

TMOS: A Transactional Garbage Collector

John N. Zigman, Stephen M. Blackburn *

Abstract

The safe management of storage reclamation is a difficult and error prone problem, diverting programmer effort from the core task. Garbage collection is an effective solution to this problem, that can be applied to both primary and secondary storage.

The Mature Object Space (MOS) garbage collector is targeted toward long lived in memory objects and its derivative Persistent MOS (PMOS) is targeted toward the collection of long lived secondary storage. Resource reclamation is essential when a limited of resources are available (e.g. main memory). However, garbage collection in secondary storage systems has the side effect of improving locality and thus should improve data access performance. In addition, the MOS family of garbage collectors have some inherent reclustering characteristics.

It is common to control concurrent access to data held in long term storage by using a transaction mechanism. The transaction mechanism enforces a data consistency model and enables concurrent access to data. Through atomicity, a transaction encapsulates a set of modifications which transform the store from one coherent state to another. A garbage collector operating in this context must be aware of the transactional nature of secondary storage. However, MOS and PMOS are not designed to operate in such a setting.

TMOS is a new member of the MOS family of collectors, building on the characteristics of the MOS collector. TMOS operates on secondary storage in a two level transactional environment that allows the movement of objects and the collection of garbage without interfering with the user level transactions. Furthermore, it separates the operation of primary and secondary storage collection, facilitating the implementation of different mechanisms and policies. This paper describes the operational characteristics of the TMOS collector and the principle mechanisms that are needed to enable its behaviour.

1 Introduction

In addition to reclaiming used space, garbage collection can improve object locality either by allowing the recovered space between objects to be used for object allocation or by actively defragmenting the resulting space.

The data held in a system that can potentially be used is referred to as live data, the remain data which can not be used is referred to as garbage. The determination of which is live and which is garbage can be characterised by its *reachability*. Data that can be referred to by some set of pointer traversals from a known point (or root) is reachable, all other data is unreachable and therefore garbage.

This principle holds true for secondary storage, where live data is considered to be anything reachable from a particular point (root or roots). The integration of language and secondary storage through orthogonal persistence or the ODMG 3.0 [Cattell et al. 2000] standard for object persistence makes garbage collection more desirable.

Atkinson and Morrison [Atkinson and Morrison 1995] identify three *principles of orthogonal persistence*. A well-understood implication of the ‘principle of persistence identification’ is that the persistence of data be defined in terms of reachability—‘persistence by reachability’ or ‘transitive persistence’. This model of persistence rests on the identification of one or more ‘roots of persistence’ from which any transitively reachable data must remain reachable beyond the lifetime of a program execution (i.e. it must persist). Garbage collection is thus a key technology for orthogonally persistent systems.

The ODMG 3.0 standard defines a type-dependent form of persistent by reachability. A subset of the types declared can be made persistence capable. An object is made persistent if it is of a persistent capable type, and it is referenced by a root or from a persistence capable object that is itself to be made persistent. This form of persistence also implies the use of automatic storage reclamation.

The MOS Collector The collection of an entire object space in a single unit of work can be disruptive, particularly as the size of the object space increases. The disruptiveness of the collector can be reduced by partitioning the collection space into smaller units. The MOS[Hudson and Moss 1992] collector is partitioned in to *cars*. As a result, it is possible that cycle of garbage objects may be larger than the collection partition, as such it would become self sustaining. To collect large cycles of garbage objects, a

*The author wishes to acknowledge that this work was carried out within the Cooperative Research Center for Advanced Computational Systems established under the Australian Government’s Cooperative Research Centers Program.

grouping of cars into trains is introduced. A train can be collected when no references exist originating outside the train which refer to objects inside the train, i.e., when a train is isolated it can be collected.

The set of cars associated with a particular train are ordered from oldest to youngest. Each invocation of the garbage collector processes the oldest car of the oldest train. The objects within the car are evacuated according to the following promotion rules:

- An object reachable from a root is copied to a younger train, adding a car to that train if required.
- An object reachable from a younger train(s) is copied to one of those younger trains, adding a car if required.
- The remaining data is unreachable.

The car is then removed from the system. These rules maintain a forward movement of objects that are reachable, and objects that are known to be unreachable will be reclaimed and it will be apparent that the remaining objects are unreachable, at which point they can be collected.

Transactions are a mechanism for controlling concurrent access to data. Such mechanisms are commonly used in relational, object relational and object oriented databases. Each application-level transaction encapsulates a set of object reads, writes and creates into a single atomic action. The transaction is either applied to the object store or discarded. This along with isolation (objects referred to do not leak from one transaction to another) transform the store from one coherent state to another.

The transformation from one coherent state to another enables specific recoverable coherent states for the store. These points of durability cannot be undone by any means except subsequent transactions. These properties must be considered when incorporating and garbage collection system into a transaction store.

2 Operation

The operations that the mutator¹ and garbage collector perform determine how they interact, which in turn can impose operational constraints upon the system. The mutator can create objects and alter the data contained within objects. By altering object references, the mutator can change the relationships between the objects in the system. Thus, the mutator may cause sections of the object graph to become unreachable, i.e., garbage.

The garbage collector can remove objects that are unreachable by the mutator and move the objects within the store. However, the object graph of reachable objects is never changed by the garbage collector. As such, the mutator views the operation of the garbage collector as a logical no-op.

2.1 Once Garbage Always Garbage?

An invariant that underpins all garbage collection algorithms is that garbage is a stable property. This follows as a result of garbage being defined as unreachable data. If data is unreachable, then no reference exists to it from any live object, therefore there is no means by which it can be reconnected to the object graph.

This invariant is subverted by the introduction of *undoability* through the atomicity provided by transactions. Thus, in transactional systems, garbage (at least as seen from within a transaction) is no longer a stable property. The specific cases of this problem are dealt with by Amsaleg, Franklin, and Gruber [1995].

In TMOS we avoid this problem by only viewing durable versions of the object graph (i.e. those that can never be revoked) and reinterpreting the axiom *once garbage always garbage* in this context as *once durable garbage always garbage*. We note that any object that is detected as durable garbage in one view of the object graph is garbage in every subsequent view.

2.2 Object Referencing

Each object within the object store is referred to by a persistent identifier (PID). The PID can either refer to an entry in an index table (thus translating a PID to the actual location of the object) or it can encode the actual location of the object within the store.

An index or indirection table will be proportional to the number of objects held in the store, and may be too large to hold in main memory. This would likely result in accessing the disk to find where the object is, and again to retrieve the object. Furthermore, the index would be modified frequently as objects are created, destroyed and moved, thus becoming a performance hot spot. By contrast encoding the actual location of the object in the PID removes the need to perform extra I/O to determine where the object resides.

2.3 Incremental Collection

The MOS family of garbage collectors manage the disruptiveness of collection by reducing the primary collection process to a series of smaller increments. Each increment corresponds to the collection of a car. To facilitate this, process external graph information is needed, described below.

¹A mutator is a process(es) which manipulates the data within the system, such as an application.

Remembered Sets To process a car, each object within the car has the MOS promotion rules applied to it. This is achieved efficiently by maintaining the reachability information (called a remembered set) within each car. The remembered set for a car contains a set of references from other cars to objects within that car. Consequently, the addition or removal of a reference to an object external to a car to an object within that car may necessitate the update of that car's remembered set.

The movement of an object (when using direct references) requires all the references to that object to be updated. In this instance the remembered sets are used to determine what corrections need to be made.

Efficient I/O The MOS algorithm applies a strict ordering to the collection of cars (from oldest to newest in the oldest available train). This ordering reduces the remembered set to include only summary information from newer cars to older cars. However, the cars retrieved from the store to satisfy the mutator operation may not satisfy the restrictive collection pattern.

Both PMOS[Moss et al. 1996] and TMOS remove this restriction by maintaining complete remembered set information, and introducing the promotion rule:

- An object reachable from an older train is copied to any other car of its current train, adding a car if required.

Using PIDs that encode object location results in the PID of an object changing when the object is moved. Consequently all the objects that refer to the object must have their object references corrected. This potentially increases the amount of I/O associated with garbage collection.

The normal operation of the mutator will change object reference values which in turn require updating of the remembered sets of some cars.

The impact of I/O can be reduced to take advantage of the cars that are loaded into memory. Maintaining two in memory sets (one for remembered set changes Δrem and the other for PID changes Δloc) allow changes to be applied to cars when they are available.

2.4 Clustering

The promotion rules which govern the MOS style garbage collectors tend to cluster objects by reachability. This process should, in general, improve the general access characteristics of an object store.

The underlying principle of the promotion rules is to guarantee the completeness of the collection process. This is satisfied by promoting objects to a newer train if possible or the same train (but never to an older train). The forward movement of objects in time combined with the guarantee to collect every car ensures completeness.

The promotion rules allow significant scope for incorporating a reclustering algorithm. It is arguable that upon this basis, garbage collection, particularly with good reclustering techniques, should improve the I/O performance of a system.

3 System Structure

The components of the system, depicted in Figure 1, are grouped into the different system levels. The store and the relationship between the store and the mutator is explored in the following sections.

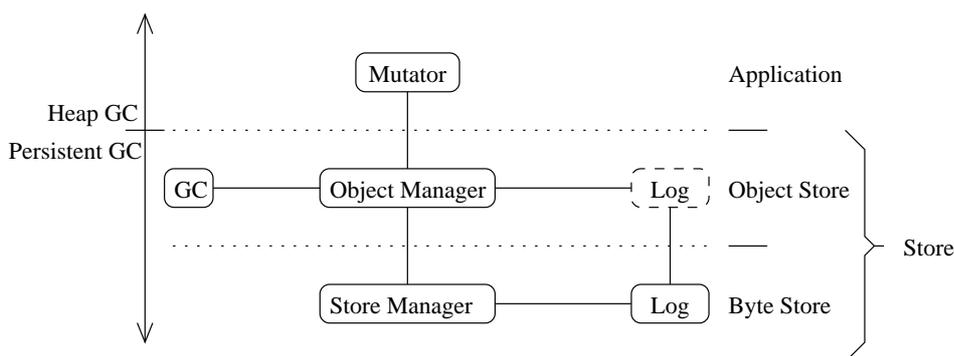


Figure 1: System Components

3.1 Two Level Store

The store consists of two logically distinct levels. The bottom level is a transactional byte store, which is an unstructured byte space. The top level is the transactional object store, which imposes a structure on the underlying byte store. The two level are outlined in the following sections.

Object Store The object manager must: understand the format of an object (a limited form of type information), in particular it must understand which parts of an object are references; the layout of objects and meta-data in the byte store; manage object level concurrency issues; generating object log entries; object level recovery from failure.

The object manager is responsible for responding to requests from the mutator. The mutator (via the object level) can: start a transaction, request an object, request write permission on an object, create new objects and request a transaction be committed or rolled back. Transactions at this level are translated into lower level transactions to be performed on the underlying byte store.

The garbage collector interacts with the object manager. However, unlike the mutator, it processes objects grouped in cars (a car would normally be one or more pages from the byte store). The object manager can provide a car of objects (this may be constructed from an older view of the objects) and the garbage collector issues a series of transactions to move the objects.

From the mutator and garbage collector transactions, the object manager derives the information necessary to produce the Δref and Δloc sets mentioned earlier. These changes are then applied by the object manager issuing small correcting transactions. Thus, in the event of failure, the in-memory sets can be reconstructed from the logs and the system can continue as normal.

Byte Store The byte store is a continuous space of bytes partitioned into pages. It can (through a transaction mechanism) read and update pages. These modifications are recorded in a log for recovery purposes.

The store recovery involves the recovery of the byte store, followed by the recovery of the object store level.

3.2 Separating the Mutator and Store

The use of an indirection mechanism allows the store mutator to move the objects around the object store without interfering with the operation of the mutator. The indirection mechanism can be implemented by global (discussed in Section 2.2) or local indirection.

The use of identifiers that encode an object's store location where the mutator can copy the identifier values is at odds with the garbage collector being able to move objects. This problem can be overcome by swizzling object references to local identifiers. This identifier would only have meaning within the context of a particular object manager (and may vary from one transaction to another). Thus, a local in-memory indirection table maintains a mapping between the logical identifiers used by the mutator and the internally used identifiers. This frees the object manager and garbage collector to relocate objects and modify their internal references without interfering with the mutator.

Cars containing modified data must be unswizzled and written to disk. This presents an opportunity to perform garbage collection upon the car, which can reduce the impact of the collection process.

4 Conclusion

This paper has outlined TMOS, a new member of the MOS family of garbage collectors. TMOS preserves the MOS family attributes: incrementality, non-disruptiveness, completeness, etc.

The notion of *durable garbage is always garbage* is used to define a view of the store so that garbage may be collected safely (albeit conservatively).

A local indirection mechanism allows the heap collection and store collection mechanisms and policies to be separated. Combined with a two level transactional system it allows the reorganization of the objects in the store, without interference between the mutator and garbage collectors.

Bibliography

- 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 the 21th International Conference on Very Large Data Bases* (Zürich, Switzerland, September 11–15 1995). Morgan Kaufmann.
- ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent systems. *The VLDB Journal* 4, 3 (July), 319–402.
- 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 Database Standard: ODMG 3.0*. Morgan Kaufman.
- HUDSON, R. L. AND MOSS, J. E. B. 1992. Incremental garbage collection for mature objects. In Y. BEKKERS AND J. COHEN Eds., *Proceedings of the International Workshop on Memory Management*, Number 637 in Lecture Notes in Computer Science (LNCS) (St Malo, France, Sept. 17–19 1992), pp. 388–403. Springer-Verlag.
- 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 International Workshop on Persistent Object Systems* (Cape May, NJ, U.S.A., May 29–31 1996), pp. 140–150. Morgan Kaufmann.