Mosaic: A Non-intrusive Complete Garbage Collector for DSM Systems

David S. Munro, Katrina E. Falkner, Matthew C. Lowry and Francis A. Vaughan

Department of Computer Science, University of Adelaide,
South Australia 5005, Australia
Email: {dave, katrina, mclowry, francis}@cs.adelaide.edu.au

Abstract

Little work has been done in garbage collection algorithms for distributed shared memory systems. Mosaic is a safe and complete garbage collection system that collects garbage in object systems that are implemented above page-based distributed shared memory systems. It is non-intrusive in its impact on application performance.

Mosaic partitions the virtual address space into separately managed regions, which form the basic unit of object storage. Garbage collection operates by moving objects between these partitions in a manner that associates live objects together leaving unreachable objects behind so that partitions may be reused. To provide for safe operation of the collector a distributed update protocol maintains sufficient local knowledge of pointer duplication and destruction to allow safe determination of object reachability without the need to globally trace the object space. Mosaic exploits the semantics of pointer operations in type-safe object languages to allow for a weakened consistency model of pointer update during garbage collection.

1 Introduction

With increasing size and complexity of applications, reliance upon explicit storage management is becoming increasingly error prone. Very large or distributed applications are typically forced to operate with only partial knowledge of storage operation. Attempting to explicitly manage storage in such applications typically leads to serious errors that can cause random program behaviour, storage leaks and inefficiency in use of memory. Such errors are notoriously difficult to detect and expensive to fix. For over thirty years, numerous garbage collection algorithms have been devised, implemented, analysed and tested to alleviate this burden from the developer. (See Wilson [21] for an excellent survey). Although distributed shared memory (DSM) systems have undergone significant development, garbage collection in DSM systems has typically continued to be performed using simplistic and often inefficient collectors [8, 13].

In a DSM system, applications view and operate over a single address space, shared and visible to a number of nodes. From this perspective it is clear that any single-processor garbage collection algorithm can operate in a DSM system, but since it must traverse the global state of the application, it operates at considerable cost to application performance.

The challenge is to develop a garbage collection algorithm with the desired properties of *safety* and *completeness* that minimises impact upon application performance. Safety ensures that a non-garbage object (a *live* object) is never collected; completeness ensures that all garbage will eventually be collected. To address the problem of correctly determining liveness in the face of concurrent application access, collection algorithms generally exploit two characteristics of object systems, namely the *stability* of garbage and the *conservatism* of reachability. Stability means that once an object becomes unreachable then it will remain in that state. Conservatism allows the collector to legitimately consider a garbage object as being live avoiding expensive co-ordination between the collector and the application. A conservative collector is not necessarily complete; there exist a number of conservative and incomplete distributed garbage collection algorithms [4, 16].

We describe Mosaic; a garbage collection algorithm that is targeted at DSM systems and is specifically designed to minimise the collector's impact on application performance. The collector is neutral with respect to the coherency protocol chosen by the DSM system for application use. There is a close relationship between the strictures of conservatism in garbage collection and the generation of multiple copies of objects in DSM systems. Mosaic exploits this relationship to achieve safety and completeness with minimal DSM interference. Whilst Mosaic is unconcerned with the DSM consistency model used by the application it introduces a novel relaxed consistency model for use in its reclamation algorithm.

Mosaic is a member of a family of collection algorithms based on the Mature Object Space (MOS) collector [5] (sometimes known as the Train Algorithm) with the specific goal of dovetailing efficiently into a DSM system. The main appeal of adapting Mosaic from a known and proven set of algorithms is that it acquires correctness guarantees. In addition, Mosaic inherits and exhibits a number of desirable properties from its predecessors, namely incrementality, completeness, asynchrony and hence potential for scalability. Research on this family of collectors has shown that it operates flexibly, correctly and efficiently in a number of contexts including main-memory systems, persistent systems and fully distributed systems [5, 7, 14, 17]. To achieve this marriage with minimal impact on the DSM operation is the subject of this paper.

2 Partitioned Garbage Collection

The principal goal of a garbage collection algorithm is to correctly distinguish those objects that can be legally manipulated by the application program (live objects) from those that cannot (garbage objects) and then using this knowledge to reclaim the garbage space. This is achieved using *reachability*; involving traversal of the object graph from a set of program *roots* to determine liveness. The set of live objects is determined from those objects reachable through the transitive closure of all roots. Safety is an essential property of any such scheme; completeness (including the detection of cycles of garbage) is a desirable and arguably essential property of any garbage collection algorithm, particularly in long-lived and distributed systems. Garbage collection algorithms should also be efficient both in terms of their run-time impact on the performance of the application and in exploiting potential to increase application locality through compaction and clustering.

Simple garbage collection algorithms assume global knowledge; that is, they have available the entire address space they operate upon. In systems where it is expensive for a collector to inspect all parts of the address space this may result in significant performance degradation. For example, the address space may be very large and mostly resident in stable storage. Similarly in DSM systems global knowledge is expensive since typically the majority of the shared address space will not be cached within any individual node. A well-understood technique for ameliorating this is partition-based collection. The address space is partitioned and each portion is collected independently, providing for a gradual and incremental reclamation of garbage. This can offer performance advantages through increased locality, limited disruption and avoidance of the need for global synchronisation. Partitioning is therefore a building block to achieving scalability [2, 3].

In order to independently collect a partition in a safe manner there must be some meta-data associated with each partition to indicate the global reachability of objects within that partition. This meta-data takes the form of a set of object references, where the members of the set point to those objects in the partition that have one or more references extant outside the partition. A collector operating upon a partition treats this set as additional roots of reachability. A live object that is inside a partition which is not referenced from any other object in that partition will not be erroneously reclaimed. Much research has been done into implementation techniques for such sets (commonly called *remembered sets* or *remsets*) [6, 15, 19]. These sets may be maintained asynchronously in distributed systems, exploiting garbage collection conservatism and thus avoiding the need for global synchronisation.

Such a partition-based collection scheme is safe and incremental, but not complete. The problem is that cycles of garbage that span more than one partition will never be collected. In a cycle spanning multiple partitions at least one member in each partition will be referenced from outside that partition. Thus, the members of the cycle will appear reachable to each partition collector and will be treated as live. To become complete a partition-based collection algorithm must augment the collection of individual partitions with some additional mechanism to discover and eliminate cross-partition cycles of garbage.

In a DSM system, the problem of collecting partitions is complicated by the fact that each node in the system is participating in a shared address space that changes dynamically. Objects may be physically resident on different nodes. Indeed, portions of a partition may not be resident in any individual node s cache or may be replicated on several nodes. Hence, in a DSM system the selection of policies for partition collection and the maintenance of meta-data become important decisions. The problem of independently collecting partitions in a safe and complete manner in the face of partial knowledge is addressed by the Mosaic collector.

2.1 The Mosaic Collector

Mosaic is one of a family of incremental collectors that are targeted at reclamation of different areas of the storage hierarchy. The MOS collector is a partitioned main-memory *copying collector* specifically designed to collect large, older generations of a generational scheme in a non-disruptive manner. Copying collectors move objects to a new location allowing the entire memory range collected to be reused en-block, as opposed to collectors that leave objects in place, attempting to reuse the space made available by individual garbage objects. Copying collectors avoid memory fragmentation and are able to exploit dynamic reclustering to improve caching performance. MOS partitions the address space into multi-page units called *cars,* which are grouped further into *trains*. Trains comprise of sets of cars where each car belongs to one and only one train. Trains are a structuring technique through which cycles of garbage can be detected and reclaimed. The PMOS [14] collector extends MOS to provide incrementality in systems that extend the object space to include secondary storage. DMOS [7] builds upon MOS and PMOS to offer incremental collection for distributed message-passing systems. DMOS is a complete, non-blocking, incremental collector for distributed object systems that does not require global tracing.

The common attributes of these collectors are safety, completeness, non-disruptiveness and incrementality (the collector reclaims space incrementally). Mosaic:

- is a copying collector and naturally supports compaction and reclustering,
- does not impose any constraints on the order of collection of the partitions and allows nodes to concurrently collect partitions of the address space,
- operates correctly in the face of multiple copies of objects being cached in a number of nodes and is compatible with DSM relaxed consistency models,
- maintains reachability information via asynchronous message-passing which is integrated into the DSM consistency protocol communication,

Like its predecessors, the Mosaic algorithm is described by the analogy of cars (conjoint partitions of the address space comprise of a fixed number of pages) that are then grouped into a flexible higher-level partitioning of trains. A unique aspect of the train algorithm is the manner in which cross-car cycles of garbage are eliminated. This is achieved through the provision of a total ordering to trains based on their age (time since creation). Trains may be referred to as younger or older than other trains. Associated with each car is a remembered set that records those objects resident in the car which are referred to from outside the car.

Cars are the units of collection. When a car is collected, any reachable objects are copied out of the car into other cars, and the entire car (with any remaining garbage) is reclaimed as free space. Choosing a car to receive a reachable object is done using the following set of rules, which may require the creation of new cars.

- If the object is reachable from a car in a younger train, the object is copied to a car in a train that is younger than it's current train but no younger then a train that holds a reference to it.
- If the object is reachable, but not from a younger train, then the object is copied into another car of its current train.
- An object is never copied to an older train.

The effect of these rules is that live objects are continually moved to younger trains. Objects that are only reachable because they are members of a cross-car garbage cycle congregate to the youngest train that holds a member of the cycle, but are promoted no further. Because of the stability of garbage, a train eventually becomes either completely empty or solely garbage. By detecting either state and reclaiming the cars of the train, completeness is achieved. The system must ensure that there are always at least two trains.

Mosaic uses two interacting collection mechanisms. One mechanism operates concurrently on a per-node basis and collects individual cars (requiring the safe maintenance of remembered sets). The other mechanism is a global mechanism that asynchronously detects and collects unreferenced trains and hence cyclic garbage. Each mechanism may be described in terms of the detection of a global predicate. Remembered set maintenance requires the detection of absence of references; unreferenced train detection requires the determination of the absence of references to any object from outside a train to within a train. Mosaic uses a set of proven protocols, adapted from the DMOS collector, that efficiently maintain these predicates. Section°4 outlines this scheme.

The train algorithm leaves decisions regarding when new trains are created, when new cars are added to existing trains, and how cars are formed from the underlying address space as matters of policy. This enables a great degree of flexibility in the manner in which objects are migrated within the address space. By collecting and relocating cars *en masse* fragmentation is avoided, and objects are naturally clustered in a manner which can be tuned to improve application locality.

The algorithm allows any car from any train to be selected for collection and requires that every car be eventually collected. The completeness of Mosaic depends on four constraints:

- objects are copied in one direction only; from older to younger trains,
- the stability property of garbage ensures that garbage is never copied to a train younger than its youngest referent,
- each car is eventually collected, and
- isolated trains are eventually reclaimed.

3 Mosaic DSM Architecture

The Mosaic garbage collector uses the car as the fundamental block of storage space within which objects are held. Constructing cars from pages is a natural progression, and allows the base element of object allocation to be seamlessly integrated with virtual memory management. Progression to a page-based DSM model of execution is similarly natural. The Mosaic system makes no demand upon the nature of the consistency model seen by the user level code and can be used with any of the currently described DSM models. However, the nature of object systems provides some significant opportunities to further relax the consistency constraints when manipulating pointers and object location.

Embedding an object system within a page-based DSM forces some performance tradeoffs with respect to object-based DSM. Page-based systems generally suffer from greater false sharing and greater cost of individual data movement. However, the ability of Mosaic to choose target cars during reassociation provides the opportunity to cluster objects for better temporal locality (ameliorating the cost of data movement by clustering objects that are used together on the same page) [1], or for dividing objects that are found to cause false sharing between separate pages. Car size can be chosen to reduce the size of the remembered set data structures since, in general, the larger a car is, and the better the clustering of objects within a car, the fewer intra-car references will occur.

3.1 Object Properties

The Mosaic system makes the following demands of the underlying object system.

- Pointer Differentiation. It must always be possible, given a pointer, to discover unambiguously all pointers within the referend object.
- Pointer Safety. User level code must not be able to manipulate or create pointers.
- Roots of Reachability. The object system must provide the garbage collector with all pointers that may exist outwith objects. Typically these are pointers in virtual machine registers, and stack frames. Further, it must be possible for these pointers to be updated to reflect the new location of an object.

These properties are common to the vast majority of object systems. The following are less common, but typically well understood, and used in a significant number of object designs.

- Crossing Map. Some mechanism exists to discover all the pointers on a page when only presented with the page address. Since objects may cross page boundaries some mechanism must exist to discover those pointers resident in the partial object. Crossing maps as described by Wilson [18] suffice.
- Pointers are virtual addresses. Pointers are direct addresses into the shared virtual address space. It is thus possible to identify the page upon which a referend object resides, and further, determine the car in which it resides.

3.2 The Car Collection Consistency Model

An object system may be considered as a virtual memory mechanism in, and of, itself. This is because user level code is prohibited from manipulating the actual contents of pointers. Code cannot modify or create pointers, it may only assign or dereference them. Object creation is not affected by users code, but is a function of the underlying object management system, and is the only time pointers are created. Most importantly, an object can be relocated with no change to the user level code semantics. Thus there is a clear duality of pointers to objects in an object system and page tables in page-based virtual memory. In combining a relaxed consistency page-based DSM system, with the relaxation of consistency available with a pointer-based object system, it is possible to exploit the benefits of both.

The Mosaic system is intended to be compatible with the various strengths of consistency described for DSM architectures. Mosaic adds a new consistency model exploited during car collection. Thus, it must be implemented as a modification to any existing consistency protocol; it is not possible to implement Mosaic above an existing DSM system. This is for a number of reasons:

- Mosaic updates pointer values in a manner that is not safe for scalar data, and thus incompatible with the consistency model for scalar data.
- Operation of the pointer tracking and other housekeeping tasks used by the Mosaic algorithm require message interchange between nodes that is causally dependant upon modification to the underlying shared memory space. Thus, these messages must be delivered in FIFO order, in the same channel as the DSM consistency messages.

Implementing Mosaic as a modification to rather than on top of an existing DSM may seem to limit the technique s usefulness, however we believe that knowledge of the DSM system s structure and execution provides significant increases in efficiency for collection.

3.2.1 Page-based DSM features

The goal of an individual car collection pass is to remove all reachable objects from within the car, thus allowing the car to be reused for object storage. The crucial point is to observe that although the objects are moved to a different location in memory a number of invariants hold.

- Moved objects are not otherwise modified.
- The source car is not modified.
- The destination car is only modified in that new objects are appended in hitherto unused (and otherwise inaccessible to user code) space.

These invariants allow a relaxed consistency to apply to the pages allocated to the collected car and its target cars for re-association. The collected car need only be held as a read-only copy in the collecting node. Whilst the car is being evacuated, nodes that hold a valid copy of pages from the car may continue to read objects resident within the car. However, no node can be allowed to gain write access to the pages of an evacuated car. Member pages of such a car are considered to be in a state special to the garbage collection algorithm. User code attempting to modify an object in an evacuating car will be redirected to the new location of the object through a mechanism described in Section°3.2.2.

The destination page must be held for write access. However, since none of the objects currently resident in the page are modified by the garbage collector, a relaxed multiple writers protocol is possible. Since the page is only modified by appending objects the calculation of changes to the page is trivial and does not require the maintenance of a ghost copy of the page, as would be required if more general updates occurred. Mosaic does however require that only one node be afforded append access to any individual car.

3.2.2 Relocation and Consistency

Since Mosaic is a copying collector it is crucial for the success of object relocation that all pointers to all relocated objects are updated in a timely and safe manner. Each page has associated with it a remset, containing a list of all pointers to objects on that page from objects in other cars. This list serves a dual purpose. First as a component in the local roots of reachability, determining those objects that will be reassociated. Secondly, as live objects are copied out of the page the Mosaic system uses the list to create a change list of pointers that must be updated to reflect the new object location. Once the page has been traversed this list is sent to each node which contains a valid copy of any source pointer in the list. This change list is the equivalent of an update consistency message.

During object reassociation the source pages may remain available for read access in other nodes. Before modification of any object that resided within the evacuated car can occur, the remote node must correctly update its references to the object to reflect its new location. Upon reception of a pointer update message a node may begin to asynchronously update the pointers it holds. Note that this update may proceed independently of any access or consistency protocol locking that may exist on the affected pages. Again this is safe because of the special nature of pointers.

Pointer update must be done whilst the local computation is blocked at a known safe point. This is the only time where the operation of the application is blocked by the collector. Update at a safe point allows modification of pointers in application registers, and avoids race conditions between the application and the collector. It also prevents the application potentially seeing two inconsistent copies of the object, one at the original location and one at the new.

Update of object location is slightly complicated because a node may have created new copies of a pointer during the time in which the update message is being created or processed. However, the pointer tracking mechanism will have generated appropriate change messages that will arrive before the update acknowledge message. If such a change message is received the collector node builds a new update request and restarts the update cycle.

During object evacuation a node with a valid copy of the car may attempt to modify an object in the car. Since the pages in the car are only available for read access the application will block with a write access fault. At this point two options exist to allow the application to correctly find and modify the correct object.

- The application is designed so that it can restart the instruction in such a way that it automatically reloads the pointer to the referend object from the source object, which will have been changed during the update pass. Many virtual machines are designed or can be implemented to restart in this manner [12, 11].
- The virtual machine is designed to understand and correctly follow forwarding pointers. Forwarding pointers (also known as tombstones) are degenerate objects that are placed in the address space where the required object used to reside. The execution engine recognises the forwarding pointer and is redirected to the new location. If forwarding pointers are used, the virtual memory system replaces the page being accessed with a synthesised page. This page contains only forwarding pointers directly synthesised from the update message. Once the update has completed the synthesised page may be safely removed.

The collecting node must also safely keep track of changes to the location of objects. Since nodes are free to continue execution during car collection, pointers to objects within the car may continue to be copied and destroyed. Messages reflecting this activity may be in transit during object relocation, and thus the owner node must keep its own copy of the update list, which is uses as a forwarding list to enable correct delivery of pointer tracking messages to the object at its new location. Each node acknowledges completion of an update message to the originator. Since messages are delivered in FIFO order, this acknowledgment acts to define the time when no more pointer tracking messages will refer to the object at its old address, thus allowing the collecting node to determine when it no longer needs the update list.

4 Mosaic Predicate Maintenance

As described in Section°2.1, Mosaic maintains two global predicates to ensure safe and complete operation. These predicates detect:

- The existence of any reference to an object in a car,
- The absence of any reference to objects within a train from outside that train.

These predicates are maintained through the use of modified protocols taken from the DMOS garbage collector. These protocols are designed to operate correctly without recourse to global synchronisation, and as such are designed to correctly reflect the causality of events occurring on separate nodes.

4.1 Pointer Tracking.

Detection of references to objects is the problem of global remembered set maintenance. In principle, each pointer manipulation should be reflected in updates to the referend object s remembered set entry. Pointer manipulation events take two forms, the duplication of a pointer, and the destruction of a pointer (through overwriting by a reference to a different object.) Both these actions can occur through the normal operation of user code on a node. However pointer duplication may also occur through the action of the DSM protocol, since whenever a page is delivered to a node all pointers on that page must be considered as having been duplicated. The need to find all pointers on a page when it is copied between nodes is the reason for the crossing map capability described earlier.

From a high level view the pointer tracking algorithm merely signals to the appropriate remembered set each creation and deletion event, noting that a page entered onto an output queue or copied into an intermediate buffer should be regarded as a duplication event.

To maintain the causal relationship of pointer events between pairs of hosts, these events are tagged with the node at which they occurred and the protocol treats the enqueing of a page for delivery to another node as a duplication event at the receiver node, not at the sender. Delivery of these events occurs in FIFO order in the same channel as other DSM coherency messages and thus any node is assured of maintaining a causally consistent view of the existence of pointers. The stability property of garbage provides the mechanism by which the remembered set maintenance system may be assured that no pointers exist, despite the lack of globally synchronised of state. This is because once the sum of pointers to an object on other nodes reaches zero, no legal mechanism exists to manufacture a pointer and thus the count will remain zero.

For the purposes of implementing the Mosaic collector, a significant number of optimisations are applied to the basic principles of the pointer tracking algorithm. In practice, pointer tracking events are only of importance when a cross car reference is involved. Additionally, only the transition from some pointers to no pointers is important to the safety of collection. Thus nodes are free to coalesce pointer-change events, and are only actually required to signal changes to and from zero.

Maintenance of the update information required to modify referencing pointers is similarly open to heavy optimisation. In principle each pointer change event includes the address of the modified pointer so that when the referend object moves all pointers referring to it can be changed. Again this knowledge may be maintained locally by a node, and utilised in response to a pointer update message that only need identify the original and new pointer locations.

4.2 Empty Train Detection.

Pointer tracking enables the correct detection of an absence of pointers to objects in a car from other cars. Empty train detection uses this remset information to determine absence of *external* references, i.e., references from outside this train to inside it. A train whose cars have empty remsets is easily determined and reclaimed.

Trains can also become isolated when they solely contain garbage; i.e. there is no valid path of reachability from application roots to any object in the train. This can be the case if and only if the remset of each car in the train only contains entries that regard other cars in the same train — a cycle of garbage. Since such a trainis in a stable state then that property is readily discovered using any standard distributed termination protocol.

5 Implementation Strategies

We have presented the design of the Mosaic garbage collector. Similar to its predecessors, Mosaic leaves much of the implementation and policy decisions, such as car size, train size etc., to the implementor. Here we offer some potential approaches to a working Mosaic system.

5.1 Object Creation

The address space of the Mosaic DSM system is visible and shareable by all nodes in the system. However each node in the system reserves exclusive access to a set of cars specifically for creating new objects. These cars are notionally defined as belonging to a reserved train of infinite age. The main idea of these *nursery* areas is to enable contention free allocation of new objects; a thread running on a node can freely create objects without the need to synchronise with other nodes.

Work on generational garbage collection scheme [10, 19] has shown that for most applications the majority of objects that become garbage do so very soon after being created. In other words, most objects die young. This implies that frequent collection of nursery cars will reclaim a significant amount of space. If nurseries are isolated from the remaining object space it is possible to use an optimised single process garbage collector that operates independently of the distributed object space. Such a design can collect the vast majority of garbage very efficiently.

Mosaic ensures that nursery cars can be collected independently of other (shareable) cars using a *copy-out* scheme similar to that used in the Casper system [20]. In this system other nodes are prevented from gaining access to a pointer that directly refers to an object within the nursery area. Pointers into the nursery area may be created, but the system detects if such a pointer exists on a page when that page is provided to another node (as part of DSM coherency operation). Such an assignment from an existing object to a newly created object can only occur on the node that created the new object, since no other node has a legal path to that object. The system then acts to ensure that the referend object is copied out of the nursery, and that any pointers on the page provided to the remote node are updated before the page is sent.

In this way objects migrate from the nursery into the shareable DSM space. As noted by Koch [9] clustering strategies that optimise object location may be applied during copy-out. For instance a limited closure of objects reachable from the initially copied object may be moved at the one time, eagerly providing other reachable objects to the distributed computation on the same page. Care must be taken not to copy too deeply to avoid losing the benefits of a nursery area.

5.2 Car Collection

Since there is a separate area for the creation of new objects then collection of a car is not necessarily triggered by object allocation. Car collection can be initiated by any number of policy choices that attempt to optimise cache occupancy and mutator activity. Similarly there is a freedom in the choice of which car to select for collection and flexibility within the collection rules to choose target cars.

The important constraint is that no two nodes concurrently select the same car for collection and also avoid the same target cars for reassociation. Any such solutions must of course be deadlock free. One way of achieving this simply is a design where each car has an allocation pointer at a fixed location in the car.

For reassociation there are a number of opportunities to make judicious choices about which car(s) to copy these objects to. The collector could choose to allocate new cars from the free list and assign these cars to the appropriate trains. Alternatively some of required pages for the target cars may already be in that node's cache. In addition it is worth noting that the train rules as stated above can be somewhat relaxed. The forward progress of the train algorithm depends on objects being copied in one direction only, from older to younger trains. If an object is referenced from a root or younger train then it is sufficient to copy that object out of its current train to a car of *any* younger train. There is a tradeoff here between progressing an object to a car of the same train as its referent and reducing inter-node traffic.

6 Related Work

Much work has been done in the area of multi-processor garbage collection, including development of collectors suitable for large object spaces, distributed systems and multi-tier storage systems. Comparatively little work has been done in the area of garbage collection for DSM systems. Although collectors for single-processor systems can be applied on DSM systems, the costs of coherency management and distribution to the address space partitioning make this an infeasible solution.

The LEMMA system [13] uses a copying collector that requires a strongly consistent DSM system. The address space is not partitioned, meaning that the entire address space must be collected in one collection. After a global synchronisation, each node in the DSM system undertakes a parallel collection. Each node must synchronise with the others when it requires an object not stored locally. Additionally, in this system the collector must acquire write locks on pages, invalidating any application read locks.

Larchant [4] uses a copying collector with an object-based relaxed consistency model (entry consistency). This collector is a partitioned collector and allows multiple nodes to concurrently collect the one partition. It is unclear in this collector how differences in the copying locations for objects are resolved. This collector is not complete and requires both read and write barriers.

More recent work is described in [8]. This paper presents a collector that may be implemented on top of an existing DSM system. It proposes a modified reference counting model that does not support complete collection.

As can be seen from this discussion, although garbage collection has been recognised as necessary in DSM systems, little work has been done in developing collectors that are efficient, scalable and complete. Many collectors are intrusive and require global synchronisation.

7 Summary and Future Work

We present that design of Mosaic; a safe and complete garbage collection system for object systems that are implemented above page-based distributed shared memory systems. Since it is derived from a family of previously defined collectors it inherits much of their desirable attributes. It is a partitioned collector that allows nodes to concurrently reclaim regions and is able to collect all garbage including cross-partition cycles. The implementation and analysis of this system will enable further evaluation of Mosaic and comparison with existing collectors in the DSM arena.

Mosaic is neutral with respect to any DSM consistency model and hence can be readily incorporated into weak-consistency models. It exploits the special nature of pointers in type-safe object systems resulting in a proposed new relaxed consistency model for use by DSM collectors. It is non-intrusive in that it incrementally and concurrently reclaims space with minimal interference to application progress.

8 Acknowledgements

The work is supported in part by ARC Large Grant A10033014.

9 References

[1] B. Buck and P. Keleher. Locality and Performance of Page- and Object-Based DSMs. *Proc. First Merged Symposium IPPS/SPDP*, pages 687-693, March 1998.

[2] P.B. Bishop *Computer Systems with a Very Large Address Space and Garbage Collection*, PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1977.

[3] J.E. Cook, A.L. Wolf, and B.G. Zorn. Partition selection policies in object database garbage collection. In Proceedings of the 1994 ACMSIGMOD International Conference on Management of Data (SIGMOD'94), May 1994, pp.371-382.

[4] P. Ferreira and M. Shapiro. Garbage Collection and DSM Consistency. In *Proc. First Symposium on Operating Systems Design and Implementation*. (OSDI), pages 229-241, November 1994.

[5] R.L. Hudson and J.E.B. Moss. Incremental garbage collection for mature objects. 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.

[6] A.L. Hosking and R.L. Hudson, "Remembered sets can also play cards", Proc. ACM OOPSLA'93 Workshop on Memory Management and Garbage Collection", Washington DC, October 1993.

[7] R.L. Hudson, R. Morrison, J.E.B. Moss & D.S. Munro "Garbage Collecting the World: One Car at a Time". Object Oriented Programming : Systems, Languages and Applications (OOPSLA), Atlanta (October 1997), pp 162-175.

[8] D. Kogan and A. Schuster, "Remote Reference Counting: Distributed Garbage Collection with Low Communication and Computation Overhead", *Journal of Parallel and Distributed Computing*, 60(10):1260-1292, October 2000.

[9] Koch, B., T. Schunke, et al. (1990). Cache Coherence and Storage Management in a Persistent Object System. Implementing Persistent Object Bases. A. Dearle, G. Shaw and S. B. Zdonik, Morgan Kaufmann: 103-113.

[10] Lieberman, H. and C. Hewitt (1983). "A Real-Time Garbage Collector Based on the Lifetimes of Objects." Communications of the ACM 26(6): 419-429.

[11] R. Morrison, D. Balasubramaniam, M. Greenwood, G.N.C. Kirby, K. Mayes, D.S. Munro and B.C. Warboys, *ProcessBase Reference Manual (Version 1.0.6)*, Universities of St. Andrews and Manchester Report 1999.

[12] R. Morrison, R.C.H. Connor, G.N.C. Kirby, D.S. Munro, M.P. Atkinson, Q.I. Cutts, A.L. Brown and A. Dearle, "The Napier88 Persistent Programming Language and Environment", In *Fully Integrated Data Environments*, M.P. Atkinson, R. Welland (eds), pp 98-154.

[13] D.C.J. Matthews and T. Le Sergent. LEMMA: A Distributed Shared Memory with Global and Local Garbage Collection. *Proc. Int. Workshop on Memory Management (IWMM'95)*, pages 297-311, September 1995.

[14] J.E.B. Moss, D.S. Munro and R.L. Hudson, "PMOS: A Complete and Coarse-Grained Incremental Garbage Collector for Persistent Object Stores", In *Proceedings of the Seventh International Workshop on Persistent Object Systems*, pp 140-150, June 1996.

[15] J.E.B. Moss, Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach, In Proc. Second International Workshop on Database Programming Languages, Glenedon Beach, OR, pages 269-285, June 1989.

[16] D. Plainfoss , and M. Shapiro, "A Survey of Distributed Garbage Collection Techniques", In Proc International Workshop on Memory Management, Kinross, Scotland (IWMM95), pp 211-249

[17] J. Seligmann and S. Grarup, "Incremental Mature Garbage Collection Using the Train Algorithm", In *Proceedings of ECOOP'95, Ninth European Conference on Object-Oriented Programming*, pages 235-252, August 1994.

[18] Singhal, V., S. V. Kakkad, et al. (1992). Texas: An Efficient, Portable Persistent Store. Persistent Object Systems. A. Albano and R. Morrison, Springer-Verlag: 11-33.

[19] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, Apr. 1984. ACM SIGPLAN Not. 19, 5 (May 1984).

[20] F. Vaughan, T. Lo Basso, A. Dearle, C.Marlin and C. Barter, "Casper: a Cached Architecture Supporting Persistence", *Computing Systems*, 5(3):337-359, 1992.

[21] P.R. Wilson. Uniprocessor garbage collection techniques. 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.