Incremental Garbage Collection of a Persistent Object Store using PMOS

David S. Munro‡, Alfred L. Brown†, Ron Morrison‡ & J. Eliot B. Moss¥

‡School of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SS, Scotland
Email: {ron, dave}@dcs.st-and.ac.uk

†Department of Computer Science, University of Adelaide,
South Australia 5005, Australia
Email: fred@cs.adelaide.edu.au

¥Department of Computer Science, University of Massachusetts,
Amherst, MA 01003, U.S.A.
Email: moss@cs.umass.edu

Abstract

PMOS is an incremental garbage collector designed specifically to reclaim space in a persistent object store. It is complete in that it will, after a finite number of invocations, reclaim all unreachable storage. PMOS imposes minimum constraints on the order of collection and offers techniques to reduce the I/O traffic induced by the collector.

Here we present the first implementation of the PMOS collector called PMOS#1. The collector has been incorporated into the stable heap layer of the generic persistent object store used to support a number of languages including Napier88. Our main design goals are to maintain the independence of the language from the store and to retain the existing store interface. The implementation has been completed and tested using a Napier88 system.

The main results of this work show that the PMOS collector is implementable in a persistent store and that it can be built without requiring changes to the language interpreter. Initial performance measurements are reported. These results suggest however, that effective use of PMOS requires greater co-operation between language and store.

1 Introduction

Automatic storage management provides an abstraction over physical storage that enables the programmer to manipulate space without the need to consider the low level details of placement and reuse. *Garbage collection* is that part of storage management that is concerned with identifying referenced data from unreferenced data, allowing space to be reorganised and reused.

Numerous garbage collectors for main-memory programming languages and systems have been designed, built, and measured (see Wilson [Wilson92] for a survey of these techniques). Extending garbage collection to a persistent store raises additional concerns:

- The size of many persistent stores suggests that semi-space techniques will be unworkable because they approximately double space requirements. Likewise, "stop-the-world" style collection would result in prohibitively long pauses.
- Object movement in copying or compacting garbage collectors may cause updates to persistent pointer locations. Where these locations are held in persistent objects on secondary storage, the updates may incur high overhead.
- Persistent stores exhibit some notion of stability whereby a consistent state can always be reconstructed after a crash. Most existing collector algorithms are not

inherently atomic and are thus unsuitable in this context. The work of Kolodner [Kol89, Kol92] and Detlefs [Det89] are notable exceptions.

The PMOS [MMH96] (Persistent Mature Object Space) collector is concerned with the collection of space from a persistent object system and is tailored to address the above issues.

This paper reports on the first implementation of the PMOS collector, named PMOS#1. The implementation validates the PMOS algorithm and demonstrates that it can be built in a simple and straightforward manner. The paper also clarifies what requirements the algorithm places on the mutator's manipulation of pointers. We discuss the details of the design and policy decisions adopted and describe the incorporation of PMOS into the generic persistent object store used to support a number of languages including Napier88 [MBC+93], Galileo [ACO+85], Quest [Car89], and Mozzie [HRH97]. One of the main attributes of this store is that its architectural layering reflects the persistence abstraction by ensuring that the programming language levels of the architecture are separate from the details of how objects are stored. Our principal design aim of the implementation is to retain this flexible store interface thus preserving the independence of the store from the language.

The collector has been built and tested using a standard release Napier88 system. At the time of writing our implementation is just complete and the results of initial sample test runs are included.

2 The PMOS Collector

PMOS 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 common features of these collectors are:

- **Safety**: the collector does not collect live (reachable) objects.
- **Completeness**: the collector reclaims all garbage within a finite number of invocations.
- **Non-disruptiveness**: the collector bounds the amount of collection work, thereby bounding the time and space requirements, for each invocation.
- **Incrementality**: the collector reclaims space incrementally.

The specific attributes of PMOS are

- It 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.[1]
- It provides techniques to reduce the I/O impact of pointer updates and object movement.
- Atomicity is provided without binding it to a particular recovery mechanism.

[1] Work by Cook, Wolf, and Zorn [CWZ94] suggests 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.

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 in [MMH96] 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.

The PMOS collector uses the following rules for copying data from a car during collection:

1 Data locally reachable[2] from roots is copied to a younger train, adding a car to that train if required.

2 Data locally reachable from younger trains is copied to those trains, adding a car if required. If an object is reachable from more than one younger train, it may be copied to any younger train from which it is reachable.

3 Data locally reachable from older trains is copied to any other car of its current train, adding a car if required.

4 Data locally reachable from other cars of the same train is copied to any other car of the train, adding a car if required.

5 The remaining data is unreachable and is reclaimed immediately.

It should be noted that the above rules are followed in order. To complete the collection of cyclic garbage one more rule is required. This rule can be applied at any time:

> If no object in a train is reachable from outside the train, reclaim the entire train. If necessary, create another train to ensure that there are always at least two trains.

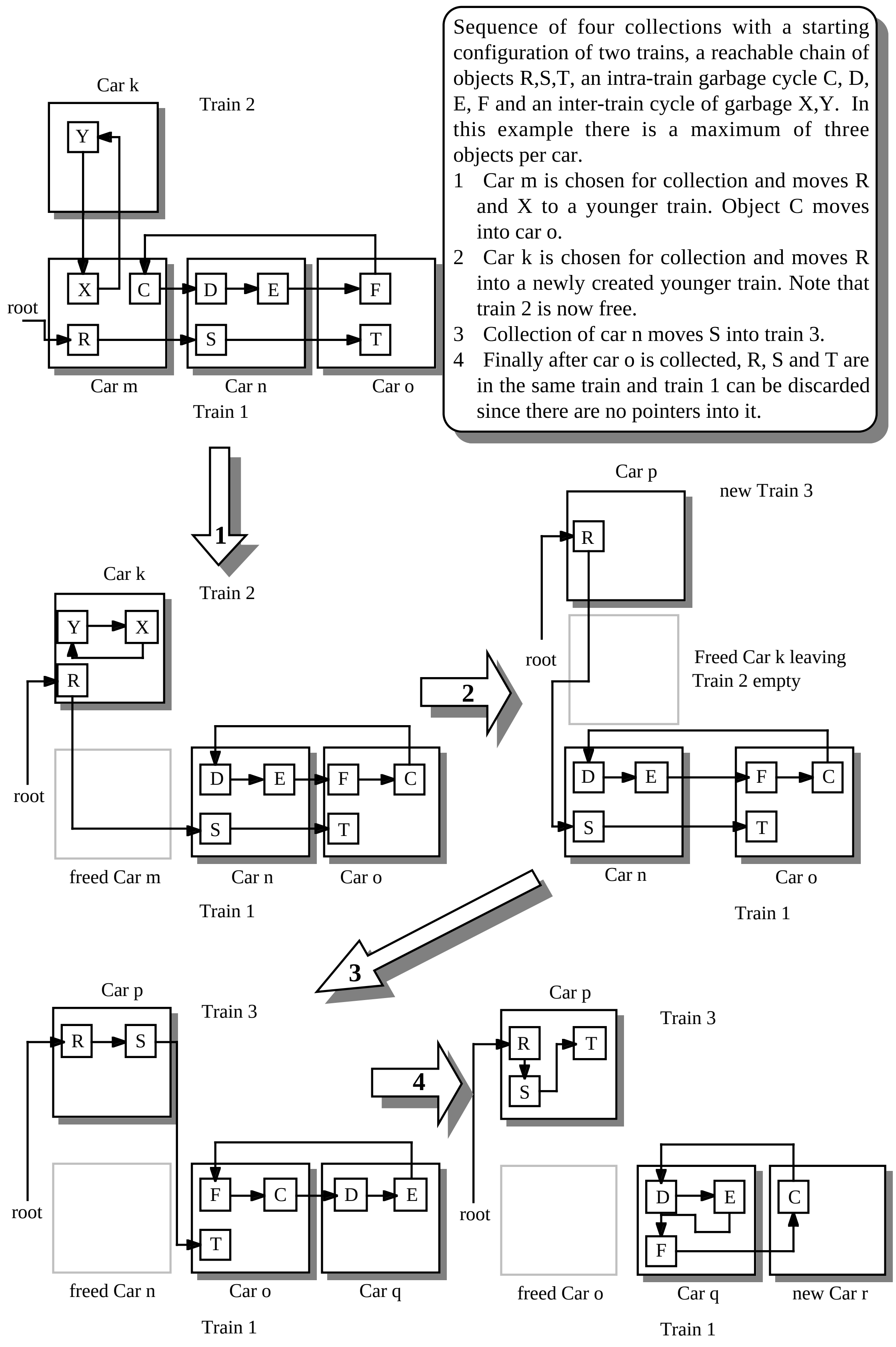
The algorithm allows any car from any train to be selected for collection and to ensure completeness requires that every car is eventually collected.

Figure 1[3] illustrates the algorithm, showing a sequence of four collections, which collects intra-train and inter-train cycles of garbage, and reclusters the live objects.

---

[2] Object Y is locally reachable from pointer X if X refers to Y, or there is a chain of pointers that leads from X to Y where the referents of these pointers al lie within the same car.

[3] Note this drawing is adapted from a similar one found in [HMM+97]


Sequence of four collections with a starting configuration of two trains, a reachable chain of objects R,S,T, an intra-train garbage cycle C, D, E, F and an inter-train cycle of garbage X,Y. In this example there is a maximum of three objects per car.

1 Car m is chosen for collection and moves R and X to a younger train. Object C moves into car o.
2 Car k is chosen for collection and moves R into a newly created younger train. Note that train 2 is now free.
3 Collection of car n moves S into train 3.
4 Finally after car o is collected, R, S and T are in the same train and train 1 can be discarded since there are no pointers into it.

Figure 1: Example Sequence of Mature Object Space Collection

2.1 Remsets, Δref and Δloc sets

To facilitate the independent collection of individual cars, PMOS defines a per-car remembered set (remset) which is used to record which cars have pointers into a given car. In a persistent system cross-car references must be preserved over system crashes and hence remsets must be recoverable. The size of the remsets is obviously application dependent and there are a number of implementation choices. At one extreme the remsets for all the cars could be kept together in some contiguous area of the store. Alternatively there may be some performance gains to be made by storing the remsets with their associated cars potentially allowing for efficient loading. However, as such, maintaining complete and accurate remset information may be expensive as this requires fetching, updating and writing back a remset any time there is a pointer creation or deletion to an object in a car on secondary storage.

The PMOS collector suggests a compromise whereby remsets are stored with their cars but changes to the remsets can be applied opportunistically. When a pointer to an object in a non-resident car is created or deleted, this event is recorded in a set (ideally held in-memory) we call the Δref set and the update to the remset is deferred. The remset is only fetched when its car is faulted in (e.g., when an object in the car is accessed). The Δref set is examined at this point for entries indicating that there have been new or deleted references to the car whilst it was on secondary memory. Any such entries are applied to the remset and removed from the Δref set. Thus the Δref set records pointer changes that have not been recorded in a car's remset.

A further observation made in the PMOS description is that pointer updates do not need to be recorded immediately in the Δref set. The claim is that Δref set entries can be written when a modified car is written back to secondary memory. As a car is read in its outgoing references (i.e., cross-car references) are summarised into a table. If the car has been modified and is about to be written back, its outgoing references are summarised again and the differences recorded in the Δref set.

PMOS also defines another in-memory set, the Δloc set, which plays a similar role to the Δref set. As a copying collector, PMOS moves potentially reachable objects from one car to another. Any pointers that reference an object that is moved need to be updated. Cars containing such pointers can be found by scanning the remset entries of the car being collected. Rather than fetch these cars to update the pointer values, the Δloc set records entries with the old and new locations of objects that have moved and which cars point to those objects. When a car is fetched into memory it scans the Δloc set and applies any relevant pointer updates, removing the entry from the Δloc set.

In essence the effects of pointer update and object movement are constrained to the object itself and the Δref and Δloc sets. The number of entries in the Δ sets is again implementation dependent but can be controlled by the collector. If these sets grow large then the collector can fetch in a number of cars to update their remsets and pointers and hence reduce the Δ set sizes. The challenge for the implementor is to balance the trade-off of Δref and Δloc sizes against the cost of importing and updating remsets.

3 Target Object Store

We chose to incorporate PMOS into the generic persistent object store used to support Napier88 partly because of the authors' knowledge of its design and implementation but also because the abstractions provided by the architectural layering simplified the task. The persistent object store is founded on an open layered architecture. The architecture is generic whereby different implementations of a layer may be interchanged without the need to alter the layers above or below. This divides the architecture between the architectural layers that provide the persistent object store and those facilities that may be programmed by a supported programming language. Thus, a data format can be altered by the compiler without the need to alter the persistent store. The design of the architecture and the initial implementation of each layer are described in detail by Brown and Morrison [BM92].

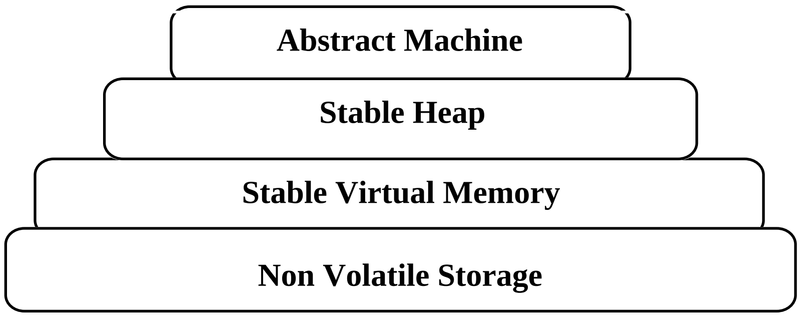


Figure 2: The Napier88 persistent object store layered architecture

In the open architecture design a programming language's abstract or target machine communicates with the object store through the stable heap layer (Figure 2). The stable heap interface provides a set of persistent object management functions that allow objects to be created, read and written, along with a checkpoint mechanism that ensures that all reachable objects from a *stable root* are saved atomically onto the non-volatile storage. The stable heap also provides interface functions to invoke both a stop-the-world garbage collection and an incremental collector, but until now only an offline mark-compact collector has been implemented.

The stable virtual memory (SVM) provides a contiguous range of addresses for use by the stable heap that can always be restored after a soft failure to a self-consistent state. The interface provides functions for reading and writing to the SVM along with a stabilise mechanism that atomically establishes a new consistent state.

This architectural layering demands that the collector is implemented entirely in the stable heap layer and that any data structures used by the collector that are required to persist can simply be allocated from the SVM. Thus no extra mechanism is needed to provide collector atomicity.

4 Design Considerations

The description of the PMOS algorithm in [MMH96] concentrates on the collector mechanism leaving a range of policy decisions that must be defined by the implementor, such as car size, number of trains etc. These are discussed in Section 5. In designing PMOS#1, however, a number of important issues came to light that are not explicitly covered in the PMOS description. These are issues that form the foundation of any PMOS implementation.

The first of these stems from the recognition that PMOS handles in-memory cars differently from cars on secondary storage. For example a cross-car pointer update results in a remset update if the referent car is resident; otherwise a Δref set entry is recorded. Similarly Δloc entries are only recorded for each pointer from a non-resident car to a moved object; otherwise the pointer update is immediately applied. In PMOS, then, every pointer manipulation and dereference results in a residency check. The management of the space of cars in memory and the space of cars on secondary storage and movement of cars between these spaces is fundamental to any PMOS implementation.

Secondly, any incremental collector that partitions the address space needs to use some method of keeping track of the graph of reachable objects in the face of changes made by the mutator. In particular the collector must build a conservative approximation of the set of pointers into the partition being collected. Write-barrier algorithms [HMS92], commonly used in partitioned collectors, trap pointer updates in order that the collector is informed of these updates. In the context of PMOS, the collector must have some technique for tracking cross-car pointers into the car being collected.

Thirdly, in PMOS the use of the Δref and Δloc sets and the deferring of remset updates helps reduce the I/O impact of the collector. In addition, the technique of summarising a car's outgoing references as it is fetched and written back allows the deferred update to the Δref set. The implication then is that this technique can track all pointer updates without the need of adding write barrier techniques to the mutator. However this is not quite the case. Summarising outgoing references and updating remsets as a car is written back ensures that a car on secondary memory has an accurate remset with respect to other cars on secondary memory. In contrast if a car in memory creates a pointer to an on-disk object this assignment is unnoticed until the car is written back and hence a reachable object may be regarded as unreachable. To ensure that a resident car always has an accurate remset, new or deleted references to that car must be eagerly located.

In essence the PMOS collector implicitly assumes an underlying architecture as shown in Figure 3. In this architecture the language interpreter utilises a transient heap for newly created and cached objects. Objects are faulted in from the persistent store on demand and are written out to the store on checkpoint or promotion. More importantly, PMOS assumes that there is some communication, such as an upcall function invoked by the collector, with the interpreter to identify the roots for collection.

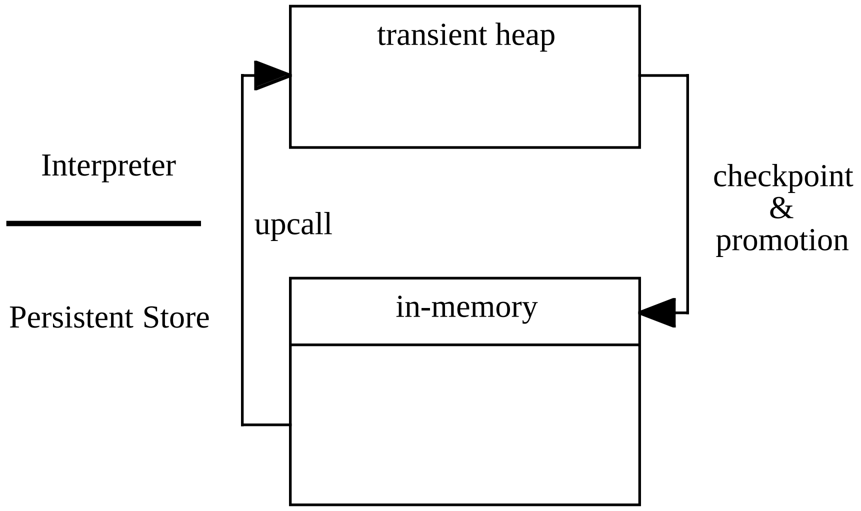


Figure 3: PMOS Implicit Architecture

In PMOS#1, one of our overriding design aims is to implement the collector without requiring changes to the interpreters using the store. The existing store interface does not provide an upcall function as described above and hence the collection roots cannot be readily obtained. This also implies that pointer updates to resident cars cannot be trapped as they occur. The solution we adopt is to designate cars as either *imported* or *exported*. Imported cars are "logically" in memory, their remsets are regarded as inaccurate and they form part of the roots for collection. Exported cars are logically on secondary memory and have accurate remsets with respect to other cars on secondary memory and are not involved in collection. An invocation of the collector selects one or more cars for evacuation and imports it/them if they are not in memory. Each imported car, including the car(s) being evacuated, has its remset entries scanned. Any remset entry that records a pointer from an exported car is added to the roots for the collection. In doing this only pointers to objects in imported cars are followed. Since all pointers from exported cars into the imported cars have already been designated as roots for the collection then pointers to objects in exported cars need not be followed. The reachable objects in the car(s) being evacuated are then found by traversing pointers from the roots.

In effect the definition of locally reachable given above is changed to "Y is locally reachable from X if X points to Y or a chain of pointers exists, all within imported cars, from X to Y". Thus any object reachable in the imported cars will be found even although their remsets are not necessarily up-to-date.

The downside to this approach is that all object creation needs to be directly allocated from the persistent store in order for PMOS to keep track of the collection roots. Thus a significant number of objects that are (very) transient incur the overhead of making them recoverable.

Since a car must be in memory for its objects to be accessible, the set of imported cars will necessarily form part of an application's working set. Thus major policy decisions for the implementor are how to manage the set of imported cars, choosing when and which cars to export and choosing which cars to evacuate. The set of imported cars will have an impact on the size of the root set calculated at each collector invocation. This consideration must be set against the size of the Δref and Δloc sets which also occupy main memory and are only reduced by importing cars that “consume” their entries. This in turn has an impact on the car selection policy. Choosing which car to evacuate next may have a significant effect not only on how much space is reclaimed but also on the size of the Δsets.

5 Initial Design Configuration

Our goal was to build the first working version of the hence a number of the policy decisions have been chosen to engineer the implementation in order to give a base from which further measurements can be made. These policies and our design decisions are itemised in Table 1. The following sections expand on these decisions and outline some of the data structures used in the implementation.

Policy	**Decision**
Car size	8K, 24K and 56K bytes
Remset size	8176 bytes
Maximum number of trains	32
Collector invocation time	After 4Mb of objects created
Object allocation policy	Lead car otherwise create new car
Car selection policy	All cars from youngest and oldest train
Train creation time	After collection of the oldest train
Remset entries	No duplicates

Table 1: Policy Decisions

5.1 Cars

In PMOS#1, remsets are stored contiguously with their associated cars. 8K is reserved for remset entries and management fields (Figure 4). We measured three versions of the collector with varying the car sizes of 8K, 24K and 56K. Space within a car is allocated contiguously and the FSP field holds a pointer to the start of free space within the car. The flags indicate if the car has been modified, if outgoing pointers from the car have been created or deleted and if the remset has been modified. The count field records the number of remset entries. If the number of cross-car references to objects in this car exceeds the number of available remset entries then an overflow remset buffer is allocated from the store and the *next* field assigned a pointer to the buffer.

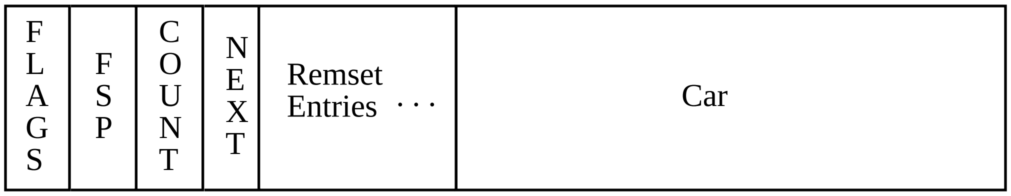


Figure 4: Car and Remset Structure

For objects larger than a single car we adopt a policy whereby cars holding the large object are marked for evacuation like any others but instead of copying an evacuated object and leaving a forwarding address the objects are marked and the cars are linked to the train the object would have been copied to.

NEXT CAR	TRAIN	Δ REF	Δ LOC	ORT

Figure 5: Car Table Entry

We maintain a contiguous area in the store called a *car table* which has one entry per car and is indexed by car number (Figure 5). Each entry in the car table indicates the train number that the car belongs to and a pointer to the next car in the train. The cars of a train therefore form a linked list. Similarly any Δref and Δloc entries associated with this car are also kept as a linked list with the head stored in the car table. Lastly the car table holds a pointer to the buffer containing the summary of the car's outgoing references (ORT).

5.2 Trains

PMOS#1 allocates a contiguous area of store for a trains table. This is an array of entries, one per train, recording the train number and a pointer to the first car of the train. Each train table entry also records a pointer to an old root for the train that may be needed to guarantee progress.[4] The train table entry also records an external reference count for the train, i.e., the number of references from cars of other trains to objects in this train. If this count goes to zero then the entire train can be reclaimed.

5.3 Roots

All reachable data can always be found by traversing the stable heap root object and we could have used this as the only root for collection. However we recognise that Napier88 defines a number of objects that are heavily referenced such as the abstract machine root object, nil, nullstring etc. These objects are essentially *immortal* since they will never become unreachable and will also incur large remsets because of their popularity. Copying such objects from car to car could impact on performance and disruptiveness. Our solution is to define a root train whose cars contain objects that form the roots for collection. The cars of this train are never collected and since the objects in these cars never move then there is no need to maintain their remsets.

There is a problem, though, in preserving the language/store independence as these immortal objects are language specific and need to be identified to the store. To solve this we introduce a convention whereby all objects created upto the first stabilise of a newly created store are considered as roots and allocated in cars of the root train. This convention allows any language using the store to declare its statically known popular objects.

5.4 Allocation and Invocation

The allocation policy adopted always creates objects in the lead car of the youngest train. In other words a new car is created if there is insufficient room in the current car and the cars of the train are not searched for a best fit. The Napier88 interpreter invokes the collector after allocating at least 4Mb of objects.

The collector policy in this initial implementation evacuates all cars in the youngest train and all cars from the oldest train at each invocation.[5] The expectation is that by always choosing the youngest train then a significant amount of space from temporary objects is reclaimed quickly. A new train is created at the end of a collection invocation

[4] Seligmann and Grarup [SG95] found a problem in the original MOS algorithm which can be avoided by recording a reference from a car of a another train to a car in this train.

[5] We are aware that this impacts on our claim of non-disruptiveness. With inaccurate remset information it is difficult to accurately detect empty trains and hence by always evacuating the oldest train we can guarantee progress.

and subsequent objects are allocated in cars of this train. A maximum of 32 trains is used and hence every 32 collections the entire store will have been collected.

6 Test Runs and Measurements

A version of the OO1 benchmark [CS92] written in Napier88 was run and measured on a Sparc10 with a 60 MHz SuperSparc CPU, 1MB of external cache, 96MB of RAM, 384MB swap space and a 4GB Seagate Hawk which held the store. The OO1 benchmark executes on three sizes of database consisting of small parts and connections between them. Each part has eight fields: a part id, a type, an (x,y) integer pair, a build date and three out-going connections to other parts. Each connection has a type and a length. To provide some notion of locality the connections to other parts are chosen so that each connection has a 90% chance of referencing a nearby part. A 20 MB database containing 20000 interconnected parts is used along with the queries:

`lookup`:	A set of 1000 random part identifiers is generated. 10 transactions are then executed; each of which fetches the set of parts from the database.
`scan`:	All parts in the database are fetched once in index order.
`traverse`:	10 transactions are executed. Each transaction selects a part at random and recursively traverses the connected parts, down to a depth of 7 (total of 3280 parts, with possible duplicates). A null procedure is called for each part traversed.
`insert`:	10 transactions are executed. Each transaction enters 100 new parts into the database and commits. Each new part is connected to 3 other (randomly selected) parts.
`insert100`:	Generates the same workload as insert, except that 100 transactions are executed.
`update`:	500 update transactions are executed. Each transaction reads and updates 10 parts chosen at random from a contiguous range of 10% of the parts in the database. The index of the first part in each range is chosen at random for each transaction.

The benchmark suite, comprising of a set of separately executed Napier88 programs, one for each query, was run over a store with the PMOS collector and a standard release store. The elapsed times for each were obtained from the store profiling code using the Unix getrusage and getitimer library functions. Note that the abstract machine of a standard Napier88 release employs a small separate cache of objects from the stable heap, called the local heap. The idea of this cache is effect essentially a first generation for newly created objects. The local heap incorporates several optimisations not yet employed in PMOS that aid its efficiency. The standard release store uses an offline stop-the-world collector and allocates objects from a contiguous heap.

program	Standard Release	64K	32K	16K
populate store	625.17s	603.83s	599.94s	627.71s
full collect	5.11s	137.45s	154.49s	148.79s
setup	172.76s	147.58s	149.04s	138.65s
insert	14.22s	49.15s	71.06s	38.82s
lookup	10.12s	17.47s	20.26s	11.97s
scan	15.51s	21.66s	20.53s	16.59s
insert100	134.69s	525.17s	443.19s	366.88s
lookup	10.16s	16.87s	11.82s	12.43s
scan	23.37s	31.18s	23.30s	22.75s
update	199.59s	361.66s	305.66s	345.92s
lookup	10.99s	13.22s	10.57s	11.49s
scan	24.87s	29.22s	22.02s	21.08s

Table 2: OO1 Benchmark results

The results of PMOS#1, our first working implementation, are shown in Table 2 giving the elapsed time in seconds for programs to populate the standard Napier88 environment, the cost of a full store garbage collection and the times for the OO1 benchmark suite. Figures for the standard Napier88 release are shown along side three versions of PMOS#1 using car/remset sizes of 64K, 32K and 16K respectively.

In general the lookup, traverse and scan results are not significantly worse than the standard release but the insert and update programs perform badly. This is where we see the cost of direct allocation from the persistent store. In addition two other factors affect the results, firstly that because of incomplete remset information we select all the cars of the oldest and youngest trains for collection at each invocation to ensure progress; secondly each stabilise exports all cars and re-reads all cars belonging to the youngest train. These are naive policies that we are actively improving and the challenge is to find solutions that provide the atomicity, incrementality and non-disruptiveness. In addition, this store implementation uses after-image shadow paging as the recovery mechanism and we suspect that there is performance loss through the shadowing of remsets and imported cars after each stabilise.

7 Conclusions and Future work

PMOS is an incremental copying garbage collecting algorithm targeted at reclaiming secondary memory space from persistent object stores whilst reducing its I/O impact. What we have demonstrated here is that the collector can be built into a persistent object store and meet the stated goals on stock hardware and without the need for special operating systems support. However it is clear from our initial results that to make effective use of PMOS we need to augment the store interface to provide a co-operative upcall function between the store and the language interpreter that allows the identification of collection roots.

Now that we have an initial implementation the work on tuning the collector and ascertaining efficient policies can begin. There are any number of choices here but in particular we are interested on the effects of varying the car size, the overhead of the Δ sets and efficient data structures to hold them. We would like to develop good policies to determine and control the working set of in-memory cars and measure the tradeoff of remset sizes and Δref sets. Experiments with different recovery methods and their performance effects under PMOS should be investigated. It would also be interesting to try out different car selection polices such as those suggested by Cook [CWZ94].

8 Acknowledgements

This work was carried out when Dave Munro was visiting the University of Adelaide in May 1998. Prof. C.J. Barter must be acknowledged for his assistance and hospitality.

9 References

[ACO+85] Albano A., Cardelli L. & Orsini R. “Galileo: A Strongly Typed, Interactive Conceptual Language.” ACM Transactions on Database Systems, vol. 10, no. 2, 1985, pp230-260.

[BC92] Bekkers and Cohen, editors. In *Proceedings of the International Workshop on Memory Management*, St. Malo, France, 1992. Published as number 637, *Lecture Notes in Computer Science*, Springer-Verlag, 1992.

[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.

[BM92] Brown, A.L. & Morrison, R. "A Generic Persistent Object Store", Software Engineering Journal, Special Issue on Object-oriented Systems, Vol.7, No.2, (March 1992), 161-168.

[BMM+92] Brown, A.L., Mainetto, G. Matthes, F., Mueller, R. & McNally, D.J. An Open System Architecture for a Persistent Object Store, Vol. 2: Software Technology, 25th Hawaii International Conference on System Sciences, Kauaii, Hawaii, (January 1992), 766-776.

[BR91] Brown, A.L. & Rosenberg, J. “Persistent Object Stores: An Implementation Technique”. In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 199-212.

[Car89] Cardelli, L. “Typeful Programming”. DEC SRC Report, No. 45, May 1989.

[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.

[CS92] Cattell, R.G.G. & Skeen, J. “Object Operations Benchmark”. ACM Transactions on Database Systems 17,1 (1992) pp 1-31

[CWZ94] Johnathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn. Partition selection policies in object database garbage collection. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD ‘94)* (Minneapolis, MN, May 1994), pp. 371-382.

[Det89] Detlefs, D.L. Concurrent, Atomic Garbage Collection. PhD thesis, Dept of Computer Science, Carnegie-Mellon (1989)

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In [BC92].

[HMS92] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanovi. A comparative performance evaluation of write barrier implementations. In Object Oriented Programming : Systems, Languages and Applications (OOPSLA), pages 92-109

[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.

[HRH97] Hollins, M., J. Rosenberg, and M. Hitchens, "Subtyping and Protection in Persistent Programming Languages", Proceedings of the Hawaii International Conference on System Sciences, Hawaii, Jan. 1997.

[Kol89] Kolodner, E.K., Liskov, B. and Weihl, W. “Atomic Garbage Collection: Managing a Stable Heap”. In Proceedings of 1989 ACM SIGMOD International Conference on the Management of Data, June 1989, pp15-25.

[Kol92] Kolodner, E.K. “Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap”. Ph.D. Thesis, MIT (1992).

[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.

[UJ88] Unger D. & Jackson F. “Tenuring policies for generation-based storage reclamation.”, In Proc of Conference on Object-)Oriented Programming Systems, Languages and Applications (OOPSLA’88), pp 1-17, 1988

[Wilson92] Paul R. Wilson. Uniprocessor garbage collection techniques. In [BC92].