# TMOS: A Transactional Garbage Collector

John Zigman[1], Stephen M Blackburn[2], and J Eliot B Moss[2]

[1] Department of Computer Science
Australian National University
Canberra ACT 0200, Australia
`John.Zigman@cs.anu.edu.au`

[2] Department of Computer Science
University of Massachusetts
Amherst, MA, 01003, USA
`{steveb,moss}@cs.umass.edu`

**Abstract.** Defining persistence in terms of reachability is fundamental to achieving orthogonality of persistence. It is implicit to the principles of orthogonal persistence and is a part of the ODMG 3.0 data objects standard. Although space reclamation in the context of persistence by reachability can be achieved automatically using garbage collection, relatively few papers address the problem of implementing garbage collection in a transactional storage system. A transactional GC algorithm must operate correctly in the face of failure, and in particular must deal with the problem of transaction abort, which by undoing changes such as the deletion of references, subverts the GC reachability axiom of 'once garbage always garbage'.

In this paper we make two key contributions. First, we present a generic approach to the design of transactional collectors that promotes clarity, generality, and understandability, and then using this approach, we present a new transactional garbage collection algorithm, TMOS. Our design approach brings together three independent components—a mutator, a transactional store, and a GC algorithm. TMOS represents the application of the Mature Object Space family of GC algorithms to the transactional context through our approach to transactional GC design.

## 1 Introduction

The practicality and value of abstraction over persistence is increasingly being acknowledged in the mainstream of the database community [Snodgrass 1999; Maier 1999]. While different persistent systems vary in their adherence to the principles of orthogonal persistence, most, including the Java Data Objects (JDO) [Sun Microsystems 1999] and ODMG-3 [Cattell et al. 2000] standards, define persistence in terms of *reachability*. Persistence by reachability (PBR) is a simple corollary of the principle of persistence identification [Atkinson and Morrison 1995] which states that persistent objects are not identified explicitly, but implicitly through transitive reachability from some known persistent root (or roots). With PBR, once an object ceases to be reachable it is by definition not identifiable and is thus garbage. This paper addresses the problem of automatically collecting such garbage in a transactional setting.

Transactions are a means of managing concurrency and recovery in a persistent system. Central to transactions are the notions of atomicity and serializability. While transactions are ubiquitous in database systems, the rigidity of conventional ACID transactional semantics [Härder and Reuter 1983] and their basis in isolationist rather than cooperative approaches to concurrency control appears to have retarded their integration into programming languages [Blackburn and Zigman 1999]. Nonetheless, transactions represent one important and practical solution to the problem of concurrency control in orthogonally persistent systems, and furthermore are established in persistence standards such as JDO and ODMG-3. Thus the integration of garbage collection into a transactional setting is important.

The argument for space reclamation in *primary* storage is principally based on the relatively high cost of memory. However, for *secondary* storage, the space cost argument is less significant, and the impact of a reclamation mechanism on access characteristics may become the overriding concern. Continued manipulation of a persistent object graph will inevitably result in the store becoming fragmented, and in the limit, each live object will reside on a distinct store page. Garbage collection will improve the clustering of secondary storage, and thus improve retrieval performance. Such clustering can occur either as an explicit function of the garbage collector, or implicitly as a side effect of object reclamation and subsequent space reuse by new objects.

**The Problem of Transactional Garbage Collection** A transactional garbage collector must address two fundamental issues, that of safely and efficiently collecting garbage and that of maintaining consistency in the face of concurrency and transactions. The interdependencies between these issues are complex and lead to involved algorithms that are often hard to describe. The resulting algorithms make it hard to distill the issues and choices and separate those that are central to correctness from those are motivated by efficiency. Thus it is often hard to understand such algorithms and have confidence in their correctness. This is the high-level problem which this paper addresses. We do this through the use of an appropriately abstract framework for relating garbage collection and transactions.

In addition to presenting an abstract framework, we identify and address a number of concrete issues common to transactional garbage collector implementations.

*Identifying Garbage in the Face of Rollback* The axiom *once garbage always garbage* is key to the correctness of garbage collection algorithms. It says that once a collector has identified some object as unreachable, that object may be *safely* collected—once the object is unreachable it has no means for becoming reconnected to the object graph. However, the introduction of *transactions* demands a re-examination of the axiom.[1] Through *atomicity* transactions exhibit all-or-nothing semantics: either the effects of a transaction are seen completely or not at all. *Isolation* is used to ensure consistency by

[1] For simplicity we specifically address ACID transactions [Härder and Reuter 1983] throughout this paper. Generalization of this approach to include the many flavors of advanced transaction models will in many cases be trivial, but a thorough analysis of the intermix of garbage collection with such models is beyond the scope of this work.

guaranteeing that other transactions are not exposed to partial results that are subsequently rolled back. A transactional garbage collector must therefore either operate in strict isolation from user transactions (which could place very expensive constraints on the ordering of user and GC activities), or be aware that partial results may be rolled back, and so the axiom of once garbage always garbage must be carefully reinterpreted. Fortunately, transactions also have the property of *durability*, which means that once a transaction has successfully committed it can't be undone. This, combined with the once garbage always garbage axiom means that it is safe to use any committed image of the store for the identification of garbage, even *stale* (superseded) images.

*Additions to the Object Graph* The allocation of new objects poses different problems. If a transaction rolls back, all trace of that transaction must be undone, including the allocation of any objects within the transaction. A second problem relating to allocation is that not all objects created in the course of a transaction will be reachable from persistent roots when the transaction commits. Clearly it would be desirable to avoid the overhead of persistent object allocation for such transient objects.

A simple way of dealing with the first problem is to allocate space in the store for the object during the transaction and then reclaim that space if the transaction is rolled back. A better approach, which addresses both problems, is to defer the allocation of store space until the transaction commit. Any object created by the transaction that is not reachable from a store object modified by the transaction cannot be reachable from the store's root, and is thus garbage. This approach also has a disadvantage, as it raises the question of how newly allocated objects will be identified (i.e., what form an OID or reference to such an object would take) prior to the allocation of a corresponding store object at commit time.

*Collecting Both Transient and Persistent Data* When implementing garbage collection for a persistent language, the problems of disk and heap garbage collection can be dealt with either together or separately. On one hand a single mechanism collects a unified persistent space which includes transient heap data, while on the other hand the problems can be seen as distinct and addressed separately.

We take the approach of distinct heap and disk collectors on the basis of two significant observations. The first relates directly to the previously mentioned problem of additions to the object graph—if persistent space is only allocated for objects that are persistently reachable at commit time, the distinction between persistent and transient data is strong. The second hinges on the *substantially* different cost factors for heap and disk collection. The time and space tradeoffs to be made for each are vastly different. Thus a dependence on random access to objects may be quite reasonable for a heap collector, but totally unrealistic for disk-bound objects.

The remainder of this paper is structured as follows. In section 2 we discuss related work. We then discuss our framework for designing transactional garbage collection algorithms in section 3. In section 4 we apply our framework, adapting the MOS [Hudson and Moss 1992] garbage collection algorithm to a transactional context. In section 5 we briefly discuss opportunities for future work and then conclude in section 6.

## 2 Related Work

**Transactional Garbage Collection** The literature records three substantially different approaches to transactional garbage collection. Kolodner and Weihl [1993] adapt a semi-space copying collector to a transactional persistent heap. In their system the mutator and collector are tightly coupled and must synchronize to ensure that the mutator sees a coherent heap image. By using a persistent heap they make no distinction between the collection of transient and persistent data.

Amsaleg, Franklin, and Gruber [1995] tackle the problem of transactional GC in the setting of an orthodox OODB. They examine the interaction of the garbage collector with the store at a detailed level, enumerating cases which cause erroneous behavior in the face of transaction abort and rollback. This leads to a set of invariants that govern the marking mechanism and the object deletion phase of their incremental mark and sweep collector. Although capable of operating over a distributed partitioned object space, their algorithm is unable to detect cycles of garbage which span partitions—a problem which was subsequently addressed by Maheshwari and Liskov [1997].

Skubiszewski and Valduriez [1997] define the concept of *GC-consistent cuts* as a means of establishing a consistent context in which to apply the *three color marking* concurrent collection algorithm [Dijkstra et al. 1978]. Their approach is safe, concurrent and complete, but it is not incremental (no garbage can be reclaimed until the mark phase has completed with respect to the entire database). *GC-consistent cuts* are similar to the incremental coherent snapshot mechanism we describe in section 4.3.

While Amsaleg et al. do highlight key issues for transactional garbage collection, neither theirs nor any of the other papers propose general methodologies for intersecting garbage collection with transactions. Rather, they each present specific solutions based on ad-hoc methodologies for dealing with this complex interaction.

**Object Clustering** Yong, Naughton, and Yu [1994] examine a variety of different garbage collection algorithms in the context of a client-server system, and conclude that incremental partitioned garbage collection provides greater scope for reclustering and better scalability than other algorithms. Cook, Wolf, and Zorn [1994] extend this work by examining a range of partition selection policies including: no collection, random, mutated (most pointer updates) and best possible (oracle). Cook, et al. show that a practical and appropriate policy can improve the overall I/O performance of the store. This indicates that incremental partitioned garbage collection can have practical benefits for object store performance, and so motivates further examination and development of such systems.

Lakhamraju, Rastogi, Seshadri, and Sudarshan [2000] discuss the general problem of on-line reorganization in object databases and present IRA, an algorithm which can be applied to copying collectors, online clustering algorithms and other contexts where it is necessary to move objects in the presence of concurrent access to those objects. IRA is able to work when object references are physical (i.e. there is no indirection between user-level and store-level object identification). In TMOS we address this problem through the use of indirection.

**Mature Object Space Collector** MOS is a copying collector that is incremental, complete and safe [Hudson and Moss 1992]. PMOS extends MOS to work in the context of a persistent heap [Moss et al. 1996].[2] PMOS is designed to be atomic, but without binding to any particular recovery mechanism. PMOS achieves incrementality and minimizes I/O overheads by delaying the update of partition meta-data and changes to references due to garbage collector activity. It requires a degree of synchronization and co-operation between the mutator and the garbage collector. Munro et al. [1999] explore an initial implementation of PMOS within a system without write barriers. In their implementation the heap is scanned to provide consistent meta-data for the local garbage identification process to operate correctly.

This paper extends the literature by generalizing the problem of transactional collection and developing an abstract framework through which arbitrary collection algorithms can be safely applied to a transactional context. We achieve this generality by providing an abstraction of garbage collection which separates *local* and *global* issues and the problem of *identifying garbage* from that of *removing/reclaiming garbage*. Moreover, we apply this framework to the MOS algorithm, yielding TMOS, which is safe, concurrent, complete and incremental.

## 3 An Approach to Transactional GC

In this section we present a framework and methodology for the design of garbage collection algorithms for transactional persistent stores. We outline significant issues involved in the design and describe the interrelations between them. Our framework is intentionally abstract and general, and rests upon a simple system model for transactional garbage collection based on a *collection algorithm*, a *store*, and a *mutator*. By raising the level of abstraction we reduce the assumptions made with respect to any particular component, thereby admitting as wide a range of collection algorithms, storage systems, and classes of mutator as possible. Our goal is to facilitate rapid *and correct* design by identifying and addressing the primary issues associated with the intermix of garbage collection and transactions. A strong and safe design provides a good basis for subsequent optimization, at which point the dependencies on each of the system components may be tightened as necessary to minimize impedance between the components as far as possible.

### 3.1 A System Model for Transactional Garbage Collection

The system has three distinct components: a mutator (of which there may be multiple instances), a transactional store, and a garbage collector. The system model is based on the transactional object cache architecture [Blackburn and Stanton 1999], which while providing strong transactional semantics, allows efficient implementations by virtue of the mutator having direct access to the store's cache. A consequence of this architecture is a clear separation of store and heap collection mechanisms. Indeed, this model does

[2] PMOS sacrifices the unidirectional write barrier optimization of the original MOS algorithm in order gain the flexibility of being able to collect cars in any order.

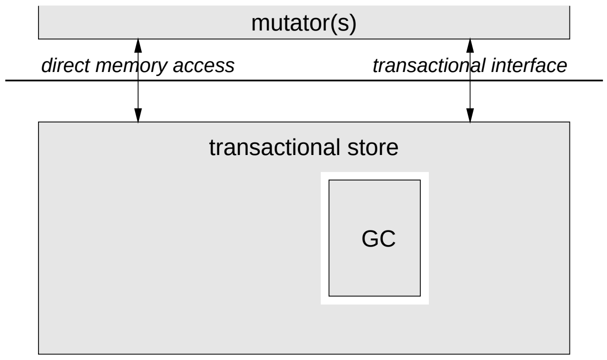

**Fig. 1.** Generic system model for transactional garbage collection, consisting of a mutator (or mutators), a transactional store, and a garbage collection algorithm.

not preclude the existence of a mutator without a collection mechanism for transient data (a C++ program for example).

*Mutator* Each mutator has direct access to store objects cached in main memory. Access to the objects is mediated through a transactional protocol [Blackburn and Stanton 1999]. Before a mutator transaction reads *(updates)* an object for the first time, it informs the store of its read *(write)* intention with respect to that object. The store will guarantee availability and transactional coherency of the object from the time that it acknowledges the mutator's intention until the mutator transaction commits or aborts.[3] A mutator may concurrently execute multiple transactions, however the mutator must ensure transactional isolation is maintained. The protocol is sufficiently abstract to allow the store to employ 'copy-out' or 'in-place' object buffering, and either optimistic or lock-based coherency policies.

*Transactional Store* The store must support conventional transactional semantics with respect to a graph of uniquely identifiable objects made available to the mutator as described above. Ideally the store would support multiple concurrent mutator transactions, and implement layered transactions, although neither of these are requirements of the our system model.

*Garbage Collection Algorithm* The garbage collection algorithm removes objects that are no longer usable by any mutator. An object that has become permanently unreachable (i.e., as consequence of a committed transaction) from the persistent root is clearly unusable by the mutator and is thus considered garbage. To the extent that the garbage collector performs any operations beyond reclaiming garbage (such as compacting for

[3] This does not preclude the use of a store that implements a STEAL policy. However, such a store must use a transparent mechanism such as memory protection to ensure that the page is transparently faulted back into memory upon any attempt by the mutator to access it.

example), those operations must also be completely transparent with respect to the mutator. Garbage collection activity must therefore be visible to the mutator as only as a logical 'no-op' (it consumes resources but does not otherwise visibly affect state).

Conceptually, a garbage collection algorithm must undertake two distinct activities: the *identification* of garbage, and the subsequent *collection* of that garbage. The algorithm must *safely* (perhaps conservatively) identify unreachable objects and conceptually mark them as garbage. Collection may include secondary activities such as compaction and clustering. In general, activities will occur in two domains: *locally* and *globally*. Algorithms typically use partitions (e.g., *cars* in MOS, *generations* in generational collectors) to increase incrementality, which is particularly necessary in the context of significant temporal overheads (due to disk or network latency, for example). Local activities (such as intra-partition collection) depend only on local information while global activities (such as the reclamation of cross-partition cycles) utilize local progress to work towards global properties of completeness. The mapping of the two activities onto these domains is as follows:

– *Local identification* of garbage involves a local (typically intra-partition) reachability analysis of objects with respect to some known (possibly empty) set of references to objects within the local domain.
– *Local collection* of garbage may simply involve deleting those objects not identified as reachable by the local identification phase.
– *Global identification* of garbage involves the identification of garbage not identifiable locally (such as cross-partition cycles).
– *Global collection* of garbage simply involves removing those objects (or partitions) that are not reachable.

## 3.2 Designing a Transactional GC Algorithm

The design of a transactional garbage collector must address both *correctness* and *performance*. We approach correctness by first establishing the correctness of the collection algorithm on which the transactional collector will be based, and then ensuring that the correctness is not compromised by its introduction into a transactional context. However, correctness alone is not adequate. The reclaimation of space (automatically or not) can only be justified by an improvement in long term performance—there is little point in employing a reclaimation mechanism if the system will perform better without it.[4] In achieving this, the collection mechanism must impose minimal interference in terms of overall throughput and/or responsiveness of the store operation. We now address the separate concerns of correctness and performance in transactional garbage collector design.

**Correctness** We identify the key correctness issues with respect to: *local identification* of garbage, *local collection* of garbage, *global identification* of garbage, and *global collection* of garbage.

[4] While the need to reclaim space in an in-memory context is clear, secondary storage is by comparison both very cheap and very expensive to access, making the benefits less clear cut.

*Local identification* correctness depends on the *collector* being able to see a consistent and valid view of the object graph and the roots for the region being collected in the face of mutator updates and rollback.

*Local collection* correctness depends on the *mutator* being able to see a consistent and valid view of the object graph in the face of the collector deleting (and possibly moving) objects. The deletion of objects identified as garbage raises no significant correctness issues because unreachable objects cannot be seen by the mutator. However, ancillary actions associated with the collection process (such as the movement of live objects to reduce fragmentation) have the potential to impact the mutator. Our system model includes a guarantee of consistency with respect to the objects seen by the mutator for the duration of a transaction, so any change to object addresses must not be visible to a mutator. However, because transactional isolation prevents the mutator from storing object references across transactions, the collector need only be concerned with *intra*-transactional consistency of object addresses seen by the mutator.

*Global identification* corrrectness depends on the *collector* being able to see a consistent and valid view of the reachability metadata (e.g. inter-partition reachability information). Such metadata may be changed as a consequence of a mutator updating an object reference.

*Global collection* correctness depends on the *mutator* being able to see a consistent and valid view of the object graph in the face of the collector deleting (and possibly moving) objects. Correctness issues for global collection thus mirror those of local collection. For some algorithms this phase is made even simpler because the collection of an unreachable partition and its constituent objects does not lead to the movement of any live objects.

Each of these correctness criteria can be trivially (though inefficiently) addressed by isolating the garbage collector actions from the mutator by performing the collector actions within a user-level transaction.

**Performance** Recall that garbage collection should be justified in terms of long-term performance improvement. In the case of persistent data, locality improvements resulting from the positive clustering and defragmentation effects of collection are likely to be the dominant source of performance improvement. We now address issues of performance under the headings of two dimensions which we seek to maximize—*responsiveness* and *throughput*—which correspond to the fundamental temporal dimensions of latency and bandwidth respectively. The relative importance of each of these concerns will depend heavily on the context in which the collector is to be applied. A system with a high degree of user interaction may value responsiveness over throughput. By contrast, an account management system may require high throughput, but have more relaxed responsiveness requirements.

*Responsiveness* The problem of maximizing responsiveness is fundamentally one of local optimization (a goal which may run counter to global optimization efforts). Maximizing mutator responsiveness corresponds to minimizing collector disruptiveness. While the problem of minimizing collector disruptiveness is a goal common to conventional heap collectors, the widely differing cost factors at play and correspondingly different expectations with respect to responsiveness suggest that the issue should be examined freshly for the transactional context. We identify three broad strategies for improving responsiveness: *minimizing the grain of collection increments*, *minimizing mutator/collector resource contention*, and *minimizing collection activity when mutator demand is high*. The first of these strategies will minimize any mutator pause time, but may have the deleterious effect of reducing collection efficiency by shifting the identification/collection efforts from local to the more expensive global domains. Many resources are shared by the mutator and collector, including: CPU, memory, I/O bandwidth, and locks. The computational efficiency of the collection algorithm will impact on CPU contention. An opportunistic algorithm may significantly reduce both memory and I/O bandwidth consumption, and appropriate use of transactions (most notably the use of layered transactions) can dramatically reduce the level of lock contention. The applicability of the third strategy is highly application dependent, but in general, responsiveness will be improved if collections opportunistically exploit lulls in mutator activity.

*Throughput* The problem of maximizing throughput is one of global optimization, a goal sometimes at odds with the objective of responsiveness. For example, a clustering policy may yield substantial improvements in *average* performance but might occasionally impact negatively on responsiveness. While a great many aspects of transactional collector implementation will impact on long-term performance, it seems that two relating to I/O are likely to be dominant: *minimizing I/O cost by improving data locality*, and *minimizing I/O overheads through opportunistic collector operation*. Because a collector's capacity to improve object locality through compaction and defragmentation is likely to have a major impact on long term performance, a collector that actively compacts or is able to accommodate a clustering mechanism is likely to yield better throughput improvement. Because it is essential that the collection process does not generate more I/O than it avoids, opportunistic policies, both with respect to reading and writing, are likely to be highly advantageous.

**Correctness and Performance** As indicated earlier, each of the four correctness issues can be addressed by isolating collector activity through user-level transactions. This solution is trivially correct because transactional semantics guarantee the coherency of the image seen by any transaction. Unfortunately this approach is heavyweight, requiring read locks to be acquired for each of the objects in the region of collection, leading to strict serialization with any transaction updating objects in that region. Less disruptive approaches exist for each of the four phases.

The correctness of local and global *identification* will be assured if identification occurs with respect to a *snapshot* of some previously committed state of the region of collection (for local identification, the object graph, and for global identification, inter-partition meta-data). This approach is trivially correct because the 'once garbage always garbage' axiom ensures that garbage identified with respect to any coherent image of the committed state of the region will always be garbage. While a snapshot could be created by briefly obtaining a read-lock on the region of collection and copying its contents, a coherent 'logical' snapshot can be established cheaply by viewing each object in the region as it is unless a write intention is declared with respect to the object, in which case the 'before-image'[5] of the object is used. The same before-image must continue to be used even if the updating transaction commits (in such a case the before-image will have to be copied by the garbage identification process before the image is removed as part of the commit process).

The primary correctness issue for local and global *collection* is that of ensuring that the mutator is not exposed to the relocation of any live object by the collector. An alternative to encapsulating these phases within a user-level transaction is to use some form of address indirection between the mutator and the collector, which would insulate the mutator from such perturbations. This could be done through the explict indirection of OIDs, or by using virtual memory techniques.[6]

This section has outlined the elements of a two-step methodology for composing garbage collection algorithms and transactional storage systems. First the candidate algorithm must be decomposed into local and global identification and collection phases. Then for each of the four phases, means must be identified for ensuring the respective correctness criteria outlined above. The simplicity and clarity of this approach allows the bulk of design work to be focused on the efficiency of the particular solution.

## 4 The TMOS Transactional Garbage Collector

Having outlined a generic design approach, we now apply that approach and develop a new transactional garbage collection algorithm, TMOS. After describing pertinent aspects of the context in which TMOS is developed, we discuss the design of the algorithm by addressing each of the correctness and performance issues outlined in the previous section.

### 4.1 Implementation Context

The primary design goal for TMOS is that it improve the long term performance of transactional store operation. This objective was a significant factor in our choice of the Mature Object Space (MOS) [Hudson and Moss 1992] collector as the basis for TMOS ('Transactional MOS'). Secondary issues include our desire to be able to integrate clustering algorithms [He et al. 2000] into the collector. We now describe the context in more detail in terms of each of the three components of the system model that underpins our generic design approach.

[5] A 'before-image' contains the last committed state of an object, and is typically created by the underlying transaction system as soon as an intention to write to that object is declared. The image is used to re-establish the object state in the event that the transaction is rolled back. This idea is similar to that of *GC-consistent cuts* [Skubiszewski and Valduriez 1997].

[6] Although it would avoid the overhead of an indirection, using virtual memory to achieve this would require each concurrent transaction to execute within a separate address space.

*Mutator* The design of TMOS places few preconditions on the mutator beyond the requirement that it interface to the storage layer through a cache and transactional interface according to the transactional object cache architecture. Our implementation of TMOS targets PSI [Blackburn and Stanton 1999], a particular instantiation of such an interface. Beyond compliance to such an interface, we make no presumptions about the mutator. Our specific goals include language run-times for orthogonally persistent Java and Java Data Objects implementations, as well as arbitrary applications such as benchmarking tools written in C and C++.

*Transactional Store* The TMOS algorithm in its most general form places no requirements on the store beyond the requirement that it support normal transactional storage semantics. However, significant performance improvements depend on the store supporting (and giving the collector access to) two-level transactions and making before-images of updated objects available to the collector.

*Garbage Collection Algorithm* The MOS algorithm was chosen because of its incrementality, completeness, and flexibility. MOS divides the object space into partitions called *cars*, the size of which determines the granularity of collection. The extension of MOS to allow cars to be collected in any order (as long as all cars are eventually collected), greatly facilitates opportunism in TMOS [Moss et al. 1996]. Furthermore, the flexibility of the MOS object allocation and *promotion* (copying) rules is substantial, allowing clustering algorithms to be integrated into the collector. Space constraints preclude a comprehensive account of the MOS algorithm here, so we refer the reader to other publications for details of the algorithm [Hudson and Moss 1992]. Instead we will briefly describe MOS in terms of the various phases of garbage collection identified in section 3.1.

– *Local identification* of garbage depends on identifying the reachability status of each object with respect to external roots and the car's *remembered set* (list of incoming references from other cars).
– *Local collection* of garbage *always* results in the reclamation of an entire car. All externally reachable objects within the car are moved to other cars according to the MOS promotion rules, and dead objects are reclaimed with the car.
– *Global identification* of garbage depends on grouping cars into *trains*, migrating any cycle into a single train, and identifying any trains that are not reachable using reference counting.
– *Global collection* of garbage simply involves the removal of unreachable trains in their entirety, including their constituent cars.

### 4.2 The Design of TMOS

Following the design approach outlined in section 3.2, we now address correctness and performance issues for the TMOS algorithm.

**Correctness** In the following sections we describe how TMOS addresses correctness issues relating to each of the four phases of collection. Our approach to each of these rests directly on the templates for assuring correctness described in section 3.2. We therefore only elaborate in areas where TMOS extends or otherwise deviates from those templates. The details of *mechanisms* integral to the correctness strategy for TMOS (highlighted below in bold text) are described in section 4.3. The local garbage identification and collection phases of MOS are tightly coupled and would most likely be implemented as a single operation in a TMOS implementation. However, each of the two components raises separate concerns, so for clarity we describe them here separately.

*Correctness of Local Identification* The critical issue for the local garbage identification phase is having a coherent image of the target car available for the duration of the identification process, in spite of any mutator attempt to update its contents. In section 3.2 we describe a **coherent snapshot mechanism**, a means of effectively implementing this approach.

*Correctness of Local Collection* MOS moves objects as part of its car collection phase, so the biggest issue for TMOS with respect to the correctness of this phase is that of maintaining coherency of mutator addressing in the face of object movements by the collector. We solve this coherency problem by placing a level of **indirection between mutator and store level addressing** and **deferring the movement of objects** accessed (in either read or write mode) by a mutator transaction.

*Correctness of Global Identification* The use of a reference count to detect train isolation in MOS is dependent on identification taking place while the system is in a stable state with respect to the count—i.e., objects are not being moved into or out of the train. MOS uses a *closed* flag to prevent allocation into a train; however, objects can be promoted (copied) into a closed train from older trains. Once a closed train becomes isolated, it will remain isolated, irrespective of mutator activity. The isolation of a closed train can only be undone by the promotion of an object into the train by the collector. Thus in the case of TMOS, correctness of global garbage identification is independent of mutator and transactional activity, but depends solely on the correctness conditions of the MOS algorithm.

*Correctness of Global Collection* Once a train (and its constituent cars) have been identified as garbage in the MOS algorithm, the collection process simply involves recovering space associated with the train and its cars. This has no impact on the mutator (as the mutator can have no references to the cars), and is therefore trivial from a correctness standpoint.

**Performance** We now address how performance issues impact the design of TMOS.

*Responsiveness* Many implementation details impact on responsiveness, but from a design standpoint the dominant concern is minimizing disruptiveness by addressing incrementality and collection opportunism. The first of these indicates minimizing car sizes. The MOS algorithm is flexible in this respect, however local identification is inevitably more efficient than global identification, so reducing car sizes may negatively impact on overall collection efficiency. The second of these indicates the importance of opportunism with respect to I/O by collecting in-memory cars where possible. Together these objectives suggest that a car should be sized to fit into a single *disk page* (the store's unit of transfer between memory and disk). If a car is not contained within a single disk page, opportunistic collection of a car will be conditional on all of the car's pages being memory resident simultaneously. Opportunism with respect to write I/O suggests that pages that have been updated be preferentially targeted for collection.

*Throughput* I/O opportunism will also impact significantly on throughput. Thus I/O related operations that do not impact on responsiveness should also occur opportunistically. A significant class of such operations is pointer and remembered set maintenance. An outgoing pointer referring to an object in another car must be updated if the remote object is moved. Similarly a remembered set entry (recording an incoming pointer from a remote object) must be: updated if the referring object moves; created if a new remote reference is created; or deleted if the referring object's reference is deleted. In the MOS algorithm such maintenance would occur at the time of the change to the remote object; however, in the context where much (perhaps most) of the object space is on disk, such an approach is impractical. To this end, PMOS [Moss et al. 1996], which faced the same problem, introduced the concept of buffering $\Delta ref$ and $\Delta loc$ entries in memory. These entries correspond respectively to mutations of cross-car pointers and movements of objects containing cross-car pointers. Each time a car is brought into memory, unprocessed $\Delta ref$ and $\Delta loc$ entries are applied to the car's objects and remembered set as necessary. If the number of outstanding entries becomes significant (impacting on memory consumption, for example), cars can be pro-actively brought into memory and updated.

Collector efficiency can also be improved and I/O costs further minimized by collecting multiple cars where possible. Multiple car collections improve clustering opportunities when objects are promoted, and by aggregating multiple updates may reduce the number of I/Os that occur. However, multi-car collection is more disruptive to the mutator and so is at odds with the goal of responsiveness.

### 4.3 The Implementation of TMOS

We now discuss key implementation details for TMOS. After briefly outlining the system structure we describe three key aspects of the TMOS implementation: the object addressing scheme, actions initiated by mutator behavior, and garbage collection. The three mechanisms identified in section 4.2, *a coherent snapshot mechanism*, *indirection between mutator and store level addressing*, and *deferring the movement of objects*, are addressed in the following discussion.

**System Structure** Figure 2 indicates the basic system model for the TMOS collector. Significant components include the cache shared between the mutator(s) and the transactional store, the *OID* to *PID* map, and the log, which is used by the collector to acquire object before-images and $\Delta ref$s.

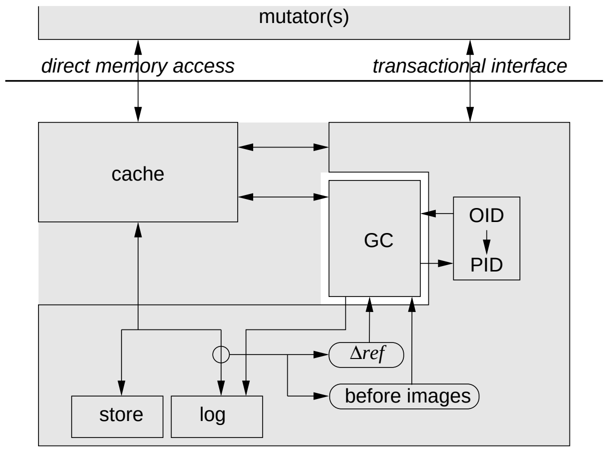

**Fig. 2.** System model for TMOS, indicating relationship between the collector and key sub-components of the transactional store.

**Object Addressing Scheme** As described in section 4.2, decoupling mutator and store level object addressing facilitates transparency of collector-induced object movement. We describe such a scheme in terms of mutator addresses (*OID*s) and store addresses (*PID*s), and mechanisms for maintaining their relationship.

*Mutator-Level Object Addresses* The mutator's notion of an *OID* is that of a value (possibly stored in an object field), which when presented to the transactional interface will uniquely identify a store object for retrieval (or update) through that interface. An *OID* is only valid within the scope of a single transaction. The size of an *OID* is implementation dependent, but would usually be 32 or 64 bits.[7] The process of object retrieval within a transaction is bootstrapped by the retrieval of a root object.[8] Assuming the structure of the root object is known to the mutator, traversal of the object graph can begin by accessing any reference field of the root object and using the *OID* contained within it to retrieve another object. Our means of insulating the mutator from the movement of objects by the collector requires support for a many to one relationship between *OID*s and *physical* memory addresses. This can be achieved with *virtual* memory to *OID* maps being many to one (with indirection via per-transaction maps),

[7] There is a distinct advantage in having the OID the same size as the mutator's reference type, as store objects can be mapped to objects in the mutator space without resizing.

[8] This can be done by using a symbolic *OID* such as `ROOT_OID`.

or one to one (avoiding indirection). The drawback of hardware address translation is that it depends on each concurrent transaction executing within a separate address space and it constrains object relocation choices to those that preserve the object's intra-page location because it can only re-map the high-order bits of the address.

*Store-Level Object Addressing* A *PID* must encode the location of an object within the store. Local and global garbage identification in MOS depends on tracking inter-car and inter-train references. If the train and car of the referenced object could be encoded in the *PID*, it would make the garbage identification process substantially more efficient by avoiding maps to associate *PID*s with trains and cars. We thus compose a *PID* as follows:

$$PID = < TID, CID, index >$$

where *TID* identifies the train containing the object, *CID* identifies the car, and *index* identifies a specific object within the car.

The *PID* and *OID* utilize the same space, so should be the same size. If only 32 bits are available, care must be taken in choosing the width of each of the components of the *PID*. Because MOS is a copying garbage collector, addresses can be recycled.

*OID, PID Mapping* The *OID* to *PID* map is many-to-one and it is dynamic. Because the validity of an *OID* is limited to the scope of a single transaction, different transactions may have different *OID* to *PID* mappings. This makes it possible to manage coherently two images of the same object, seen by different transactions in different parts of memory. Such a feature allows an object to be 'moved' by the collector while a transaction retains a read-lock on that object. The locking transaction and a subsequent reader will see separate identical images of the object (this idea is further motivated below).

**Mutator Activity** Aside from the use of address indirection, the existence of the collector impinges on the relationship between the mutator and the store in three principle ways: the faulting of objects into memory, the generation of $\Delta ref$s by the mutator as a result of changes to references, and the flushing of updated objects back to disk.

*Car Faulting* Faulting an object into memory in response to a mutator request necessitates the faulting of the car containing that object if the car is not already in memory. This process involves three steps:

1. Fault the appropriate store page from disk into memory.
2. Apply any outstanding $\Delta loc$ entries to all objects in the car.
3. Swizzle all references from *PID*s to *OID*s.[9]

At this stage the mutator may access the requested object. However, before the car can be subject to garbage identification, one more step must be performed (this need not be performed if the car is not collected):

4. Apply any outstanding $\Delta ref$ entries to the car's remembered set.

[9] This process can alternatively be performed lazily, swizzling objects only when they are first accessed by the mutator.

*Δref Generation* Any changes made by the mutator to object references must be propagated to the collector if those changes are committed. The creation (or deletion) of an inter-car reference will generate an insertion into (deletion from) the target car's remembered set. As described in section 4.2, such changes are applied immediately only if the target remembered set is in memory, otherwise a Δ*ref* entry is placed on a queue for processing when the target remembered set is faulted in. Because such mutator changes are only relevant when committed, they can be processed as part of the generation of the commit log record (which involves identifying mutator changes and encapsulating them in log Δs).

*Car Flushing* Updated cars ultimately must be flushed to disk. Because references are swizzled as part of the faulting process, the flushing process must unswizzle all references from *OID*s to *PID*s before writing a car to disk.

**Garbage Collection** Finally, and most critically, we now describe the garbage collection process. At both the local (car) and global (train) levels of the algorithm we combine both identification and collection phases in the implementation.

*Car Collection* Car collection can be initiated with respect to any car (or cars). However, before collection can proceed, the car must be in memory, be up to date with respect to Δ*loc*s, in a consistent state (transactionally), and closed to allocation of new objects. If a car is in a swizzled state at the time of collection, each reference will be translated from *OID* to *PID* as it is scanned (*PID*s are used to do reachability analysis). Thus the cost associated with collection will depend on the starting state of the target car.

If a mutator transaction holds a write lock on any object in the target car when collection commences, then that object's before-image is used in place of the object during the garbage identification phase. For the duration of the collection, mutator transactions are prevented from upgrading read locks on any objects and from committing any updates to objects in the car (new read locks are permitted).

The reachability status of all objects in the car is then established using the car's remset. The car is then *evacuated* by copying reachable objects to other cars according to the MOS promotion rules (utilizing the flexibility in the rules to maximize clustering where possible). If no mutator locks are held over the car, the car is reclaimed. As long as mutator locks remain on any objects in the car, those objects remain in place in the evacuated car *and* in the car to which they were promoted. In the case of a read lock, new transactions will be given references to the promoted objects (with two images of the same object co-existing in a read-only state). In the case of a write lock, new transactions will be excluded from accessing the object until the lock is released, and if the transaction commits successfully, the new object state is copied to the car to which to object was promoted. As soon as the last mutator lock is released, the car is reclaimed.

The promotion of each object generates a Δ*loc* for each car referring to the object and a Δ*ref* for each object it refers to. The deletion of an object generates a Δ*ref* for each object it refers to.

*Train Collection* Train collection is substantially simpler than car collection. Train isolation is detected using train reference counts which are trivially maintained as a byproduct of $\Delta ref$ processing (recall that *PID*s encode train identifiers). Once a train is detected as being isolated, it and its constituent cars can be reclaimed. Isolated cars may be on disk, in which case their reclamation may occur lazily. Correctness of the algorithm will depend on either $\Delta ref$ entries being generated for all objects referred to by reclaimed cars, or by records of car reclamation being kept so that any remembered set entries corresponding to reclaimed cars may be disregarded. The former approach implies a need for all reclaimed cars to be scanned (faulted in from disk if necessary), while the latter may require car reclamation records to be kept for very long periods and may involve relatively expensive lookups when scanning remembered sets.

*Recoverability* Collections impact on five aspects of store state: remembered sets (via $\Delta ref$s), object pointers (via $\Delta loc$s), meta-data describing train membership, meta-data describing car membership, and low-level space allocation meta-data. By maintaining all such data structures (e.g., table of outstanding $\Delta ref$s, low-level space allocation map, etc.) as low-level persistent data in the two-level store, the data structures become fully recoverable. By making each collection invocation a single low-level transaction, the effects of each invocation trivially take on transactional recovery properties.

## 5 Future Work

The TMOS garbage collector is currently being implemented as part of Platypus [He et al. 2000], an object store that has been developed at the Australian National University. Once completed, the TMOS garbage collector implementation will be evaluated. The TMOS collector embodies a high degree of policy flexibility for both allocation and collection. We see the exploration of this policy space and the identification of appropriate heuristics as important research goals. The flexibility of the TMOS collector also provides opportunities for exploring clustering algorithms. We see an implementation of TMOS in Platypus as an excellent vehicle for furthering work on clustering [He et al. 2000].

## 6 Conclusions

In addition to minimizing disk usage, garbage collection is likely to be an important mechanism for maximizing long term I/O performance of persistent systems. TMOS is a garbage collector that applies the completeness and incrementality of the mature object space (MOS) collector to a transactional storage context. The complexity of designing a garbage collection algorithm for a transactional setting has led us to develop a generic methodology for the design of transactional collectors. The methodology begins with an initial garbage collection algorithm that is known to be correct and a mutator and a store that are strongly separated by a transactional interface. Correctness and performance issues are clearly separated allowing the characteristics of the newly derived transactional garbage collection algorithm to be well understood. This paper thus contributes both a new garbage collection algorithm and a generic methodology for designing transactional garbage collection algorithms.

# Bibliography

[Amsaleg et al. 1995] AMSALEG, L., FRANKLIN, M. J., AND GRUBER, O. 1995. Efficient incremental garbage collection for client-server object database systems. In U. DAYAL, P. M. D. GRAY, AND S. NISHIO Eds., *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland* (1995), pp. 42–53. Morgan Kaufmann.

[Atkinson and Morrison 1995] ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent systems. *The VLDB Journal 4*, 3 (July), 319–402.

[Blackburn and Stanton 1999] BLACKBURN, S. M. AND STANTON, R. B. 1999. The transactional object cache: A foundation for high performance persistent system construction. In R. MORRISON, M. JORDAN, AND M. ATKINSON Eds., *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems, Aug. 30–Sept. 1, 1998, Tiburon, CA, U.S.A.* (San Francisco, 1999), pp. 37–50. Morgan Kaufmann.

[Blackburn and Zigman 1999] BLACKBURN, S. M. AND ZIGMAN, J. N. 1999. Concurrency—The fly in the ointment? In R. MORRISON, M. JORDAN, AND M. ATKINSON Eds., *Advances in Persistent Object Systems: Third International Workshop on Persistence and Java, Sept. 1–3, 1998, Tiburon, CA, U.S.A.* (San Francisco, 1999), pp. 250–258. Morgan Kaufmann.

[Cattell et al. 2000] CATTELL, R. G. G., BARRY, D. K., BERLER, M., EASTMAN, J., JORDAN, D., RUSSELL, C., SCHADOW, O., STANIENDA, T., AND VELEZ, F. Eds. 2000. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers.

[Cook et al. 1994] COOK, J. E., WOLF, A. L., AND ZORN, B. G. 1994. Partition selection policies in object database garbage collection. In R. T. SNODGRASS AND M. WINSLETT Eds., *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994* (1994), pp. 371–382. ACM Press.

[Dijkstra et al. 1978] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. 1978. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM 21*, 11, 966–975.

[Härder and Reuter 1983] HÄRDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys 15*, 4 (Dec.), 287–317.

[He et al. 2000] HE, Z., BLACKBURN, S. M., KIRBY, L., AND ZIGMAN, J. N. 2000. Platypus: Design and implementation of a flexible high performance object store. In *Proceedings of the Ninth International Workshop on Persistent Object Systems, Lillehammer, Norway September 6–9, 2000* (2000).

[He et al. 2000] HE, Z., MARQUEZ, A., AND BLACKBURN, S. M. 2000. Opportunistic prioritised clustering framework (OPCF). In *ECOOP2000 Symposium on Objects and Databases - Object-Oriented Programming, Sophia Antipolis, France, June 13, 2000, Proceedings*, Lecture Notes in Computer Science (LNCS) (2000). Springer. To appear.

[Hudson and Moss 1992] HUDSON, R. AND MOSS, J. E. B. 1992. Incremental garbage collection of mature objects. In Y. BEKKERS AND J. COHEN Eds., *International Workshop on Memory Management, St. Malo, France Sept. 17–19, 1992*, Volume 637 of *Lecture Notes in Computer Science (LNCS)* (1992, 1992), pp. 388–403. Springer.

[Kolodner and Weihl 1993] KOLODNER, E. K. AND WEIHL, W. E. 1993. Atomic incremental garbage collection and recovery for large stable heap. In P. BUNEMAN AND S. JAJODIA Eds., *SIGMOD 1993, Proceedings ACM SIGMOD International Conference on the Management of Data, May 26-28, Washington, DC*, Volume 22 of *SIGMOD Record* (June 1993), pp. 177–186. ACM Press.

[Lakhamraju et al. 2000] LAKHAMRAJU, M. K., RASTOGI, R., SESHADRI, S., AND SUDARSHAN, S. 2000. On-line reorganization in object databases. In *SIGMOD 2000, Proceedings ACM SIGMOD International Conference on Management of Data, May 14-19, 2000, Dallas, Texas, USA*, Volume 28 of *SIGMOD Record* (May 2000). ACM Press.

[Maheshwari and Liskov 1997] MAHESHWARI, U. AND LISKOV, B. 1997. Partitioned garbage collection of large object store. In J. PECKHAM Ed., *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, Volume 26 of *SIGMOD Record* (June 1997), pp. 313–323. ACM Press.

[Maier 1999] MAIER, D. 1999. Review - An approach to persistent programming. *ACM SIGMOD Digital Review 1*.

[Moss et al. 1986] MOSS, J. E. B., GRIFFETH, N. D., AND GRAHAM, M. H. 1986. Abstraction in recovery management. In C. ZANIOLO Ed., *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, Volume 15 of *SIGMOD Record* (June 1986), pp. 72–83. ACM Press.

[Moss et al. 1996] MOSS, J. E. B., MUNRO, D. S., AND HUDSON, R. L. 1996. PMOS: A complete and coarse-grained incremental garbage collector. In R. CONNOR AND S. NETTLES Eds., *Seventh* (Cape May, NJ, U.S.A., May 29–31 1996), pp. 140–150. Morgan Kaufmann.

[Munro et al. 1999] MUNRO, D. S., BROWN, A. L., MORRISON, R., AND MOSS, J. E. B. 1999. Incremental garbage collection of a persistent object store using PMOS. In R. MORRISON, M. J. JORDAN, AND M. P. ATKINSON Eds., *Advances in Persistent Object Systems: Eigth International Workshop on Persistence Object SystemsSept. 1–3, 1998, Tiburon, CA, U.S.A.* (San Francisco, 1999), pp. 78–91. Morgan Kaufmann.

[Skubiszewski and Valduriez 1997] SKUBISZEWSKI, M. AND VALDURIEZ, P. 1997. Concurrent garbage collection in o2. In M. JARKE, M. J. CAREY, K. R. DITTRICH, F. H. LOCHOVSKY, P. LOUCOPOULOS, AND M. A. JEUSFELD Eds., *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece* (1997), pp. 356–365. Morgan Kaufmann.

[Snodgrass 1999] SNODGRASS, R. T. 1999. Review - PM3: An orthogonal persistent systems programming language - design, implementation, performance. *ACM SIGMOD Digital Review 1*.

[Sun Microsystems 1999] SUN MICROSYSTEMS. 1999. Java Data Objects Specification, JSR-12. http://java.sun.com/aboutjava/communityprocess/jsr (July), Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

[Weikum 1991] WEIKUM, G. 1991. Principles and realization strategies of multilevel transaction management. *TODS 16*, 1, 132–180.

[Yong et al. 1994] YONG, V.-F., NAUGHTON, J., AND YU, J.-B. 1994. Storage reclamation and reorganization in client-server persistent stores. In *Proceedings of the 10th International Conference on Data Engineering*, IEEE Computing Society (Feb. 1994), pp. 120–131.