

Work related to garbage collection is performed by a *collector* thread, which is run at low priority to avoid delaying application requests. The collector scans modified and new objects in the log for inter-partition references.

Inlists, outlists, and translists are stored as linked lists of fixed-sized *block* objects. A translist block stores a compact array of references in no particular order. An inlist (outlist) block stores references to the contained translists paired with the source (target) partition ids. Blocking allows these lists to grow or shrink without copying; we use relatively small blocks (64 bytes) to reduce fragmentation. When updating a list, we log only the modified and new blocks. These blocks are logged using transactions, except that we bypass the concurrency control mechanism since only the collector accesses them. One problem with blocking is that the blocks of a list may be mixed with those of other lists and should be consolidated periodically.

A delta list is implemented as a hash table. To merge a delta list into a translist efficiently, we iterate through the translist: for every reference x in the translist, we remove x from the delta list, if present. At the end, we append remaining references in the delta list into the translist.

We have not yet implemented crash recovery and the actions needed on truncating the stable log. In our experiments, however, we accounted for the expected log overhead from these actions.

5.3 Workload

Amsaleg et al. pointed out the lack of a standard benchmark for database garbage collectors [AFFS95]; such a benchmark remains absent today. Therefore, we designed a micro-benchmark specifically for evaluating the overhead of maintaining inter-partition reference information.

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, the benchmark allows us to control the distribution of reference modifications systematically: both the spatial locality of references, i.e., where they point, and the temporal locality, i.e., which references are modified in time order.

The workload consists of selecting a *cluster* of objects at random and initializing their references; a cluster comprises a fixed number of contiguous objects. The process is repeated until the database is fully initialized. Random selection of clusters simulates the effect of concurrent applications creating or modifying groups of clustered objects. The cluster size is a measure of temporal locality in reference modifications.

Spatial locality is determined by how the target object of a reference is chosen. Objects are numbered sequentially, and object n refers to a random object $n + i$ using a chosen probability distribution for i. (Object numbers wrap around when they overflow or underflow the bounds of the database.) The database is partitioned linearly; each partition contains p objects.

We used the following distribution of references. With some probability s, a reference points within the containing page. With probability $1 - s$, a reference points to an object that is i apart according to the exponential distribution:

$$prob(i) = \frac{1}{2d} e^{-\frac{|i|}{d}}, -\infty < i < \infty$$

We call d the *deviation*. While s governs the number of inter-partition references, d governs their spread: a small deviation keeps them to nearby partitions, while a large deviation spreads them further. All experiments reported in this paper used a deviation equal to the number of objects in a partition, which resulted in each partition containing references to its 16 neighbors on the average. Moreover, given this deviation, a fraction $1/e$ of references under the exponential distribution fall in the same partition. Thus, the overall *cross fraction* of references f that cross partitions is $(1 - s)(1 - 1/e)$. Table 1 summarizes the parameters employed for the workload.

In our database of 256 Mbyte, a cross fraction of 10% leads to about 4 Mbyte of translists, inlists, and outlists. The lists represent 1.5% overhead with respect to the database

Parameter	Value(s) [Default]
Database size	256 Mbyte
Partition size	1 Mbyte
Object size	30 bytes
Objects per page	1K
Objects per cluster	32–8192 [1024]
Cross fraction f	0–15% [7.5%]
Deviation d	32K objects

Table 1: Workload parameters.

size, which is a small overhead on disk. However, for realistic database sizes, the amount of this information would be significant compared to primary memory.

5.4 Experimental Configuration

We designed experiments to determine the *relative* performance of our scheme with and without delta lists for a range of workload parameters. We refer to the scheme with delta lists as DELTA, and that without delta lists as NODELTA.

The experiments ran 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 ms. The model for the log disk has a bandwidth of 5 Mbyte/s and average rotational latency of 5 ms; we ignore its seek time because the log is written sequentially.

To compute the correct overhead of maintaining inter-partition information, 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.

The collector is given a fixed amount of primary memory to store lists. It is also given space in the memory-resident log for modified lists. In our experiments, we allocated 1 Mbyte each for the list memory and the list log.

In NODELTA, the collector uses the list memory to cache translists, inlists and outlists. It also uses a small delta list (1 Kbyte) for efficient addition of references to translists. (In a real implementation without delta lists, translists could be implemented as B-trees for efficient addition.)

In DELTA, the collector uses the allocated space to store delta lists and to cache other lists. We determined how to best allocate this space between delta lists and other lists experimentally. Figure 5 shows the variation in collector overhead as delta lists are allocated a bigger fraction of available memory. (Here, the cross fraction was fixed at 7.5%.) The overhead is the lowest when three quarters of the space is allocated to the delta lists. Therefore, we used this division in our experiments.

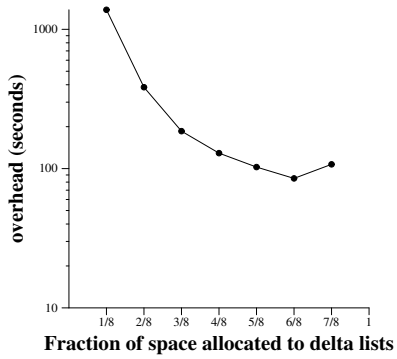


Figure 5: Effect of space allocated to delta lists.

5.5 Performance Results

This section reports on the results of running the benchmark for a range of workload parameters. We show how the collector overhead varies with the cross fraction, the cluster size, and the amount of list memory available. For each of these parameters, we compare the performance of DELTA and NODELTA.

Figure 6 shows the results of running the benchmark in the DELTA configuration as the fraction of references crossing partitions, f, is increased. The processor overhead comprises scanning overhead, which is a constant cost due to scanning objects, and list processing, which is mostly due to manipulating various lists in memory. The disk reads are due to fetching pages containing list blocks, and the disk writes are due to installing dirty pages on the disk. The log forces are due to committing transactions containing modified blocks; the log overhead is too small to be visible in the figure. Most of the overhead is due to processing and disk reads. The overhead increases steeply with the number of inter-partition references as the database disk becomes an increasing bottleneck.

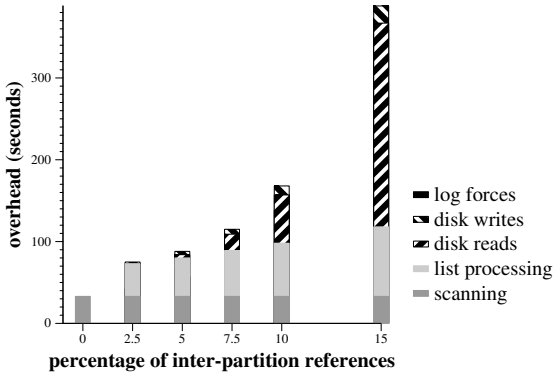


Figure 6: Breakdown of collector overhead.

Figure 7 compares the disk read overheads in the DELTA and NODELTA configurations as inter-partition references increase. When the amount of information exceeds the allocated cache space (beyond 2.5% cross fraction), NODELTA

begins to thrash on memory. At 7.5% cross fraction, the disk read overhead for NODELTA is higher by a factor of more than 100. (The processor overhead of NODELTA is also much higher than DELTA because delta lists are efficient for addition of references into translists. However, we discount this advantage because other techniques could be used in NODELTA to improve processing time.)

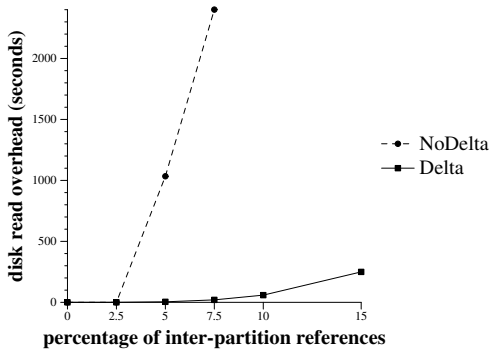


Figure 7: Effect of spatial locality on disk reads.

Figure 8 shows the overheads for DELTA and NODELTA as the cluster size varied from 32 to 8K objects (note log scale). Here, the fraction of inter-partition references was fixed at 7.5%. Higher clustering reduces overheads by increasing temporal locality. DELTA retains good performance for a low cluster size of 32 objects, while NODELTA was unable to finish in 3000 seconds.

In fact, DELTA has lower overhead for a cluster size of 32 than for 1K. This is because a low cluster size results in a large number of small delta lists. Since we merge delta lists to the same partition at the same time, a lower cluster size provides greater opportunity to create translists to the same partition close together. This results in fewer disk reads on later merges. For very small cluster sizes, this advantage outweighs the disadvantage from poor temporal locality. The results indicate that good placement of translists is important to performance; probably a system should recluster them periodically.

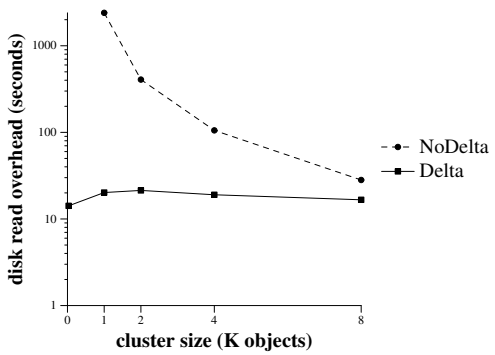


Figure 8: Effect of temporal locality on disk reads.

Finally, Figure 9 shows how the performance varies with the amount of memory allocated to the collector. The fraction of inter-partition references was fixed at 7.5%. As more memory is allocated to hold lists, the difference in the performance of DELTA and NODELTA is reduced. When the collector is given enough space to store all translists in memory (3 Mbyte), NODELTA has lower overhead than DELTA by a small margin. This is because DELTA allocates some of the space for delta lists and therefore has a smaller translist cache. However, the disadvantage is negligible compared to the advantage when less memory is available.

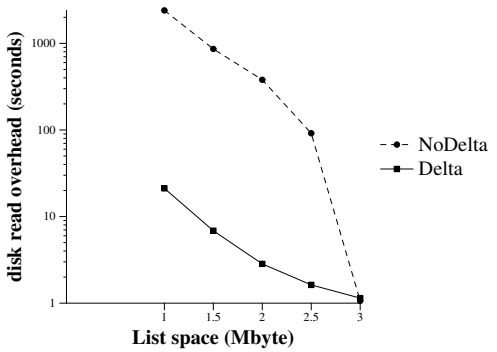


Figure 9: Effect of list memory on disk reads.

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 first proposed dividing a large address space into independently collectible partitions [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.

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 only timestamps of the global roots are advanced. An advantage of using timestamps over mark bits is that, in effect, multiple marking phases can proceed concurrently in a staggered manner. 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 partitioned collection and global marking [JJ92]. The scheme relies on global marking to remove unnecessary inlist references, whether or not they form an inter-partition garbage cycle. Global marking 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 references 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 [ONG93]. The collector picks up the modifications made by applications by using an update log. The scheme was designed for an unpartitioned heap that fit in the primary memory.

Yong *et al.* compared incremental copying, reference counting, and partitioned collection in a client-server object store and found partitioned collection to perform the best [YNY94]. They used remembered sets that recorded the objects containing inter-partition references, which had to be fetched and scanned before tracing a partition.

Amsaleg *et al.* designed a partitioned collector for a transactional, client-server database [AGF95]. Their work focuses on supporting transactional mechanisms such as rollback. The collector uses the log to process inter-partition references in modified objects; the authors point out the need for efficient maintenance of stable inlists.

Maheshwari and Liskov proposed identifying objects that are highly likely to be cyclic garbage and migrating them [ML95]. An object is suspected to be cyclic garbage if it has a large distance: The distance of an object is the minimum number of inter-partition references on any path from a persistent root to that object. Although the scheme avoids unnecessary migration, it still requires patching up references to the moved objects.

Ferriera and Shapiro designed a collector for a distributed shared memory system that caches replicas of data segments [FS96]. Each segment has an inlist and outlist, and segments cached at the same node can be grouped to form a unit of tracing. However, such a group may not contain all of a garbage cycle. For a compound cycle such as a doubly-linked list to be collected, all component 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 page at a time; a page is the unit of both fetching and tracing. Each page has an inlist that identifies the source pages for each incoming reference. Outlists are not stored on disk; instead, whenever a page is read into the cache, it is scanned to compute its outlist. When a modified page is evicted, it is scanned again to compute differences from the old outlist; the differences are stored in tables similar to delta lists to avoid disk accesses.

PMOS compacts objects across pages. This provides better compaction, but it changes the names of moved objects. Pages containing references to a moved object are scanned and updated. (Cached pages must be updated before they are used, while those on disk are updated when fetched.) PMOS collects inter-page cyclic garbage by grouping pages 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 page, objects are moved to the newest pages in the trains that refer to them. Thus, collecting a page may involve accessing multiple target pages. Further, the collector precludes application control over the placement of objects.

7 Conclusion

This paper has presented new techniques to solve two problems with partitioned garbage collection in large object stores: the overhead of maintaining information about inter-partition references, and the collection of inter-partition cyclic garbage.

The information about inter-partition references is maintained on disk so that it uses little space in primary memory and can be recovered after a crash. We provide a compact yet efficient organization for this information using inlists, outlists, and translists. We avoid disk accesses to maintain these lists by keeping information about new inter-partition references in delta lists.

The performance of maintaining this information is important because it is a steady-state activity that must be carried out as objects are modified. The paper presents a benchmark to evaluate this performance under a range of workload parameters. The results show that, when available memory is limited, delta lists allow processing of inter-partition references with much less disk overhead than possible otherwise.

We have augmented partitioned collection with a new global marking scheme to collect cyclic garbage across partitions. 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. Further, it preserves the localized nature of partitioned collection.

Acknowledgements

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

References

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

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

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

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

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

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

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

[FS96] P. Ferreira and M. 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] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In *Proc. IWMM*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[Hug85] R. J. 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] N.-C. Juul and E. 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] E. K. Kolodner and W. 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.

[LGG+91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. SOSP*, pages 226–238. ACM Press, 1991.

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

[LMN96] B. Liskov, U. Maheshwari, and T. Ng. Partitioned garbage collection of a large stable heap. In *Proc. IWOOOS*, 1996.

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

[ML94] U. Maheshwari and B. 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] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of PODC'95 Principles of Distributed Computing*, pages 57–63, 1995.

[MMH96] J. E. B. Moss, D. S. Munro, and R. 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] T. Ng. Efficient garbage collection for large object-oriented databases. Technical Report MIT/LCS/TR–692, MIT LCS, 1996.

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

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

[Sob88] P. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT, AI Lab, 1988.

[Ung84] D. 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.