Implementing Garbage Collection in the PerDiS system

Xavier Blondel
INRIA-Rocquencourt and CNAM-Paris, France
Xavier.Blondel@inria.fr

Paulo Ferreira
INESC, Lisboa, Portugal

Marc Shapiro
INRIA-Rocquencourt, France

30 March 1998

Abstract

We describe the PerDiS persistent distributed store and its garbage collection algorithm. The PerDiS store behaves like an object-oriented persistent shared memory, and is accessed transactionally. Applications get direct access to persistent data in their own memory, and use the usual C/C++ pointer dereferencing and assignment to navigate and modify the object graph. PerDiS provides persistence by reachability based on the Larchant algorithm. We focus on the issues of implementation for large-scale sharing. The main problems we face involve concurrency, ordering and consistency. We also address performance/modularity trade-offs.

1 Introduction

This article describes the integration of the Larchant distributed garbage collection algorithm [8] into PerDiS, a persistent distributed store. PerDiS [16, 22] is a Long Term Research ESPRIT project, aiming at the design and implementation of a large-scale **PER**sistent **DI**stributed **S**tore, for cooperative engineering applications.

Persistence is based on reachability [2]. This relieves the programmer of the burden of file and memory management; it is inherently safe and efficient, contrary to manual approaches. Each application gets access to shared objects directly in its own memory (provided it has sufficient access rights), giving a simple, secure and efficient way of sharing data.

Scalability is a major factor of the design. Applications share a huge amount of data on a large-scale network, where possibly thousands of computers exchange data via a network such as the Internet. An important assumption to make this work is that concurrent write access to the same data is infrequent; when it does occur, it is within a small, well-connected area. This assumption is justified by the social structure of large design projects [19].

This paper addresses the implementation of the garbage collector in the PerDiS platform. It is organized as follows. After this introduction, Section 2 describes the PerDiS platform and explains the needs for a distributed garbage collector. Section 3 describes the Larchant garbage collection algorithm. Section 4 examines the implementation of the collector's constructive component; Section 5 does the same for the destructive component. Section 6 examines related work. Section 7 concludes and describes future work.

2 The PerDiS design

In this section, we describe the design of the PerDiS platform. The result of the first year of the project is an implementation called the Preliminary Platform implementation or PPF (available for download at http://www.perdis.esprit.ec.org/download/). In order to ease the first implementation, we made some simplifications[1], sometimes at odds with scalability. Two more full releases are planned at one-year intervals.

2.1 Basic components

The PerDiS platform supports shared objects connected by references. Entry points into this object graph are known as *roots*. An application program can access a root by naming its URL, and is then able to navigate the graph by following references from object to object.

Objects are grouped into *clusters*. A cluster is an arbitrary set of logically-related objects. Applications explicitely allocate objects in clusters. Whereas objects constitute the basic access unit for applications, clusters are the system management unit. A cluster has a name and protection attributes; it is the user-visible unit of memory management, storage, caching, sharing and protection. A cluster is managed like a distributed shared memory, i.e., it is cached by each client application that opened it. A cluster maybe of inifinite size, but only the part of it accessed by applications are cached at a given site[2].

For an application program, a reference is simply a pointer in memory. The PerDiS memory manager, based on the Larchant design [7, 8, 21], distinguishes intra-cluster references (connecting objects within a single cluster) from inter-cluster references (crossing cluster boundaries). An inter-cluster reference is represented by a *stub* of the source cluster, and by a *scion* in the target cluster. A stub contains the location of the corresponding scion; this information is used to swizzle persistent addresses.[3] A scion serves as a root for partitioned garbage collection. It identifies the cluster where the reference originates. There is a one-to-one correspondence between stubs and scions.[4] Stubs and scions are internal to the memory management subsystem and are invisible to application programs; in particular, there is no run-time overhead in dereferencing an inter-cluster reference: the application only sees raw pointers.

2.2 Architecture

The PerDiS architecture, shown in Figure 1, is based on two main components: the *User Level Library* (ULL) and the *PerDiS Dæmon* (PD). The ULL is linked to the application code, providing the PerDiS execution environment. The PD provides the applications with the PerDiS functionalities. There may be several applications but a single PD per site. Data is cached at both the ULL and the PD levels.

The ULL has an API module that interfaces to a Cluster Module and a Transaction module; they in turn talk to other modules in charge of caching, memory management, swizzling, etc. Relevant to garbage collection is the Stub/Scion Creation module described later. The PD is composed of a Transaction Module, a

[1] Simplifications are described in footnotes when needed.

[2] As a simplifying assumption, the garbage collector may cause an entire cluster to be cached. This is dicussed in Section 7.

[3] Swizzling refers to the translation of a persistent address in the store, into a virtual address in the application addressing space. The virtual address is computed only once and placed in the corresponding pointer, so the application can dereference the pointer with no overhead.

[4] However, as we will see later, the creation and destruction of a scion may be delayed with respect to the creation and destruction of stubs.

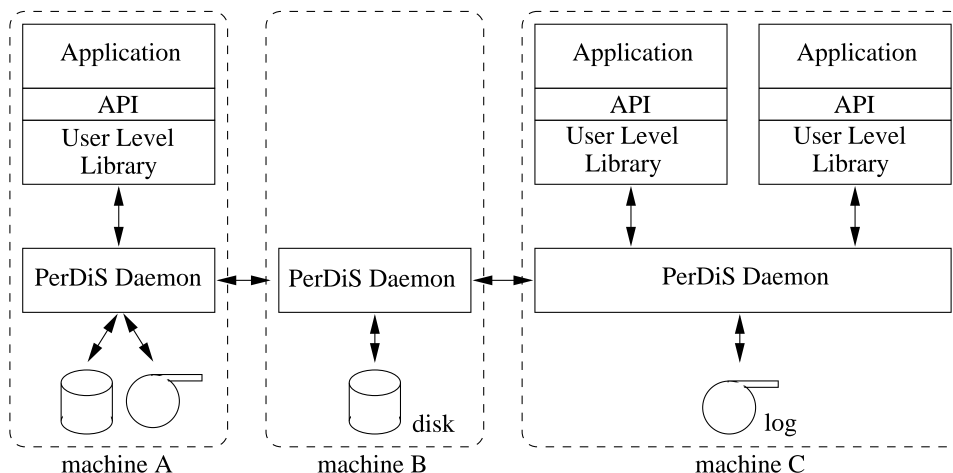

Figure 1: *PerDiS architecture: application processes link the User Level Library (ULL), which connects to the PerDiS Dæmon (PD) on the same machine. A PD connects to storage and to other PDs.*

Cache Module, a Cluster Module, a Storage Module, and a Garbage Reclamation Module. The Cluster Module (of which there is a copy in both the ULL and the PD) is solely aware of the storage format of clusters. A Cluster Module has interfaces to allocate an object in the cluster, to discover the type of an object, etc. Cluster Modules communicate with one another via Cache Modules; a Cache Module carries and caches uninterpreted pages, and manages page locks, applying a policy described in Section 2.3.

In the current implementation, swizzling [14], i.e. translating persistent addresses in the store into virtual addresses in the application addressing space, is performed by the ULL when an application takes a lock on an address range, similarly to swizzling at page fault time in the Texas system [26]. For future releases, we are considering moving the swizzler into the PD to improve performance. As a matter of fact, swizzling into the PD will reduce the number of swizzling operation, since the ULLs will be asked to map the same pages always at the same virtual addresses, if possible.

Distributed garbage collection is split into a "Stub/Scion Creation" component, that protects potentially reachable objects, and a "Garbage Reclamation" component, that detects and deletes garbage objects in the persistent store. Stub/Scion Creation discovers new pointers by analyzing modified data, as it is propagated between caches; in the current implementation it executes in the ULL, intercepting data sent to the PD when a transaction commits. In contrast and by design, Garbage Reclamation runs in the PD, because that is where the persistent store is managed.[5]

This article is about the implementation of the garbage collector for the persistent store. We therefore ignore what happens in the ULL, apart from Stub/Scion Creation, and focus on the operation of the PD. Since there is a single PD per site, we use the words PD and site interchangeably.

[5]We are interested in garbage collection of the persistent store only. This is distinct from collecting the transient memory of application processes. The former is part of the system, whereas supporting the latter or not is a language or application policy.

2.3 Data sharing

A PD caches a cluster[6], in order to provide local applications access to its data. PDs maintain consistency between replicas of a cluster, by using techniques taken from distributed shared memories. More specifically, PDs maintain "entry consistency" [4] among replicas of a cluster. An application can write in a cluster only if the PD holds the *write token* for that cluster, and can read only if the PD holds one of the *read tokens*.[7] At any time and for a given cluster, there can be either a single write token, or any number of read tokens.

Two PDs, the *home site* and the *owner site* play a special role. Each cluster has a statically-assigned home site, in charge of safe-guarding the definitive version of the cluster on disk. The owner site is dynamic, and defined to be the last site to hold a write token for that cluster; therefore, it is unique. The owner is guaranteed to hold in its cache the most recently-commited version of the cluster; the owner communicates it to the cluster's home as part of commit processing. In the absence of failures, the cached copy of the cluster at the owner and the on-disk copy at the home are equal.

Finding the owner of a cluster follows Li and Hudak's Dynamic Distributed Manager algorithm [13]. Each site maintains for each cluster a probOwner field, which refers to another site that may be the owner site, or which will forward requests, using its own probOwner. A request eventually reaches the owner site, which shortcuts the chain by directly answering the requesting site.[8]

It is important to underline that the DSM techniques used here are meant to replicate data among the PDs. Data are then available to the application linked to the ULL, in order for it to run a transaction onto them. Currently, transactions rely on the consistency mechanisms to perform their own locks, hence decreasing the efficiency of both transaction and caching. We are currently working on the feasibility of separating transaction locks from the locks of the replication mechanism.

3 The Larchant Garbage Collection algorithm

When the object graph is complex, manual memory management is very difficult. It is well-nigh impossible if the application is parallel, distributed and data is persistent.

By freeing only unreachable objects, a garbage collector (GC) guarantees that a program can safely use any pointer it can access. This property, called referential integrity, is essential for program correctness; it also improves performance by removing any need for run-time checks. Furthermore, GC avoids memory leaks by automatically deallocating unused memory.

Our system uses the Larchant GC algorithm [8, 9], designed for a shared, replicated memory. It is distributed and minimizes induced I/O, communication and synchronization costs.

Larchant uses partitioned GC [5] (also called hybrid GC [17]), with reference listing for inter-partition references, and tracing for intra-partition references. A partition includes a varying number of clusters.

Larchant imposes minimal consistency and synchronization requirements, as shown by Ferreira and Shapiro [7]. A given cluster can be replicated at any number of sites. Larchant remains correct even if the replicas are not mutually coherent.

[6] As we said above, this is a simplifying assumption for the preliminary platform.

[7] The user running the application must also hold appropriate access rights, but this is another story.

[8] A site does not need to record the probOwner of every cluster. To access a cluster without the probOwner information, it suffices to ask the home site, which is fixed and well-known. The home site does maintain the probOwner information and is able to forward requests correctly.

This is because Larchant applies the so-called Union Rule, whereby an object is garbage only if it is unreachable in the union of all replicas [9].

3.1 Reference listing and tracing

Larchant runs a tracing GC algorithm on every site. The trace can use either mark-and-sweep or copy-collection [12, 25]. It collects each cluster independently from others; furthermore, each replica of a cluster is traced independently of other replicas. The root of a trace is the set of scions along with PerDiS root objects. Since we are collecting the persistent store (not an application's private replica), we do not have to worry about mutator roots, such as stacks or local variables.

Inter-cluster references are managed using reference listing.[9] The scions of a cluster represent incoming references. After an application process creates a new inter-cluster reference (by pointer assignment), the corresponding stub-scion pair must eventually be created. In our design, a component called *Stub/Scion Creation* traces modified pointers to discover new inter-cluster references and to create stubs and scions. Every modified pointer must be scanned by Stub/Scion Creation before being stored on disk or being propagated to some other site.

Conversely, removing an outgoing reference eventually deletes the corresponding stub and scion. Thereafter, this deleted scion is no longer a root for the intra-cluster collection, enabling the algorithm to spot more garbage objects. The component that deletes unreachable stubs and scions and actually collects garbage object is called *Garbage Reclamation.* By design, Garbage Reclamation runs inside the PD.

A race condition between a scion creation message and a scion deletion message could lead to the following scenario: first, the scion is deleted; then, the object referred to by the scion is also deleted; finally, a new scion is created, which refers to the deleted object. To avoid this, the Larchant algorithm imposes ordering constraints on stub/scion creation and deletion, as described by Ferreira and Shapiro [9]. At any site, and for each GC cycle, all scion creations must occur before any scion deletion. Furthermore, GC-related messages related to a given cluster must be delivered in causal order.

3.2 Inter-cluster cycles of garbage

Reference listing, just like reference counting, does not collect garbage cycles. More precisely, in our case we miss inter-cluster garbage cycles. To solve this problem, the Larchant algorithm group-collects several clusters at one time. Such clusters are grouped with an heuristic that tries to minimize extra I/O: we trace the group of cluster replicas that is currently in the local cache.

This group heuristic does not guarantee to collect all persistent cycles of garbage. Indeed, if a cycle is distributed among two clusters that never appear together in a same cache, we miss it. We have started a series of measurements [18] of real applications to evaluate the frequency and impact of such occurence. If the missed cycles turn out to be a real problem, we will study more agressive heuristics, for instance forcing some other clusters into the cache to improve the probability of holding complete cycles on a single node, this to ensure completeness of the reclamation.

[9] Reference listing [17] is an extension of reference counting, where the count of an object is replaced by the list of the locations of originating references. Using our vocabulary, each scion represents an element in this list.

3.3 Improving the Larchant algorithm

The Larchant algorithm ensures safety and completeness of the garbage collection. Nevertheless, it does not address specific implementations issues, such as concurrent accesses to clusters by a collector and applications, and efficient propagation of stubs and scions. The algorithm describes in this article addresses more specifically this issue.

4 Stub/Scion Creation

In this section, we describe how stub/scion creation is implemented in the PerDiS platform. Stubs and scions must be created in a way that makes them available in any replica of the cluster. Hence, we decided to create stubs and scions at the owner.

Stub creation is trivial. Creating a stub in a cluster occurs if an outgoing reference has been added to this cluster, i.e. this cluster has been write-accessed. Since the coherence algorithm is based on single-writer/multiple-readers concurrency control, this occurs only at the owner replica. So, before a transaction releases its locks, the stub/scion creation component of the ULL walks the graph of each write-locked cluster from its scions, thereby discovering new outgoing references and creating the corresponding stubs. At the same time, references are "unswizzled" from their in-memory representation (a pointer) to their persistent one (a cluster identifier and an object identifier within the cluster). Since this owner replica was write-accessed, there are no other replicas.

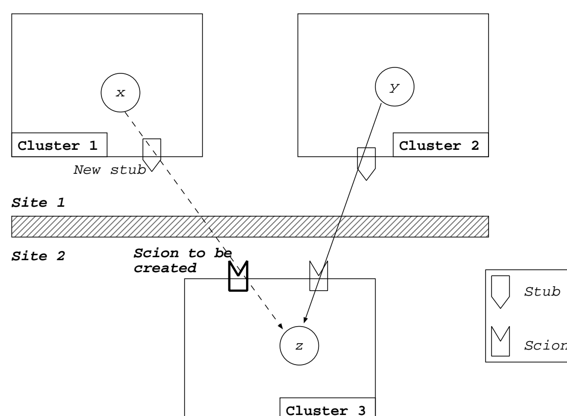

Figure 2: *An example of the need of a remote scion creation*

Unfortunately, the local site cannot directly create the corresponding scions. Indeed, the cluster to contain the nascent scion is not necessarily owned locally; it might not even be cached locally.

The example in Figure 2 illustrates this situation. Site 1 accesses two clusters, Cluster 1 and Cluster 2. The latter contains one object y, with a reference to the object z in Cluster 3, owned by Site 2. Let us assume that Site 1 performs operation $x := y$, where x is in Cluster 1. Cluster 1 now has an outgoing reference to z, drawn as a dotted line in the figure. Stub/scion creation at Site 1 will create the new stub locally, but the scion (drawn in bold) must be created on Site 2. Therefore Site 1 sends a create-scion message to the PD at Site 2.

To avoid distributed races, the Larchant algorithm imposes ordering constraints on scion creation and deletion, as mentioned in Section 3. So, before the transaction

that created the new reference releases its locks, the Stub/Scion Creation module transmits create-scion requests to the owner sites of the target clusters. It waits for an acknowledgement in order to ensure causal ordering[10].

When an owner site receives a scion creation request, it must not cause conflicts with any concurrent applications. The request is queued and acknowledged. Later, when no application holds a write lock on this cluster, but before propagating this cluster to another site or to disk, and before the next garbage reclamation round, the request is dequeued and the scion created.

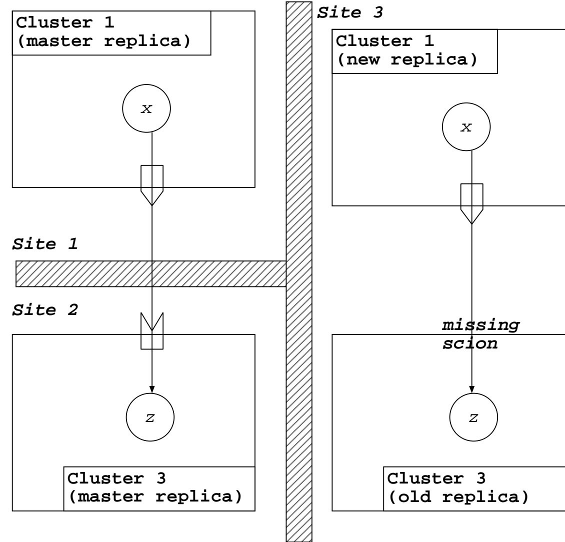

Figure 3: *An example of a missing scion*

Remote stub creation messages create the possibility of a race with coherence messages. Consider the example in Figure 3. Site 3 uses the clusters of the example in Figure 2. Since Cluster 1 was write-accessed on Site 1, by entry consistency its replica on Site 3 must be up-to-date (any old replica would have been invalidated). On the other hand, suppose that previously some application on Site 3 read Cluster 3, which now remains, unmodified, in the cache. The data in Cluster 3 is up-to-date, but not the metadata; in particular, scion $x \to z$ is not yet present. Now some application at Site 3 reads Cluster 1 on Site 3. As it follows the reference from x, the Cluster Manager at Site 3 accesses the newly created stub, but does not find the scion in Cluster 3.

This race condition is easily resolved. When the Cluster Manager detects the missing scion, it retrieves up-to-date meta-data from the owner of Cluster 3.[11] Meta-data is propagated lazily but safely.

5 Garbage Reclamation

Now we describe how Garbage Reclamation is implemented in the PerDiS platform. The two main issues were limiting inter-module dependencies, and consistency problems.

[10] This causes the transaction commit, hence the application, to be blocked until all acknowledgemnts have been received, an unacceptably expensive operation. We are currently working on sending create-scion messages when the transaction is applied rather than at commit time.

[11] Each stub-scion pair has a unique identification, ensuring that a missing scion can be detected.

5.1 Garbage collection interfaces

We want the garbage collector to be independent of other modules as possible, i.e. we do not want that modifying a module implies a modification to the garbage collector. Garbage collection is a complex operation with many dependencies, so the internals of module operations are hidden from the collector, thanks to the interfaces described here, which do not depend on a particular implementation of a module.

For instance, garbage reclamation depends on the Cache Module for the list of clusters currently cached at this site, for collecting inter-cluster cyclic garbage, as explained in Section 3.2. It depends on the Cluster Module for access to meta-data such as stubs, scions, type information, free list, etc. Therefore, it would seem natural to add reclamation to its list of functions; however the code would be very intricate and difficult to modify. Our decision was to favour independence over performance and to define clear interfaces between garbage collection, cluster management, and caching. The added cost is probably negligeable compared to expensive operations such as swizzling or transmitting data over the network.

In the rest of this section, we provide the full list of interfaces between the garbage collection modules (stub/scion creation and garbage reclamation) and the other modules. Note that the current collector implementation assumes that clusters are found in swizzled form. Furthermore, garbage reclamation assumes that all pointers are valid, having been checked at the time of stub/scion creation. If either of these assumptions were untrue, a further interface, for de-referencing pointers, would be needed.

5.1.1 General interfaces

The following interfaces allow the collector to create or delete a stub or a scion in a cluster:

```
bool cluster::new_stub (...values...);
bool cluster::new_scion (...values...);
bool cluster::delete_scion (scion* sc);
bool cluster::delete_stub (stub* st);
```

The parameter list ...values... represents the initialization values of the data type.

In the following list, each pair constitutes an iterator, over objects in a cluster, and over pointers in an object, respectively:

```
scion* cluster::first_scion ();
scion* cluster::next_scion ();
void* cluster::first_allocated_object ();
void* cluster::next_allocated_object ();
void* cluster::first_reference (void* obj);
void* cluster::next_reference (void* obj);
```

The following interface:

```
void* cluster::start_address_of (void* ptr);
```

converts an arbitrary pointer, possibly pointing inside an object instead of to its beginning,into a pointer to the beginning of the object.

The following group of interfaces allow GC to test the status of a cluster.

```
bool cache::is_cluster_locally_owned (cluster_id &cid);
bool cache::is_cluster_locked (cluster_id &cid);
```

The first one tells whether the master site of the cluster is the local site or not. The second one checks whether a cluster is locked by a transaction or not.

5.1.2 Stub/Scion Creation interface

Stub/scion creation is triggered in the ULL, at commit time, and continues in the PD. Before a transaction sends modified data towards the associated log in the PD, it calls the following procedure:

```
bool ss_module::process_dirty_cluster (cluster* c, transaction* t);
```

which walks the graph reachable from this cluster's scions. If any new outgoing pointers have appeared, then it creates the corresponding stub with cluster::new_stub, then calls:

```
bool perdis_gc::process_new_stub (const stub& st, cluster_id& cid);
```

This procedure generates the create-scion message to the owner of the pointed-to cluster.

When a PD receives create-scion, it takes a latch —a notification lock, as described in Section 5.1.3— on the cluster where the scion is to be created. If the cluster is locally owned (checked with cache::is_cluster_locally_owned), then it creates the scion. Otherwise, it forwards the message to the owner, using the cache::cluster_owner interface.

5.1.3 Garbage Reclamation interface

Garbage Reclamation does a group trace of all clusters currently cached at this site for which there is no write conflict. It uses the following interfaces to manage this group:

```
int cache::latch_cached_clusters ();
bool cache::list_cached_clusters (cluster_id *cllist);
bool cache::unlatch_cluster (cluster_id &cid);
```

The first primitive puts a notification latch on all clusters in the cache, and returns the number of such clusters. The second returns the list of notification-latched clusters. The third releases the notification latch on a cluster.

The following procedure:

```
bool perdis_gc::conflict_notification (cluster_id &cid);
```

is a call-back from the cache, to notify garbage reclamation that an application is trying to write-lock the cluster, or that this cluster is invalidated. The cache must wait until conflict_notification this function returns before continuing. In the current implementation, gc_notify aborts reclamation. In the future, this static call-back function will be replaced by registering a function dynamically, which could backtrack changes done to this cluster only.

Finally:

```
bool cluster::reclaim_object (void* obj);
```

is used by garbage reclamation to reclaim the space of an unreachable object.

The cache implements the invalidation latch and the write latch needed by the reclaimer.

5.2 Consistency issues

As explained in Section 3, the Larchant GC algorithm is correct independently of coherence. In other words, the collector at some site is satisfied with whatever replicas it finds locally, whether they are up-to-date with remote replicas or not.

However, we are not totally free of consistency worries. The collector accesses clusters in the Cache Module in order to garbage collect them. The Cache Module has an interface based on pages; it does not guarantee internal consistency unless these pages are locked.

If an object straddles a page boundary, behaviour would be undefined if the first page is an old version and the second page a new version. Similarly, consider a pointer contained in some page, pointing to an object on a separate page; if the

target object does not appear in the second page, dereferencing the pointer could cause an error. Thus, although a cluster needs not be coherent, data and meta-data must be internally consistent.

Meta-data such as free lists, type descriptors and stubs must also be consistent with the cluster's data. Scions are managed differently, since a scion incoming into a cluster may be created or deleted without any write access to the cluster. As was explained in Section 4, scion creation and deletion do not need to be acted upon immediately at all replicas.

In the rest of this section, we examine these problems in more detail. Section 5.2.1 deals with the application modifying data concurrently with the collector. Section 5.2.2 deals with the collector modifying data concurrently with applications. Section 5.2.3 examines how replication impacts reclamation.

5.2.1 Concurrent updates by applications

One problem is concurrent writing of a cluster by an application, while the garbage reclamation is reading the same data and meta-data. One possible solution would be for garbage reclamation to take an ordinary transactional lock on clusters it reads. However, since garbage reclamation is designed to eagerly read all the data present in the cache, this approach could potentially block many applications. Another solution, which we proposed in an earlier design [21], is to run garbage reclamation inside an optimistic transaction; it would not block concurrent applications, and would abort in case of a concurrent write by an application. The current platform implements this solution in a simplified form, using the notification latches described hereafter.

For future versions of the platform, we are planning a slightly more subtle solution. Garbage reclamation only takes local, non-transactional locks (latches). This works as follows: when garbage reclamation starts, it asks the Cache for all locally replicated clusters that are not write-locked by an application. They are protected by a local *notification latch*. A notification latch does not prevent applications from accessing the cluster; however if a local application write-locks such a cluster, the cache sends a notification, forbidding garbage reclamation to consider this cluster. If garbage reclamation had deallocated any objects in this cluster, the deallocation is undone (only for this cluster, without aborting the whole collector). There is no danger of unsafe reclamation thanks to the Larchant safety rules [9], which ensure that scion creation always precedes reclamation.

When garbage reclamation is over, it takes an exclusive write-latch on any cluster it modified. It updates the cache with the cleaned clusters. The write latch suspends temporarily applications trying to read- or write-lock a cluster. Then the persistent store itself is updated, in order to make garbage reclamation persistent. Note that the latch is local and does not cause remote caches to be invalidated, even though clusters are written by garbage reclamation. The latch only serializes local writes performed by garbage reclamation, avoiding that an application reads a half-written cluster.

5.2.2 Improved concurrent deallocation

In the case of an in-place garbage collector, we propose an even more efficient solution, that allows the collector to reclaim garbage even while an application has a write lock.

The problem we address is concurrency between an application allocating objects, while garbage reclamation is concurrently deallocating. Write conflicts are limited to the free list. Although garbage reclamation reads all sorts of data and

meta-data, an in-place collector's only writes are to add a garbage object to its cluster's free list.

To solve this problem without slowing down applications, we use a semantical concurrency control, based on the knowledge of the semantics of accesses performed by garbage reclamation. Therefore we designed the free list for concurrent updates without relying on transactions.

To allocate an object, an application takes a write lock. To deallocate, garbage reclamation simply appends a deallocation record on a local, secondary free list. The next process to take the write lock merges together the main free list and any secondary free lists.

This optimization is not yet implemented in the current platform.

5.2.3 Local deallocation

We still have the problem of propagating the work done by garbage reclamation to all existing replicas. An important decision is to update only clusters for which this cache holds the owner replica. This way, we are sure the modifications will be propagated, since a site needing a replica of a cluster asks its current owner.

However, if the local site detects garbage in a cluster that is not locally owned, its work is wasted. In the case of cyclic garbage, this is not much of a problem. Even if we do not reclaim an entire garbage cycle, we break it, by suppressing the locally owned garbage objects of the cycle.

6 Related work

We describe here various work dealing with some distributed persistent stores and their garbage collection.

6.1 Other distributed persistent stores

Certainly, the most widespread distributed persistent stores are the various distributed filesystems, for instance NFS [20] or AFS [11]. They enable several users working on different machines to access and share remote filesystems. They provide, to a certain extent, fault-tolerance and coherence. Like our system, they are based on the assumption of low write contention. However, they lack transactional semantics. Their interface is file-oriented, which is not adapted to the storage of complex data structures, such as the ones used in the cooperative engineering applications supported by PerDiS. Obviously, they do not provide garbage collection.

Object-oriented client-server databases, such as O_2 [6], provide distributed access to shared data. They provide superior interfaces enabling complex data structures. However, they do not scale well, because data is centralized at a single server. In a large-scale environment, such as a co-operative design project distributed over the Internet, the central server is a bottleneck. Furthermore, maintaining several database servers with security in mind is quite difficult. In contrast, PerDiS has been designed to be scalable, and security was also a guideline of the design from the very beginning.

Several powerful persistent stores exist, but most of them, such as Texas [23], are not distributed. Furthermore, security support was not a major concern during the design of these stores.

The PerDiS design borrows many ideas from these various stores. Our goal is to integrate the advantages of each system in order to have a persistent distributed store fitting the needs of our target domain, cooperative CAD applications.

6.2 Garbage collecting a persistent store

A number of distributed garbage collection algorithms have been proposed [17]. Until recently, none took into account issues of caching, replication and coherence. Recently, Yu and Cox proposed a conservative collector for the TreadMarks (non-persistent) DSM system, which has some similarities to Larchant. Garbage collection algorithms for client-server databases [1, 5, 10, 24] are usually non-distributed, being executed at the database server.

The PerDiS garbage collector is based on the Larchant design [8, 9]. Many of the other Larchant concepts, such as dividing the memory into clusters, replicating clusters, and the stub and scion data structures, were carried into the PerDiS design. The Larchant system was a small-scale research prototype. An important outcome of PerDiS will be to provide a strenghthened, large-scale implementation useable for actual industrial applications.

The consistency issues described in Section 5.2 are closely related to concurrent garbage collection. Several studies have been done in this field; however they do not consider the impact of replication and memory coherence on garbage collection. Some recent results include Baker's Treadmill [3], which is fairly efficient, but has the drawback of causing unpredictable delays because, in some cases, a collection must be finished before accessing the data. O'Toole at al. propose transaction-based collector for a persistent heap [15]. Our garbage reclamation algorithm is closely inspired by theirs. To improve performance, we replaced transaction locks by notifications, and implemented reclamation using secondary free lists.

7 Conclusion

We presented the way we fitted an existing distributed garbage collector algorithm into a persistent distributed store. We noticed that such an operation raises a lot of ordering and consistency problems. We also noticed that lowering module dependencies is a trade-off between modularity and performance.

A version of the platform, which does not yet provide swizzling but already include garbage collection, is available for download [16].

Future work includes elaborating the design of the secondary free lists, to enable concurrent writes. We will also have to deal with future evolutions of the PerDiS platform, mainly fine-grain locking, fault-tolerance and security. We managed to have a flexible enough design for the garbage collector, in order to be able to integrate these evolutions into future PerDiS garbage collector, with only minor changes to the general design.

The major drawback of our current GC design is that it needs to have a whole cluster to perform garbage reclamation. Consequently, any first access to a cluster generates a huge amount of communication, in order to get a replica of the whole cluster. This is at odds with our concern for scalability. To fix this problem, we plan to divide a cluster in several *bunches*, which can be seen as sub-clusters, each with its own free list and stub and scion lists. Bunch structure and size will be discussed on the basis of real applications measurements, in order to fit the performance needs of these applications.

References

[1] Laurent Amsaleg, Olivier Gruber, and Michael Franklin. Efficient incremental garbage collection for workstation-server database systems. In *Proc. 21st Very Large Data Bases (VLDB) Int. Conf.*, Zürich (Switzerland), September 1995.

[2] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[3] Henry G. Baker. The treadmill: Real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3):66–69, March 1992.

[4] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, Pittsburgh, PA (USA), September 1991.

[5] Jonathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn. Partition selection policies in object database garbage collection. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 371–382, Minneapolis MN (USA), May 1994. ACM SIGMOD.

[6] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[7] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241, Monterey CA (USA), November 1994. ACM. http://www-sor.inria.fr/publi/GC-DSM-CONSIS_OSDI94.html.

[8] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proc. 16th Int. Conf. on Dist. Comp. Syst. (ICDCS)*, Hong Kong, May 1996. http://www-sor.inria.fr/publi/LPRDSMGC_icdcs96.html.

[9] Paulo Ferreira and Marc Shapiro. Modelling a distributed cached store for garbage collection. In *European Conference on Object-Oriented Programming (ECOOP)*, Brussels (Belgium), July 1998. http://www-sor.inria.fr/publi/MDCSGC_ecoop98.html.

[10] Olivier Gruber and Laurent Amsaleg. Object grouping in EOS. In *Proc. Int. Workshop on Distributed Object Management*, pages 184–201, Edmonton (Canada), August 1992. ftp://rodin.inria.fr/pub/publications/conferences/IWDOM.92.ps.gz.

[11] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[12] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester (GB), 1996. ISBN 0-471-94148-4.

[13] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[14] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[15] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 161–174, Asheville, NC (USA), December 1993.

[16] PerDiS Project. The PerDiS project: a persistent distributed store. http://www.perdis.esprit.ec.org/.

[17] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), September 1995. http://www-sor.inria.fr/publi/SDGC_iwmm95.html.

[18] Nicolas Richer. Mesure et analyse des caracteristiques d'une memoire d'objet. Rapport de Recherche 3315, inria, Rocquencourt (France), December 1997. http://www-sor.inria.fr/publi/MACMO_rr3315_97.html.

[19] F. Sandakly, R. Greening, P. Poyet, and F. Kersting. Application requirements analysis. PerDiS Deliverable PDS-R-97-001, PerDiS Project, May 1997. http://www.perdis.esprit.ec.org/deliverables/docs/T.A.1/WPA1.html.

[20] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), June 1985. USENIX.

[21] Marc Shapiro and Paulo Ferreira. Larchant-RDOSS: a distributed shared persistent memory and its garbage collector. In J.-M. Hélary and M. Raynal, editors, *Workshop on Distributed Algorithms (WDAG)*, number 972 in Springer-Verlag LNCS, pages 198–214, Le Mont Saint-Michel (France), September 1995. http://www-sor.inria.fr/publi/LRDSPMGC_wdag95.html.

[22] Marc Shapiro, Sytse Kloosterman, and Fabio Riccardi. PerDiS — a persistent distributed store for cooperative applications. In *Proc. 3rd Cabernet Plenary W.*, Rennes (France), April 1997. http://www-sor.inria.fr/publi/PPDSCA_cabernet97.html.

[23] K. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proc. of the Fifth International Workshop on Persistent Object Systems Design, Implementation and Use*, pages 13–28, San Miniato Pisa (Italy), September 1992.

[24] Marcin Skubiszewski and Nicolas Porteix. GC-consistent cuts of databases. Rapport de recherche 2681, Institut National de la Recherche en Informatique et Automatique, rocquencourt, April 1996. ftp://ftp.inria.fr/INRIA/publication/RR/RR-2681.ps.gz.

[25] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), September 1992. Springer-Verlag. ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps.

[26] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 364–377, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Comp. Society Press.