Storage Reclamation and Reorganization in Client-Server Persistent Object Stores *

Voon-Fee Yong †　　Jeffrey F. Naughton　　Jie-Bing Yu

Department of Computer Sciences
University of Wisconsin, Madison, WI 53706
{ yong, naughton, jiebing@cs.wisc.edu}

Abstract

In this paper we develop and evaluate a number of storage reclamation algorithms for client-server persistent object stores. Experience with a detailed simulation and a prototype implementation in the Exodus storage manager shows that one of our proposed algorithms, the *Incremental Partitioned Collector*, is complete, maintains transaction semantics, and can be run incrementally and concurrently with client applications. Furthermore, it can significantly improve subsequent system performance by reclustering data, rendering it attractive even for systems that choose not to support automatic storage reclamation.

1 Introduction

Experience with object-oriented programming languages has demonstrated that explicit storage management by programmers is a difficult and error prone task — anyone who has spent time trying to find a storage leak in a non-trivial C++ program can attest to that fact. Fortunately, in C++ programs the effect of a storage leak is limited to individual runs of programs, since all garbage is implicitly collected when the program terminates. The problem becomes more serious in persistent object stores, and the client-server model adds new dimensions to the problem. These new aspects include: (1) the objects are disk-resident rather than main memory-resident; (2) persistent object stores, being the core of OODBMS, have to maintain transaction semantics and recoverability; (3) the client-server paradigm introduces parallelism (concurrent running of multiple clients) and thus creates more opportunities for inconsistencies. These problems render traditional solutions to garbage collection either inapplicable or hopelessly inefficient [11].

*This work was supported by NSF grant IRI-9157357

†Current address: Tandem Computers Incorporated, 10100 N. Tantau Ave. LOC 251-04, Cupertino, CA 95014. E-mail: YONG_VOONFEE@tandem.com

In this paper we develop and evaluate a number of algorithms for storage reclamation (garbage collection) in client-server persistent object stores. These are based upon reference counting, mark-and-sweep, and copying collectors, all adapted to applications in a client-server environment.

We evaluate these algorithms by both detailed simulation of a client-server OODBMS and implementation in the Exodus storage manager [5] (Exodus SM). The results of these experiments will be described in Section 6. Briefly, the results show one of our algorithms, *Incremental Partitioned Copying*, has the following desirable properties: (1) It reclaims all garbage. (2) It maintains transaction semantics and recoverability. (3) It is incremental and non-disruptive. (4) It reclusters and compacts the database as it collects. The performance gain we observed due to this reclustering suggests that this *Incremental Partitioned Collector* can be of great benefit even for systems that do explicit storage management.

We define garbage by reachability: anything that is not reachable from persistent roots is garbage. One could argue that there is no garbage in persistent object stores, since anything stored there must by definition persist; in our experience, this argument is most often made by those with little experience with persistent programming languages. One proof of the existence of garbage in persistent object stores is that many OODBMS vendors include garbage collection in their products.

The rest of this paper is organized as follows: Section 2 describes some related work, while Section 3 introduces our system model. The algorithms we studied are described in Section 4; Section 5 gives our experiment configurations, and Section 6 gives experimental results. We conclude in Section 7.

2 Related Work

Most of the work in the area of storage reclamation has been focused on volatile memory systems [1, 2, 3, 7, 9, 13, 16, 19, 20, 21, 24]. Comparatively little attention has been paid for implementing automatic storage reclamation for large persistent object storage systems, and none that we are aware of considers the client-server model.

The most directly relevant work is by Butler [4], who showed that copying collectors are necessary for achieving better performance for future traversal of the database. Our simulation confirmed this. However, the mark and sweep algorithms considered in that apper are not incremental, and none of the algorithms deals with recovery or concurrency control.

Zorn [25] proposed the use of a deferred sweep algorithm to make the sweep phase of the mark and sweep algorithm incremental, very similar to our incremental mark and sweep algorithm described in Section 4.2.2. We also propose a hybrid approach between generational copying and mark and sweep algorithm.

Kolodner's work [14] describes a stop-the-world *Semispaces* [2] copying collector. In addition to being disruptive, it requires the *force/no steal* buffer management policy for recovery. In [15], an incremental fault tolerant scheme based on Appel, Ellis and Li's copying algorithm was presented. This incremental algorithm has some properties similar to our adaptation of Baker's incremental copying algorithm. However, it deals with uniprocessor systems rather than client-server systems, and it requires the use of a complicated and expensive (in terms of *read barriers* and extra logging traffic) recovery mechanism. Our copying algorithms do not require the use of *read barriers* and extra log records, since a consistent view of the database is achieved through the use of standard database recovery mechanisms.

3 System Model

In this section we give a brief description of our client-server persistent object store model for which the storage reclamation algorithms were designed. Figure 1 shows a general architecture for a client-server persistent object store. This is very similar to the client-server Exodus SM. Persistent objects reside in secondary storage at the server, which has several data and log disks. The client-server connection uses pages as the unit of data transfer. The server maintains enough type information in its system catalogs to be able to identify the pointers in the objects stored in the system. The client and the server each maintain separate buffer pools. We assume a simple two-phase locking mechanism with page-level locking for concurrency control, and *write ahead logging* for crash recovery.

The server is multi-threaded, with a separate collector thread activated to perform garbage collection. Client applications may cache object identifiers in their local variables across transactions. A *call back* mechanism is assumed to allow the collector thread to communicate with the clients for verification of potential garbage during collection.

4 Persistent Storage Reclamation Algorithms

The algorithms we studied fall into three broad categories: Reference Counting, Mark and Sweep, and Copying algorithms. We describe each in a subsection below.

4.1 Reference Counting

We studied two variations of the reference counting algorithms: Server-Oriented Reference Counting, and Client-Optimized Reference Counting. Both are based upon the deferred reference counting algorithm of Deutsch and Bobrow [7]. We chose this algorithm as a basis because it reduces the frequency of updating reference counts, it can be easily adapted to accommodate the concept of transactions, and, unlike naive reference counting schemes, it does not store reference counts with objects. This is especially important in a client-server environment because it minimizes both the communication and IO cost of updating reference counts at server.

The two algorithms make use of three tables of object identifiers: a Zero Count Table (ZCT), a Multi Reference Table (MRT) and a Variable Reference Table(VRT). The ZCT contains OIDs for objects with zero reference counts (potential garbage.) They are not confirmed inaccessible since references to these objects may still exist in the VRT, which contains the OIDs referenced by local variables and parameters of active methods. The MRT contains OIDs for objects with reference counts greater than or equal to two.

In our variant of the algorithm, each client workstation contains a local VRT, while the ZCT and MRT are stored at the server. By doing this, we can prevent multiple clients from duplicating effort, avoid the message cost for keeping consistent copies of ZCT and MRT on clients in synch with server, and eliminate client-to-client communication during the verification process. The three tables are implemented as hash tables with OIDs as hash keys.

The reference counting algorithm has two phases: the continuous counting phase and the periodic collection phase. During the counting phase, the ZCT and MRT are updated using a *garbage collection record list (GCRL)*. The *GCRLs* are built at the clients, and sent to the server at transaction commit time. All opera-

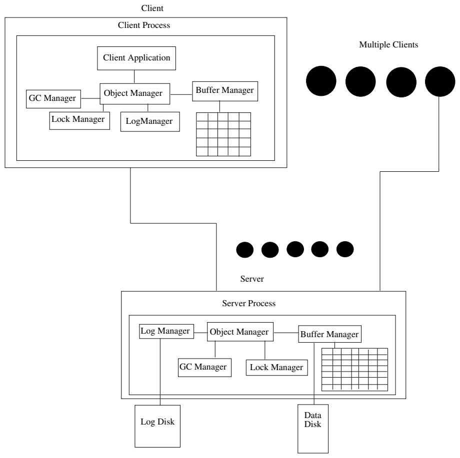

Figure 1: Client-Server Object Store Simulator

tions that will affect reference counts are recorded in the *GCRL*. To accomplish this, software *write barriers* are inserted into client application programs during compilation wherever object creation, pointer creation, or pointer replacement are performed. Unlike the case for main-memory garbage collection, in persistent stores the overhead associated with adding such *write barriers* is insignificant.

The server periodically activates collection, either when the amount of free space available on disk falls below certain predefined threshold, or to match the rate of storage allocation by client applications. During this phase, the collector first verifies the inaccessibility of objects in ZCT by sending the ZCT to all client workstations. Each client workstation returns the subset of zero reference count OIDs that are neither in its local VRT nor in its unsent *GCRL*. The server can then reclaim space occupied by the confirmed garbage.

After a collection, the new ZCT will contain original OIDs that were not in the intersection of the clients' replies and OIDs whose reference counts have fallen to zero due to the transitive decrementing of reference counts during collection phase. When the collector reclaims storage, the ZCT and MRT are continuously updated by new *GCRLs* from committed transactions. This is safe because these new *GCRLs* can only create more garbage, not convert garbage objects to live objects. We do not log the ZCT and MRT because they can be reconstructed after recovery with a traversal of the persistent object graph.

As mentioned previously, we considered two versions of the reference counting algorithm. They differ only in the type and amount of information passed back to the server. In the Server-Oriented version, the client simply sends a *GCRL* that contains all operations that affect the reference count of objects. As a result, the list may contain multiple entries for the same object. In the Client-Optimized version, upon transaction commit the client summarizes the changes before sending them to the server. This is intended to reduce both the communication between client and server and the processing overhead at server.

4.2 Mark and Sweep

We developed two variants of the conventional mark and sweep garbage collection algorithm. In both cases, we use depth first search (DFS) traversal for marking *live* objects, since more complicated marking algorithms that involve link-inversions [19] are very expensive in terms of disk I/Os and require a great deal of states to be logged. The details of the two algorithms are presented in the following subsections.

4.2.1 Server-Oriented Halting Mark and Sweep

This algorithm simply stops all client applications when garbage collection is activated. The collector runs at the server and starts by traversing the database from the set of designated persistent roots. All objects reachable from the persistent roots are marked *live*, and all remaining unmarked objects are collectible. As in the reference counting schemes, we again choose to avoid logging, since the mark phase can start from scratch during recovery from a crash.

After the marking phase, traditional mark and sweep garbage collectors perform a sweep phase, collecting all unmarked (garbage) objects. This sweeping phase can be very expensive over a disk-resident database. In our algorithm we reduce the cost by incorporating the marking phase with the disk space management. We start the mark phase by assuming that the entire disk is free. During the marking phase, as each object is marked, it is treated as if it were a newly created object, and it is removed from the free list. Hence, after the marking process has successfully traversed the database, it will successfully create a new free-list (or bitmap) of the disk. While this scheme avoids the sweeping phase, clearly it has to be incorporated with disk space management schemes that making use of a free list.

4.2.2 Incremental Mark and Sweep

The principle drawback of the above scheme is that it is terribly disruptive — the entire database has to be off-line during garbage collection. For that reason, we also considered a variant of the *incremental update write barrier* algorithms described in [22]. In particular, the algorithm we considered is similar to that developed by Dijkstra [9]. The algorithm allows both the mutators (client applications) and the collector to execute simultaneously.

This incremental mark and sweep algorithm makes use of a *tri-color* scheme [22]. Three marking colors are used to identify objects that are made reachable when the collector is traversing the object graph: *WHITE* for all untraversed objects, *BLACK* for marked objects with all references examined, and *GREY* for marked objects which still have some references not yet been examined. An important consideration is where and how to store the color information for objects. Storing it with the objects on secondary storage incurs a tremendous maintenance and reference cost, since color checking of an object is an operation that will be performed frequently. Furthermore, at the end of each collection phase, the cost of resetting all objects' color field to *WHITE* would require a scan and a write of the entire database. We use a memory resident color bitmap instead. Its size is relatively small — only 2 bits required for each object.

In this algorithm, the server runs a collector that traverses the set of persistent roots, using the depth first marking scheme. At the same time, client applications can manipulate the persistent objects. A *write barrier* [22] detects changes made to the persistent objects in clients. The *write barrier* detects the modification of object references in an object, and adds the OIDs of the new object references to a local grey list. This grey list is then shipped back to the server at transaction commit time, at which point the collector adds it to its list of objects that are yet to be traversed. At the end of the mark phase, the collector at server queries the client, asking for all unsent object identifiers in local grey lists (including references from the client applications' process stacks.)

The collector at the server uses a short read-lock on the current page. Fortunately, this read-lock can be released as soon as the collector finishes examining the page, since the semantics of the collector operation does not require a transaction-consistent view of the database. The combination of this read-lock by the collector and the *write barrier* in the clients are sufficient to ensure that no *live* object can be marked as *dead*, regardless of the interleaving order of client and collector operations.

4.3 Copying

The final category of garbage collection algorithms we considered are copying collectors, which move *live* objects between two separate storage spaces. In addition to reclaiming garbage, this category of algorithms has the added goal of improving the locality of the database by reclustering *live* objects. We consider two different copying collectors: one based upon Baker's incremental copier, the other an algorithm inspired by generational collectors.

4.3.1 Baker's Incremental Copier

The algorithm described in this section is an extension of Baker's incremental copying garbage collector [2]. It requires that the system use logical object identifiers. For garbage collection, logical OIDS have the advantage of allowing the movement of objects without the use of forwarding addresses or the need to update all references to the transported objects.

In our version of Baker's algorithm, the database disk is first partitioned into two equal spaces: a *fromSpace* and a *toSpace*. *Live* objects are copied incrementally from the *fromSpace* to the *toSpace*. New objects are allocated directly into the *toSpace*. The client applications and the server collector execute in parallel. Objects are locked using a write-lock temporarily as they are copied by the collector. The collector acquires a write-lock for the page containing the object to be copied and another for its destination page in the *toSpace*. After moving a *live* object, the collector updates the address of the transferred object in the correspondence table mapping OID's to disk addresses. In this way, all future reference to the object will be directed to the *toSpace*.

In Baker's standard algorithm, a *read barrier* is needed to detect dereferences of references to objects that have been copied out of the from space. In our case, the combination of the indirection offered by the correspondence table and the write-lock on pages actively being copied eliminates the need for this *read barrier*. The *write barrier* is still needed to capture changes made by the client applications mutators to the database during collection. All information captured using the *write barrier* is sent to the collector when the transaction commits. At the end of the copying traversal of the object graph, a *call back* scheme is used to obtain all object references stored in the client application's local variables.

During the copying, the algorithm can utilize a variety of clustering strategies. In our experiments, we used the depth first search algorithm for reclustering for its simplicity and effectiveness.

This algorithm is expensive for several reasons. During normal execution, only half of the storage space can be used. Second, changes to the correspondence table need to be logged for crash recovery, which can be very expensive for large databases. We will address this problem in section 4.3.3.

4.3.2 Partitioned Incremental Copying

In this section we describe a new partition-oriented copying algorithm. It provides the reclustering and compacting benefits of Baker's incremental technique without its space overhead and its delay before reclaimed storage is available. This scheme uses one of the basic ideas behind the generational garbage collection algorithms [20, 21, 24]: instead of copying the entire database, limit the scope of collection to a small region of the storage that is expected to provide the maximum free space for the collection effort. However, unlike generational garbage collectors, which select the region by age of the object stored, the partition collector selects the region with the highest rate of mutation.

Our motivation for this change is the following: generational garbage collectors make the fundamental assumption that most objects die *young*. It is not clear whether it is the case for persistent object stores. It is possible that most of the *young* objects used by an application are transient objects in the applications virtual memory heap. Furthermore, as we shall see below, the partitioned collector is only able to cluster objects that appear in the same partition. This means that if we adopt the generational garbage collector's partitioning scheme, an object cannot be clustered with a related set of objects unless it is of about the same age.

In the partition copying algorithm, we first divide the persistent object store into several contiguous partitions, which may be of different physical sizes. The basis of partitioning may be logical (e.g. grouping together objects of the same class), or physical (e.g. dividing the disk into partitions of equal sizes). In our experiments we have chosen a physical partitioning scheme for its simplicity and generality. The algorithm also uses a *write barrier* to capture changes made by the client applications. However, the *write barrier* in this case will only capture changes that modify a reference to an object in the partition currently being scavenged. It is also used to construct and maintain a *remember set* for each partition in the persistent store. The *remember set* of a partition is the set of all OIDs of objects in other partitions that reference objects stored in this partition[21]. Clients send collection-specific information to the server at transaction commit time.

Like the previous incremental traversal garbage collection algorithms, the partition algorithm uses the *tricolor* scheme to differentiate between traversed and untraversed objects. New objects are always allocated as *grey* and untraversed.

The *remember set* itself is also garbage collected during the collection of its partition, removing all identifiers of objects which no longer reference any object in the scavenged partition. Since logical OIDs are used, a transferred object's entry in the correspondence table needs to be updated, and it needs to be logged for recovery purposes. After recovery, once the correspondence table is restored, the location of all objects will be known and scavenging of a partition can restart. Therefore, this scheme achieves fault tolerance with standard database recovery mechanisms.

So far as described, the partition copying algorithm is not complete since it may fail to collect garbage resulting from cycles of object references in different partitions. We think such cycles will be rare in practice – the partitions can be very large, so even with random clustering the probability of such a cycle arising will be small. With any kind of clustering that attempts to cluster related objects near to one another, this probability will be even lower. However, to avoid problems in the worst case, the algorithm can be augmented so that if a series of collections is ineffective at reclaiming space, the partition size can be doubled (merging adjacent partitions) and another collection attempted. This doubling can continue until the entire database is a single partition, in which case all garbage is guaranteed to be collected. Since we expect this to be a rare occurrence, in our experiments, we do not include test cases where this doubling was necessary.

4.3.3 Recovery

The copying collection algorithms described in this section move objects from one area of the database to another, which, as mentioned in Kolodner's paper [15], can interfere with the proper recovery of the database after system failures. The recovery mechanisms used by our algorithms as described above are simpler than those proposed by Kolodner. The use of logical OIDs avoids the problems associated with inconsistent forwarding addresses; our requirement that the collector sets appropriate locks while objects are being moved guarantees that the garbage collector never moves an object that is being updated by an uncommitted transaction.

5 Experimental Configuration

As mentioned in the introduction, we use both simulation and an implementation in Exodus SM to evaluate those algorithms described. We consider each in turn.

5.1 Simulation

In order to obtain a fair comparison of the the algorithms and an accurate estimate of the cost of garbage collection in a persistent object store, we built a detailed simulator using C++/CSIM [17]. The simulator is based on the client-server model of an object server [8] with page-level locking. The components of the simulator correspond to those in the persistent object store model of Figure 1. The client buffer pool consists of 150 4KByte pages, and the server pool is 500 4KByte pages. Both the client and server buffer managers use the LRU replacement policy. The log manager at the client site is used for simulating the generation of log records due to updates by the client application. The size of a log record is based on that created by Exodus's recovery mechanism [12].

We have simulated a per-update logging system without the page-level optimization used in Exodus. Simulating per-update logging accentuates the difference in the number of log records generated by different collection algorithms. We simulated a 2-phase locking scheme for concurrency control. The GC manager at the client site simulates the appropriate *write barrier* for the different collection algorithms. The GC manager on the server is responsible for spawning the collector for the collection algorithms. The disks simulated are based on the Fujitsu M2266 drive. The details of this simulated disk were obtained from the ZetaSim simulator developed at the University of Wisconsin-Madison.

5.2 Implementation in Exodus

The Exodus SM is a multi-user object storage system with transaction support, concurrency control, and recovery. Objects are distinguished by physical OIDs. Client applications linked with client storage manager module communicate with the server module via remote procedure call to manipulate objects stored on server. The client module provides the upper application layer with a library of interface calls for object manipulation and transaction support. The server is a multi-threaded process that supports I/O, files, transactions, concurrency control, and recovery. Each client request is executed within the context of a thread. See [10] for details.

We did a prototype implementation of the Client-Optimized reference counting (Subsection 4.1) and the incremental mark and sweep (Subsection 4.2.2) in the Exodus SM. We chose these algorithms because they can be easily implemented into the current Exodus SM (which only supports physical object identifiers.)

To implement the two garbage collection algorithms, we provided a set of library routines so that an application can easily link with them to materialize garbage collection. We modified a subset of the C++ implementation of the OO7 benchmark [6] by utilizing those routines to perform garbage collection and measure its cost. Due to the lack of a type manager in the current Exodus system, we built a hand-coded type manager. Our implementation is by no means complete in the sense that it cannot be used directly in the current Exodus release. But we believe that the code we added is sufficient to allow easy construction of a complete version of garbage collection in the Exodus Storage Manager.

For the Client-Optimized reference counting, we provide three client interface routines for the software barriers: sm_GCInsertNew(), sm_GCIncrement(), sm_GCDecrement(). Each takes an OID as input, and adds or updates the corresponding entry (OID, count) pair in *GCRL*. The ZCT, VRT and MRT are all implemented as two in-memory linear hash tables. A special collector thread was added in the server to do the actual collection. Once activated, the collector thread is awaken periodically, checking the disk space utilization first to decide whether to proceed or not. If the free space threshold has been observed, the server initiates one cycle of collection.

For the incremental mark and sweep, we provide a client interface call sm_WBarrier() to add OIDs caught by the *write barrier* to its grey list. A grey stack is implemented at the server for the DFS traversal. The server also maintains a color bitmap for each disk volume. Upon activation of the mark phase, persistent roots are first pushed into the empty grey stack. A special mark thread is then activated to do one cycle of marking by updating the grey stack and the color bitmap. The mark phase ends when the grey stack gets empty and then turns into sweep phase. Since Exodus does not have the kind of disk space management scheme mentioned in 4.2.1 which can be incorporated with the sweep phase, the conventional sweep phase is still performed. An optimization of examining only the interesting (occupied) pages with the help of an existing free page bitmap is used to reduce the number of pages actually traversed. Both the mark phase and the sweep phase are done in cycles to realize the incremental nature of the algorithm. System parameters can also be set to control the amount of work to be done in each cycle and the cycle interval time.

In the algorithms described in Section 4, a *call back* mechanism is needed for verification purpose between

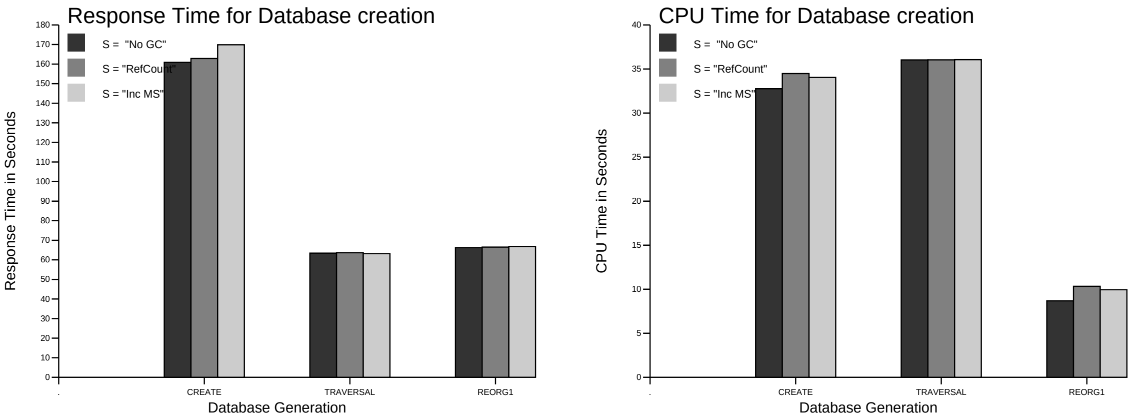

Figure 2: Response Time of Different Transactions during Normal server execution

client and server at transaction commit. However, Exodus does not support such a mechanism at this stage, so we were unable to implement the verification process as described. Instead, we chose to send a snap-shot of the client application's VRT back to the server during transaction commit, in addition to the *GCRL*. This snap-shot copy is retained on each client link at server. The server only needs the most up-to-date copy for verification process. The storage space required for VRTs from all clients may be substantial if the client applications can freely assign persistent OIDs to local variables, as is the case in Exodus [5]. A similar change is adopted for the incremental mark sweep algorithm in Exodus, where all OIDs assigned to local variables during a transaction are captured by the *write barrier* and then sent back to the collector thread at the server for marking as well.

6 Experiments and Results

In this section we describe the results of our experiments. The workload in each case is a subset of an early version of the OO7 Benchmark [6]. Space precludes a full description of that benchmark in this paper. It suffices to know that the OO7 benchmark database contains a graph of *atomic parts*. The graph itself is partitioned into composite parts, with about twenty atomic parts per composite part. Connections in this graph have a random component, though the composite parts provide high degree of locality in the graph. In our tests we made use of four of the benchmark operations:

1. Generate database. Generate an instance of the database. The process of generating such a database bottom up results in some pre-generated atomic parts not being connected to any other parts due to the random connection property of the object graph.

2. Traverse database. Do a depth first search over all the atomic parts in the database. This is a read-only traversal.

3. Reorganize the database (1). Deletes half the atomic parts in the database, then reinserts them.

4. Reorganize the database (2). Delete and reinsert all the atomic parts in the database. In addition to deleting and reinserting more objects than reorganization (1), the insertions and deletions in reorganization (2) are ordered so that it is likely that the resulting database will be badly clustered with the atomic parts for a given composite part spread over many pages.

In this section we want to illustrate through our implementation in Exodus the low overhead of incremental garbage collection. All the measurements shown in this section were obtained with the client application running a DEC 3100 workstation with 16MBytes of main memory and the server running on a DEC 5000 with 32MBytes of main memory and a 100MByte disk, connected through Ethernet. 150 4KByte pages constitute the buffer pool at the client station and 500 4KByte for the server buffer pool.

6.1 Normal Operation Overhead Experiment — Exodus

Figure 2 shows the response time of several different transactions, varying in degree of read and write operations, performed by the client application. The database created in the creation transaction is a small 007 [6] database of 2.8MBytes, created by a series of

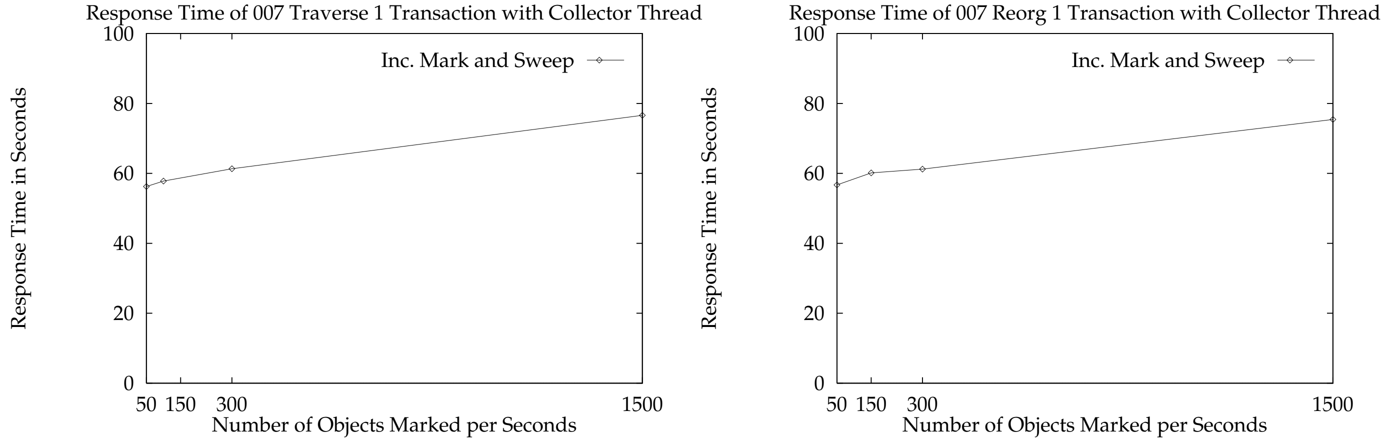

Figure 3: Response Time of 007 transactions during Collection(accessing the Same objects)

update transactions. The second transaction is a DFS traversal of the database, followed by a replacement transaction that replaces half the atomic objects (5000) in the database. The response time of these transaction are measured under normal server execution, in which the collector thread is not activated. Even without the collector running, there is still overhead due to maintaining counts and write barriers and so forth.

The graph in Figure 2 shows a very small difference in response time for each of the following cases: (1) When no garbage collection activities are performed. (2) When the write barrier for the reference counting algorithm are activated. (3) When the write barrier for the incremental mark and sweep scheme is activated. This result shows that the cost of the *write barrier* is virtually undetectable due to the relatively expensive communication and disk operations that are associated with database transactions. In particular, for the read-only transactions there is no CPU overhead at all. The graph also shows that the database creation cost is slightly higher for using any garbage collection schemes than using no garbage collection. This difference is resulted from the increase in communication cost for the GC records sent to the server. Note that more GC records are sent by the mark and sweep write barrier than those sent by the Client-Optimized reference counting scheme.

We regard these figures as encouraging: they imply that the overhead caused by garbage collection while garbage is not actually being collected is minimal. Furthermore, since these overheads should be similar across all algorithms, it is reasonable to expect that the other four algorithms and in particular the *Incremental Partition Copying Algorithm* also has this desirable low overhead property for normal execution.

6.2 Overhead During Collection Experiment — Exodus

Figure 3 shows the effect on response time of the traversal and reorganization (1) transactions when the collector thread is activated at the server. Unlike Figure 2, in this case, the collector at server is actually running concurrently with the client during some portion of the test. The exact portion of the overlap varies with the frequency of activating the collector. The graphs plot the response time of the client transaction as a function of the amount of garbage to be collected each cycle. Note that since the collector and the client are accessing the same database, sometimes they can actually serve to prefetch for each other. This explains for the better performance observed in Figure 3 than in Figure 2 when no collection is taking place. To calibrate the graph, note that running the collector at 1500 objects per second corresponds roughly to continuous execution of the collector.

We also ran the same experiment over a database consisting of two modules (for a total of 5.6MBytes), and made the collector and client run over different modules. The corresponding graph appears in Figure 4. In this case, there are no cache hits for shared accesses, and the performance of the client degrades more rapidly than that in Figure 3.

These figures are encouraging, too: they imply that unless the garbage collector is run at a significant fraction of the time, it has minimal impact on the client applications. Furthermore, since again the overhead in the incremental mark and sweep algorithm is similar to that of the incremental partitioned copying collector, it is also reasonable to expect that the incremental partition copying algorithms can also be run with low overhead during collection as well.

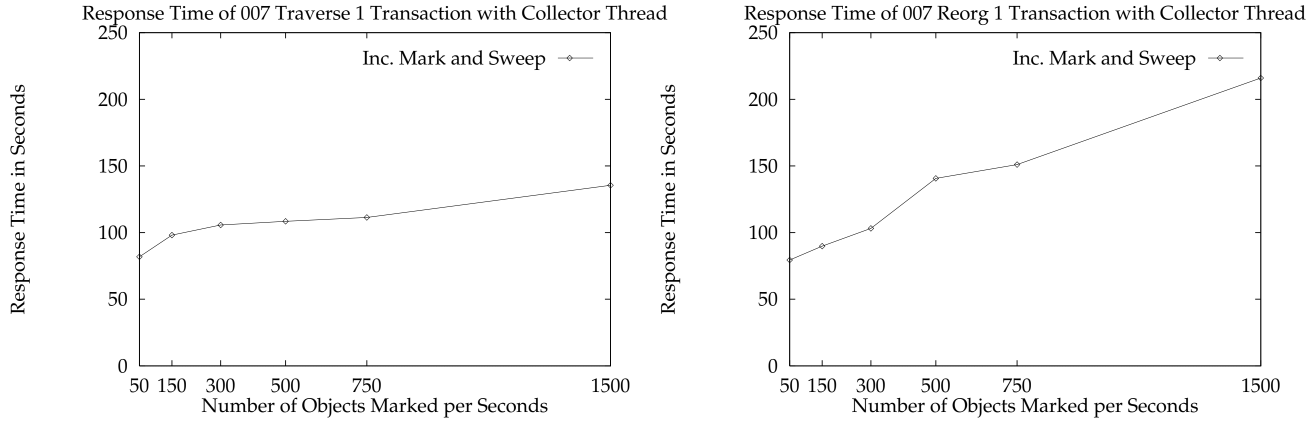

Figure 4: Response Time of 007 transactions during Collection (accessing Different objects)

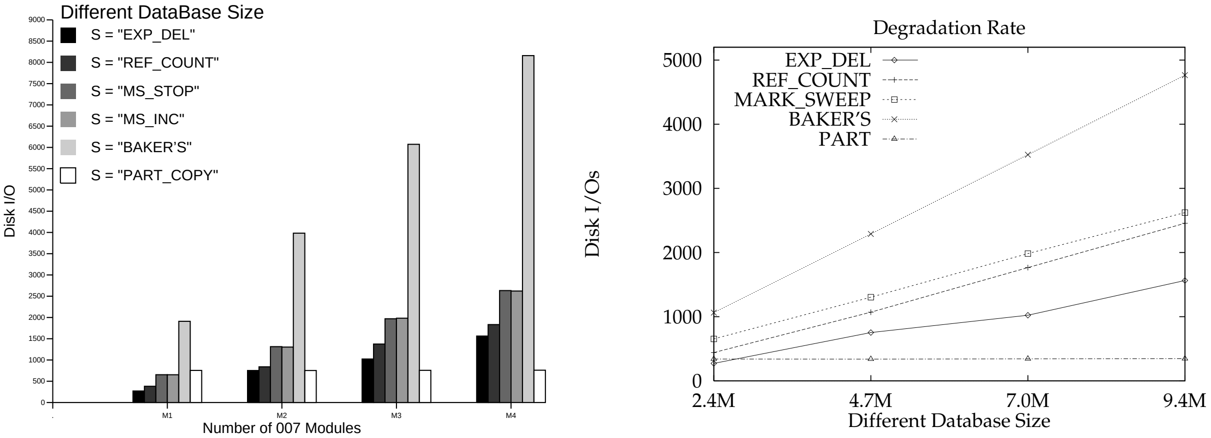

Figure 5: Scalability of Garbage Collectors

6.3 Database Size — Simulation

The experiments in this section are aimed at providing an indication of the scalability of each algorithm. Figure 5 shows the number of disk I/Os required by each algorithm in response to databases of increasing size. We also observed tests with database size up to 51MBytes with similar linear scale up. For the remainder of the experiments we just use small databases mainly to speed the process of collecting results.

Since Server-Oriented and Client-Optimized reference counting have the same amount of disk I/Os, only one graph line is plotted for both. In the figure, *explicit delete* is the algorithm that assumes a programmer explicitly calling a delete procedure on an object as soon as it is rendered garbage. This algorithm benefits a lot from caching, since it is highly likely that the object being deleted is in the cache at the time of deletion.

In this experiment, the partition incremental collector used a physical partition size of 300KBytes. As shown in Figure 5, all other algorithms scale up linearly as the database size increases, except for the partition incremental copying collector, in which case the disk I/Os remain roughly constant. The reason is that the cost of partition copying is dependent on the size of the partition being scavenged and its associated remember set rather than the size of the database. The tradeoff here is that since it only scavenges one partition at a time, the incremental copying collector in general does not reclaim as much space as the other algorithms.

6.4 Recluster for Performance — Simulation

The experiments in this section are designed to compare the ability of each collection algorithm to improve the locality of the *live* objects after the collection phase. The workload used in this test are the two reorganization operations from an early version of the 007 benchmark [6], which were designed to test the compaction and reclustering capability of object-oriented database systems. The size of the database used in this test is approximately 2.5MBytes with a total of 32943 objects created. In reorganization 1, half of the atomic objects (5000) in the entire database are replaced by newly created atomic objects. The pattern of replacement allows the newly created objects to be clustered together. Figure 6 shows the hit rate for traversing the entire database after the reorganization and after the execution of the respective garbage collection algorithms. The hit rate before the reorganization is 97%, while after the reorganization the hit rate for the schemes that did not recluster dropped to 93.5%. Note that both the incremental partitioned copying algorithm and Baker's algorithm have a significantly higher hit rate after the garbage collection.

Figure 6 also shows the number of 4KByte pages of data communicated between the server and client for the traversal. Without reclustering, the number of pages communicated before the replacement is 910 pages and that after the replacement is 1939 pages. The database grew from 2.5MBytes to 3.4MBytes. With reclustering, the database size dropped and the number of pages transferred approached the number transferred before the reorganization.

In reorganization 2, all the atomic objects (10000) in the database were replaced. The pattern of replacement also differ from reorganization 1, making it much harder for the system to cluster well during the reorganization. This test is designed to accentuate the difference in clustering capability of different object-oriented database system. Figure 7 shows the hit rate of traversing the entire database after the reorganization and after the execution of the respective garbage collection algorithms. The hit rate before the traversal was 97%, and after the replacement for the algorithms that could not recluster this dropped to 84.9%. Again, Baker's and the incremental partitioned copying algorithms were able to recluster and after running those algorithms the hit rates were much better. Figure 7 shows the same test but in terms of the number of 4K pages of data communicated between the server and client for the traversal. Again, only Baker's and the incremental partitioned copying algorithms were able to recluster, and the performance of the traversal after running these algorithms was dramatically better than than the performance after running the others.

7 Conclusion

Reference counting, though has the nice property of having a cost proportional to the number of objects modified, is incomplete since it lacks the ability to collect cyclic garbage. In addition, our simulation showed that the space overhead needed for storing the reference counts of objects can be quite high, especially when the degree of connectivity of the database object graph is high. For example, in our simulation, for a small database of 2.4MB, the MRT tables occupied 124 4K pages. Another important short coming of reference counting is its inability to improve locality of reference or remove fragmentation. However, it is still incremental in nature and compared to mark and sweep or Baker's incremental copying, its cost is still much lower. Finally, like the partition incremental copying collector, it provides a fast track to reuse free space.

The mark and sweep algorithm in general has a high cost since it needs to traverse the entire database. In addition, it does not recluster nor does it remove fragmentation. In the case of stop-the-world mark and sweep, it is simply too disruptive for database transactions. Hence, one can rarely recommend mark and sweep collection for persistent object stores.

Baker's incremental collector has the capability to recluster the objects for locality improvement. It is incremental in nature, thus it can amortize its cost over time. However, when free space is needed urgently due to a sudden burst of storage allocation, it is inefficient since it can only reuse its *fromSpace* after the end of the entire object graph traversal. In addition, it requires disk space twice as large as that required to hold the database, and half of that space is always non-usable to user. This is a much more serious problem in disk-storage than in virtual memory storage.

Partition incremental copying can be viewed as a compromise approach to Baker's algorithm, since it trades improvement of locality for time and space (fewer disk I/Os are needed to reclaim space, and a much smaller *toPartition* is needed). Although the improvement of locality of reference offered by the partition collector is less than that of Baker's algorithm, our simulation shows that through careful selection of the partition size a significant gain in locality is possible. This is true especially in the case where partitions are logical groupings of objects that have a high locality of reference and few inter-partition references. A more detailed description and study of the partition incremental copying algorithm will be provided in a coming paper.

8 Acknowledgements

We gratefully acknowledge the essential contributions of these people: Mike Carey, Marvin Solomon, Prakash Sundaresan, Mike Franklin, Zorn Benjamin and Manolis Tsangaris for their constructive comments and ad-

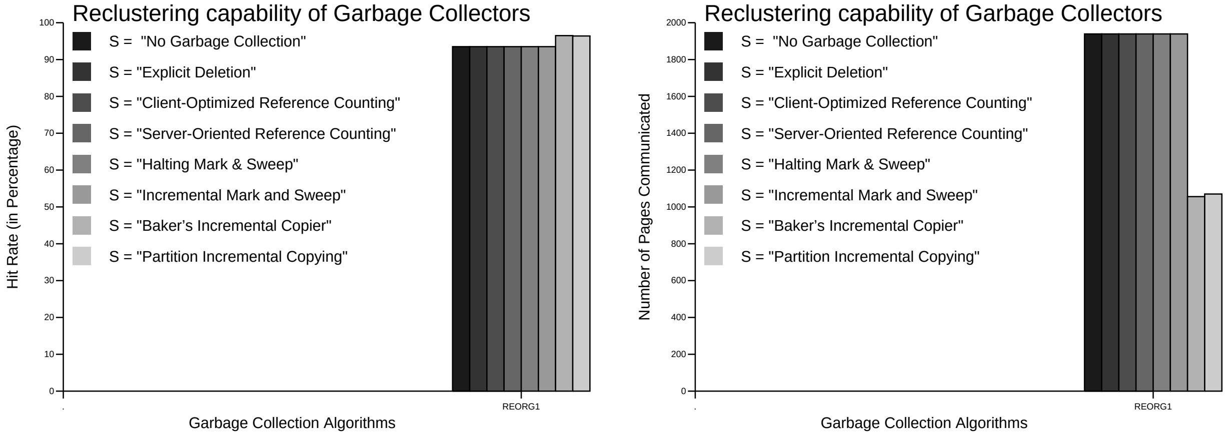

Figure 6: Hit Rate and Communication Cost for 007 Reorganization Test (1).

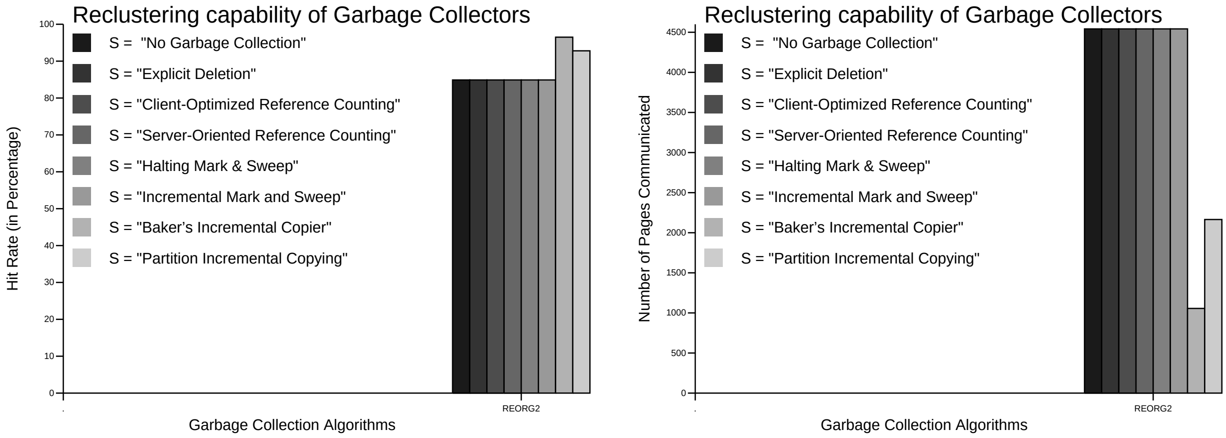

Figure 7: Hit Rate and Communication Cost for 007 Reorganization (2).

vice; Barbara Liskov for her inspiring talk on Thor, driving us to come up with a better copying collector; Ambuj Shatdal for being the perfect graphic consultant; Mike Zwilling for his patience in demonstrating the profiling of the Exodus server using IPS and C.K.Tan for his knowledge of the Exodus recovery system.

References

[1] Appel, A.W., Ellis, J.R. and Li, K. "Real-time Concurrent Collection on Stock Multiprocessors," Proceedings of SIGPLAN Conference on Programming Language Design and Implementation 1988, Atlanta, Georgia, June 1988, pp.11-20

[2] Baker, H.G. "List processing in real time on a serial computer," Communication of the ACM, 21(4), April 1978, pp.280-294

[3] Brooks, R.A. "Trading Data Space for Reduced Time and Code Space in Real-time Garbage Collection on Stock Hardware," Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, pp.108-113, August 1984, Austin, Texas

[4] Butler, M.H., "Storage Reclamation in Object Oriented Database Systems," ACM-SIGMOD Proceedings, San Francisco, May 1987, pp.410-425

[5] Carey, M.J., DeWitt, D.J., et. al. "The Exodus Extensible DBMS Project: An Overview," Readings in Object-Oriented Database Systems, Morgan Kaufmann, San Mateo, California, 1990, pp.474-499

[6] Carey, M., Dewitt, D.J. and Naughton, J.F. "The 007 Benchmark", to appear in SIGMOD '93.

[7] Deutsch, L.P. and Bobrow, D.G. "An Efficient, Incremental, Automatic Garbage Collector" Communications of the ACM, Vol. 19, Number 9, September 1976, pp.522-526

[8] DeWitt, D.J., Futtersack, P., Maier, D. and Velez, F. "A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems," Proc. 16th International Conference on VLDB, Brisbane, Aug. 1990

[9] Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S. and Steffens, E.F.M. "On-the-fly garbage collection: An exercise in cooperation," Communication of the ACM, 21(11), November 1978, pp.966-975

[10] Exodus Project Group, Exodus Storage Manager Architectural Overview, Exodus Project Document, University of Wisconsin-Madison, Nov. 1991

[11] Franklin, M.J., Copeland, G., Weikum G. "What's Different About Garbage Collection for Persistent Programming Languages?" MCC Technical Report

[12] Franklin, M.J., Zwilling, M.J., Tan, C.K., Carey, M.J., Dewitt, D.J. "Crash Recovery in Client-Server Exodus," ACM-SIGMOD International Conference on the Management of Data, San Diego, 1992

[13] Johnson, D. "The Case for a Read Barrier," Fourth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS IV), pp.96-107, Santa Clara, California, April 1991

[14] Kolodner, E., Liskov, B. and Weihl, W. "Atomic Garbage Collection: Managing a Stable Heap," Proceedings of the ACM-SIGMOD Conference, Portland, Oregon, 1989, pp.15-25

[15] Kolodner, E., Weihl, W. "Atomic Incremental Garbage Collection and Recovery for Large Stable Heap," To appear in Proceedings of ACM-SIGMOD Conference, Washington D.C., 1993

[16] Moon, D.A. "Garbage Collection in a Large Lisp System," In Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, pp.235-246, Austin, Texas, August 1984

[17] Schwetman, H. "C++/CSIM: An Object Oriented Simulation Package" MCC/Octopus Project

[18] Schuh, D., Carey, M. and "Persistence in E Revisited - Implementation Experiences," UW-Madison Computer Sciences Department Technical Report 957

[19] Standish, T.A. "Data Structure Techniques," Addison-Wesley, Reading, Massachusetts, 1980, pp.212-218

[20] Ungar, D. and Jackson, F. "Tenuring Policies for Generation-Based Storage Reclamation," OOPSLA 1988 Conference Proceedings, San Diego, California, 1988, pp.1-17

[21] Ungar, D. "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm," ACM SIGPLAN Notices, Vol. 19.5, May 1984, pp.157-167

[22] Wilson, P.R. "Uniprocessor Garbage Collection Techniques," Proceedings of the International Workshop on Memory Management, September 1992

[23] Wilson, P.R., Lam, M.S. and Moher, T.G. "Effective Static-graph Reorganization to Improve Locality in Garbage-Collected Systems," Proceedings of the SIGPLAN 1991 Conference on Programming Languages Design and Implementation, Toronto, Ontario, Canada, June 1991, pp.177-191

[24] Wilson, P.R. and Moher, T.G. "Design of the Opportunistic Garbage Collector," OOPSLA Conference Proceedings 1989, New Orleans, Louisiana, Oct. 1989

[25] Zorn, B. "Comparing Mark and Sweep and Stop and Copy Garbage Collection," LISPC90, Nice, France, June 1990, pp.87-98.