A First Implementation of the DMOS Garbage Collector

John N. Zigman *

Department of Computer Science
Australian National University
Canberra ACT 0200 Australia

Email: John.Zigman@cs.anu.edu.au

Abstract

The DMOS, (Distributed Mature Object Space [Hudson et al. 1997]), garbage collection algorithm represents a step forward in distributed garbage collection. The algorithm possesses many properties which are desirable and/or essential for a distributed garbage collection mechanism. These properties are: safety, completeness, incremental, asynchronous, decentralization and non-disruptiveness.

To assess the effectiveness of the DMOS algorithm we have produced an initial implementation. The implementation consists of a memory manager and a mutator. The mutator may be either a simulator or execution engine of some form.

The paper first outlines the system model. Then some of the data representation decisions are explained. The memory layout and management mechanisms are detailed. Then some of the policy issues that arise in an implementation of DMOS are discussed, and the concerns arising from the implementation are covered. Finally some future work is outlined.

1 Introduction

The DMOS algorithm is one of the most recent distributed garbage collection algorithms. Its design is a step forward in distributed garbage collection technology.

As with all algorithms it is important to evaluate the algorithm through an actual implementation. The implementation of an algorithm provides a way of determining difficulties and key issues in the algorithm.

This paper covers a first attempt at implementing the DMOS algorithm. This has necessitated the implementation of both a memory management layer and a simulator/mutator.

In the sections that follow we first described the motivation, platforms and purpose of the implementation. The body of this paper discusses the implementation of the algorithm, its operation and the support facilities. Followed by a brief examination of some of the policy decisions that must be made. Then the problems encountered with the implementation are listed. Lastly a brief conclusion and future work are outlined.

2 Motivation

It is well known that explicit memory allocation and deallocation is difficult and error prone. Ideally the reclamation of memory that is no longer usable is done automatically. There has been a great deal of effort over the years put into developing algorithms for reclaiming memory. Few algorithms address a distributed graph of memory references, however DMOS is one such algorithm.

The implementation of an algorithm fleshes out the related issues and allows for practical evaluation. The evaluation of the algorithm in terms of: space reclamation rate, proportion of compute time used and scalability is important.

* The author wishes to acknowledge that this work was carried out within the Cooperative Research Center for Advanced Computational Systems established under the Australian Government’s Cooperative Research Centers Program.
2.1 Platforms

The platforms upon which the implementation is tested perform an integral role in the evaluation of the algorithms. The algorithm has been run on two platforms:

1. **AP+** a 32 site Fujitsu machine with a 2D Torus network utilizing worm-hole routing. Each site runs the Linux.
2. **AP3000** a 12 site Fujitsu machine with a 2D Torus network utilizing worm-hole routing. Each site runs either Linux or Solaris. This machine does not have the synchronization and broadcasting bus of the AP+.

The AP+ possesses a synchronization and broadcasting network, however these were not used as the experiments were to focus on the more general form of a tightly coupled machine.

2.2 Purpose

The general focus of research is on scalable parallel computing. One goal within this context is the implementation of a distributed object store on a tightly coupled machine. For a complete object store a distributed garbage collection mechanism must be incorporated into the store. The examination of DMOS and other garbage collection algorithms is an important step along that path.

3 Implementation

This section builds a picture of the overall implementation of the DMOS algorithm. The system was implemented using C++, with sections written in C to ensure adequate performance. The initial development work was done on a PC running RedHat Linux. It was subsequently ported to AP+ Linux and Solaris.

The following section describes the general structure of the implementation. The subsequent sections then cover the different facets of this implementation.

3.1 General Structure

To implement the algorithm it was necessary to make a complete Memory Management Layer along with a method for manipulating the objects. In the system developed there is a clear separation between the Mutator and the Memory Management Layer. This allows for separate implementations of different Mutators. The two levels are covered in the following sections.

3.1.1 Mutator

The Mutator performs the operations upon the objects within the system. The mutator may be any of the following variants:

- **Simulator** where the operations performed upon the objects are specified by a higher level interface. This mechanism facilitates testing and can also be used for running object manipulation traces.
- **Execution Engine** where the user level program is executed, and that program either through an interpreter or direct calls interacts with the Memory Management Layer.

The Mutator may be multi-threaded and each thread may interact directly with the Memory Management Layer.

For this implementation only the Simulator has been implemented, and traces from real application executions used to perform the object manipulations. The Simulator is only single threaded.

3.1.2 Memory Management Layer

Each site in the distributed system must have its own memory management. The Memory Management Layer must implement: Memory Allocation, Garbage Collection and Communication between it and the other memory management layers.

Two threads are used to implement the three subsystems. The three task are each managed by a thread: communication and garbage collection each have a dedicated thread, where as memory allocation is handled by the calling thread.
The memory is segmented into equal sized blocks. Each block is called a car. A Train consist of a arbitrary number of cars, (at least one). There are at least two trains present within each compute site.

The allocation of a new object places that object within an existing car or a new car. Containing of objects with cars de-fragments the heap when collection garbage.

Individual cars are selected for garbage collection. A car when selected is marked and swept, where reachable objects are re-associated according to the DMOS rules. Disruption of the mutator is at worst the time to collect one car.

3.2 Single Address Space

A single address space enables objects to be uniquely identified across the entire distributed machine. This presents the objects as a single set. Interactions with the local Memory Management Layer allow objects to be cached locally hiding the distribution of the objects from the mutator.

To enable a single address space an object must be uniquely identifiable. The following sections described the train identifier and the object identifier which form the basis for single address space references, additionally the segment identifier is covered for later use.

3.2.1 Universal Train Identifier

The DMOS algorithm associates objects with sites, trains and cars. Each train is managed by a particular site. The UTID, (universal train identifier), is combination of:

- **Site**: The site that manages the train,
- **Train**: The train number within the context of the management site.

Thus each train has a global identity without global synchronization. The UTID is defined as \( \tau : \psi_{\tau} \) where \( \tau \) is the train number and \( \psi_{\tau} \) is the site managing that train.

3.2.2 Object Identifier

An OID is an internal reference to an object and is not visible outside the object space, nor is it visible at the mutator level. An OID can encode several pieces of information. However the encoding of the information can effect different aspects of the collector and memory management layer performance.

The information that may be contained within an OID are: site the object is resident on, train, car, offset and the memory location of the object. It is necessary to encode the site that contains the object into the OID to allow the fast direction of requests and meta information. It is also necessary to allow DMOS to perform cross-site garbage collection.

If the OID specifies the address in which the object is located it will increase the access speed, however if that object is to be moved then all references to the object must be corrected. The movement of objects can be used to de-fragment memory. Copying collectors by there nature copy objects, (either logical or actual objects), this can be used as a natural de-fragmentation mechanism.

In this implementation the information encoded in the OID is:

- **Site**: The site that owns/holds the object.
- **Train**: The train containing the object.
- **Car**: The car containing the object.
- **Offset**: The object number.

The OID is defined as \( < \psi, \tau, \kappa, \phi > \) where \( \psi \) is the site number, \( \tau \) is a universal train identifier, \( \kappa \) is the car number and \( \phi \) is the offset.
3.2.3 Segment Identifier

A segment identifier, (SID), is a global label for a particular segment. Each segment belongs to a particular train, that train may be managed by a different site to the one that the segment resides on. The SID must represent the following information:

- **Train:** The UTID of the train it belongs to,
- **Site:** The site that the segment is resident on,
- **Segment:** The segment number within this site and train.

The SID is defined as tuple \( < s, \psi, \tau > \) where \( s \) is the segment number, \( \psi \) is the site where the segment resides and \( \tau \) is the train the segment belongs to.

3.3 DMOS Operation and Messages

The DMOS Garbage Collection algorithm is based on the MOS (Mature Object Space) and PMOS (Persistent Mature Object Space) algorithms. The algorithm has many essential and desirable properties, these are: distributed, decentralized, complete, asynchronous, incremental, non-disruptive and safe.

The object space is divided into blocks called trains, and is further subdivided into sections called cars. Each object is associated with a particular site, train and car. The division of the object space into small sections (Cars), enables the collector to do small discrete amounts of work at each step in the collection process.

The DMOS collector is based on the copy collection process. Objects that are reachable are reassociated (or copied) to a different part of the object space. The objects remaining in the collected space are garbage and are removed.

There are two levels to the collector: the Train and Car levels, these are described in the following sections.

3.3.1 Car Collection

In this section the rules for re-associating objects during the process of garbage collecting a Car are described. Then the reference information that needs to be propagated to other sites in the system is described. Lastly the messages that are used to forward this information and the differences to DMOS are highlighted.

**Re-association Rules**

Cars are the smallest unit of collection. The order in which cars are collected is not specified, however the system must guarantee to collect all cars eventually.

The different points in the system that can hold references can loosely be categorized as one of: heap, stack, registers, communication buffers or objects. For simplicity these are grouped into two categories: roots (including temporary) and objects. Thus a root is a reference contained in either: heap, stack, register or a communication buffer.

The collection of a car involves determining which objects that are associated with the car are reachable from outside the car, either from a root or another object. The objects that are reachable are then re-associated to other trains or cars. An object is always re-associated to a newer car or train than the one it is from. This condition ensures that live objects are associated with more recent cars, and garbage is left in old cars and trains.

The re-association of objects is dictated by the following set of rules:

1. Objects locally reachable from roots are re-associated with any local car, (on the same site), of any younger train, adding a car to that train if required.
2. Objects locally reachable from younger trains are re-associated with any local car of those trains, adding a car if required. If an object is reachable from more than one younger train, it may be re-associated with any younger train from which it is reachable.
3. Objects locally reachable from older trains are re-associated with any other local car of the current train, adding a car if required.
4. Objects locally reachable from others cars of the same train are re-associated with any other local car of the current train adding a car if required.
5. The remaining objects are unreachable and are reclaimed immediately, and the car is deleted.

Note: the objects are only re-associated within a site.
DMOS Reference Messages

Moss, Hudson, Morrison, and Munro [1998] specifies a mechanism for informing the site $\psi_H$ containing object $o$ that a reference is being sent to another site $\psi_B$. This mechanism and its complement can be characterized as a remote allocation and local deallocation model.

A site $\psi_A$ sends a reference to object $o$ to site $\psi_B$, this action by site $\psi_A$ causes a reference to object $o$ to be allocated on site $\psi_B$. Thus sending a reference from one site to another is the remote allocation of a reference.

A site $\psi_B$ may deallocate/remove a reference to object $o$ that is held locally. This then is the complementary action of the remote allocation, i.e. local deallocation.

DMOS specifies two messages, $[+, o, \psi_B]$ and $[-, o, \psi_B]$, which are used to inform the site $\psi_H$ containing object $o$ that an allocation or deallocation, (respectively), has taken place on site $\psi_B$. Site $\psi_A$ sends $[+, o, \psi_B]$ to $\psi_H$ when it sends a reference to object $o$, (remote allocation), to site $\psi_B$. Site $\psi_B$ sends $[-, o, \psi_B]$ to $\psi_H$ when it removes a local reference to object $o$, (local deallocation).

For this mechanism to operate safely messages sent from one site to another must be received in the order in which they were sent. The messages are still asynchronous and allow further optimization and batching. Each site must maintain the corresponding information about the remote references to object contain in it.

Implementation

For a particular object $o$ to be re-associated according to the rules of re-association it is necessary to know: if the object is referred to by a root, and which trains contain references to it.

There are two distinction situations to consider: intra-site reference and inter-site references.

Intra-site reference information is all available locally. It is not necessary to replicate the relevant references, however for efficiency it is necessary to reduce the search space for the relevant information. To do this each car has associated with it a list of local cars that contain references to objects associated with the car, and local roots are held in a root table.

Inter-site reference information is more problematic. For an object $o$ the containing site $\psi_H$ must know if any references to object $o$ are held on other nodes. This requirement ensures that the car collection process will not reclaim a potentially live object, (this is necessary to ensure the safety criteria).

From the earlier description of the information required by the re-association rules we can see the basic DMOS remote allocation and local deallocation mechanism does not communicate enough information. More detailed information must be available: if the object is referenced by a root, and the trains that contain references to the object.

The DMOS messages $[+, o, \psi_B]$ and $[-, o, \psi_B]$ alone are inadequate, as such their meaning is augmented and another message is introduced. The messages are all sent to site $\psi_H$ containing the object $o$ and are defined as follows:

1. $[+, k, o, \psi_e]$ Is either sent by a site $\psi_e$ or site $\psi_r$ when it sends $k$ references to object $o$ to site $\psi_e$. This message indicates that $k$ remote root reference have been create on site $\psi_e$ to object $o$.

2. $[-, k, o, \psi_e]$ Is sent by site $\psi_e$ when $k$ root references to object $o$ are deallocated at site $\psi_e$.

3. $[=, \tau, \eta, \gamma, o, \psi_e]$ Is sent by site $\psi_e$ when either the most recent train $\tau$ with a possible epoch number $\eta$ and the flag $\gamma$ indicating more than one train has references to object $o$ has changed.

Note: Epoch numbers are used in the train collection process which is covered in a subsequent section, they are included here only for completeness of the message definition.

3.3.2 Cross Site Train Collection Method

Cyclic garbage that can fit into a single car may be collected by the car collection process. It is also possible that the cycle may be too big to fit into a single car or that the cycle exists across several sites of the system. In either case it is necessary to have a train collection mechanism. A train can be reclaimed if there are no references from outside the train to objects within the train, (i.e. the train is isolated).

There are two distinct situations to consider: intra-site and inter-site train collection.

The intra-site train reclamation process uses information held within the particular site. This enables the detection of train isolation to be done within that site, and does not require any inter-site communication.

The inter-site train reclamation process is more problematic, this requires all sites that participate in the train to agree that the train is isolated. To coordinate the participating sites must be involved in a distributed termination algorithm to safely determine if the train is isolated. Although any termination algorithm can be selected, DMOS describes the operation of a token ring termination algorithm.
Segments

A train segment is used to facilitate the process of train reclamation. A train segment is a set of cars belonging to one train on one site. Each site may have an arbitrary number of segments (the site that is the master of the train must have one segment if any other site has a segment). The segments are joined together to form a (token) ring.

A site that wishes to participate in a train managed by another site must join a segment to that train. A site may join a particular segment to a token ring for a particular train by sending a Join message to the managing site of the token ring. The managing site will insert the segment into the token ring and respond to the request indicating the successor segment with a Succ message.

If a site has an empty segment it may remove it from the token ring by sending a Leave message around the token ring. When the message arrives at the predecessor of the segment to be remove it sets its successor to the successor of the segment that is leaving and then sends a Left message to the segment that is leaving.

DMOS defines the following messages for managing the token ring are defined as:

- [Join, ] (where \( \sigma = < s_1, \psi, t : \psi_m > \)), is sent by site \( \psi_a \) to site \( \psi_m \) when it wants to create a segment \( \sigma \) in train \( t : \psi_m \).

- [Leave, \( \sigma', \sigma'' \)], (where \( \sigma = < s_2, \psi_b, t : \psi_m > \), \( \sigma' \) and \( \sigma'' \) are segments), is either initially sent by site \( \psi_b \) to remove segment \( \sigma \) from the train or forwarded to a subsequent site to indicate the removal of segment \( \sigma \).

- [Left, \( \sigma', \sigma'' \)], (where \( \sigma' = < s_2, \psi_b, t : \psi_m > \) and \( \sigma'' = < s_3, \psi_c, t : \psi_m > \) are segments), is sent from predecessor of \( \sigma \) to site \( \psi_t \) to indicate the segment is to be removed.

- [Succ, \( \sigma, \sigma'' \)], (where \( \sigma = < s_1, \psi_a, t : \psi_m > \) and \( \sigma'' = < s_3, \psi_d, t : \psi_m > \)), is sent in response to a Join message to indicate that the segment \( < s_1, \psi_a, n : \psi_m > \sigma \) has joined the ring and its successor is \( \sigma'' \).

During the process of leaving or joining a token ring additional steps must be taken to ensure that the messages passed around the token ring maintain the same order. To do this two messages are introduced: Flush and FlushDone. The segment prior to the leaving or joining segment may be requested to either hold or flush its buffer before the joining or leaving operation is performed. The messages to enable this synchronization are:

- [Flush, ] (where \( \sigma = < s_1, \psi_a, t : \psi_m > \)), this is a request to send a [FlushDone, ] message to the sender of the Flush message, enabling blocking dependent upon the receivers buffer being processed.

- [FlushDone, ], (where \( \sigma = < s_1, \psi_a, t : \psi_m > \)), indicates that the buffering of messages to segment \( \sigma \) is no longer needed.

Epochs

The basic train collection method must be extended to cope with the unwanted relative problem. The unwanted relative problem occurs when an object, (during the car collection process), is associated with a previously isolated train and that association results in the train ceasing to be isolated. This action prevents the previously isolated train from being collected.

The epoch mechanism is introduced in Moss, Hudson, Morrison, and Munro [1998] to resolve the unwanted relative problem. It does this be dividing the cars in a train into two groups within each participating site. The older group is isolated, (i.e. there are no pointers into it from external sources), and the newer group may not be isolated. All participating sites have to agree on the division between the two groups. Once this is achieved, the older set can be safely removed. The process continues until the entire train is empty and the participating sites leave the token ring.

The epoch mechanism uses an epoch number, the larger the number the more recent the epoch. For a car uniquely identified by the identifiers for site \( \psi \), train \( \tau \) and car \( \kappa \), the car has a corresponding epoch number denoted: \( \eta_{\psi, \tau, \kappa} \) where

\[
\eta_{\psi, \tau, \kappa} > 0
\]

The epoch number associated with the car represents the time when the car was created.

To enable the determination of the epochs that can be collected, each train \( \tau \) has a current epoch number for each site \( \psi \) participating in the train. The current epoch is used in the determination of the division of the older and newer epoch groups. The current epoch number for a particular site \( \psi \) and train \( \tau \) is denoted: \( \eta_{\psi, \tau} \) where

\[
\eta_{\psi, \tau} \geq 0
\]

Note: if \( \eta_{\psi, \tau} = 0 \) then the cars are in the newer group.
The rules for re-associating an object are given in section 3.3.1. These rules must be augmented to integrate use of the epoch mechanism. An object \( o \) that is to be reassociated with a new car of a particular train \( \tau' \) must be associated with the correct part of the train, otherwise the unwanted relative problem will remain. Therefore where an object \( o \) associated with site \( \psi \), train \( \tau \) and car \( \kappa \) and is to be reassociated with site \( \psi \), train \( \tau' \) and car \( \kappa' \), (note: re-association is only intra-site), then either of the following rules must be satisfied:

- If \( \tau \neq \tau' \) then \( \eta_{\psi,\tau',\kappa'} > \eta_{\psi,\tau,\kappa} \), (i.e. it is to be associated with the newer group of epochs),
- If \( \tau = \tau' \) then \( \eta_{\psi,\tau,\kappa} = \eta_{\psi,\tau,\kappa'} \), (i.e. it is to be associated with the same epoch).

It may be necessary to add a car to satisfy either of the above rules.

The older group of cars in a train on a particular site must be isolated, (i.e. there must not be any external pointers to any object within that group). A pointer from object \( o \) to object \( o' \), where object \( o \) is associated with site \( \psi \), train \( \tau \) and car \( \kappa \) and object \( o' \) is associated with site \( \psi' \), train \( \tau' \) and car \( \kappa' \), then that pointer is said to be external iff:

\[
\tau \neq \tau' \quad \vee \quad \eta_{\psi',\tau',\kappa'} < \eta_{\psi,\tau,\kappa}
\]

To determine an appropriate initial value for the current epoch \( \eta_{\psi,\tau} \) for site \( \psi \) for a particular train \( \tau \) we must determine that no external pointers exist from the newer to the older group. Note: if a site \( \psi \) becomes aware of an external pointer for epoch \( \eta_{\psi,\tau} \), then the site sets \( \eta_{\psi,\tau} = 0 \).

The token ring is used to negociate a common epoch for a particular train on all the participating sites where all the objects in that epoch and earlier epochs are isolated, and therefore can be reclaimed. A site \( \psi \) can start a token at segment \( \sigma \) for epoch \( \eta \) if there are no external pointers from epochs \( \eta > \eta_{\psi,\tau} \) to any objects with an epoch \( \eta' \leq \eta_{\psi,\tau} \). The current epoch is then set to \( \eta_{\psi,\tau} = \eta \).

When a segment \( \sigma' \) receives a token which was last started be segment \( \sigma \), where \( \sigma \neq \sigma' \) one of the following rules is applied:

- If \( \eta_{\psi,\tau} = \eta \) then the token is passed on.
- If \( \eta_{\psi,\tau} \neq \eta \) and the site could initiate a token with \( \eta \) then the token is passed on.
- If \( \eta_{\psi,\tau} \neq \eta \) and the site could not initiate a token with \( \eta \) then either
  - the token is retained,
  - a new token is initiate at segment \( \sigma' \) for epoch \( \eta' \) according to the rules for starting a token.

A token that completes a traversal of the token ring indicates to the last initiating node that an epoch for reclamation has been agreed too. When a segment \( \sigma \) receives a token that was initiated at segment \( \sigma \) then it sends a collection token around the token ring to indicate that anything up to and including epoch \( \eta \) can be reclaimed.

The messages that are used to implement the token passing are defined as:

- \([\text{Epoch}, \tau, \sigma, \eta]\), is for train \( \tau \) and was last re-started at segment \( \sigma \) with epoch \( \eta \).
- \([\text{EndEpoch}, \tau, \sigma, \eta]\), is for train \( \tau \) and was confirmed at segment \( \sigma \) and all epochs at of earlier than \( \eta \) are to be re claiming.

### 3.3.3 Changing OID

An OID uniquely identifies an object, however it may be necessary to change an object’s OID. This is a result of the information contained within the OID and that information changing.

Minimally the OID must encode the site that contains the object it identifies. It is possible that that object may be moved to another site, (e.g. for storage or performance reasons), resulting in the object being assigned a new OID.

When an object’s OID changes to OID’ then all the references to that object must be updated. To safely update all the references on all the sites, an OID change protocol is invoked and managed at the site that contained the object before the change to OID’. This protocol operates as follows:

- Initially all sites that are known to have references to OID are informed of the change to OID’.
- As the site managing the protocol becomes aware of other sites that have references to OID it informs those sites of the change to OID’.
The algorithm terminates when all references to $OID$ have been removed, (this is determined by the normal pointer tracking algorithm). The site managing the protocol then informs all sites that knew of $OID$ that the change to $OID'$ is complete.

Two messages are needed to operate the $OID$ change protocol, these are:

- $[Move, OID, OID']$ This message is sent to all the sites that contain a reference to the object that has $OID$ is to be changed.
- $[EndMove, OID, OID']$ When the original home site has sent all necessary move messages, an End Move message is sent to all sites involved in the update.

### 3.4 Communication System

The Communication System consists of two parts: one manages the incoming and outgoing messages, the other provides a simple abstraction of the underlying communication mechanism.

Messages sent from one mutator to another may require DMOS messages to be sent to maintain correct pointer tracking. It may be necessary to translate the information contained in the message into another form before transmission and after reception. The mutator should not be relied upon to perform these additional operations.

The DMOS operations presented are either for pointer tracking or for propagating information safely through the use of termination algorithms. It is desirable to be able to replace termination algorithms, to enable the evaluation of the performance of the different termination algorithms in the DMOS context.

The communications system queues outgoing and incoming messages. Sending messages when required and receiving messages when they arrive. To deal with these requirements the processing of these messages is managed by a set of message handlers. For each message type it is possible to associate three message handlers. The handlers are called at different stages of the process. The handlers are:

1. $Pre$: is called when the message is passed to the communication system to be sent.
2. $Post$: is called after the message has been sent.
3. $Del$: is called immediately prior to the deletion of the message from either the send or receive queue.
4. $Rec$: is called when the message is received.

This enables different communication modules to handle different underlying mechanisms, and different communication behaviours to be instantiated.

### 3.5 Meta-Data Tables

It is necessary to maintain some meta-data to enable the correct functioning of the DMOS algorithm, as well as the reference handle mechanism. The following sections briefly outline the meta data that is maintained.

#### 3.5.1 $OID \leftrightarrow Handle$

Object identifiers are hidden from the mutator. All accesses to objects are done through handles. The handles provide a level of indirection that can facilitate the operations of the memory management and collector levels of the system. Although the use of handles adds a level of indirection, for a small number of accesses it is no less efficient than swizzling, however as the number of accesses increase it becomes less efficient than swizzling.

The use of handles requires a mapping from each object identifier to the corresponding handle. This meta data is maintained in a table held by the Memory Management Layer. The structure used to hold this information is a $B^+$ Tree indexed on the $OID$.

#### 3.5.2 $OID \leftrightarrow IncomingReferenceInformation$

The DMOS pointer tracking algorithm and the extensions to it, require information regarding inter-site references are made available to the sites containing the objects that are referenced. This information can be a summary of the available information to minimize the space required and the messages sent to allow pointer tracking safety and the re-association rules to be applicable. The summary information for references to an object is collected on a per site basis, as such for each site the following information is held:
6th IDEA Workshop, Rutherglen

- $r$ a count of Remote Pointer Allocations,
- $\tau$ the train identifier of the most recent train referencing the object,
- $\eta$ the most recent epoch number, (if applicable), of the most recent train referencing the object,
- $\gamma$ a flag indicating whether two or more references to the object exist in different trains, (used for determining isolation).

The site containing an object knows whether there are any external references to that object, and what trains those references are held in. This is to provide both safety and adequate information for re-association.

### 3.5.3 $OID \mapsto$ OutgoingReferenceInformation

The outgoing information is held for two reasons: to enable only summary information to be sent to the site holding the object, and to enable updates to the references in the event of an $OID$ change message being received.

The summary information sent to the site holding the object must be easily calculated, additionally the information is only sent if there is a change in the summary information. From the previous section and the requirement for possible $OID$ updates the following information is held for each object held remotely:

- $r$ the number of remote allocations made,
- $k$ the number of local root copies made,
- $< \tau, \eta, n >$ a set of trains, epoch numbers and counts for references to the remotely held object. Each entry is a triplet identifying the train $\tau$ and the epoch $\eta$ of the cars containing the references along with a count $n$. The train $\tau$ and epoch $\eta$ define the set.
- $< \tau, \kappa >$ a set of local train $\tau$ and car $\kappa$ identifiers indicating which cars hold the references.

### 3.5.4 $OID \mapsto OID$

This is used for either migrating or modifying the $OID$ associated with a particular object. Each object has a unique $OID$, when an object is move it may necessitate a change in the $OID$. An obvious example is the relocation of an object from one site to another.

Associated with each mapping is the set of sites which are known to have references to $OID$. This is maintained to ensure that sites previously informed of the $OID \mapsto OID$ are no informed again.

### 3.6 Memory Management

All memory management is done internally. Initially large sections of memory are allocated to hold data required for the system operation and the user level objects.

The allocated memory is divided into a small number of large pools. Each object is allocated in one of the pools. Each pool is divided into a number of equal size segments. Objects are allocated to the segment that is large enough to contain it. Selection of the segment is by a simple stack based mechanism.

There is a finite number of Cars allocated. Each Car is a fixed size, (for simplicity large objects are not dealt with). Other objects such as the $B^+$Trees holding meta data.

### 4 Issues

The implementation of any algorithm involves decisions ranging from policy decisions through to technical problems related to the platforms selected. These issues are cover in the following sections.

#### 4.1 Policy Decisions

To implement DMOS several policy decisions must be made. These issues are classified as policy decisions as they affect the operation of the DMOS algorithm but not the completeness. These policies are:

- Which car to collect next?
• Which train to re-associate an object with when there is more than one possibility?

• Which car to associate a newly created object?

• When to close a train?

• When to add a new train?

The intra-site garbage collection processes on car at a time. For completeness the local garbage collection process must guarantee to collect every car, however the order in which these cars are collected is not specified. If the unreachable objects, (garbage), is not evenly distributed among the cars then ideally we process the cars that contain the most garbage first. What heuristics and methods can be used to achieve this?

The DMOS rules for re-associating objects can result in more than one candidate train. The re-association of an object with one of many trains should consider: the likely life of the object and the expected times in which the trains are to be collected. The more accurately this can be predicted the shorter the time the unreachable objects will be kept.

A newly created object is initially associated with a particular car and train. The system may have several candidate trains. Similarly to the previous policy, the selection of the train and car ideally will allow the particular car to remain uncollected until just after the object becomes unreachable. Again this relies on the estimated life span of the object and the collection times of the cars.

The inter-site garbage collection process, (train collection across sites), ideally is invoked when the probability of the train being collectible is high. How is this determined, so that a train can be closed and inter-site collection for this train proceed?

The DMOS collector requires that there are at-least two trains on each site at any given time. It is possible and may even be desirable to have more trains. How is the need for another train determined, what heuristics are needed?

These issues must be address fully to obtain the best performance from the DMOS garbage collection mechanism.

4.2 DMOS Issues

The DMOS rules for object re-association require train identifier and epoch numbers to be known on the site containing the object referred to. The pointer tracking algorithm, while providing safety, does not include enough information for the re-association rules to function. This has meant the introduction of additional messages to communicate this information.

4.3 Representational Problems

In any implementation the data representation must be chosen. The \textit{OID} is 64 bits in size and contains: the site, train, car identifiers and the offset for that car. The fixed size of the data items constrains the range of values any one of the parts of the \textit{OID} can have.

The DMOS algorithm assumes that train and car identifiers can go on increasing without limit. This is clearly not the case for this implementation, and it would not be practical in general. This leaves the possibility of values cycling, this implies that some form of synchronization may be needed to maintain the ordering properties of the system.

This implementation does not address this issues and assumes the system is not running long enough, and that the problem size is such that value cycling will not occur. However a robust production system, may have to address this issue.

4.4 Technical Problems

The technical issues that hindered the implementation of the system were:

• Thread Safe Libraries,

• Templates.

The Linux 5.0 standard c libraries were not thread safe. The problem manifested itself primarily with the input output routines used in conjunction with \textit{stderr}. The system would crash when writing trace/debug information.

Template classes were heavily used for \textit{B+ Trees} and memory management. The template handling of GCC is incomplete and caused several redesigns to work around the problems. Problems such as the initializing of statics in templates.
5 Conclusion and Future Work

The implementation of the DMOS algorithm is not straightforward. The interactions between the different components is involved. It is difficult to ensure that the operation is deadlock free when several threads are used to handle the different facets of the algorithm.

Different policy decisions and methods will change the effectiveness of DMOS.

In this implementation the termination algorithms are encoded into the message handlers. To implement different termination algorithms it is necessary to change the messages used and write corresponding message handlers. The selection of different termination algorithms may provide more effective message passing and less contention.

The investigation of different OID encoding levels and representations may provide performance gains. Changing the physical association of objects with cars to and abstract association will change: message passing and car collection characteristics.

The investigation of these issues within the context of different applications is yet to be done. Insights into the different performance characteristics may provide for different policies for different applications.

Bibliography

