Efficient Incremental Garbage Collection for Client-Server Object Database Systems

Laurent Amsaleg
INRIA Rocquencourt
Laurent.Amsaleg@inria.fr

Michael Franklin[*]
University of Maryland
franklin@cs.umd.edu

Olivier Gruber[†]
IBM Research Division
gruber@almaden.ibm.com

Abstract

We describe an efficient server-based algorithm for garbage collecting object-oriented databases in a client/server environment. The algorithm is incremental and runs concurrently with client transactions. Unlike previous algorithms, it does not hold any locks on data and does not require callbacks to clients. It is fault tolerant, but performs very little logging. The algorithm has been designed to be integrated into existing OODB systems, and therefore it works with standard implementation techniques such as two-phase locking and write-ahead-logging. In addition, it supports client-server performance optimizations such as client caching and flexible management of client buffers. We describe an implementation of the algorithm in the EXODUS storage manager and present results from an initial performance study.

1 Introduction

A primary strength of Object Oriented Database Management Systems (OODBMS) lies in their ability to model complex objects and the inter-relationships among them. This modeling power, while allowing for a more natural representation of many real-world application domains, also raises new problems in the design of fundamental lower-level functions such as storage management and reclamation. It has long been recognized that explicit storage management (e.g., *malloc()* and *free()*) places a heavy burden on the developers of large programs. Manual detection and reclamation of garbage increases code complexity and is highly error-prone, raising the risk of memory leaks and dangling pointers. For these reasons *automated* garbage collection for programming languages has long been an active area of investigation [Wil92].

The shared and persistent nature of databases provides further motivation for automated garbage collection. Because a database is shared, the knowledge of data interconnection may be distributed among many programs and/or users, making it difficult for programmers to determine when to explicitly reclaim storage. Furthermore, sharing and persistence increase the potential damage caused by storage management errors, as mistakes made by one program can inadvertently affect others. These considerations argue for a very cautious approach to storage reclamation, however, neglecting to reclaim wasted space can degrade performance due to increased I/O.

In an OODBMS, the notion of persistence (and hence, garbage) can be tied to *reachability*. Reachability is a simple, implicit way to express persistence that is orthogonal to object type [ZM90]. The database is an object graph in which some objects are designated as persistent roots. Objects that can be reached by traversing a path from a root can persist beyond the execution of the transaction that created them. Objects that are not reachable from a root or from the transient program state of an ongoing transaction are garbage; such objects are inaccessible and thus, they can be safely reclaimed. Reachability-based persistence is used in the GemStone [BOS91] and O2 [BDK91] systems, and garbage collection is required in the Smalltalk binding for the ODMG object database standard [Cat94]. In general, however, most existing systems still rely on explicit object deallocation and/or off-line storage management utilities. The lack of acceptance of reachability-based persistence is due in part to the absence of efficient implementation techniques that can be integrated with existing systems. The work described here is aimed at addressing this need.

1.1 OODBMS Garbage Collection Issues

In recent years, there has been increasing research activity in OODBMS garbage collection [But87, FCW89, ONG93, KW93, YNY94]. OODBMS have several features that can impact the correctness and/or performance of traditional garbage collection approaches:

Concurrency: A garbage collector must co-exist with concurrent transactions and must not adversely impact their ability to execute.

Client Caching: OODBMS typically cache and update data at clients. The dynamic, replicated, and distributed nature of these updates makes client-based garbage collection problematic. Server-based collection is difficult because a server may have an inconsistent snapshot of the database.

Atomic Transactions: The rollback of partially completed transactions violates the fundamental assumption that unreachable objects always remain unreachable.

[*]This author's work was partially performed as a Visiting Researcher at INRIA Rocquencourt, and was supported by NSF RIA IRI-94-09575 and a grant from the University of Maryland General Research Board.

[†]This author's work was performed while he was with INRIA Rocquencourt.

**Proceedings of the 21st VLDB Conference
Zürich, Switzerland, 1995**

Persistence: Modern garbage collection algorithms assume that the volume of live (i.e, non-garbage) objects is small. Persistence invalidates this assumption.

Fault Tolerance: OODBMS provide resilience to system failures. Garbage collection must also operate in a fault tolerant manner.

Disk-Resident Data: Much of the data in an OODBMS is on disk — the collector cannot ignore I/O.

1.2 Overview of the Paper

In this paper, we describe a client-server OODBMS garbage collector that has the following characteristics:

- It is server-based, but requires no callbacks to clients and performs only minimal synchronization with client transactions.
- It works in the context of ACID transactions [GR93] with standard implementation techniques such as two-phase locking and write-ahead-logging.
- It is incremental and non-disruptive; it holds no locks on data and requires minimal logging. It has been designed to be efficient with respect to I/O.
- It is fault tolerant — crashes of clients or servers during garbage collection will not corrupt the state of the database.
- It can co-exist with performance enhancements such as inter-transaction caching at clients and "steal" buffer management between clients and servers.
- It has been implemented and measured in the client-server EXODUS storage manager.

Similar to other recent work on DBMS garbage collection [CWZ94, YNY94] we use a *partitioned* approach in order to avoid having to scan the entire database before reclaiming any space. In contrast to the other work, which advocated copying or reference counting, we use a non-copying Mark and Sweep algorithm.

The remainder of this paper is structured as follows: Section 2 describes our architectural assumptions and motivates our choice of garbage collection approach. Section 3 describes three problem scenarios that must be addressed when adapting this approach to a client-server OODBMS. Section 4 describes the partitioned Mark and Sweep collection algorithm. Section 5 details the implementation of the algorithm in EXODUS. Section 6 presents an overview of our performance studies on the implemented system. Section 7 discusses related work. Section 8 presents conclusions and future work.

2 Setting the Stage

2.1 Client-Server OODBMS Architecture

Client-server OODBMS architectures typically use *data-shipping* — data items are sent from servers to clients so that query processing (in addition to application processing) can be performed at the clients. We focus on *page servers*, in which physical data units (i.e., pages) are transferred between servers and clients [DeW90, CFZ94]. Upon request, data pages required by an application are replicated and brought into the client's local memory. These pages may be cached at the client both within a transaction and across transaction boundaries. Clients perform updates on their cached page copies. If clients follow a "steal" buffer management policy with the server, then these updated pages can be sent back to the server at any time and in any order, during the execution of the updating transaction. This flexibility, while providing opportunities for improved performance, also raises potential problems for garbage collection algorithms.

2.2 Assumptions

Our solution depends upon several assumptions about the OODBMS architecture:

Assumption A1: All *user* operations that access or modify object pointers are done using two-phase locks that are held until the end of transaction (i.e., degree 3 consistency [GR93]).

Assumption A2: The validity of object identifiers (OIDs) is maintained only for those OIDs that reside in the database or in the transient state (e.g., program variables, stacks, etc.) of *active* transactions. For example, an OID extracted from the database and stored elsewhere is not guaranteed to be valid for use by a subsequent transaction.

Assumption A3: The system follows the write-ahead-logging (WAL) protocol between clients and the server. That is, a client sends the server all log records pertaining to a page before it sends a dirty copy of that page to the server, and sends the server all log records pertaining to a transaction prior to the commit of that transaction.

Assumption A4: Client buffering follows a "force" policy: all pages dirtied by a transaction are *copied* to the server prior to transaction commit. The force is to the server's memory (i.e., no disk I/O is required) and clients can retain copies of the pages.

Assumptions A1 and A2 are fundamental to the algorithm's correctness; A3 and A4 are important details on which the implementation of the algorithm is based. A3 must be supported by any client-server DBMS that implements WAL-based recovery at servers (e.g., EXODUS [Fra92], and ARIES/CSA [MN94]). A4 simplifies recovery and avoids the need for client checkpoints in such systems. The tradeoffs involved in relaxing A4 are discussed in [Fra92], [FCL93], and [MN94]. For the purposes of garbage collection, relaxing A4 would require additional coordination with the buffer manager to determine when certain page copies have arrived at the server.

2.3 Choosing a Collection Technique

The main approaches to storage reclamation are reference counting and tracing-based garbage collection. Reference counting was deemed inappropriate for our purposes due to its inability to collect unreachable cycles (i.e., mutually-referential garbage objects) and the overhead of maintaining reference counts in a fault tolerant manner. Tracing collectors can be divided into Mark and Sweep algorithms, which first mark all reachable objects and then reclaim the unmarked ones, and copy-based algorithms which copy live objects to new locations before reclaiming an entire storage area. Given the assumptions

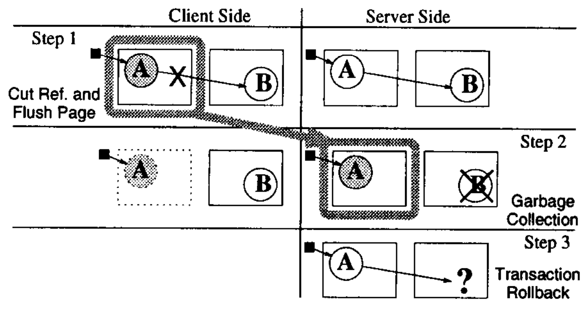

Figure 1: Transaction Rollback Problem

of the previous section, we believe that a Mark and Sweep approach can be more efficient and easier to integrate into an existing OODBMS for the following reasons:

- Copying causes objects to move between pages, which can require locks and can complicate recovery [MRV91, KW93, YNY94].
- The clustering produced by a copying-based collector may be in conflict with database requirements or user-specified hints (e.g., [BD90, GA94]). For example, generation scavenging [Ung84], tends to cluster objects by their age.
- Although copying can reduce fragmentation, *slotted pages*, which are used in many database systems, allow objects to be moved within a page, mitigating the potential fragmentation problems of sweeping.

3 Problem Scenarios

The most straightforward way to implement a Mark and Sweep garbage collector in an OODBMS would be to run it at a server as a single monolithic transaction. Such an implementation, however, would result in a synchronization bottleneck that could severely impact the ability for user transactions to run. Ideally, we would like to run the Mark and Sweep algorithm outside of the transaction mechanism; however, a traditional Mark and Sweep algorithm running without transaction protection in a client-server OODBMS could run into a number of correctness problems. We have identified three specific types of problems that can arise in a client-server OODBMS that supports the "steal" buffering policy (in which pages dirtied at clients can be sent back to a server at any time during a transaction's execution). These problems are described in the following sections.

3.1 Transaction Rollback

Allowing clients to send updated pages to the server at any time makes it possible that a garbage collector running at the server will encounter dirty (i.e., uncommitted) pages. If a page containing uncommitted updates is garbage collected, the subsequent rollback of those updates may be difficult. An example of this problem is shown in Figure 1. In this figure (and in the subsequent ones) the solid black squares represent the persistent root and objects are represented by circles. Updated objects are shaded in order to distinguish between before and after states of the update operations.

In step 1 of the figure, a transaction running at the client updates object A by removing its reference to object B and then flushes a copy of the updated page to the server. In step 2, after receiving the updated page, the server runs the garbage collector which reclaims object B, as it is unreachable. In step 3, the transaction is rolled back. If the rollback does not take garbage collection into account (as shown in the figure) then object A will contain a dangling reference to an object that no longer exists. This problem arises because transaction rollback violates a fundamental invariant of garbage collection, namely that *unreachability is a stable property of an object.* Note that this is a non-trivial problem for implementing rollback, as the page containing object B was never actually updated by the aborted transaction.

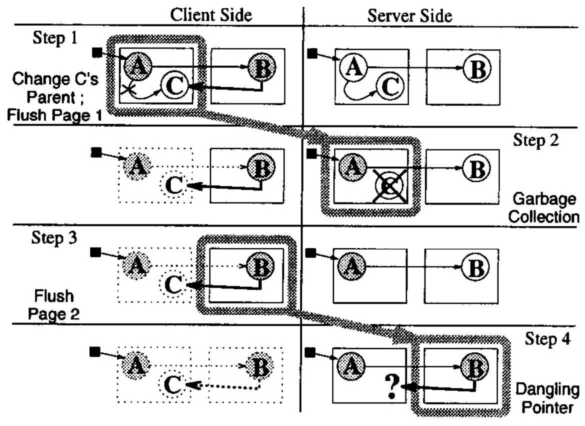

Figure 2: Multi-Page Update Problem

3.2 Partial Flush of Multi-page Updates

Similar problems can arise even in the absence of aborts, when there are updates to inter-object references that span page boundaries. The garbage collector may see an inconsistent view of such updates, as there is no guarantee that all of the relevant pages will be sent to the server prior to garbage collection. There are two specific problems that can arise: 1) partial flush of updated objects, and 2) partial flush of newly created objects. Figure 2 shows an example of the first problem. In step 1, the client changes the parent of object C from object A to object B, which resides on a different page than C. The page containing C is then sent to the server, but the page containing B (the new parent) is not. In step 2 the server runs garbage collection, and reclaims object C because it appears to be unreachable. In step 3 the page containing B is sent to the server. As shown in step 4, the database is now corrupted, as object B contains a dangling reference.

The second problem that can arise due to partial flushes involves the creation of new objects, and is shown in Figure 3. In step 1, the client creates a new object C that is referenced by object A, which resides on a different page. The page containing the new object C is then sent to the server, but the page containing the parent object A is not sent. In step 2, the garbage collector incorrectly determines that object C is unreachable, and reclaims it. This will result in a dangling reference when the page containing A is sent to the server.

It should be noted that in general, it is not possible to produce an ordering of page flushes that would avoid these problems, as there can be circular dependencies among pages due to multiple updates. Furthermore, unlike the transaction rollback problem discussed

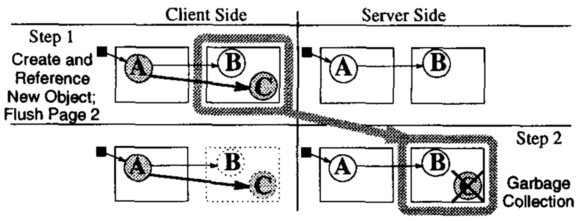

Figure 3: New Object Allocation Problem

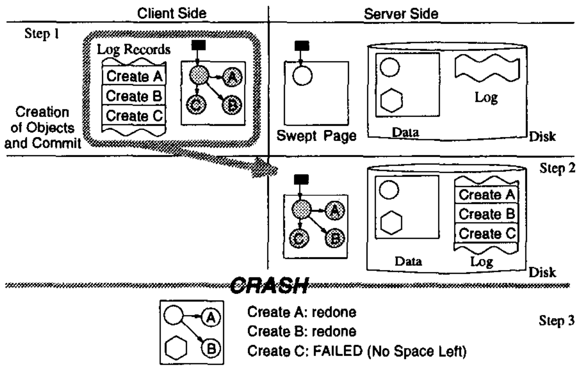

Figure 4: Redo Failure During Recovery

previously, both of these scenarios can occur even if the garbage collector reads only committed versions of pages.

3.3 Overwriting Collected Pages

A third set of problems involves the overwriting of garbage collected pages. One way that this can occur is during transaction REDO. If an operation uses space on a garbage collected page that is not reflected on stable storage at the time of a crash, then REDO could fail due to a lack of free space on the page. This problem is illustrated by Figure 4.

In step 1, the client creates 3 objects on a copy of a page that had previously been garbage collected at the server. The disk-resident copy of the page, however, does not yet reflect the collection — it still contains a garbage object that consumes space (represented by the hexagon in the figure). In step 2, the client commits, sending along the updated page and the associated log records; the log records are forced to stable storage. In step 3, the system crashes before the updated page reaches the disk and recovery is initiated. REDO fails because it attempts to re-create the three new objects on the *uncollected version* of the page, which has free space for just two of them.

A less serious instance of the overwriting problem can arise when the server sweeps a page and a client updates its (unswept) cached copy of that page. In this case, the client will overwrite the work of the collector when it sends the dirty page back to the server. Unlike the previous problem, this overwriting is a matter of efficiency rather than correctness, however, the probability of such overwriting is increased significantly by inter-transaction page caching, which allows clients to retain cached page copies over long periods of time.

4 Partitioned Mark and Sweep

We now present a partitioned Mark and Sweep algorithm that does not execute as a transaction, yet avoids the problems described in the previous section. The algorithm is initially presented for a monolithic (i.e., non-partitioned) database, and then extended to allow for independent collection of database partitions. First, however, we discuss the intuition behind the algorithm.

4.1 Enforcing Correctness

A Mark and Sweep algorithm associates a "color" with each object in the object space. In our algorithm, an object can have one of two colors: live or garbage; At the start of the collector, all object colors are initialized to be "garbage". Colors are maintained only during garbage collection and are kept in special *color maps* that are not part of the persistent object space. Mark and Sweep is a two phase algorithm. In the first (or *marking*) phase, the object graph is traversed from the root(s) of persistence. All objects encountered during this traversal are marked (i.e., colored) as live objects. In a database system (ignoring partitions for the moment), the roots of persistence are: 1) special database root objects that are entry points into the database object graph, and 2) program variables of any *active* transactions which may contain pointers into the database. Once the entire graph has been traversed, the *sweeping* phase scans the object space, reclaiming all objects that are still marked as garbage. At the end of the sweeping phase, garbage collection is complete.

Our algorithm is an incremental Mark and Sweep that has been extended to enforce the preservation of three invariants. These invariants are similar in spirit to invariants that have been proposed for traditional incremental collectors [Dij78, Bak78]. In contrast to that earlier work, however, these invariants reflect the transactional nature of database accesses. The invariants are:

Invariant I1: When a transaction cuts a reference to an object, the object is not eligible for reclamation until the first collection that is *initiated* after the completion (i.e., commit or abort) of that transaction.

Invariant I2: A new object is not eligible for reclamation until the first collection that is *initiated* after the completion of the transaction that created it.

Invariant I3: Space reclaimed by the sweeper can be reused only when the freeing of that space is reflected on stable storage (i.e., in the stable version of the database or the log).

Invariant I1, in conjunction with assumption A1 (Section 2.2), protects against the rollback problem described in Section 3.1, as it ensures that only objects made unreachable by committed transactions are eligible for reclamation. It also (in conjunction with assumption A4) avoids the partial flush problem of Section 3.2; a reference cut will be ignored by any garbage collector that runs concurrently with the cutting transaction and a collector that runs after the transaction commits will see only the after images of all pages updated by the transaction (due to A4). Thus, no collection will be done based on a partially flushed update.

In a similar manner, Invariant I2 protects against the incorrect reclamation of *newly created* objects due to partial flushes. Taken together, I1 and I2 allow the collector to run incrementally and concurrently with transactions, without having to obtain locks. I1 and I2 protect objects from the problems that could arise when the collector observes uncommitted states of the database. Furthermore, because I1 and I2 protect objects for the duration of any transaction that could cause them to be reclaimed, *the collector can ignore the program state (i.e., program variables, stacks, etc.) of any transactions that are concurrently executing at the clients.*[1]

It is important to note that I1 and I2, along with the assumptions listed in Section 2.2 ensure that any object that is considered to be garbage at the end of a garbage collection pass is truly unreachable, and can safely be collected. The key insight is that assumption A1 requires that all accesses to pointers in the database be performed in the context of strict two-phase locking, and A2 requires that all object pointers used by a transaction be obtained from the database. Observe that an object becomes garbage only when the transaction that cuts the *last* reference to the object commits. Such a transaction must have obtained a *write* lock on the last reference, so therefore, A1 ensures that *at the time that the last reference is cut, no other concurrent transaction could have accessed the reference.* A2 ensures that no *subsequent* transactions will be able to use the reference. Thus, A1 and A2 taken together ensure that the only transaction that can possibly "re-attach" an object to the object graph is the transaction that caused it to become disconnected from the graph. Assumption A4 ensures that any garbage collection pass that begins after the commit of such a transaction will see all of the pages updated by that transaction, and thus, will see the reattachment.

While I1 and I2 protect against problems that could arise during normal transaction operation and rollback, I3 protects against problems that could arise during crash recovery — it ensures that REDO reclaims space freed by sweeper before attempting to REDO any operations that may have used that space.[2] The crux of the garbage collection algorithm is the efficient maintenance of these invariants, which we detail in the following subsections.

4.2 Protecting Cut References

In order to enforce I1, we introduce a garbage collector data structure called the Pruned Reference Table (PRT), which contains entries for cut references. Each entry contains the OID of the referenced object and the Transaction ID (TransID) of the transaction that cut the reference. By considering the PRT as an additional root of persistence, the garbage collector is forced to be conservative with respect to uncommitted changes. That is, any object that has an entry in the PRT will be marked as live and will be traversed by the marker (if it isn't already marked live) so that its children objects will be marked as well. Therefore, a single PRT entry transitively protects all of the objects that are reachable from the protected object.

In order to make the necessary entries in the PRT, all updates to pointer fields in objects must be trapped. Traps of this form are typically implemented using a *write barrier* [Wil92, YNY94]. A write barrier detects when an assignment operation occurs and performs any bookkeeping that is required by the garbage collector. Recall that the garbage collector (and hence, the PRT) reside at the server while updates are performed on cached data copies at clients. Thus, a write barrier in the traditional sense would be highly inefficient. To solve this problem, we rely on the fact that clients follow the WAL protocol (Assumption A3). The WAL protocol ensures that log records for updates will arrive at the server prior to the arrival of the data pages that reflect the updates. At the server, the incoming log records are examined, and those that represent the cutting of a reference will cause a PRT entry to be made. Note that unlike previous work that exploits logs (e.g., [KW93, ONG93]), this algorithm processes log records as they arrive at the server — prior to their reaching stable storage.

When a transaction terminates (commits or aborts), its entries in the PRT are flagged. These flagged entries are removed prior to the start of the next garbage collection (i.e., the start of the next marking phase). The PRT entries can not be removed exactly at the time of transaction termination even though all dirtied pages are copied to the server on commit. This is because an on-going marker may have already scanned the relevant parts of the object graph using the previous copies of the objects. The next time the marker runs, however, it is known that it will see the effects all of the updates made by the committed transaction, so the PRT entries for that transaction can be removed. Also, if no collection is in progress when a transaction terminates, the PRT entries for that transaction can be removed immediately.

4.3 Protecting New Objects

To protect new objects, we introduce another structure called the Created Object Table (COT). The COT is used to enforce I2. As with the PRT, the COT is maintained at the server and is updated based on log records received from the clients. When a log record reflecting the creation of an object arrives at the server, an entry is made in the COT. This entry contains the OID of the new object and the TransID of the transaction that created it. In contrast to the PRT, which is used during marking, the COT is used during the sweeping phase of garbage collection.

The sweeping phase scans the object space linearly in order to maximize I/O bandwidth. It reclaims any objects that have been left colored as garbage by the marker. For each page, the sweeper checks to see if there are entries in the COT for any of the objects on that page, and if so, it does not reclaim those objects, regardless of their color. The collector does not need to traverse the objects protected by the COT because an object referenced by a newly created object can only be: 1) another new object (which must be in the COT), 2) an object that is also referenced elsewhere (which will be colored "live" by the marker), or 3) an object referenced solely by this new object. An object in this last category must have had a reference read from another object and then cut. Such an object must therefore be protected by the PRT.

In order to avoid unnecessary I/O the sweeper checks the color map and COT entries for the page before reading the page from disk. If the page has no garbage objects on it, then the sweeper simply moves on to the next page.

[1] It should be noted that I1 and I2 are conservative conditions for solving the partial flush problem; relaxing them is an interesting area of future work.

[2] The sweeping of a page never needs to be undone since I1 and I2 ensure that any reclaimed objects were disconnected from the object graph by committed transactions.

If the page has no live objects on it, then the sweeper deallocates the page — without reading it in from disk. The sweeper can be run incrementally, with the sweep of a single page being the finest granule of operation.[3] The WAL rule (assumption A3) ensures that the sweeper sees all COT entries for the pages it sweeps. Removal of COT entries is handled in the same manner as PRT entries.

4.4 Fault Tolerance

The garbage collector has a simple approach to fault tolerance: in the event of a server crash during a garbage collection, the interrupted collection cycle is simply discarded. If a system crash occurs during the marking phase, then no changes had yet been made to data pages, so there is no danger of corrupting the database and thus, no work to be done for recovery. If a system crash occurs during the sweeping phase, then some pages will have been swept and some will have not. Unswept pages can simply be ignored — they will still contain garbage objects, but these will be reclaimed by the next garbage collection. As described in Section 3.3, however, pages that have been swept could cause errors during restart REDO. This problem is avoided by enforcing I3.

Recall that I3 requires the effects of a page sweep to be reflected on stable storage before the space freed on the page is re-used. In a sense, it requires that the sweep of a page be treated as if it were a "nested top-level transaction" [GR93]. If slotted pages are being used, then one way to enforce I3 is to have the sweeper log the color map for each page that it modifies. The color map serves as a *logical* log record for the sweep. Color maps contain a single bit for every slot in the page, and therefore are quite small. Furthermore, no before image data is logged, as page sweeps never need to be undone.[4]

It is correct to restart a new garbage collector *from scratch* after recovery is complete, because at that point, the server has a transaction-consistent snapshot of the database so the old PRT and COT entries are not needed. Thus, an important side effect of this simple approach to fault tolerance is that *the PRT and COT do not need to be managed in a fault tolerant manner.* That is, during normal execution, the PRT and COT entries can be kept in memory and do not need to be logged.

4.5 Preventing Overwrite of Collected Pages

Logging protects the system from problems that can arise during transaction REDO, however, it does not protect from sweeper work being wasted due to page overwrites. This problem can be reduced in two ways. First, the sweeper can try to obtain a "no-wait" *instant* read lock on a page. Such a lock is released immediately upon being granted, and the sweeper does not block if the lock is not granted. If the lock is not granted, then some transaction has a write lock on the page. In this case, the sweeper simply skips the page, as any work that it does will only be overwritten when the transaction holding the lock commits. Secondly, the overwrite problem is exacerbated in systems that perform inter-transaction client caching. For such systems it is possible to have the sweeper exploit the cache consistency mechanism in order to reduce the potential for clients to update unswept page copies. Once such mechanism is described in Section 5.

[3] A latch must be held on the current page being swept in case a new copy of that page arrives at the server.

[4] As described in Section 5, logging can be even further reduced if media recovery is not to be supported.

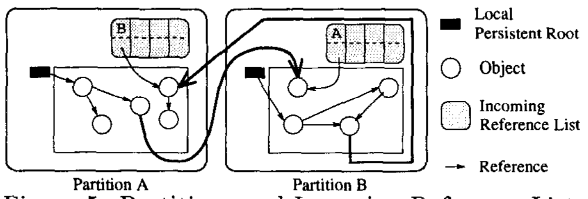

Figure 5: Partitions and Incoming Reference Lists

4.6 Collecting Partitions

In this section we briefly discuss how to extend the monolithic algorithm to allow independent garbage collection of disjoint partitions of the object space. We assume that partitions are sets of pages. The actual partitioning of the object space can be done according to physical considerations (e.g., file extents) or logical considerations (e.g., by class or relation). Partitions must be disjoint, however, objects may reference each other across partition boundaries. In order to allow for partitions to be collected independently, each partition must have an associated list of incoming references that originate from other partitions. This list is called the *Incoming-Reference List* (IRL). Conceptually, the IRL contains the OID of the (local) destination object and the ID of the partition in which the (foreign) source object resides for each such reference. The IRL of a partition serves as an additional root of persistence for the partition-local collection. Similar schemes are often used by distributed garbage collection algorithms to handle inter-node references (e.g., [SGP90, ML94]). Figure 5 shows an example of two partitions containing objects with cross-partition references. Note that the objects themselves point directly to each other and do not involve the IRL.

The IRL mechanism is transparent to programmers, and therefore, updates that may require IRL modifications must be trapped. This is handled in the same manner as the PRT and COT: the server examines incoming log records. When the creation of a cross-partition reference is detected, an entry is made in the appropriate IRL. The removal of IRL entries is performed by the garbage collector. The marking phase traverses a partition from the persistent roots (including the IRL); when it encounters a reference to an object in a different partition, it marks the corresponding entry in the remote IRL and stops traversing that path. At the end of the marking phase, any unmarked remote IRL entries originating from the currently collected partition can be removed, provided that the transaction that created the entry has committed (similar to removing PRT and COT entries).

There are two drawbacks to this approach. First, in contrast to the PRT and the COT, IRLs must be fault-tolerant. Upon recovery all IRLs must be restored to their state at the time of the crash; otherwise, some remotely referenced objects may be collected erroneously. Thus, IRLs must be maintained in database pages (rather than with in-memory structures), and updates to them must be logged. Secondly, as is well known, this type of approach can not collect cycles of garbage that are distributed across multiple partitions. Separate algorithms such as Hughes' distributed collector [Hug85] can be used to collect such cycles periodically. In general, however, the partitioning of the object space should be done in a way that minimizes the number of cross-partition references in order to limit the number of such cycles and to keep the IRLs small.

5 Implementing the Garbage Collector

As stated in Section 1, the design goals for garbage collection include: 1) it should impose minimal overhead on client transactions, 2) it should be efficient and effective in collecting garbage, and 3) it should be relatively straightforward to integrate the collector in existing client-server database systems. In order to assess the algorithm in light of these requirements, we have implemented a single-partition version of it in the client-server version of the EXODUS storage manager [Fra92, Exod93].

5.1 The EXODUS Storage Manager

Our initial implementation is based on the EXODUS storage manager v3.1 [Exod93]. EXODUS is a client-server, multi-user system which runs on many Unix platforms. It has been shown to have performance that is competitive with existing commercial OODBMS [CDN93]. EXODUS supports the transactional management of ***untyped*** objects of variable length, and has full support for indexing, concurrency control, recovery, multiple clients and multiple servers. Data is locked using a strict two-phase locking protocol at the page or coarser granularity. Recovery is provided by an ARIES-based [Moh92] WAL protocol [Fra92]. The EXODUS server is multi-threaded — every user request is assigned a thread when it arrives at the server; the server also has its own threads for log management, etc. The server supports asynchronous I/O (using multiple I/O processes) so some threads can run while other threads are waiting for I/O. EXODUS extends a traditional slotted page structure to support objects of arbitrary length [Car86]. "Small" data items (those that are smaller than a page) and the headers of larger ones are stored on slotted pages. Internally, objects are identified using physical OIDs that allow pages to be reorganized without changing the identifiers of objects.

EXODUS is a page server; updates are made by clients to their local copies and the resulting log records are grouped into pages and sent asynchronously to the server responsible for the updated pages. Dirty pages can be sent back to the server at any time and in any order during the execution of a transaction; a WAL protocol ensures that all necessary log records arrive at the server before the relevant data page. At commit time, copies of any remaining dirty pages are sent to the server. The client retains the contents of its cache across transaction boundaries, but no locks are held on those pages. Cache consistency is maintained using a check-on-access policy (based on "Caching 2PL" [Car91]).

For recovery purposes all pages are tagged with a Log Sequence Number (LSN) which serves as a timestamp for the page. The server keeps a small list of the current LSNs for pages that have been recently requested by clients. When a client requests a lock from the server, the server checks this table, and if it can not determine that the client has an up-to-date copy of the page, it sends a copy of the page along with the lock grant message.

To summarize, the EXODUS storage manager supports the fundamental assumptions on which the garbage collector depends (see Section 2.2), and also has desirable properties such as slotted pages. In addition, the EXODUS server uses non-preemptive threads and asynchronous I/O, which simplify the implementation of the garbage collector. However, the system also provides features that present challenges for garbage collection, such as client caching, a steal policy between clients and servers, asynchronous interactions between clients and servers, a streamlined recovery system, and optimizations to avoid logging in certain cases.

5.2 Implementation Overview

The implementation of the garbage collector in EXODUS is currently a proof-of-concept implementation. It is well integrated with concurrency control and recovery and has been heavily tested; including its fault tolerant aspects. However, there are some limitations of the current implementation. First, as stated above, the framework is in place to support cross-partition references (we currently use an EXODUS "volume" as a partition) and do much of the checking that is needed to manage IRLs, but the IRL scheme is not yet fully implemented. Second, it collects only small-format objects. The extension to large-format objects, is straightforward but was not necessary for our purposes. Third, because the Exodus storage manager does not know the types of the objects that it stores, we store a bitmap in the initial bytes of the data portion of each object. The bitmap indicates which OID-sized ranges of bytes in the object contain actual OIDs and is used by the marker during its traversal. These bitmaps are created automatically when objects are allocated using a C++ constructor. Finally, as discussed in Section 6, the scheduling of the garbage collector with respect to other EXODUS thread activity is not fully tuned to balance collection and transaction processing.

The modifications that were made to EXODUS can be classified into two categories: 1) garbage collection-oriented bookkeeping during normal processing, and 2) the garbage collection algorithm itself. During normal processing, the server scans incoming log records to determine if the logged updates require any entries to be made in the garbage collector data structures.

In order to add garbage collection to the EXODUS server, we created a new type of server thread called the `gcThread`. When a collection starts on a partition, a new `gcThread` is spawned. The `gcThread` initializes the garbage collector data structures, runs the marker phase and then runs the sweeper phase. At the end of the sweeper phase, the `gcThread` terminates. At present, the scheduling of the `gcThread` works as follows: when the `gcThread` gets the processor, it starts a timer (currently set at 50 msec). If the timer expires during the marking phase, then the marker finishes examining the current object and then gives up the processor. If it expires during the sweeping phase, then it finishes sweeping the current page and then gives up the processor. The `gcThread` is woken up (after a specified sleep time) when there are no client requests waiting for the processor.

The implementation required approximately 4000 lines of new or modified code on the server-side; the bulk of which was for the `gcThread` itself. The client-side required only 200-300 lines of new or modified code. The algorithm was implemented with only minor changes to the description in Section 4. Three implementation issues, however, deserve mention. First, we exploit the cache consistency mechanism to reduce the potential for clients to overwrite the work of the sweeper. The sweeper updates the sequence number (LSN) on each page that it modifies. Any transaction that subsequently tries to lock such a page will then receive the swept copy even if it already has a cached copy of the page. To avoid recovery problems, however, the sequence number placed on the page by the sweeper must be guaranteed to be lower than the sequence number that will be placed on the page by any subsequent client update.

The second issue arises because EXODUS has no general support for allowing non-recovery-related server threads to create log records. Although there are workarounds, we chose to avoid logging completely in the `gcThread`. We accomplish this by setting a flag in a page's header when the page is swept; the flag is cleared when the page is read from disk. If a client obtains a page that has the flag set, then it logs the slot allocation information (from the page header) prior to performing any updates on a page. In this way, log records are only generated for those swept pages that had not been written to disk, prior to being obtained by a client. Note that the log record is generated only by the first transaction to update such a swept page.

The third issue results from an optimization that EXODUS uses to reduce logging during bulk loading and other create-intensive operations. When a new page is allocated to hold new objects, the individual object creations are not logged; rather, the entire page is logged when it is copied back to the server. This optimization, while providing better performance for EXODUS, deprives the garbage collector of the information on individual object creations. As a result, in the EXODUS implementation of the collector, we enter page IDs in the COT rather than individual OIDs. When the sweeper encounters a page that has an entry in the COT, it simply skips it. Furthermore, when a newly allocated page arrives at the server, the server scans all of the objects on the page to determine if any new IRL entries are required. At present, the detection of cross-partition references is implemented, while the creation of IRL entries is not.

6 Performance Measurements

In this section we describe an initial study of three different performance aspects of the implementation: 1) the overhead added to normal client processing, 2) the stand-alone performance of the collector, and 3) the performance of the collector and client transactions when running concurrently.

For all of the experiments presented here, the EXODUS server was run on a SPARCstation LX with 32MB of main memory. The log and the database were stored on separate disks, and raw partitions were used in order to avoid operating system buffering. The size for both data and log pages was set to 8KB. All times were obtained using *gettimeofday()* and *getrusage()*.

The experiments are run on one or more synthetic database partitions consisting of simple linked-lists of objects. Each object is 80 bytes long, and along with EXODUS page and object headers, 84 objects can fit on a page. The objects are allocated in contiguous pages in an EXODUS file. The pages are fully packed with objects, however, we vary the percentage of garbage objects *in each page* as an experimental parameter. We also vary the "clustering factor" of objects in pages. This factor determines the number of live objects on a page that the marker can scan before crossing a page boundary. For example, with "1/2 Clustering", half of the live objects in a page can be traversed before a page boundary is crossed.

6.1 Bookkeeping Overhead

The first experiment measures the overhead that is incurred during normal transaction operation *with no garbage collection running.* The overhead in this case is due to extra work required to maintain the garbage collector data structures (e.g., the PRT and COT). This includes the extra log-related work that clients must perform and the processing of log records at the server. In this experiment, a single client process was run on a SPARCstation IPC with 32MB of memory; it was connected to the server over an Ethernet.

Four different operations were tested: 1) object allocation, 2) modification of references in existing objects, 3) modification of non-reference data in existing objects, and 4) read-only access to objects. For each test, the operation was performed on every object in the partition before committing. In the tests shown here, the data partition contained 100,000 objects (1190 pages) and was fully clustered. Client and server cache sizes were 1500 pages — more than enough to hold all of the accessed pages, so all clusterings would have similar performance here. Each benchmark was run 10 times and the results averaged. For each experiment, we report times for both a cold and a hot server cache (except for allocate, which creates all of the pages it accesses).

The results are shown in Table 1. For each test, times are presented for the both unmodified and modified (with GC code) EXODUS systems. In addition, results are shown for cases with and without committing the transactions. With garbage collection, transaction commit incurs the extra overhead of flagging PRT and/or COT entries. As can be seen in the table, the overhead imposed on normal operations by the garbage collection code is quite small in all of the cases tested. The highest overheads were seen for the allocation of new objects; this is due to the full-page logging for newly allocated pages in EXODUS. For this reason, the server must scan the entire new page in order to locate any cross-partition pointers; individual object creations on already existing pages would not incur this cost.

6.2 Off-Line Garbage Collector Performance

The second experiment examines the cost of the garbage collector when it is run without any concurrent user transactions. In this case, we varied the clustering, size, and % garbage of the partition. All experiments were run with a server cache of 1000 pages. The partition size was varied from 500 to 10,000 pages.

Figure 6 shows the performance of the *marking* phase of the collector with a fully clustered partition for various percentages of garbage. In this case the marker performance scales linearly with the partition size for all % garbage values (except for 100% which remains near 0, as there is nothing for the marker to do). Response time improves as the garbage % is increased because the marker traverses only live objects, and there are fewer of these.

The performance of the *sweeping* phase of the collector

	Cold Server Cache			*Hot Server Cache*		
Operation	**Original**	**w/GC**	**Overhead**	**Original**	**w/GC**	**Overhead**
Allocate	38176	40382	5.8%			
w/Commit	54989	59049	7.4%			
Update Ref	52543	53165	1.1%	34367	34920	1.6%
w/Commit	62736	63381	1.0%	44607	45160	1.2%
Update Value	51091	51616	1.0%	33104	33438	1.0%
w/Commit	61365	61998	1.0%	43327	43671	0.7%
Read-only	27586	27792	0.7%	13403	13565	1.2%
w/Commit	27622	27828	0.7%	13439	13601	1.2%

Table 1: Client Slowdown (msec), Cold and Hot Server Cache

for the corresponding cases is shown in Figure 7. First, notice that all % garbage values have similar sweeper performance except for 0% and 100%. If the color map shows that page has 0% garbage, then the sweeper does not bother to fetch the page from disk. Likewise, if a page contains only garbage, the sweeper can free the page without fetching it from disk. The difference in performance here, comes from the overhead for freeing a page in EXODUS. In all other cases, the sweeper performance is virtually independent of the percentage of garbage in a page, as it must fetch each page that needs to be swept. When the partition is smaller than 1000 pages, all of the pages that the sweeper would need to fetch are already in the buffer because of the marker. Once the partition exceeds the cache size, then this pre-fetching effect is completely lost (in this case), as the layout of the objects in the partition cause the marker and sweeper to scan the partition in the same order.

Figure 8 shows how the performance of the marking phase varies for different clustering factors. When the partition is small enough to fit in memory (1000 pages, here) then marking performance is not affected by clustering. However, once the partition exceeds the size of the buffer, then the marker begins to incur I/O due to inter-page references. As can be seen in the figure, full clustering is the best case for the marking phase, as it allows marking to process pages sequentially, minimizing I/O. As the clustering factor is reduced, the I/O requirements of the marker increase. This effect is to be expected and raises the issue of the garbage collector's impact on concurrent user transactions. This issue is addressed in the following section.

6.3 On-Line Performance

The third set of experiments examines the performance when client transactions and the collector execute concurrently. Although garbage collection can improve performance in the long run by reducing the amount of wasted space in the database, performance can suffer while the collector is running due to synchronization with transactions and the load it places on the server. Given that our collector performs no synchronization with client transactions, and (as shown in Section 6.1) imposes small off-line costs on clients, the main impact that it will likely have on client responsiveness is the load placed on the server when the collector is running.

Because the collector is fully incremental, the negative impact on client performance can be traded off against the execution time of the collector by varying the aggressiveness with which the collector is scheduled at the server. Favoring transactions at the server will reduce the slowdown experienced by clients when the collector is running. However, this slowdown will be incurred over a longer time period, as the collector will take longer to complete its job. Furthermore, slowing down the collector can hurt performance by impacting its ability to quickly free up wasted space.

In order to find a balance between transaction and garbage collector execution, we experimented with the scheduling of the **gcThread** at the server. The results of the off-line collector experiments (described in the previous section) led us to concentrate on the costs of the collector in an I/O-bound setting. In this experiment, the transaction workload was generated by five client processes, each running on a separate SPARCstation 1+ with 32MB of memory. Each client repeatedly ran transactions consisting of a read-only, full traversal on a private partition. In addition, we continuously ran the garbage collector on a sixth partition. We used partitions consisting of 1200 pages (10MB) each, with full clustering and 5% garbage.[5] The client caches were only 200 pages and the server buffer was set to a total of 1200 pages. Given the sequential nature of the transactions and the garbage collector in this workload, the hit rate at both client and server buffers was effectively zero.

Section 5.2 described the integration of the **gcThread** with the EXODUS scheduler. In order to vary the scheduling of the collector, we adjusted the value of the sleep time (i.e., minimum delay) that the scheduler imposes on the garbage collector before allocating it a new processor time slice. This delay is imposed on the collector whenever there are client requests waiting for service. Multiple client requests are allowed to run before the **gcThread** is given the processor.

In this experiment we varied the collector sleep time from 0 msec to 1 second. Figure 9 shows the results of this experiment for the range from 0 msec to 100 msec. The lowest line shows the performance of the collector running alone, with a 200 page server buffer (i.e., the same amount as is allocated when running concurrently with the transactions). The flat dotted line shows the average response time for a client when the five clients are run without the garbage collector. The other two lines show the performance of the transactions and the collector when they are executed concurrently. As would be expected, when the collector sleep time is small, its response time is low, and the transactions suffer. As the sleep time is increased, this relationship reverses. What is striking in this figure is that at a sleep time of about 20msec, the transactions and the collector (which have similar

[5] To keep the garbage percentage fixed at 5% in this experiment, the sweeper does not actually reclaim the garbage objects, however it still performs all checking and marks the pages dirty.

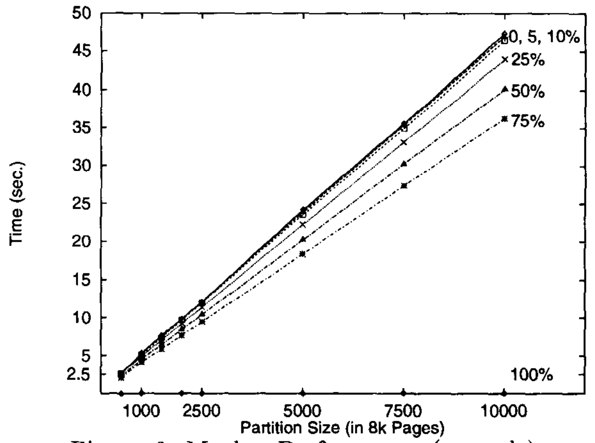

Figure 6: Marker Performance (seconds)
(Full Clustering, Varying % Garbage)

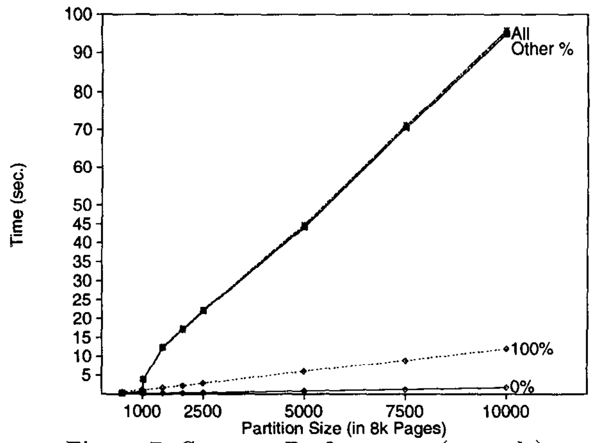

Figure 7: Sweeper Performance (seconds)
(Full Clustering, Varying % Garbage)

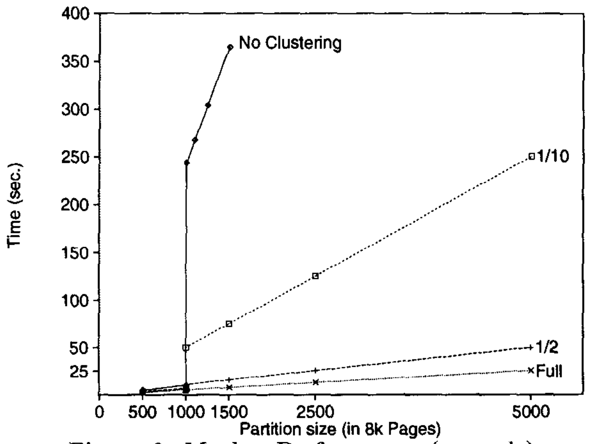

Figure 8: Marker Performance (seconds)
(Varying Clustering Factor, 0% Garbage)

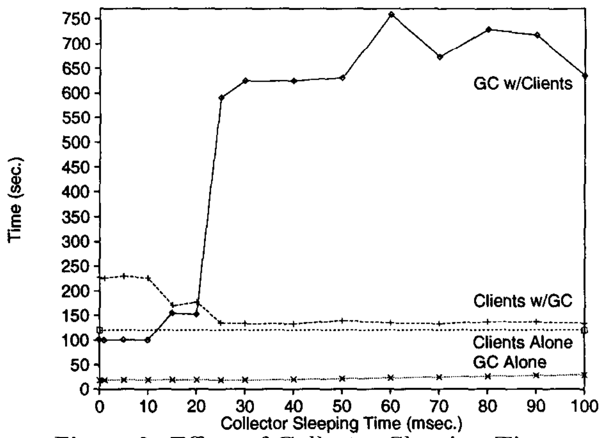

Figure 9: Effect of Collector Sleeping Time
(Full Clustering, 5% Garbage, Multiple Partitions)

I/O patterns in this experiment) are roughly balanced. Beyond this point however, the transactions quickly approach their minimal response times, while the collector jumps to a response time of between 600 and 760 seconds.[6] At present we are still investigating the interactions with the EXODUS thread scheduler that cause the sudden jump in collector response time. However, this experiment (and others) showed that a sleep time of 20 msec provides a reasonable balance between collector and transaction scheduling. Therefore, we used that value in our subsequent experiments.

The final set of experiments that we describe here, explores the interaction of the collector and transactions using several different databases in both CPU-intensive and I/O-intensive settings. The transactional load in these experiments is the same as that used in the previous experiment. In order to run in a CPU-intensive mode, however, we ran all five clients and the collector on the same, server memory-resident partition. These results and the results for the I/O-bound case (all running on separate partitions, as before) are shown in Table 2.

The Single Partition numbers in the table show that in the CPU-intensive case, the collector overhead remains around 10% or lower. The Multiple Partition numbers show that as expected, the collector overhead is higher in an I/O-bound system, ranging from 12% to nearly 48%.

This set of tests represents a study of the worst case performance of the garbage collector for several reasons. First, the garbage collector is run continuously — after completing the collection of a partition, it immediately begins a new collection. In practice, the garbage collector would be run only periodically, and, as discussed in Section 6.1, only a small overhead is imposed on client transactions when the collector is not running. Secondly, in these tests, the client transactions read the same number of pages regardless of the amount of garbage in the pages; thus, only the performance costs, but not the benefits of garbage collection (i.e., reduced I/O) are shown here. Finally, the tests present a constant, heavy load to the server, which is an undesirable (but sometimes unavoidable) condition under which to run garbage collection. For example, in the multiple-partition case, the server is completely I/O-bound.

For these reasons, we believe that the overhead imposed upon clients while the collector is running is reasonable. It is important to stress that unlike garbage collectors that require synchronization with client transactions, in this collector the bulk of the overhead is due to the I/O requirements of detecting and reclaiming garbage objects. Such overhead would be incurred by *any* tracing-based garbage collector, and can be adjusted due to the

[6]The collector performance continues to oscillate within this range for the cases we tested (up to a 1 second sleep time).

	Single Partition				Multiple Partitions			
Experiment	Alone	w/GC	Overhead	Collector	Alone	w/GC	Overhead	Collector
0% Garb., Full Clust.	48.19	52.59	9.13%	29.27	126.96	167.74	32.12%	75.48
5% Garb., Full Clust.	47.79	52.90	10.69%	30.99	120.77	178.09	47.46%	152.58
0% Garb., 1/2 Clust.	99.11	105.65	6.59%	34.00	255.29	286.28	12.13%	316.82
5% Garb., 1/2 Clust.	97.23	104.34	7.31%	30.90	250.43	299.26	19.49%	461.54

Table 2: Client and Collector Performance (Seconds)

incremental nature of the collector.

7 Related Work

As stated in the introduction, garbage collection has been intensively studied in the context of traditional programming languages [Wil92]. The invariants that an incremental collector must respect were first proposed by [Dij78, Bak78]. Our work addresses the efficient implementation of similar invariants in a (transactional) client-server DBMS context. The earliest study of garbage collection for object-oriented databases was done by Butler [But87]. This work simulated the behavior of several kinds of collectors running against a centralized OODBMS, but did not consider interactions with concurrency control, recovery, and caching mechanisms. More recent work has investigated fault-tolerant garbage collection techniques for transactional persistent systems in centralized [KW93, ONG93] and distributed [MRV91, MS91] architectures. This work addresses fault tolerance but does not consider dynamic page replication and caching as arises in a client-server environment. Heuristics for selecting which partition is the most cost-effective for a partition-based algorithm to collect have been studied in [CWZ94]. This work, however, did not address the details of collection algorithm itself.

A reference counting collection scheme for MIT's Thor system is described in [ML94]. Thor is a distributed OODBMS which uses optimistic concurrency control to regulate accesses to objects. This paper focuses on distributed collection across servers in a client-server environment rather than on collection that is local to a server. Each time a client fetches an object from a server, the server records the OID of the object and all the OIDs that are referenced by the fetched object in a local table. Server tables are cleaned as a side effect of local collections. These tables can be viewed as a before image log, which avoids the reclamation of pruned objects prior to transaction commit. The algorithm uses a "no-steal" policy so that modified objects are not sent to the servers prior to commit. This policy avoids the problems due to partial flushes of updates (Section 3.2) at the expense of reduced flexibility in client cache management. [ML94] describes the algorithm but does not discuss an implementation and provides no performance analysis.

The work that is most relevant to our algorithm is [YNY94]. This paper examines the performance of several reclamation algorithms for client-server persistent object stores. Some of the results are obtained from an implementation of an incremental partitioned Mark and Sweep algorithm, although very few details of this algorithm or its implementation are given. Most of the results are obtained using a simulation of several algorithms.

Similarly to our algorithm, their partitioned Mark and Sweep collector runs at the server and can execute concurrently with client transactions. However, their algorithm differs in that it uses a special *write barrier* and that the collector obtains (non-two phase) locks on data items. The write barrier traps updates at the clients and adds any new object references to a local list. This list is shipped to the server when a client commits or when the client receives a callback message from the server. The server requests these lists and the contents of application process stacks from the clients before the collector enters its sweep phase. The lists and other references are used during the marking phase as additional roots of persistence. In terms of locks, the marking phase obtains and holds a read lock on a page while it is accessing the page. These locks cause the marker to synchronize with the transactions. In contrast, our partitioned Mark and Sweep algorithm does not hold any locks on pages, does not send callbacks to clients, and can ignore the program state of on-going transactions. Measurements of the implementation showed that the write barrier has only minimal impact on client performance; our measurements support this result.

Several other algorithms are examined in [YNY94], including a partitioned copy-based collection algorithm. This algorithm obtains non-two-phase exclusive transactional locks for moving objects and uses callbacks, it also requires the use of logical OIDs. Based on the results of the simulation studies, the copy-based algorithm is advocated over partitioned Mark and Sweep due to its ability to recluster the database. In contrast, we have chosen to allow clustering to be treated separately by the system in order to gain the efficiency and relative ease of implementation of Mark and Sweep.

8 Conclusions

OODBMS features such as client caching, "steal" management of client buffers, transactions, and fault tolerance raise three major problems for the development of an efficient garbage collector for client-server OODBMS environments. These problems are tied to the rollback of transactions, the partial flushing of multi-page updates, and the potential for overwriting of garbage collected pages due to recovery and/or client caching.

We described a garbage collection algorithm based on a partitioned Mark and Sweep approach. By exploiting the flow of log records between clients and server, we are able to enforce the correctness of our algorithm. The collector is incremental, but requires very little synchronization with client transactions (e.g., it holds no locks), performs minimal logging, and requires no client callbacks or special hardware. The algorithm has been implemented in EXODUS, and integrated with the concurrency control and recovery of that system. Furthermore, the garbage collector affected fewer than 300 lines of code in the client side of the system, and required no changes to the complex protocols already in place, such as those for recovery, concurrency, caching, or clustering.

An initial study showed that the collector bookkeeping mechanisms added little overhead to client operations, and that the performance impact of the collector is primarily due to the I/O load for detecting and reclaiming garbage, which is unavoidable in any tracing collector. The study also raised scheduling issues that must be addressed in order to moderate the impact of garbage collection I/O activity on overall system performance.

In terms of the current implementation, remaining tasks include implementing the mechanisms for handling cross-partition references, and more closely investigating the interaction between the `gcThread` and EXODUS thread scheduling to balance garbage collection and client transaction processing. In terms of future work, we plan to investigate the relaxation of some of the assumptions and invariants on which the collector is based in order to support a wider range of existing OODBMS. We also plan to use the implementation to investigate data partitioning, partition selection, and garbage collector scheduling strategies, and to more closely explore alternative collection approaches such as scavenging.

Acknowledgements

The authors thank Mike Zwilling for providing invaluable assistance and information about EXODUS on many occasions and Patrick Valduriez for providing us the opportunity to collaborate on this work.

References

[Bak78] H. Baker. List Processing in Real Time on a Serial Computer. *CACM*, 21(4), Apr. 1978.

[BD90] V. Benzaken and C. Delobel. Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2. In *4th Int. Workshop on Persistent Object Systems*, Martha-Vineyard, MA, Sep. 1990.

[BDK91] F. Bancilhon, C. Delobel, and P. Kannellakis. *Building an Object-Oriented Database : the O_2 story.* Morgan Kaufmann, 1991.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *CACM*, 34(10), Oct. 1991.

[But87] M. Butler. Storage Reclamation in Object Oriented Database Systems. In *SIGMOD Conf.*, San Francisco, CA, May 1987.

[Cat94] R. Cattell. *The ODMG Object Database Standard, Rel 1.1.* Morgan-Kaufman, San Mateo, CA, 1994.

[CDN93] M. Carey, D. DeWitt, J. Naughton. The OO7 Benchmark. In *SIGMOD Conf.*, Washington D.C., May 1993.

[Car86] M. Carey, et al. Object and File Management in the EXODUS Extensible Database System. In *VLDB Conf.*, Kyoto, Japan, 1986.

[Car91] M. Carey, et al. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *SIGMOD Conf.*, Denver, CO, Jun. 1991.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *SIGMOD Conf.*, Minneapolis, MN, May 1994.

[CWZ94] J. Cook, A. Wolf, and B. Zorn. Partition Selection Policies in Object Database Garbage Collection. In *SIGMOD Conf.*, Mineapolis, MN, May 1994.

[DeW90] D. DeWitt, et al. A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. In *VLDB Conf.*, Brisbane, Aug 1990.

[Dij78] E. Dijkstra, et al. On-the-Fly Garbage Collection: An Exercise in Cooperation. *CACM*, 21(11), Nov. 1978.

[Exod93] EXODUS Project Group. EXODUS Storage Manager Architectural Overview, 1993.

[FCL93] M. Franklin, M. Carey, and M. Livny. Local Disk Caching in Client-Server Database Systems. In *VLDB Conf.*, Dublin, Ireland, Aug. 1993.

[FCW89] M. Franklin, G. Copeland, and G. Weikum. What's Different About Garbage Collection For Persistent Programming Languages? Tech. Rep. ACA-ST-062-89, MCC, Austin, TX, Feb. 1989.

[Fra92] M. Franklin, et al. Crash Recovery in Client-Server EXODUS. In *SIGMOD Conf.*, San Diego, CA, Jun. 1992.

[GA94] O. Gruber and L. Amsaleg. *Object Grouping in Eos,* pages 117–131. In [ODV94], May 1994.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufman, 1993.

[Hug85] J. Hughes. A Distributed Garbage Collection Algorithm. In *Functional Languages and Computer Architectures*, LNCS 201, Springer-Verlag, Sep. 1985.

[KW93] E. Kolodner and W. Weihl. Atomic Incremental Garbage Collection and Recovery for Large Stable Heap. In *SIGMOD Conf.*, Washington D.C., Jun. 1993.

[Moh92] C. Mohan, et al. ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1), Mar 1992.

[ML94] U. Maheshwari and B. Liskov. Fault-Tolerant Distributed Garbage Collection in a Client-Server Object-Oriented Database. In *PDIS Conf.*, Austin, TX, Sep. 1994.

[MN94] C. Mohan and I. Narang. ARIES/CSA: A method for Database Recovery in Client-Server Architectures. In *SIGMOD Conf.*, Minneapolis, MN, May 1994.

[MRV91] L. Mancini, V. Rotella, and S. Venosa. Copying Garbage Collection for Distributed Object Stores. In *SRDS Conf.*, Pisa, Italy, Sep. 1991.

[MS91] L. Mancini and S. Shrinivastava. Fault-tolerant Reference Counting for Garbage Collection in Distributed Systems. *Computer Journal*, 34(6), Dec. 1991.

[ODV94] T. Özsu, U. Dayal, and P. Valduriez. *Distributed Object Management.* Morgan-Kaufman, 1994.

[ONG93] J. O'Toole, S. Nettles, and D. Gifford. Concurrent Compacting Garbage Collection of a Persistent Heap. In *SOSP Conf.*, Asheville, NC, Dec. 1993. ACM Press.

[SGP90] M. Shapiro, O. Gruber, and D. Plainfossé. A Garbage Detection Protocol for a Realistic Distributed Object-support System. Tech. Rep. INRIA-1320, INRIA, Rocquencourt, Nov. 1990.

[Ung84] D. Ungar. Generation Scavenging: A Non-disruptive High-Performance Storage Reclamation Algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Env.*, Apr. 1984.

[Wil92] P. Wilson. Uniprocessor Garbage Collection Techniques. In *Int. Workshop on Memory Management*, volume 637 of *LNCS*, St. Malo, France, Sep. 1992. Springer-Verlag.

[YNY94] V. Yong, J. Naughton, and J. Yu. Storage Reclamation and Reorganization in Client-Server Persistent Object Stores. In *Data Engineering Conf.*, Houston, TX, Feb. 1994.

[ZM90] S. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, 1990.