Garbage Collection for Storage-Oriented Clusters

William Brodie-Tyrrell, Henry Detmold, Katrina Falkner, David S. Munro
School of Computer Science, University of Adelaide
Adelaide SA 5005, Australia
{william, henry, katrina, dave}@cs.adelaide.edu.au

Abstract

Storage-oriented clusters present unique challenges to the implementation of storage management. Such clusters manage a vast amount of data, most of which is located on secondary storage. Manual storage management in storage-oriented cluster environments is complex, error-prone and tedious. As a result there is a clear need for automatic storage management (garbage collection) for such clusters.

The goals of a garbage collector for use in a storage-oriented cluster are safety, completeness and scalability in the face of distributed cycles of garbage in secondary storage. Of the few extant distributed secondary storage garbage collectors, none meet all of the stated goals whilst also operating efficiently.

This paper describes the design and implementation of a new distributed garbage collector based on the train algorithm, specifically for use in storage-oriented clusters. The collector presented here extends the train algorithm, employing an asynchronous distributed termination detection algorithm for isolated train detection, a mechanism for deferring the update of metadata and a new external root tracking mechanism to permit interaction with clients that cache and swizzle pointers. Our experiments demonstrate that these extensions successfully adapt the train algorithm for efficient operation in a storage-oriented cluster, fulfilling the stated goals of safety, completeness and scalability.

1 Introduction

The focus of research into cluster computing has concentrated on computational clusters, while storage-oriented clusters have received less attention. Clustering of storage provides scalability of I/O performance without the use of expensive special-purpose hardware. Computational clusters provide supercomputer-class performance at commodity cost, similarly storage-oriented clusters have the potential to provide mainframe-class I/O for commodity cost. Significantly, there is a growing need for such cluster technology to support an important class of applications that exhibits strongly connected graphs of data objects within very large data sets. Finite Element Analysis for the simulation of physical systems is a widely used member of this class. However, storage-oriented cluster technology is currently less mature than computational cluster technology and many of the key problems in the implementation of storage-oriented clusters have yet to be adequately addressed. Our research attempts to solve key problems in storage management for storage-oriented clusters.

Our model for programming storage-oriented clusters is based on the orthogonal persistence abstraction over storage (Atkinson, Bailey, Chisholm, Cockshott & Morrison 1983), permitting the form of programs to be independent of the lifetime of data they manipulate and the lifetime of data to be independent of its type. Extending this abstraction to distribution makes the form of a program independent of the underlying stable storage mechanism and the number of storage sites participating. The benefits of the persistence abstraction have been well documented (Atkinson & Morrison 1995) and are particularly appropriate for the class of long-running applications expected on storage-oriented clusters.

To manage the storage resources of long-running applications, it is essential to identify and reclaim the storage allocated to those objects that are no longer needed. The consequence of not doing so is that applications will fail unexpectedly and unnecessarily due to exhaustion of free space; the approach of discarding the entire address space at the end of execution is

not acceptable in this environment. Accurately identifying the objects that are no longer required so that they may safely be reclaimed is a complex, error-prone and tedious task even for small programs where the data structures are simple. Furthermore, when the majority of those objects are located in secondary storage, there are significant engineering challenges to performing both identification and reclamation efficiently. Finally, ascertaining liveness of an object in a distributed system (such as a cluster) requires global knowledge. However, in order to maintain scalability (and hence efficiency) this global knowledge must be distilled over a series of asynchronous local steps. Moreover, these steps should involve only the minimal subset of sites necessary to identify a particular garbage component.

Any implementation of storage management at the application level must correctly address all of the problems identified above. At best, this burdens application programmers with a tedious and time-consuming problem. A more likely outcome (and one which is much worse) is that the application level storage management is incorrect in some aspect, resulting in unsafe reclamation of live objects or failure to reclaim garbage or both. It is therefore virtually essential to take a systematic and automatic approach to storage management, namely, the implementation of a distributed garbage collector.

The train algorithm (Hudson & Moss 1992, Seligmann & Grarup 1995), from which our research is derived, exhibits the desirable properties of *safety, completeness, asynchrony and scalability.* These align with the majority of the requirements identified above, hence justifying the use of this garbage identification technique. The train algorithm has previously been shown to be amenable to implementation in a wide range of architectures, each with their own unique set of challenges. Existing adaptations of the train algorithm include generational (Hudson & Moss 1992), uniprocessor persistent (Moss, Munro & Hudson 1996), distributed main memory (Hudson, Morrison, Moss & Munro 1997) and distributed shared memory (Munro, Falkner, Lowry & Vaughan 2001) garbage collection. The challenge is to adapt the train algorithm to use in storage-oriented clusters, a context where it has not yet been seen.

The contributions of this paper are the design, implementation and initial evaluation of a new distributed secondary storage garbage collector based on the train algorithm. This new collector adapts and combines a number of proven techniques to provide:

- safety through pointer tracking,

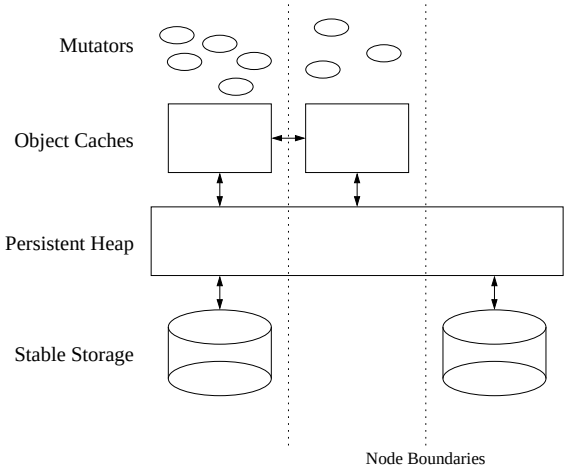

Figure 1: Layered ProcessBase Architecture

- completeness through isolated train detection, underpinned by Safra's Algorithm (Dijkstra 1987),
- tolerance of secondary storage latency through a deferred metadata update similar to that used in PMOS (Moss et al. 1996), and
- scalability of persistent garbage collection by incrementally gathering external reference information.

To provide an efficient programming abstraction, it is necessary for sites to cache data in primary memory. A further contribution of the distributed store described here is a new technique for tracking pointers in client caches, which is necessary to permit the decoupling of garbage collection in the persistent heap from that in the client caches.

This paper describes the architecture and experimental platform in which the garbage collector exists, the challenges confronting such a collector and their solutions as well as initial experimentation performed to confirm the validity of our approach.

2 Architecture

Our architecture includes distribution of both storage and computation across the nodes of a cluster. The computational model is based on the ProcessBase (Morrison, Balasubramaniam, Greenwood, Kirby, Mayes, Munro & Warboys 1999) programming language. The storage model is an orthogonal persistence (Atkinson et al. 1983, Atkinson & Morrison 1995) abstraction supported by the ProcessBase runtime system and a distributed stable heap.

The focus of the work described in this paper is in the heap layer. In particular, we have designed and implemented a novel secondary storage storage garbage collector, DPMOS (Distributed Persistent Mature Object Space), based on the train algorithm.

The secondary storage architecture is a set of peer *sites* that are in communication with each other to synthesize a single address space from separate stable storage held on each site as per Figure 1. Client virtual machines can connect to any of the participating store nodes; a store site need not have both a connected client and local storage. The stable stores are independent and do not communicate with each other. Object locations are encoded in their identifiers, providing a trivial resolution scheme.

Each virtual machine has a write-back object cache and a collection of mutator threads that operate over the cached objects. ProcessBase is a persistent language system designed for use in *hyper-programming* (Kirby, Connor, Cutts, Dearle, Farkas & Morrison 1992) and process-modelling systems and for which there is ongoing research into distribution of computation and storage at both the persistent heap and cache levels (Howard, Detmold, Falkner & Munro 2003, Falkner, Detmold, Munro & Olds n.d.). The ProcessBase programming model is that of multiple concurrent mutators that operate directly on objects held in the caches.

Cache coherency is the responsibility of the clients, however certain events relating to the copying and destruction of references must be reported to the persistent heap to permit safe garbage collection therein.

Our secondary storage garbage collection algorithm operates within the persistent heap layer. A common approach to achieving scalability through incrementality in a garbage collector (distributed or otherwise) is to partition the object space and reclaim garbage in partitions individually. Distributed acyclic garbage is reclaimed by repeated invocation of the local garbage collector in conjunction with a pointer tracking algorithm that maintains a conservative set of *external roots* for each partition. Distributed cyclic garbage is reclaimed using an asynchronous implementation of the train algorithm.

Stable storage is implemented at each site using an after-image shadow paging (Gray, McJones, Blasgen, Lindsay, Lorie, Price, Putzolu & Traiger 1981) mechanism and page cache. The shadow paged store provides stability with checkpoint semantics and its use of a cache permits the direct mapping of data structures in the heap onto the page store's address space without significant performance loss. The cache supports page-pinning to explicitly retain critical and frequently-accessed data structures in the cache. The stable storage space is a flat address space with no concept of objects, references or reachability.

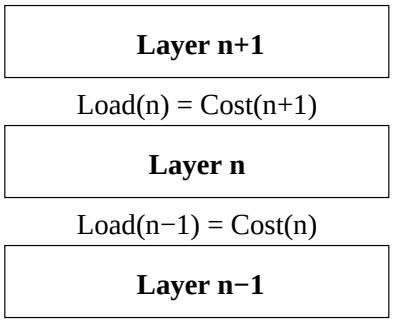

Figure 2: Layering in Measurement Model

Communication between sites is by asynchronous message passing over a network that preserves FIFO ordering of messages between endpoints. The system assumes reliable message delivery, message loss results in graceful shutdown. Similarly, the algorithm is not crash tolerant and the loss of a site also leads to graceful shutdown. Finally, Byzantine behaviour is not permitted anywhere in the system.

2.1 Measurement

The architecture has been designed in such a way as to facilitate measurement and evaluation. In particular, the layering of the architecture permits the substitution of any layer for the purposes of comparison and is the basis for quantitative performance and scalability measurements. This enables our system to be used in a detailed investigation into distributed garbage collection.

By focusing measurement on activity at the interfaces between layers, it becomes possible to directly compare the effects of different policies within a layer or even two entirely different implementations of the same layer.

Each layer in the architecture provides a service to layers above it by composing a mechanism and policies of its own with requests of lower layers. The load upon a layer n is defined as the set of requests made of that layer while the cost of a layer is the set of requests it makes of the layer below itself, see Figure 2. Each layer transforms its load into a cost by the application of policies to mechanisms, with those policies possibly specified by higher layers in the case of a compliant system (Morrison, Balasubramaniam, Greenwood, Kirby, Mayes, Munro & Warboys 2000). There may be multiple independent systems at each layer, for example the Persistent Heap operates over

the Stable Storage and Network systems, each considered to be a layer below the Persistent Heap yet independent of each other. In principle the load upon a layer could be the union of accesses from multiple higher layers but this does not occur in the DPMOS architecture: there are no circumstances where multiple systems in an upper layer operate directly upon a single system in a lower layer. The long-term goal of this research is to attempt to measure quantitatively the transformation of load into cost at the persistent heap layer and thereby to make comparisons between different distributed persistent garbage collectors.

In the case of the distributed persistent garbage collector, the load is represented in terms of object read and write requests and pointer tracking events from caches; the cost can be measured in terms of the network traffic generated and the load placed upon each of the stable stores.

3 Challenges

The design and implementation of a distributed persistent garbage collector for use in storage-oriented clusters pose a range of challenges that are addressed by our research. These relate to:

- The efficient and accurate tracking of the creation, destruction and transmission of references external to a partition.
- Accessing stable data in such a way as to be tolerant of secondary storage latency.
- The efficient operation of a scalable distributed algorithm to accurately identify distributed cycles.

This section describes in detail the three challenges addressed by our research.

3.1 Pointer Tracking

To perform safe garbage collection of a partition, information regarding references into that partition is required. Whilst it is in principle possible to acquire this information via a global trace, this renders the algorithm unscalable. Hence, to support scalability of collection, inter-partition reference information must be maintained incrementally. This process is known as pointer tracking. The algorithm described in (Hudson, Morrison, Moss & Munro 1998) provides an efficient solution to the pointer tracking problem but is suited only to a single layer store where references are not cached at a higher layer. However, the presence of caches (necessary for efficiency) in a storage-oriented cluster results in a two-level architecture, therefore it is necessary to adapt the pointer tracking algorithm to cope with references held in caches.

For scalability, it is desirable to decouple garbage collection in caches from garbage collection in the persistent heap so that both may operate independently. This ensures the scalability of the persistent garbage collector because it does not require synchronisation with and interruption of the caches. For these reasons, a new pointer tracking mechanism is required that encompasses client caches and is aware of their behaviour.

3.2 Latency Tolerance

Garbage collection in primary memory is a well-investigated area (Wilson 1992). In this field, algorithms typically assume that storage is randomly addressable with little penalty, i.e. random accesses to storage have low latency. Clearly, this assumption does not hold for disc storage. For a secondary storage garbage collector to operate efficiently, it must be implemented so as to take account of the read and write latency of secondary storage and to minimise the impact of that latency on overall performance. The main strategies available to the implementation are caching in main memory and optimising access patterns so as to prefer sequential access over random access.

The pointer tracking algorithm required to support partitioned collection entails the creation and maintenance of metadata, recording inter-partition references. As with all other data, this metadata must be kept on secondary storage. There is at least one metadata update for every inter-partition reference update. Furthermore, a given reference update and its associated metadata updates are physically dispersed. These properties complicate the implementation of an approach to maintaining metadata that is tolerant of high storage latencies, whilst simultaneously making it more important to adopt such approaches.

3.3 Collection of Cyclic Garbage

Reference counting algorithms, such as pointer tracking, are inadequate to detect cycles of garbage that span partitions. In a main memory collector one might simply ignore such cycles since they will disappear when the program exits. This is not a viable design choice in a storage-oriented cluster since the address space is persistent and so completeness is thus a necessary requirement. Hence, an additional distributed algorithm is required to handle this case and this algorithm must interact safely with the pointer

tracking algorithm. Furthermore, the distributed cycle detection algorithm must be efficient and scalable. This is an inherently challenging problem that has been the subject of extensive research (Tel & Mattern 1991, Blackburn, Hudson, Morrison, Moss, Munro & Zigman 2001, Lowry & Munro 2002, Norcross, Morrison, Munro & Detmold 2003).

4 Implementation

Here we describe our approach to solving the specific challenges of implementing a safe, complete, asynchronous and scalable distributed persistent garbage collector posed in the previous section.

The generic train algorithm, on which our collector is based, operates as follows. Objects are stored in *cars* (partitions), of which there are many per site and which do not cross site boundaries. Each car can be collected independently of any other car in the system and the inter-partition reference information (from the pointer tracking algorithm) required to do this is kept in *remembered sets* (remsets). Each car belongs to a *train*, which may span multiple sites. When a car is collected, the reachable objects in it are *reassociated* (copied) into other cars, possibly in other trains, depending on the information in the remset and then the space occupied by the collected car is reclaimed. The reassociation policy selects a destination train for each object so that distributed cycles are marshalled into a single train. In addition, reachable objects are eventually moved into other trains, leaving this train *isolated*. Train isolation is a stable property which may be established using a Distributed Termination Detection (DTD) algorithm.

4.1 Pointer Tracking

There are two cases where pointer events requiring tracking occur:

- Object writes issued by caches and pertaining to objects on a site other than the one the cache is connected to (remote writes). A remote object write involves the transmission of references from a cache to a persistent heap site and the creation of references in the latter; this case (the creation and destruction of references in the object written as well as the presence of references in network messages) is handled by an implementation of the pointer tracking algorithm defined in (Hudson et al. 1998).
- Remote object reads issued by caches. These involve the creation of references in a cache and the transmission of references if the read is remote. Because the references created are not in the persistent heap, they cannot be tracked by Hudson et. al.'s algorithm. Our solution to this is described below.

Caches in our architecture *swizzle* (translate) references from the persistent heap's format to their native machine address format in order to provide acceptable computational performance in the virtual machine. Our approach is to track references as they enter (are faulted into) and leave (are purged from) caches. Swizzling allows us to ignore the reference creations and deletions performed by mutators.

The cache pointer tracking algorithm must accurately report when a non-zero count of references to a given object are held by any cache. Furthermore, this must be done without synchronising with caches and without information regarding mutations of swizzled references.

References are tracked separately, depending on whether or not they are swizzled, only the non-swizzled references are counted by this algorithm. References to an object can be swizzled only if a copy of the object is held by the cache, so the pointer tracking algorithm conservatively assumes that if an object is present in the cache then there are swizzled references to it. If an object is purged to make space in the cache, all references to it must be deswizzled (and therefore counted).

The ProcessBase virtual machine (VM) operates only on swizzled references and so has no direct interaction with the cache pointer tracking algorithm. Since the pointer tracking algorithm is only invoked during object fault and purge, its impact on the operation of the virtual machine is minimal.

A fault will occur at any point where the VM attempts to dereference an unswizzled pointer to an object not present in the cache. If the object is already present, the pointer is swizzled, resulting in a minus message. Our system uses a simple policy by which purges are triggered only by cache exhaustion. If swizzled pointers exist in a modified object to be purged, they must be unswizzled (requiring plus messages) before the object is written back.

For each object contained in or referred to by one or more caches, a boolean value and counter, each vectorised by cache identity, is kept. These are referred to as $R_b[X]$ and $R_c[X]$ respectively. $R_b[X]$ denotes the presence of R in cache X and $R_c[X]$ contains a conservative estimate of the number of unswizzled pointers to R in X. If either of these values are non-zero, R is considered live and cannot be reclaimed by the persistent heap collector.

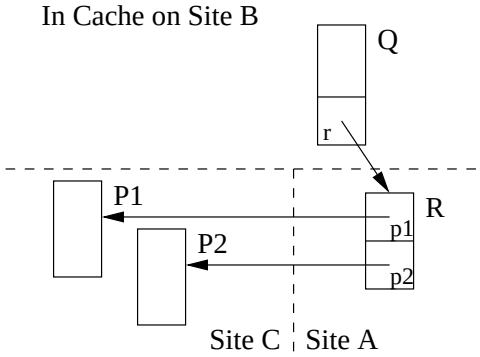

Figure 3: Example Object Graph

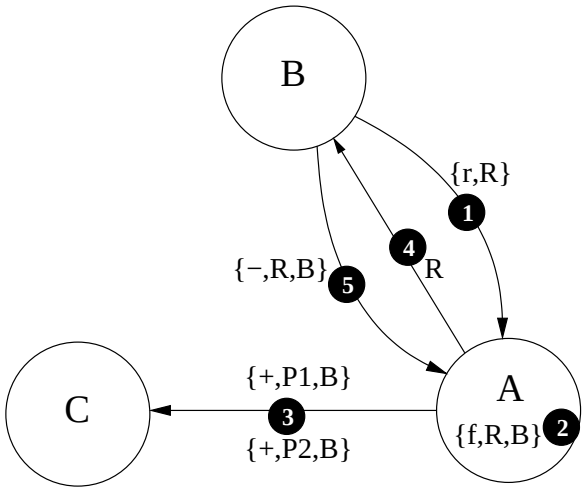

Figure 4: Fault Sequence

The tracking of references in the caches is described by four events:

- *Plus*: {+, R, B} indicates the creation of an unswizzled reference R at cache site B; it causes $R_c[B]$ to be incremented.
- *Minus*: {-, R, B} indicates the destruction of an unswizzled reference R at cache site B; it causes $R_c[B]$ to be decremented.
- *Fault*: {f, R, B} indicates that a copy of R is held in cache B; it causes $R_b[B]$ to be set.
- *Purge*: {p, R, B} indicates that B has lost its copy of R; it causes $R_b[B]$ to be cleared.

Consider the example object graph of Figure 3. Object R resides on site A and objects $P1$ and $P2$ reside on site C. A cache at site B holds a reference r (to R) in object Q. Figure 4 and Figure 5 describe the fault and purge sequences of events, respectively.

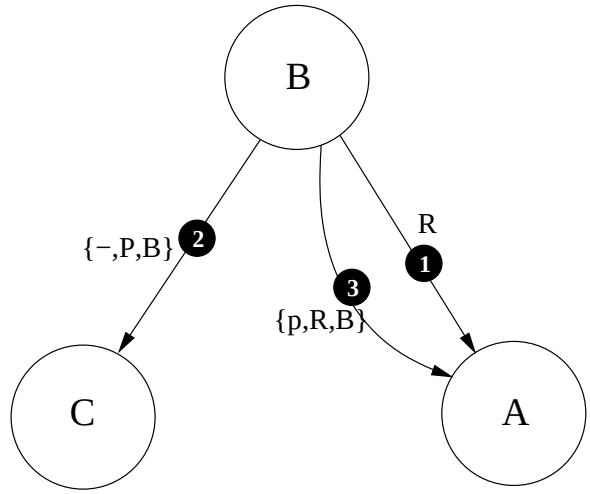

Figure 5: Purge Sequence

When a mutator on B dereferences the reference to R in Q, the reference must be swizzled. For this to occur, R must be faulted in (Figure 4) so B sends a read request (1) to A. The fault event {f ,R, B} (2) occurs at A and plus events are generated at A for every reference ($p1$ and $p2$ in Figure 3) in R. Also, a message is sent (3) to the home site C of each object Pn for which a plus event occurred. The contents of R are then transmitted (4) to B which can swizzle the reference in Q, resulting in the transmission (5) of a {-, R, B} message to A.

When B purges R (Figure 5), it writes the contents back (1) to A if R was modified. This involves pointer transmission and the creation and deletion of pointers, all of which are tracked by the protocol of (Hudson et al. 1998). Minus messages (2) are generated for each unswizzled reference in R and A is notified (3) that R has been purged from B.

Our architecture uses a checkpoint model of stability and permits the sharing of references between object caches; our implementation of the persistent heap is necessarily conservative in its treatment of references held by client caches. In contrast, transactional systems introduce a degree of synchronisation between client and server (Amsaleg, Franklin & Gruber 1995), permitting the server to make assumptions about when references can be created.

4.2 Latency Tolerance

In addressing the latency of secondary storage access, our focus is on reducing the costs of metadata update. We pursue two approaches to this:

- caching of metadata to amortize secondary stor-

age access costs, and

- batching of updates to metadata to enable secondary storage writes to be performed sequentially.

Eagerly updating remsets on secondary storage is infeasible because updates are required at up to three addresses not spatially correlated with each other: the reference update, a decrement at the remset for the pointer overwritten and an increment at the remset for the pointer created. This implies up to three seek operations for each pointer update and results in unacceptable throughput. The solution used in DPMOS is to defer and batch updates until an opportune time or when the log of deferred updates has become too large (a policy decision) and must be purged. This solution means that multiple updates to a single remset are batched into a single disc-block write which requires only a single seek. Our results show that where a block write is dominated by seek time such batching improves performance.

DPMOS remsets have a tree structure in memory, permitting O(log n) search and update but are flattened for placement on secondary storage. There is a fixed size cache of remsets in memory. Remsets are purged from the cache when a new remset is required to be read in.

If a reference update occurs and the desired remset is not in memory, the update is appended to a structure in memory referred to as a Δref set. This set contains a vector of records, each describing an increment or decrement operation on a remset entry. Δref sets are the mechanism for deferring and batching updates to remsets, the assumption being that many operations will be stored in a Δref set before the corresponding remset is loaded and updated. On loading a remset into the cache, any applicable Δref is applied. When the space occupied by Δref sets becomes too large (again, a policy decision), remsets are faulted in and updated.

Δref sets are always applied to their respective remsets on checkpoint and therefore never reside on stable storage. Our initial experiments lead us to believe that checkpoint speed may be improved at the cost of complexity and recovery speed by storing Δref sets stably instead of eagerly applying them at checkpoint. We will investigate this alternative approach in future work.

The stable storage subsystem used by DPMOS is a shadow paging system with a page cache and checkpoint durability semantics. Partitions being collected are pinned into the cache to permit a garbage collection algorithm designed for primary memory to operate efficiently. The partitioned nature of the store and its incremental collection means that only a small amount of persistent space needs to be in cache at any given time.

PMOS (Moss et al. 1996) is a uniprocessor persistent implementation of the train algorithm that uses deferred metadata updates to gain performance. PMOS maintains two such caches: the Δref and Δloc sets. The former is similar to DPMOS's Δref system and the latter describes object motion due to garbage collection. DPMOS does not require a Δloc set as there is an indirection between persistent object identifiers and their physical location; in contrast, PMOS must update all references to an object when it is moved by the garbage collector and the Δloc set defers these updates.

4.3 Collection of Cyclic Garbage

The collection of distributed cycles requires a mechanism to identify isolated trains. The reassociation policies employed by the train algorithm cause distributed cycles to move into a single train and remain there if unreachable while live objects will be reassociated out of the train. A partition along train boundaries is thereby formed between reachable and unreachable objects. The reclamation of a whole train is the process by which inter-partition and distributed cycles are reclaimed.

Liveness of a site in a computation subject to DTD is equivalent (Tel & Mattern 1991, Blackburn et al. 2001, Lowry & Munro 2002) to the presence of reachable objects in a partition of a distributed garbage collector. Several such algorithms already exist and can be readily adapted to our system to detect isolation detection for the purpose of garbage collection. In (Norcross et al. 2003), the notion of *club rules* is investigated to permit the use of different DTD algorithms for the purpose of train isolation detection. Each collector partition corresponds to a DTD site and a train is equivalent to the DTD algorithm's computation scope.

Reassociation of an object into a train constitutes spontaneous activity in the context of a DTD algorithm so is not permissible if a DTD algorithm is to be used to detect train isolation. Therefore a train is *closed* (Lowry & Munro 2002) during the execution of the DTD algorithm, thereby prohibiting the reassociation of objects into that train. Should the DTD fail (i.e. not detect isolation), the train is reopened and reassociation recommences. A correct asynchronous DTD algorithm is safe with respect to the exportation of references from a train since such exportation

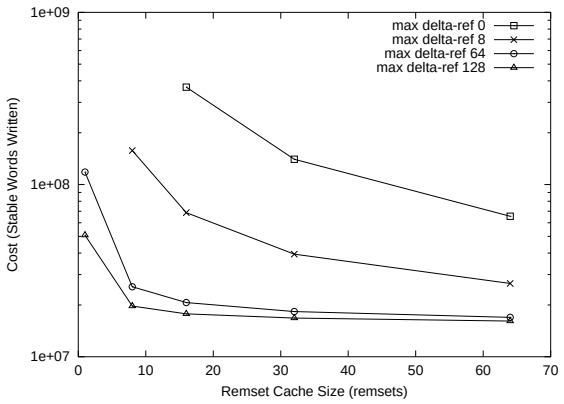

Figure 6: Effect of Caching Metadata (FEA)

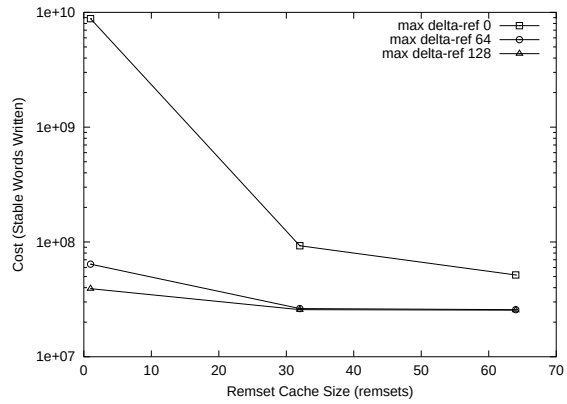

Figure 7: Effect of Caching Metadata (OO7)

can occur only if there is an existing external reference to the train.

We have chosen Safra's Algorithm (Dijkstra 1987) for the purpose of detecting isolated trains. This is an asynchronous ring-based wave algorithm for distributed termination detection. It is particularly appropriate for our purposes not only because it is asynchronous but also because it closely matches the structure of trains in our implementation. The closing and opening of trains is performed by the execution of a wave over the ring of sites participating in a train. The FIFO nature of message delivery and the use of waves over a ring to manage train membership as well as open and close trains ensures that train management events do not overlap with the opening or closing of a train.

5 Experiments and Results

Our experimental platform consists of a pair of Athlon XP systems: one of 1527MHz and 1024MB RAM and the other 1470MHz and 512MB RAM connected via switched fast ethernet. Wall-clock time is the only machine-dependent result as the I/O costs are deterministic (depending only on configuration and the load data structures) and therefore independent of the hardware used for execution.

To verify integrity of the store, the OO7 (Carey, DeWitt & Naughton 1993) "medium" data structures were created, installed into the store and verified. To measure performance in the face of highly connected data, a small finite-element simulation was written in ProcessBase and checkpointed. It creates a 2D planar mesh of triangles, each connected to all of its neighbours and objects representing its vertices. As an initial experiment, the effect of the remset caches and Δref sets were measured by observing the variation in stable I/O with respect to the size of both remset caches and Δref sets.

Figure 6 shows at least one order of magnitude reduction in execution cost (as measured by the load on the stable storage layer) is possible by using the latency tolerant metadata update schemes described in this paper. The results also show that there is little benefit in increasing Δref set and remset cache size beyond a certain minimum.

Experiments with OO7 (Figure 7) in conjunction with extremely small remset caches and Δref sets show approximately 2.5 orders of magnitude difference in I/O cost between the highest and lowest cost configurations. The OO7 data structure has lower connectivity and therefore fewer inter-partition references than the FEA test, so as expected the use of metadata caching has less effect.

Further experimentation into caching policies (including Δref sets and remset caches) and other system policies is ongoing.

6 Related Work

Distributed persistent object stores have been designed and implemented in the past but their garbage collectors have in general suffered from a lack of completeness, scalability or both.

Larchant (Shapiro & Ferreira 1995) and PerDIS (Ferreira, Shapiro, Blondel, Fambon, Garcia, Kloosterman, Richer, Roberts, Sandakly, Colouris, Dollimore, Guedes, Hagimont & Krakowiak 2000) are distributed persistent stores with garbage collection but they are not complete in the face of large distributed cycles and are only probabilistically complete with smaller distributed cycles.

Thor (Maheshwari 1993) is a distributed transactional object store that can perform dis-

tributed garbage collection by controlled migration (Maheshwari & Liskov 1997*b*), distributed marking (Maheshwari & Liskov 1997*c*) or back-tracing (Maheshwari & Liskov 1997*a*). Controlled migration requires that an entire distributed cycle be migrated to a single site to be reclaimed by that site's local garbage collection and is therefore not scalable to large connected components of garbage. Distributed marking operates in phases over all sites in the system, regardless of the sites that the component to be reclaimed actually spans. The global nature of garbage detection in that scheme means that it is not scalable to large numbers of nodes. Thor's back-tracing algorithm is complete and scalable but space- and time-inefficient because of its requirement to store state for each remset entry involved in a back trace. Safety of the back trace also requires a pointer tracking algorithm that is theoretically more expensive than that used by DMOS and DPMOS.

7 Further Work

The context of this research is a detailed comparison between two different approaches to obtaining complete collection in the face of distributed cycles of garbage while maintaining efficiency and scalability. The architecture and measurement framework described in this paper is the basis for the quantitative comparisons between the train algorithm and a form of back-tracing based on Thor.

8 Conclusions

Powerful scientific applications in common use demand scalable, complete garbage collection for storage-oriented clusters. Scalability and completeness of collection are made necessary by the types of data structures employed by these applications and efficiency and scalability are required of the system which stores them.

The contributions of DPMOS are:

- The design and implementation of a new pointer tracking algorithm to permit safe garbage collection in the persistent heap, decoupled from collection in caches.
- The novel combination of the Train Algorithm, Distributed Termination Detection, Deferred Metadata Update and our new Pointer Tracking algorithm to provide safe, complete, scalable and efficient garbage collection for storage-oriented clusters.

This paper describes the first implementation of a system that uses this novel combination and shows that this new scalable and complete approach to distributed persistent garbage collection is valuable since it permits the support of a large and powerful class of applications commonly targeted by high performance and distributed systems.

9 Acknowledgments

The work is supported in part by ARC Linkage International Award LX0349049: Extending a family of garbage collectors.

References

Amsaleg, L., Franklin, M. & Gruber, O. (1995), Efficient incremental garbage collection for client–server object database systems, *in* 'VLDB95', Zurich, Switzerland, pp. 42–53.

Atkinson, M. P., Bailey, P., Chisholm, K., Cockshott, W. & Morrison, R. (1983), 'An approach to persistent programming', *Computer Journal* **26**(4), 360–365.

Atkinson, M. P. & Morrison, R. (1995), 'Orthogonally persistent object systems', *VLDB Journal* **4**(3), 319–401.

Bekkers, Y. & Cohen, J., eds (1992), *Proceedings of International Workshop on Memory Management*, Vol. 637 of *LNCS*, Springer-Verlag, St Malo, France.

Blackburn, S., Hudson, R., Morrison, R., Moss, J., Munro, D. & Zigman, J. (2001), Starting with termination: A methodology for building distributed garbage collection algorithms, *in* 'Proc. ACSC2001'.

Carey, M. J., DeWitt, D. J. & Naughton, J. F. (1993), 'The OO7 benchmark', *SIGMOD Record* **22**(2), 12–21.

Dijkstra, E. W. (1987), Shmuel Safra's version of termination detection, Technical Report EWD 998, The University of Texas at Austin.

Falkner, K., Detmold, H., Munro, D. & Olds, T. (n.d.), Towards compliant distributed shared memory, *in* 'WSDSM'02, 2nd Int'l Conf on Cluster Computing and the Grid (CCGRID)'.

Ferreira, P., Shapiro, M., Blondel, X., Fambon, O., Garcia, J., Kloosterman, S., Richer, N., Roberts, M., Sandakly, F., Colouris, G., Dollimore, J.,

Guedes, P., Hagimont, D. & Krakowiak, S. (2000), PerDiS: Design, implementation and use of a PERsistent DIstributed Store, Vol. 1752 of *LNCS*, pp. 427–452.

Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F. & Traiger, I. (1981), 'The recovery manager of the System R database manager', *ACM Computing Surveys* **13**(2), 223–242.

Howard, D., Detmold, H., Falkner, K. & Munro, D. S. (2003), Using the compliant systems architecture to deliver flexible policies within two-phase commit, *in* A. E. James, B. Lings & M. Younas, eds, 'BNCOD', Vol. 2712 of *Lecture Notes in Computer Science*, Springer, pp. 245–252.

Hudson, R. L., Morrison, R., Moss, J. E. B. & Munro, D. S. (1997), Garbage collecting the world: One car at a time, *in* 'OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications — Twelth Annual Conference', Vol. 32(10) of *ACM SIGPLAN Notices*, ACM Press, Atlanta, GA, pp. 162–175.

Hudson, R. L., Morrison, R., Moss, J. E. B. & Munro, D. S. (1998), Where have all the pointers gone, *in* 'Proceedings of 21st Australasian Computer Science Conference', Perth, pp. 107–119.

Hudson, R. L. & Moss, J. E. B. (1992), Incremental collection of mature objects, *in* Bekkers & Cohen (1992), pp. 388–403.

Kirby, G., Connor, R., Cutts, Q., Dearle, A., Farkas, A. & Morrison, R. (1992), Persistent hyper-programs, *in* 'Persistent Object Systems', Springer-Verlag, pp. 86–106.

Lowry, M. C. & Munro, D. S. (2002), Safe and Complete Distributed Garbage Collection with The Train Algorithm, *in* 'Proceedings of ICPADS'02', Taipei, Taiwan, pp. 651–658.

Maheshwari, U. (1993), Distributed garbage collection in a client–server, transactional, persistent object system, Technical Report MIT/LCS/TR–574, MIT Press.

Maheshwari, U. & Liskov, B. (1997*a*), Collecting cyclic distributed garbage by back tracing, *in* 'Proceedings of PODC'97 Principles of Distributed Computing', pp. 239–248.

Maheshwari, U. & Liskov, B. (1997*b*), 'Collecting cyclic distributed garbage by controlled migration', *Distributed Computing* **10**(2), 79–86.

Maheshwari, U. & Liskov, B. (1997*c*), Partitioned garbage collection of a large object store, *in* 'Proceedings of SIGMOD'97', pp. 313–323.

Morrison, R., Balasubramaniam, D., Greenwood, M., Kirby, G. N. C., Mayes, K., Munro, D. S. & Warboys, B. C. (1999), *ProcessBase Standard Library Reference Manual (Version 1.0.1)*, Universities of St Andrews and Manchester.

Morrison, R., Balasubramaniam, D., Greenwood, R., Kirby, G., Mayes, K., Munro, D. & Warboys, B. (2000), 'A compliant persistent architecture', *Software, Practice & Experience* **30**(4), 363–386.

Moss, J. E. B., Munro, D. S. & Hudson, R. L. (1996), PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores, *in* 'Proc. of the 6th Int. Workshop on Persistent Object Systems', Cape May NJ (USA), pp. 140–150.

Munro, D., Falkner, K., Lowry, M. & Vaughan, F. (2001), Mosaic: A non-instrusive complete garbage collector for DSM systems, *in* 'Proc. 1st CCGRID, WSDSM'01, Brisbane, Australia', pp. 539–546.

Norcross, S., Morrison, R., Munro, D. & Detmold, H. (2003), Implementing a family of distributed garbage collectors, *in* 'Proceedings of the 2003 Australasian Computer Science Conference', Adelaide, Australia, pp. 161–170.

Seligmann, J. & Grarup, S. (1995), Incremental mature garbage collection using the train algorithm, *in* O. Nierstras, ed., 'Proceedings of 1995 European Conference on Object-Oriented Programming', LNCS, Springer-Verlag, University of Aarhus, pp. 235–252.

Shapiro, M. & Ferreira, P. (1995), Larchant–RDOSS: a distributed shared persistent memory and its garbage collector, *in* J.-M. Helary & M. Raymond, eds, 'Workshop on Distributed Algorithms', number 972 *in* 'LNCS', Le Mont Saint-Michel, pp. 198–214.

Tel, G. & Mattern, F. (1991), The derivation of distributed termination detection algorithms from garbage collection schemes — (extended abstract), *in* Aarts et al., eds, 'PARLE'91 Parallel Architectures and Languages Europe', Vol. 505 of *LNCS*, Springer-Verlag.

Wilson, P. R. (1992), Uniprocessor garbage collection techniques, *in* Bekkers & Cohen (1992).