Incremental Garbage Collection in Massive Object Stores

Fred Brown
Department of Computer Science
University of Adelaide
South Australia 5005
AUSTRALIA
fred@cs.adelaide.edu.au

Abstract

There are only a few garbage collection algorithms that have been designed to operate over massive object stores. These algorithms operate at two levels, locally via incremental collection of small partitions and globally via detection of cross partition garbage, including cyclic garbage. At each level there is a choice of collection mechanism. For example, the PMOS collector employs tracing at the local level and reference counting at the global level. Another approach implemented in the Thor object database uses tracing at both levels. In this paper we present two new algorithms that both employ reference counting at the local level. One algorithm uses reference counting at the higher level and the other uses tracing at the higher level. An evaluation strategy is presented to support comparisons between these four algorithms and preliminary experiments are outlined.

1. Introduction

The garbage collection of massive object stores gives rise to a number of challenges not faced by main memory collectors [1,13,14]:

- The scale of the object store precludes a stop-the-world algorithm due to the time required. Consequently, some form of incremental collection strategy is essential. This may also require the collector to cooperate with applications manipulating the object store.
- It is highly likely that a massive object store will be part of a database and require protection from crashes. So, crash recovery may need to be integrated with the garbage collection algorithms employed.
- The scale of the object store may result in most of the object store residing on disk rather than primary memory. Therefore, care is required in order to minimise the disk I/O incurred by garbage collection.
- The scale of the object store is such that a complete garbage collection may take a very long time. It would therefore be desirable to make maximum progress on each incremental collection. This may require careful control over the order and timing of incremental collections.

In meeting these challenges, garbage collection algorithms must also ensure that they are safe and complete. A safe algorithm will never garbage collect a live object and a complete algorithm will eventually collect all garbage. Therefore a garbage collection algorithm suitable for use in a massive object store should exhibit the following properties.

- The object store will be divided, either physically or logically, into partitions that can be garbage collected independently of other partitions.
- The collection algorithm will take account of the application programs concurrently manipulating the object store.
- Cross partition and cyclic garbage will be detected via the synthesis of appropriate global knowledge.
- Crash recovery will be directly supported if the underlying object store does not provide this as an orthogonal mechanism.
- The algorithm's supporting data structures will be carefully chosen in order to minimise the I/O overheads incurred during a collection.
- A number of different policies will be supported to guide the selection of which partition to collect next.
- The collection algorithm will be safe and complete.

1.1. Algorithm taxonomy

A taxonomy of suitable garbage collection algorithms could be defined in terms of how partitions are collected and how the necessary global knowledge is synthesised. In both dimensions of the taxonomy there is a choice between two major approaches, reference counting or tracing.

Reference counting involves associating a counter with every object that records the total number of pointers referring to the object. If an object's reference counter is 0, the object is no longer in use, it is considered garbage and its storage is reclaimed. Within a partition reference counting can be effective at detecting acyclic garbage but is incapable of detecting cyclic garbage. Reference counting will require the overhead of a counter for every object in the object store.

Tracing involves following all pointer fields starting from one or more roots and identifying which objects can still be used. All objects not encountered are considered garbage and their storage is reclaimed. Tracing may involve marking all reachable objects and performing a sweep to reclaim the garbage or it may involve copying all reachable objects and reclaiming all uncopied objects.

If tracing is used, then all cyclic garbage completely contained within the partition being collected can be detected and reclaimed. However, every partition must maintain some form of remembered set to identify which objects within a partition are reachable from elsewhere. The remembered set forms the roots for the local tracing of a partition. Popular objects, those referenced by large numbers of objects in other partitions may cause severe problems for some remembered set implementation strategies.

At the global level it is necessary to synthesise sufficient global knowledge to be able to detect cross partition garbage and cross partition cyclic garbage. The partitions that are incrementally collected will be referred to as local partitions from now on. The synthesis of global knowledge can be achieved by associating objects with global partitions such that cyclic garbage is associated with global partitions that are not referenced by the rest of the object store. The identification of these garbage global partitions can then be achieved by reference counting or tracing references between global partitions. The global partitions need not be related to the local partitions in any way.

An additional dimension to the taxonomy could be the addressing mechanism. Direct addressing of objects has some potential efficiency benefits for application behaviour but can cause severe problems for the store implementation if objects must be physically moved between partitions. The task of identifying which references must be corrected when an object moves may require some form of remembered set to be maintained. This can be problematic if an object store contains a significant number of popular objects that may need to be moved. To avoid many of these problems most garbage collection algorithms designed for massive object stores require some form of indirect addressing mechanism to be used.

For the purposes of this paper the addressing dimension of the taxonomy will be ignored. Therefore we can classify garbage collection algorithms for massive object stores into four categories:

1. Local partitions are collected by tracing,
   global partitions are collected by reference counting.
2. Local partitions are collected by tracing,
   global partitions are collected by tracing.
3. Local partitions are collected by reference counting,
   global partitions are collected by reference counting.
4. Local partitions are collected by reference counting,
   global partitions are collected by tracing.

2. Existing algorithms

There are only a few garbage collection algorithms that have been designed to operate over massive object stores [1,13,14]. Each of these algorithms is based on dividing the object store into partitions which may be garbage collected without interfering with running applications. This is supplemented by an additional mechanism that is able to synthesise global knowledge in order to detect cyclic and cross-partition garbage. Support for crash recovery is delegated to a lower level mechanism and is typically not addressed by the algorithms. Attempts to minimise the I/O induced by garbage collection do appear in these algorithms but they are not based on empirical evidence and their effectiveness has not been evaluated. Finally, none of the existing algorithms directly address the issue of ensuring maximum progress. This issue is either not addressed or left to higher level policies that have yet to be evaluated.

2.1. PMOS

The Persistent Mature Object Space algorithm, PMOS [14], is an example of an algorithm from the first category. PMOS is an extension of the Mature Object Space algorithm [8] which divides the object store into fixed size local partitions known as cars which are members of time ordered global partitions known as trains. Every car has a remembered set that, for each externally referenced object, identifies which other cars point to the object. A reference counter is kept for each train so that it can be discarded when none of the train's objects are referenced from other trains.

The garbage collection of a car involves copying all objects reachable from the remembered set, or program roots, to other cars. The evacuated car is then discarded along with the remaining unreachable objects. A reachable object pointed to from a younger train is copied to a car in the younger train. If an object is pointed to from an older train or its own train, it will be copied to another car in its own train. Since objects can only be copied to cars in the same or a newer train and objects are only copied to trains that refer to them, cycles of garbage will eventually congregate in the youngest train that contained any part of the cycle. Once all live objects have been copied from the train containing the cycle, the cross-train reference counting will cause the entire train containing the cycle to be reclaimed.

To ensure that PMOS is complete, the policies that select cars for garbage collection must guarantee that they will eventually select every car and that progress will be made. See [11] for details. In addition to being complete the PMOS algorithm is also safe.

The remembered set implementation strategy has been designed with the aim of minimising disk I/O. Where possible the remembered set for a car is stored inside the car and updates are not applied until the car is brought into memory. However, no empirical studies have been yet undertaken to establish the effectiveness of this design.

The use of remembered sets enables PMOS to support the direct addressing of objects. When an object is moved to a new car, the remembered set identifies which cars contain references that need to be updated. However, the remembered set implementation can become very expensive in the presence of popular objects. For example, global objects referenced by pointer literals could be potentially referenced by every car in the object store. Similar problems could occur in object-oriented systems with static members of a class being referenced by every instance of their class. To overcome these difficulties objects that can never become garbage could be segregated as in the existing implementations [11]. However, this does not guarantee that real applications will not create their own popular objects and suffer severe performance penalties as a result.

Although PMOS is both safe and complete it is unclear how much work is required to collect all garbage in an object store. For example, collecting a doubly linked list of garbage spread over N cars could take up to O(N^2) collections before the list is in a single train and some arbitrary number of additional collections before all reachable objects were copied out of that single train so that it could be reclaimed. To address the issue of making maximum progress some experiments have been conducted into partition selection policies for PMOS [12]. However, it is still too early to identify any particular policy that is guaranteed to be more effective than random selection.

2.2. Thor

An example of an algorithm from the second category has been implemented as part of the Thor object database [9,13].

This algorithm is based on local partitions that can be formed from arbitrary numbers of disk pages. Each page provides an indirect addressing mechanism so that internal fragmentation can be managed by relocating objects within a page. Each partition maintains a remembered set that identifies cross-partition references.

The garbage collection of a partition involves two marking phases. The first phase propagates global mark bits starting from program roots and remembered set entries due to globally marked objects. If this phase finds references to objects in other partitions then the remembered set entries of those objects are globally marked. The second phase does a local trace of all objects reachable from the remembered set and discards any objects not found by either phase. Thus internal garbage, including cycles, can be collected immediately.

Cross-partition cyclic garbage cannot be detected until the global marking has been propagated to the entire object store. This can take a long time since marking a remembered set entry causes the collector to revisit the remembered set's partition. The maximum number of partition visits is proportional to the number of partitions times the maximum number of inter-partition references from any object to the root of the object store.

At the end of a global marking phase the only unmarked objects in the object store are part of cycles of cross-partition garbage. So, during the first visit to each partition during a global marking phase, all unmarked objects are immediately deleted. In effect the global marking divides the object store into two global partitions membership of which is denoted by the mark bits.

This algorithm attempts to minimise unnecessary disk I/O using a particular implementation of remembered sets. In this scheme all entries from a particular car to another car are kept together. In effect, each partition has a list of in-references and out-references. To further reduce I/O overheads, updates to these lists are delayed and performed in batches. Some performance analysis has been undertaken on the effectiveness of this strategy [13]. It should be noted however, that this remembered set implementation strategy is also subject to problems with popular objects.

The global marking strategy ensures that this algorithm is both complete and safe. However, as noted above it can take a long time before any cyclic cross-partition garbage can be reclaimed. Even after a global marking phase has been completed every partition that contained any form of cross-partition cyclic garbage must be revisited in order to reclaim it. In contrast the PMOS collector can identify cyclic garbage as soon as it is isolated within a single train. As far as we are aware no partition selection policy experiments have been applied to this algorithm.

2.3. Exodus

Another example of an algorithm from the second category has been used with the EXODUS storage manager [1,7]. This algorithm design takes the view that cyclic cross-partition garbage will be rare and if there is any, it can be collected by employing an additional garbage collection algorithm. However, the algorithm does make some significant contributions in the area of safe interaction with transactions. For example, all overwritten references are retained in a pruned references table until the transaction performing the overwrite commits. Also all new objects are retained in a created object table until the transaction creating them commits. Both tables prevent objects being reclaimed that may subsequently be found to be live due to a transaction commit or abort. Although this algorithm is safe it is not complete and will not be considered further.

2.4. Summary

The existing algorithms described above all rely on remembered sets to track cross partition references and to support local partition garbage collection. In the case of PMOS, the remembered set is essential in identifying references that must be updated when an object is moved. As noted above popular objects may cause severe problems for remembered set implementations.

A further area of difficulty is identifying an effective partition selection policy to guide the order of incremental collections. To be effective a partition selection policy must perform significantly better than random selection.

3. Reference counting algorithms

We are not aware of any examples of algorithms from categories three and four of the taxonomy that have been specifically designed for use with massive object stores. These categories employ reference counting to identify garbage within local partitions.

Two commonly stated reasons for not using reference counting for massive object stores is the overhead of maintaining reference counting in the presence of transactions and the inability of reference counting to detect cyclic garbage [1].

The first difficulty is an assumed performance issue that may or may not be significant. As will be noted later, the issue of performance evaluation is sufficiently complex that the contribution of any particular implementation strategy to overall system performance cannot be predicted. Therefore, the use of reference counting to collect local partitions should not be discounted without investigating its actual behaviour in a real system.

3.1. Collecting cyclic garbage

The second difficulty, an inability to collect cyclic garbage, can be overcome with an appropriate global partitioning mechanism. The existing reference counting algorithms that can collect cycles all use an additional mechanism to identify the cycles.

The earliest attempts relied on programmers identifying the boundaries between cycles so that each cycle effectively lives in its own partition [4]. Reference counting could then be employed to collect entire partitions. Relying on programmer input is potentially error prone, so later attempts relied on compiler hints to partition objects. However, this approach was only used in functional programming systems where referential integrity and the absence of assignment ensured that partition hints remained accurate [10]. The new algorithms presented in this paper also rely on partitioning objects but they work in the general case and are not dependent on programmer or compiler hints.

The alternative approach to partitioning is to employ tracing algorithms to ensure completeness [10]. However, this global tracing could prove prohibitively expensive in the case of a massive object store.

3.2. Remembered sets

A potential weakness of the algorithms based on local tracing is that they need to maintain remembered sets for each local partition. In the presence of popular objects these remembered sets could become very large and costly to maintain. The use of reference counting eliminates the need for remembered sets, assuming they need not be retained to support object relocation. The new algorithms presented below assume indirect addressing is used so that remembered sets can be discarded.

3.3. Partition selection policies

A major challenge in ensuring maximum progress in reclaiming storage is identifying where most garbage is located [6]. With the algorithm examples given above, an additional mechanism is required to predict where garbage may be located. In contrast, reference counting identifies the location of some acyclic garbage immediately it becomes garbage. Therefore, reference counting may provide an excellent starting point for a partition selection policy. However, an additional mechanism is still required to sort local partitions in terms of their obvious garbage content.

The potential success of a partition selection policy based on reference counting is dependent on other factors. For example, if there is a large amount of cyclic garbage in an object store, the progress of the global repartitioning required to detect it may prove more important than the rate at which acyclic garbage is detected. Of particular importance is the correlation between partitions with obvious acyclic garbage and those that should be chosen to obtain maximum overall progress in reclaiming garbage.

3.4. I/O considerations

The I/O overheads incurred by the garbage collection of a massive object store are a major consideration. A potential difficulty with reference counting algorithms is the need to continually modify reference counts on objects that may not be directly involved in a computation. The extent of this difficulty will depend on where the reference counts are located. It may be desirable to keep reference counts in a central location rather than with the objects. If indirect addressing is being used the reference counts could form part of the address tables used to locate the objects.

Although the overheads of maintaining reference counts are significant, in the context of a massive object store, these overheads must be offset against the alternative costs involved in maintaining remembered sets. Remembered sets can also incur significant overheads especially in a system with large numbers of popular objects. Therefore the choice of garbage collection algorithm for a particular system must be based on a sound understanding of the data and applications it must support. The evaluation approach given below is one way in which this understanding could be realised.

4. The new algorithms

The two new algorithms presented in this paper have been designed to operate in the same environment as the PMOS collector. In fact they were derived whilst attempting to overcome problems encountered in implementing PMOS. It is assumed that applications operate on copies of objects, held in an object cache, that are returned to the object store prior to an object store checkpoint. To support incremental collections of the object store, the object cache supplies a list of all object pointers it holds when a collection is requested. The object store does not support user transactions or concurrency control. The implementation of these issues is left to the higher levels of the architecture. Crash recovery is handled by the underlying buffering mechanisms that can be instructed to perform a checkpoint of the object store. Consequently, the new algorithms do not need to consider crash recovery, concurrency or transactions.

Both of the new algorithms employ the same implementation for object addressing, reference counting, local partitioning and the identification of global partitions.

4.1. Indirect addressing

All object addresses are object numbers that are used to index a multi-level object table. The initial object table is made up of four levels of table each with 256 entries. All entries in the leaf nodes of the table contain a triple consisting of an object's real address in the object store, an object's reference count and an object's train number. The global partitioning used by these algorithms is a simulation of the train partitioning used in PMOS, consequently the global partitions are referred to as trains.

The process of addressing an object is performed using the following steps. First, the top 8 bits of the object's number are used to index the top level of the object table. Then the next 8 bits are used to index the next level of the object table and so on until the object's triple is located. The object may then be directly addressed or its reference count or train number may be manipulated.

4.2. Reference counting

Whenever the object store is asked to update the pointer field of an object two reference count manipulations may occur. First, the object identified by the reference being overwritten has its reference count decremented by 1. If the reference count has become 0 the object is now garbage and a counter associated with the object's partition is incremented by the size of the object. Second, the object identified by the reference being written has its reference count incremented by 1. As an optimisation, the object's reference counter does not record pointers from an object to itself.

4.3. Local partitions

The object store is divided into local partitions by grouping fixed ranges of object numbers. Each partition has an associated partition selection counter that records the total size of all objects it contains that have a zeroed reference count. This total is the minimum amount of acyclic garbage the partition contains. The partition selection counters are held in a table indexed by partition number. Entries in this table contain the counter and a position in a second table. The second table is organised as a heap with its largest values at the root. Whenever a local partition has its selection counter modified, the partitions entry is moved up or down the heap as appropriate. In the worst case the cost of this movement is proportional to the logarithm of the number of partitions in the object store.

When an incremental collection is performed, the partition at the root of the heap may be a good candidate for collection. It contains more garbage directly due to zeroed reference counts than any other local partition in the object store. Regardless of how good a candidate the chosen partition is, some additional policy is required to ensure that all local partitions are eventually collected.

4.4. Global partitions – trains

The global partitions used by these algorithms simulate the trains used in PMOS. The intention is to migrate objects between trains so that cyclic garbage congregates in the same train where it can then be identified by counting references between trains or tracing cross train references. All trains are numbered in ascending order and objects can only be migrated to newer trains that point to them. The choice of how many trains there are and when to create them are policy decisions.

Unlike PMOS, the trains used with the new algorithms are virtual. Rather than associating local partitions with a train, individual objects are tagged with the number of the train they currently belong to. When the collector visits a local partition, every pointer field is checked for cross train references. If an object contains a reference to an object in an older train, the second object's train number is changed to that of the first object. Additional steps are required to track an object's movement between trains and to subsequently identify entire trains that can be reclaimed.

4.5. Reference counting and completeness

One of the new algorithms is an example of an algorithm from category 3 of our taxonomy. This algorithm uses reference counters to track cross-train references and to identify which trains are garbage. It operates as follows.

4.5.1. Global phases. A *trains* data structure is maintained which is indexed by train number and each entry contains two reference counters, old and new. A train with an old counter of 0 is not referenced by any other trains in the object store and is considered dead. The new values for the reference counters are calculated by scanning the entire object store. Each scan of the entire store is called a global phase. A global phase counter is maintained which is used to date stamp local partitions when they are visited by the collector. On the completion of a global phase, the new reference counters replace the old reference counters. The global phase counter is then incremented, the new reference counters are re-initialised to 0 and a new phase begins.

4.5.2. Collecting a local partition. A collection of a local partition involves three main tasks, collecting garbage, re-initialising cross-train reference counters and migrating objects between trains.

The first step in collecting garbage is, for every object in the local partition that belongs to a dead train, every pointer field is overwritten by a null pointer. Second, every object in the local partition with a zero reference count has every pointer field overwritten by a null pointer. The second step is extended if additional objects in the local partition have had their reference counts decremented to zero as a result of the null pointer assignments. Third, the space allocated to objects with a zeroed reference count or to objects from a dead train is reclaimed. An object's address, i.e. its object number, cannot be reused until its reference count becomes zero. This means that an object from a dead train may still have an address and a reference counter even though its storage has been reclaimed. Finally, the local partition's selection counter is set to zero.

When the collector selects a partition for collection the partition's date stamp is compared with the current value of the global phase counter. If the date stamp is out of date, this is the first visit to the partition in the current global phase. On the first visit the date stamp is brought up to date and then every pointer field in the local partition is inspected. If an object contains a pointer to an object in a newer train, then the newer train has its new reference counter incremented by 1. When this task has been applied to every partition the new reference counts will be up to date. Pointers to older trains are ignored because the next task removes them.

The migration of objects between trains is achieved by inspecting every pointer field of every object in the local partition. If object1, in train1, contains a pointer to object2 in an older train, train2, then object2 has its train number changed to be train1. Train1's new reference counter is incremented by object2's reference counter minus 1. This increase in the reference counter may be too large but the sources of all other references to the moved object are unknown. Train2's new reference counter is also incremented by the number of pointer fields in object2. This increase is to allow for any references from object2 to other objects in train2 which are now cross-train references. Any unnecessary increases in the reference counters will be corrected during the next global phase. This does not affect the correctness of the algorithm but may delay the detection of dead trains. The second and third tasks can be applied in parallel since they do not conflict.

4.5.3. Cross-train reference counting. When a pointer assignment occurs the reference counting manipulations described in section 4.2 are performed. If a new cross-train reference is being written to object1 in train1 which references object2, in an older train, train2, then object2's train number is changed to train1. In addition, train1's new reference counter is incremented by the value of object2's reference counter. If the collector has already visited the partition containing object2 in the current global phase, then train2's new reference counter is incremented by the number of pointer fields in object2.

If the collector has already visited the partition containing object1 in the current global phase, the cross-train reference counts are modified as follows. Overwriting a cross-train reference will decrement the new reference counter in the referenced train. Writing a new cross-train reference to a newer train causes the new reference counter of the referenced train to be incremented.

4.5.4. Summary. The conservative modification of cross train reference counters during object migration ensures that a live train cannot be erroneously considered dead but it may delay the detection of a dead train. In the worst case, a complete cycle of garbage must be isolated in a single train and then a new global phase completed before the cycle is detected. However, unlike the algorithm used with Thor, a global phase need only involve a single visit to every local partition. If the collection rate is acceptable, the delay in collecting cyclic garbage may be acceptable too.

The original MOS algorithm was unable to guarantee that an object would eventually migrate to a newer train. In this algorithm, the migration of objects between trains during assignment ensures that progress is always made. During a global phase, either a cross-train reference is detected during the collection of a local partition or it is detected by an assignment that copies it. Therefore, this algorithm is both safe and complete.

4.6. Train marking and completeness

The second new algorithm is an example of an algorithm from category 4 of our taxonomy. This algorithm uses tracing to track cross-train references and to identify which trains are garbage. It operates as follows.

4.6.1. Global phases. A *trains* data structure is maintained which is indexed by train number and each entry contains a list of other trains that may be referenced by objects in the current train. A train that cannot be reached by following these lists, starting from the youngest train, cannot contain objects reachable from the root of the object store and is considered dead. The contents of these train lists are calculated by scanning the entire object store. Each scan of the entire store is called a global phase. A global phase counter is maintained which is used to date stamp local partitions when they are visited by the collector. On the completion of a global phase, the train lists are traversed starting from the youngest train. Any train that is not reached during this traversal is marked as dead. The global phase counter is then incremented, the train lists are re-initialised to be empty and a new phase begins.

4.6.2. Collecting a local partition. A collection of a local partition involves three main tasks, collecting garbage, re-initialising train lists and migrating objects between trains.

The first step in collecting garbage is, for every object in the local partition that belongs to a dead train, every pointer field is overwritten by a null pointer. Second, every object in the local partition with a zero reference count has every pointer field overwritten by a null pointer. The second step is extended if additional objects in the local partition have had their reference counts decremented to zero as a result of the null pointer assignments. Third, the space allocated to objects with a zeroed reference count or to objects from a dead train is reclaimed. An object's address, i.e. its object number, cannot be reused until its reference count becomes zero. This means that an object from a dead train may still have an address and a reference counter even though its storage has been reclaimed. Finally, the local partition's selection counter is set to zero.

When the collector selects a partition for collection the partition's date stamp is compared with the current value of the global phase counter. If the date stamp is out of date, this is the first visit to the partition in the current global phase. On the first visit the date stamp is brought up to date and then every pointer field in the local partition is inspected. If an object in train1 contains a pointer to an object in a newer train, train2, then train2 is placed on train1's train list. When this task has been applied to every partition the new train lists will be up to date. Pointers to older trains are ignored because the next task removes them.

The migration of objects between trains is achieved by inspecting every pointer field of every object in the local partition. If object1, in train1, contains a pointer to object2 in an older train, train2, then object2 has it's train number changed to be train1. Train1's train list has train2 placed on it and train1 is placed on train2's train list. This ensures that if a third train referenced the moved object, train1 will become reachable from the third train's train list. It also ensures that if object2 references other objects in train2, the newly created cross-train references are reflected in train1's train list. The second and third tasks can be applied in parallel since they do not conflict.

4.6.3. Reference counting. When a pointer assignment occurs the reference counting manipulations described in section 4.2 are performed. In addition, if a new cross-train reference is being written from an object in train1 to an object, object2, in another train, train2, the following occurs. First, train2 is added to train1's train list. If train1 is newer than train2, object2 has its train number changed to train1 and train1 is added to train2's train list.

4.6.4. Summary. This algorithm is both correct and complete. As with the previous algorithm, the migration of objects during pointer assignment is sufficient to guarantee progress during a global phase. The conservative approach to recording cross-train references ensures that no live train is erroneously considered dead. In the worst case, once a cycle of garbage is isolated, a subsequent global phase is required to identify it as garbage. However, unlike PMOS and the previous algorithm, a cycle of garbage can be isolated without having to migrate the entire cycle into a single train. This may compensate for the delay in waiting for the additional global phase required to identify dead trains.

5. Evaluation strategy

Evaluating the above algorithms is problematic. Each algorithm's behaviour is dependent on the interaction of the design decisions reflected by the implementation. This may be further complicated by application specific access patterns that either highlight or mask unhelpful behaviours. For example, popular objects can cause severe problems for algorithms based on remembered sets. Therefore, these algorithms can only be effectively evaluated if the presence or absence of popular objects is taken into account. Similarly, the presence or absence of cyclic garbage can severely affect the evaluation of these algorithms. To accommodate the potential complexities the proposed evaluation strategy for the above algorithms is to run a number of specific and repeatable experiments tailored to particular applications and user data.

5.1. Experimental environment

The experimental environment is the Napier88 layered architecture [2]. This architecture has been successfully used with a number of other programming languages including Staple, Galileo and Quest [3]. The architectural layers can be easily replaced to produce a new instance of the architecture where only one component has changed. For example, any one of the four algorithms given above can replace the object store layer.

It is also possible to preserve the lower layers and replace the higher layers. So, to conduct repeatable experiments, a trace of all interface calls to a particular layer could be collected. Subsequently, a new instance of the architecture could be instantiated where the layers using the traced interface are replaced by software that replays the trace. This would allow interactive programming sessions to be recorded and accurately replayed over different object store implementations.

Traces can be collected at a number of different levels including, the object store interface, the object cache interface or at the virtual machine level. Each different level allows experiments to focus on particular performance characteristics. For example, replaying the virtual machine's behaviour would allow repeatable experiments in paging and overall I/O behaviour, replaying object cache behaviour would assist with experiments in object cache management and replaying the object store behaviour would allow experiments with specific aspects of a garbage collection algorithm.

In addition to conducting repeatable experiments, it is also necessary to analyse the structure of an object store's data. Appropriate characterisation of an object store or application behaviour will assist in identifying the conditions under which a specific algorithm can be most effective or perform badly. Some basic analysis tools have been constructed for the layered architecture and used to support experiments with partition selection policies for use with PMOS[3]. This work was an attempt to re-evaluate the results of previously published simulation work [6].

5.2. Experiments

The experiments to be conducted will be targeted at areas including, garbage collection rates, object placement policies, remembered set implementations, frequency of garbage collections, object addressing mechanisms and paging and I/O behaviours. These experiments will attempt to build on the work of others in identifying an effective evaluation strategy for garbage collection algorithms in massive object stores [5].

6. Conclusions

This paper has presented a taxonomy of incremental garbage collection algorithms for massive object stores with examples of safe and complete algorithms where they exist. In addition, two new algorithms have been presented for categories where none were previously known. The new algorithms illustrate how reference counting can be used in the presence of cyclic garbage to construct incremental garbage collection algorithms that are both safe and complete. An experimental framework for evaluating the algorithms was briefly outlined and is currently being implemented.

7. References

[1] Amsaleg, L., Franklin, M. & Gruber, O. "Garbage Collection for a Client-Server Persistent Object Store". ACM Transactions on Computer Systems, Vol. 17, No. 3, August 1999, pp153-201.

[2] A.L. Brown, A. Dearle, R. Morrison, D.S. Munro, and J. Rosenberg, "A Layered Persistent Architecture for Napier88". *Intn'l Workshop on Computer Architectures to Support Security and Persistence of Information. In Security and Persistence.* (Eds. J.Rosenberg & L.Keedy). Springer-Verlag, 155-172.

[3] A.L. Brown, and R. Morrison, "A Generic Persistent Object Store", *Software Engineering Journal*, Special Issue on Object-oriented Systems, Vol.7, No.2, March 1992, 161-168.

[4] Bobrow, D.G., "Managing Reentrant Structures Using Reference Counts", ACM Transactions on Programming Languages and Systems, Vol. 2, March, 1980.

[5] Cook, J.E., Klauser, A., Wolf, A.L. & Zorn, B.G. "Semi-automatic, Self-adaptive Control of Garbage Collection Rates in Object Databases". *In Proc. ACM SIGMOD InternationalConference on the Management of Data,* Montreal, Canada, (1996), pp 377-388.

[6] Cook, J.E., Wolf, A.L. & Zorn, B.G. "A Highly Effective Partition Selection Policy for Object Database Garbage Collection.". *IEEE Transactions on Knowledge and Data Engineering* 10, 1 (1998) pp 153-172.

[7] EXODUS Project Group, "EXODUS Storage Manager Overview", Technical Report, Computer Science Department, University of Wisconsin at Madison, Madison, WI, 1993.

[8] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In *Proceedings of the International Workshop on Memory Management, Lecture Notes in Computer Science*, Vol 637, Springer-Verlag, 1992.

[9] Liskov, B. Adya, A. Casto, M. Day, M. Ghemawat, G. Gruber, R. Maheshwari, U. Myers, A. & Shrira, L., "Safe and Efficient Sharing of Objects in Thor", In Proc. ACM SIGMOD, 1996, pp 318-329.

[10] Lins, R.D. & Vasques, M.A., "A Comparitive Study of Algorithms for Cyclic Reference Counting", Technical Report 92, Computing Laboratory, The University of Kent at Canterbury, August 1991.

[11] Munro, D.S., Brown, A.L., Morrison, R. & Moss, J.E.B. "Incremental Garbage Collection of a Persistent Object Store using PMOS". In *Advances in Persistent Object Systems,* Morrison, R., Jordan, M. & Atkinson, M.P. (ed), Morgan Kaufmann (1999) pp 78-91.

[12] Munro, D.S. & Brown, A.L., "Evaluating Partition Selection Policies using the PMOS Garbage Collector," to appear in Proc. *9th International Workshop on Persistent Object Systems (POS9),* Lillehammer, Norway (2000).

[13] Maheshwari, U. & Liskov, B. "Partitioned Garbage Collection of a Large Object Store". In Proc. *ACM SIGMOD'97,* Phoenix, Arizona (1997) pp 313-323.

[14] Moss, J.E.B., Munro, D.S. & Hudson, R.L. "PMOS: A Complete and Coarse-Grained Incremental Garbage Collector for Persistent Object Stores". In Proc. *7th International Workshop on Persistent Object Systems (POS7),* Cape May, NJ, USA (1996).