

PMOS Revisited

Fred Brown and Dave Munro

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

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

Abstract

The Persistent Mature Object Space, PMOS, garbage collection algorithm is designed to incrementally collect all garbage in a potentially large persistent object store. The goal of the PMOS algorithm is to break the collection of garbage into small enough units so that disruption to the running system is insignificant. PMOS is able to collect the small units in arbitrary orders whilst eliminating cyclic garbage and being able to guarantee progress. In this paper, we present the lessons learnt from our prototype PMOS implementation and outline a strategy for a new implementation that more readily addresses the goals of the PMOS algorithm. A key feature of the new design is the ability to support a wide range of experiments with the policies that may affect the performance of the PMOS algorithm.

1. Introduction

The Persistent Mature Object Space, PMOS, garbage collection algorithm [MMH96] is designed to incrementally collect all garbage in a potentially large persistent object store. In a very large store, a stop-the-world collector would prove extremely disruptive. The goal of the PMOS algorithm is to break the collection of garbage into small enough units that disruption to the running system is insignificant. The PMOS algorithm is an extension of the earlier Mature Object Space, MOS, algorithm [HM92]. The MOS algorithm is a main-memory collector that collects one small unit of store at a time, in a predefined order. The PMOS extends this algorithm by allowing small units of store to be collected in an arbitrary order, some of which may be on disk, without necessarily incurring large overheads. In this paper, we present the lessons learnt from our prototype PMOS implementation [MBMM98] and outline a strategy for a new implementation that more readily addresses the goals of the PMOS algorithm.

2. The PMOS Algorithm and Architecture

The PMOS algorithm and architecture is fully described in [MMH96]. For brevity we assume the reader has a working knowledge of the algorithm. Our prototype implementation highlighted a number of undocumented assumptions that must be considered in an implementation.

Firstly, the PMOS algorithm has an assumed architecture that includes a first generation, a local heap or object cache, where all new objects are created and transient objects can be quickly reclaimed. Only objects that survive a number of collections of the first generation are considered mature objects and are promoted to the Persistent Mature Object Space. The mature object space is organised into a series of fixed size cars. Cars are further organised to form two or more trains that are themselves ordered by age. The role of the PMOS algorithm is to select a single car and remove all potentially reachable objects from it. Reachable objects in a car are moved to the same train as one of the objects that references the reachable object, but never to an older train. Over time all reachable objects are removed from the oldest train which is then discarded. Also, any cycles of garbage migrate into the same train after a finite number of invocations of the collector. Whenever a train is found to have no references to it from other trains, any objects that it contains must be garbage and the train is discarded.

The PMOS algorithm treats cars in-memory very differently from those on-disk. Each car has an associated remembered set that records which of its objects are referenced by which other cars. The remembered set is held within the car. To allow updates to be made to this data structure without loading the car into memory, a separate delta-refs data structure is maintained. When a car is read into memory, its remembered set is updated from the delta-refs.

In the interests of efficiency, the PMOS algorithm also assumes that updates to in-memory cars are not detected until the car is written out to disk. Only at that point are changes to the car's external references checked and the delta-refs for other cars updated. This has the effect that any collection must traverse all in-

memory cars since references originating in an in-memory car may never have been propagated to the remembered sets of the referenced cars.

Finally, the PMOS algorithm assumes that all object references are direct addresses that must be fixed up when a reachable object is removed from a car being collected. This requires the support of a delta-locations data structure to ensure object references in cars on-disk will be fixed up when those cars are read into memory. Since all in-memory cars are traversed by a collection, object references in cars in-memory can be fixed up immediately.

3. Limitations & Lessons

The prototype PMOS implementation replaces the stable heap layer of the Napier88 persistent programming system [MBC+93,BDM+90] in order to minimise the development effort required. The stable heap presents an object space to the higher architecture that provides object creation, read and write calls together with calls to initiate checkpoints and garbage collection. In most Napier88 implementations a local heap is placed between the runtime system and the stable heap [Bro94]. Since the requirement for a local heap is unclear in the original PMOS paper it was elided from the prototype design. This presented a number of challenges and is the primary reason for poor performance.

In the absence of a local heap, all new objects are created directly in mature object space resulting in frequent invocations of the PMOS collector. Most new objects are short-lived causing the store to rapidly fill with garbage. To limit the effect of this, a policy decision was taken that all new objects are placed in the youngest train. This train is collected in full once a minimum of 4MB of new objects has been created. A side effect of this is that checkpoints of the object store require all cars in the youngest train to be written to disk including a significant number of transient objects that are about to become garbage. The cost of this effect is further exaggerated by the implementation of the underlying stable virtual memory layer.

The stable virtual memory is implemented using memory mapped files and has to eagerly shadow copy pages due to its lack of control over when updated pages are written to disk [Bro99]. Consequently, every checkpoint has to write out 4MB+ of the youngest train and then immediately write it out again as part of the eager shadow copying process. As a result every checkpoint of the stable heap required at least 10MB of writes even if little persistent data has changed.

The presence of a stable virtual memory affects the treatment of in-memory and on-disk cars. The actual movement of cars between disk and memory is left to the host operating system's virtual memory system. The act of modifying a car incurs some shadow copying overheads in order to provide stability. When a car is brought into memory the PMOS algorithm requires any outstanding delta-locations and delta-refs entries to be applied to the car, that is, the car must be modified. In order to minimise the costs of shadow copying it was decided to only treat cars as logically in-memory if they are to be modified. So just prior to modifying a car, any outstanding delta-locations and delta-refs entries are immediately applied. Cars that are not being modified are treated as logically on-disk but may be paged into memory for read-only access. When it is decided to treat a car as logically on-disk, any changes to its external object references are used to update the remembered sets or delta-refs entries for other cars.

This approach has the advantage of allowing direct addressing of objects without incurring any software translation overheads. This is especially important as the Napier88 runtime system is directly operating on objects so addressing has to be as efficient as possible. A downside of this approach is that the in-memory cars have to remain in-memory over checkpoints, that is, they are immediately shadow copied after a checkpoint in anticipation of further modifications. This is the same double write effect suffered by the youngest train.

The issue of popular objects, those with massive remembered sets, is minimised in the prototype by taking advantage of the Napier88 Persistent Abstract Machine's architecture, PAM [CBC+89]. When the PAM is initialised it creates a root object containing references to all the predefined Napier88 objects including a null pointer, null string, etc. These objects are referenced by most other objects created by the PAM, are therefore extremely popular and never become garbage. In order to eliminate the need to build remembered sets for these objects it was decided to create a root train. The root train contains every object created prior to the first checkpoint of the system and is always logically in-memory. All PMOS collections start by assuming that every reference in an object in the root train is a root for garbage collection. No other special treatment is given to popular objects, a user defined object that becomes popular will cause the prototype to fail.

The prototype PMOS implementation does not attempt to implement cross train reference counters or any sophisticated car selection policies. In order to guarantee progress the prototype PMOS collector always selects the entire oldest train for collection. The entire youngest train is also collected so that the majority of transient objects are reclaimed very quickly. This leads to significant pause times as new cars have to be allocated for two trains worth of live data which in turn incurs the eager shadow copying overheads of the stable virtual memory. A further overhead is the fact that all logically in-memory cars also have to be traversed to ensure no reachable objects are accidentally treated as garbage.

The relocation of reachable objects during a collection requires a mechanism to ensure all references to the moved objects are updated correctly. Since all the in-memory cars are traversed it is possible to leave forwarding pointers behind so that the traversal can automatically fix up the references. However, references from on-disk cars require special treatment. The delta-locations table is insufficient. Since logically on-disk cars can be paged into memory the stable virtual memory is instructed to treat cars with possibly out of date references as inaccessible. When one of these cars is subsequently referenced, a page fault is generated and the car is made logically in-memory. This leads to all outstanding delta-locations and delta-refs entries for the car being applied immediately.

4. Planned Second Implementation

A new PMOS implementation is being designed that attempts to better address the goals of the PMOS algorithm by concentrating on the intended PMOS architecture and experimentation with policies. Firstly, the new design includes a local heap. This removes the need to handle most transient objects, greatly reduces the cost of checkpoints and reduces the number of and pause times due to PMOS collections. However, a mechanism is required to allow the local heap to efficiently discover which objects a PMOS collection moves. The use of a local heap also reduces the need for the direct addressing of objects so it is not necessary to use a stable virtual memory with its attendant problems. Policies that will be considered include where to place new objects, where to move reachable objects during a collection, how frequently to collect, how to select cars for collection and how to accommodate popular objects [SG95].

The placement of new objects may involve placing objects in the youngest train subject to some fill limit. It may be desirable to limit the number of new objects placed in a particular train in the hope that objects created together will become garbage together. This may lead to more trains being reclaimed due to cross-train reference counts without the need to collect individual cars.

When a car is collected there may be a choice as to where to move the reachable objects. Moving them to the youngest train may limit future copying but may preserve them longer than is required. Alternatively, it may be better to leave the objects alone but link the collected car into a younger train. A record of how full each car is, could be used to build a cost/benefit function for use in selecting cars to be collected.

The frequency of collections could be controlled by tracking the rate at which space is recovered. If collections recover significant space each time, their frequency may be reduced. Alternatively, if little progress is being made, much more frequent collections may be necessary.

The selection of cars to be collected is essential to guaranteeing the progress and completeness of a PMOS collector. It is necessary to ensure that every car is eventually collected and that progress is eventually made. Progress can be guaranteed by remembering a cross train reference and using this as a root for collection if no progress is apparent [MMH96]

Objects that are referenced from a very large number of other cars cause problems with remembered set sizes. The most common popular objects are dealt with by maintaining a root train. However, this only deals with predefined popular objects and not user defined popular objects. An alternative to complete remembered sets needs to be considered.

5. Benchmarking

The measurement of the second implementation will use benchmarks in addition to OO1 [CS92]. OO1 is actually very checkpoint intensive and, as described above, this is the major weakness of the prototype implementation. The entire OO1 benchmark ran with only a few collections but several hundred checkpoints. The integration of PMOS into the Napier88 architecture allows any Napier88 program to be run with PMOS and measured. For example, a Napier88 implementation of OO7 [CDN93] is available.

Within the implementation detailed measures are required of pause times due to PMOS collections, due to local heap collections and due to checkpoint requests. The sizes of the remembered sets, delta-refs and delta-

locations must also be tracked. It is not clear whether distributing remembered sets among cars is better or worse than keeping all remembered set information in a single central data structure. It may be that most information is duplicated in the central delta-refs anyway so distributed remembered sets may just be unnecessary overhead. It will also be interesting to track car/train distributions to see if changing the policies on when to create new trains affects the progress and effectiveness of collections.

6. Conclusions

The PMOS algorithm presents implementors with a wide range of important policy decisions that can significantly impact the potential performance of the algorithm. Having completed a prototype implementation of PMOS our basic understanding of the algorithm has greatly improved. We are now in a position to construct a more effective implementation that can focus on the important policy issues that will make or break the algorithm.

Bibliography

- [BDM+90] Brown, A.L., Dearle, A., Morrison, R., Munro, D.S., Rosenberg, J. "A Layered Persistent Architecture for Napier88". International Workshop on Computer Architectures to Support Security and Persistence of Information, Universität Bremen, West Germany, (May 1990). In Security and Persistence. (Eds. J.Rosenberg & L.Keedy). Springer-Verlag, 155-172.
- [Bro94] Brown, A.L. "Dynamically Configurable Persistent Object Caches", 17th Annual Computer Science Conference, University of Canterbury, Christchurch, New Zealand, 19-21 January (1994), in Australian Computer Science Communications, Vol. 16, No. 1, pp629-638.
- [Bro99] Brown, A.L. "Using NFS to Expose Persistent Object Store I/O", 6th IDEA International Workshop, Rutherglen, Victoria, Australia, 27-29 January (1999).
- [CBC+89] Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A. & Morrison, R. "The Persistent Abstract Machine". 3rd International Workshop on Persistent Object Systems, Newcastle, N.S.W., (January 1989), 80-95. In Persistent Object Systems (Eds. J.Rosenberg & D.Koch). Springer-Verlag, 353-366.
- [CDN93] Carey, M.J., DeWitt, D.J. & Naughton, J.F. The OO7 Benchmark. ACM SIGMOD, May 1993.
- [CS92] Cattell, R.G.G. & Skeen, J. "Object Operations Benchmark". ACM Transactions on Database Systems 17,1 (1992) pp 1-31
- [HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In [BC92].
- [HMM+97] Hudson, R.L., Morrison, R., Moss, J.E.B. & Munro, D.S. "Garbage Collecting the World: One Car at a Time". Object Oriented Programming : Systems, Languages and Applications (OOPSLA), Atlanta (October 1997), pp 162-175.
- [MBMM96] David S. Munro, Alfred L. Brown, Ron Morrison & J. Eliot B. Incremental Garbage Collection of a Persistent Object Store using PMOS. In Proceedings of the 8th International Workshop on Persistent Object Systems, Tiburon, California, 30 August - 1 September 1998.
- [MMH96] J. Eliot B. Moss, David S. Munro, and Richard L. Hudson. PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores. In Proceedings of the 7th International Workshop on Persistent Object Systems, pp. 140-150, Morgan Kaufmann, 1996.
- [MBC+93] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. "The Napier88 Reference Manual (Release 2.0)", University of St Andrews technical report CS/93/15, 1993
- [SG95] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95) (Aarhus, Denmark, August 1995), no. 952 in Lecture Notes in Computer Science, Springer-Verlag, pp. 235-252.