

PMOS Revitalised

Fred Brown and Dave Munro

*Department of Computer Science,
University of Adelaide,
South Australia 5005, Australia*

Email: {fred,dave}@cs.adelaide.edu.au

Abstract

Traditional garbage collection techniques designed for language systems operating over transient data do not readily migrate to a persistent context. The size, complexity, and permanence characteristics of a persistent object store mean that an automatic storage reclamation system, in addition to ensuring that all unreachable and only unreachable data is reclaimed, must also maintain store consistency while limiting I/O overhead when collecting secondary-memory data.

PMOS is an incremental garbage collection algorithm specifically designed for reclaiming persistent object storage. The collector extends the Mature Object Space algorithm (sometimes known as the *train* algorithm) to ensure incrementality in a persistent context, to achieve recoverability, and to impose minimum constraints on the order of collection of areas of the persistent address space. The goal of the PMOS algorithm is to break the collection of garbage into small enough units so that disruption can be minimised. PMOS is able to collect the small units in arbitrary orders whilst eliminating cyclic garbage, guaranteeing progress and minimising the impact on the I/O.

Our initial implementation of the PMOS collector demonstrated the efficacy of the algorithm but highlighted some architectural assumptions and performance drawbacks. We have designed a new implementation incorporating numerous lessons learned. Here we report on the architecture of the second implementation and our plans for a range of experiments on collector policies.

1. Introduction

PMOS [MMH96, MBH+99] is an incremental garbage collector designed specifically to reclaim space in a persistent object store. It is one of a family of incremental collectors targeted at reclamation of different levels of the storage hierarchy. The Mature Object Space (MOS) algorithm [HM92] (colloquially known as “the Train Algorithm”) is an incremental main-memory copying collector specifically designed to collect large, older generations of a generational scheme in a non-disruptive manner. In MOS the address space is partitioned into a number of areas that can be collected independently. The DMOS collector [HMM+97] is a complete, non-blocking, incremental collector for distributed object systems that does not require global tracing.

The principal features of PMOS are safety, completeness (the collector reclaims all garbage within a finite number of invocations), non-disruptiveness and incrementality. In addition, PMOS is a copying collector and naturally supports compaction and clustering without the need for semi-space techniques. It can be implemented on stock hardware and does not require special operating systems support such as pinning or external pager control. The collector does not impose any constraints on the order of collection of the

areas thus allowing the implementor to provide a policy appropriate to the application. It provides techniques to reduce the I/O impact of pointer updates and object movement.

1.1 The PMOS Algorithm

The original PMOS algorithm and architecture is fully described in [MMH96] and we refer the reader to that paper for a detailed description of the algorithm.

The PMOS collector is described using the metaphor of *trains* made up of *cars*. The address space of the store is divided into a number of disjoint blocks (cars). One car (or more) is collected in each invocation of the collector, by copying its potentially reachable objects into other cars. Since only potentially reachable data is copied, all unreachable structures contained within the one car will be collected immediately.

To collect cyclic garbage that spans more than one car, cars are grouped together into trains. By ensuring that all the cars in a train are collected by copying the potentially reachable data into other trains, cyclic garbage will be left behind and can be collected, once it is marshalled into the same train. It is shown that to guarantee completeness it is sufficient to order the trains in terms of the (logical) time they are created. Hence we will refer to trains being *older* or *younger* than other trains.

In [MBH+99] and the last IDEA workshop we presented the lessons learnt from our prototype PMOS implementation outlining a strategy for a new implementation that more readily addresses the goals of the PMOS algorithm. In this paper we report on the architecture of the new implementation, PMOS#2, which meets all the design goals of PMOS. In particular it allows us, at each collector invocation, to select any car for collection. We outline our plans for a set of experiments on car-selection policy.

2. The PMOS#2 Architecture

The PMOS#2 architecture can be divided into three layers, an object cache with which applications interact, a stable persistent heap of objects managed by PMOS#2 and an explicitly buffered stable store. The key differences between this architecture and the previous implementation are that an object cache is used, pointer updates in the stable heap have an immediate effect on car remembered sets and the underlying stable storage is explicitly buffered rather than being memory-mapped.

2.1 The Object Cache

The presence of an object cache between the stable heap and applications allows the working set of objects to be clustered together in memory for efficient access. As a consequence, the stable heap is not always aware of the current state of all objects or the presence or absence of pointers between them. Complete knowledge is only available when the object cache writes back all new and changed objects prior to a checkpoint of the persistent store. Since a goal of PMOS is to collect individual cars with a minimum of disruption a checkpoint prior to each incremental collection is not desirable. An alternative strategy adopted for PMOS#2 is that the object cache passes an incremental collection a list of all persistent objects for which it holds a pointer. The object cache then modifies its copy of any pointers that are modified by the collection.

Generating the list of pointers to persistent objects requires the object cache to either maintain a permanent list of known pointers or to scan all its objects searching for persistent pointers. The latter option is used with PMOS#2 since garbage collection of the object cache must already look at every pointer and the change had no impact on any other part of the object cache implementation. Following an incremental collection some pointers may need to be updated requiring all pointers in the object cache to be checked.

This check is also performed during garbage collection of the object cache. In a mark-sweep collector the list of known pointers would be generated during the mark phase, the PMOS#2 incremental collector invoked and then the pointers fixed up during the collection phase. In an in-place compacting collector the list of known pointers would be generated during the first pointer reversal phase, the PMOS#2 incremental collector invoked and then the pointers fixed up during the compaction phase.

2.2 The Stable Heap

The collection of a car requires accurate information in its remembered set regarding which objects in the car may be reachable from other cars. In the previous implementation of PMOS remembered sets were only updated when a car was returned to disk. This meant that updated cars that were still in memory could contain pointers into another car that were not recorded in the other car's remembered set. To overcome this difficulty every updated car in memory had to be scanned prior to each collection. Therefore, even the collection of a single car required all in memory cars to be traced which is potentially expensive. PMOS#2 addresses this overhead by eagerly updating remembered sets thereby allowing individual cars to be collected without reference to other cars.

When a car is faulted into memory a list of all pointers to other cars is recorded in an out going references list. In PMOS#2 this list is augmented with a count of the number of occurrences of each reference. Whenever the object cache writes a persistent object back to the persistent heap each updated pointer field is checked. If the update overwrites pointer to another car, the pointer's occurrence count is decremented. If the new pointer value points to another car, the new pointer's occurrence count is incremented. If an occurrence count becomes 0 or a new pointer is added to the out going references list, then the car that is pointed to has its remembered set modified immediately. In this way every car's remembered set is always up to date with respect to the contents of other cars. Finally, when a car is returned to disk no further work is required since all remembered set updates have already been performed.

This change to the PMOS algorithm still leaves the problem of pointer updates performed in the object cache, some of which may not have been propagated to the stable heap. To ensure no objects are collected when they are really still reachable, the object cache supplies a list of all pointer values it knows about. This list of known pointers is used as an additional set of roots for a collection. Thus, PMOS#2 is actually able to collect an individual car with reference only to roots but no other cars. This addresses the primary goal of the PMOS algorithm, namely the ability to perform incremental collections of arbitrary cars with minimal I/O overheads or disruption.

2.3 The Stable Store

The first implementation of PMOS used shadow paging and memory mapped files to perform the I/O. Consequently, it was not possible to control I/O behaviour or measure it effectively. To facilitate greater I/O control and to allow it to be accurately measured, PMOS#2 uses an explicit buffering mechanism to access the underlying stable storage. The stable storage is still based on after-look shadow paging but disk I/O can be delayed until a page must be written out. The stable heap implementation now directs which pages of stable storage will be written out, when the pages will be written out, which pages must be discarded to service a page fault and how large an in-memory buffer is maintained. A side effect of this change is that the implementation should be more portable since no system dependent memory mapping facilities are used.

3. Car Selection Policy Experimentation

Over the last thirty years or so significant research has been undertaken in the design, construction and measurement of garbage collectors for main-memory programming languages and systems. Much of the cited reclamation techniques are inappropriate for use in a persistent object store since they do not address the characteristics of secondary-memory objects or the size, complexity and stability properties of these stores. Therefore there is much research still to be done to study and understand the behaviour of secondary storage reclamation. Such work aids the design of efficient algorithms but is also a crucial guide to policy decisions

Any PMOS implementation must establish a number of policy decisions such as car size, remembered set size, object allocation policy, number of cars per train, when to create new trains, tradeoffs on the sizes of the Δ refs and Δ loc set, popular object handling etc etc. Two important policy decisions that can have a significant impact on the performance of any incremental, partitioned collector are when or how frequently to invoke the collector and which partition(s) to select for collection. Simulation studies of these policies in object database collectors by Cook et al [CWZ94] suggest that a flexible selection policy that allows a collector to choose which partition to collect can significantly reduce I/O and increase the amount of space reclaimed. However there are a number of drawbacks to their experimental base :-

- Their database size is (very) small – smaller than most RAM sizes in modern PCs.
- The remembered sets are non-persistent and hence have to be reconstructed at system startup requiring a complete scan of the object database.
- There is no account of the impact of how remembered sets are stored and where they are stored.
- Buffering costs of remembered sets are uncoded.
- No account is taken for popular objects that potentially require large remembered sets.
- No cycle detection for inter-partition cyclic garbage is measured.
- No analysis or discussion is presented on the benefits/drawbacks of multi-partition selection.
- No account is taken of the relative costs of the partition selection algorithms,
- The object database access patterns used and the garbage creation mechanisms are unrealistic.

With our new PMOS#2 implementation we are in a position to re-examine their experiments and findings on car selection policy in a real system.

4. Conclusions

The PMOS algorithm presents implementors with a wide range of important policy decisions that can significantly impact the potential performance of the algorithm. At time of writing we have almost completed construction of a more effective implementation. This will give us a version with full flexibility to experiment with important policy issues that affect performance.

5. Bibliography

- [CWZ94] Cook, J.E., Wolf, A.L. & Zorn, B.G. "Partition selection policies in object database garbage collection". In Proc. ACM SIGMOD International Conference on Management of Data, Minneapolis, MN (1994) pp 371-382.
- [HM92] Hudson, R.L. & Moss, J.E.B. "Incremental Garbage Collection for Mature Objects". In **Lecture Notes in Computer Science 637**, Springer-Verlag (1992) pp 388-403.
- [HMM+97] Hudson, R.L., Morrison, R., Moss, J.E.B. & Munro, D.S. "Garbage Collecting the World: One Car at a Time". In Proc. OOPSLA 97, Atlanta, USA (1997).
- [MBH+99] 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.
- [MMH96] 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).