Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection

Paulo Ferreira, Marc Shapiro

Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection*†

Paulo Ferreira‡ and Marc Shapiro
INRIA - Projet SOR

Abstract

We consider a shared store based on distributed shared memory (DSM), supporting persistence by reachability (PBR), a very simple data sharing model for a distributed system. This DSM+PBR model is based on distributed garbage collection (GC). Within a general model for DSM+PBR, we specify a distributed GC algorithm that is efficient and scalable. Its main features are: (i) *independent collection of memory subsets (even when replicated),* (ii) *orthogonal from coherence,* (iii) *asynchrony, and* (iv) *a simple heuristic to collect cycles avoiding extra I/O costs. We briefly describe our implementation and show some performance results.*

1. Introduction

The overall goal of *Larchant* is to provide a *Persistent Distributed Store* (PDS) that makes it easy to share data objects. Larchant ensures that the access to data from different sites and/or at different times is simple, and efficient. In addition, persistence, I/O, memory management, and distribution are all transparent and automatic. With Larchant, programmers may concentrate on application issues without distraction from memory bugs, caching, coherence, remote access, I/O, etc. The applications we envisage to support are CAD-CAM systems, multimedia, financial databases, etc.

The model of Larchant is that of a *Single Address Space* (spanning every site in a network including secondary storage) with *Persistence By Reachability* (PBR) [2]. This model is very attractive given its simplicity from the programmer's point of view.

To provide the illusion of a shared address space across the network, although site memories are disjoint, Larchant implements a *distributed shared memory* (DSM) mechanism [9].

Accessing data entails finding its reference by navigation from a well-known *persistent root* (*e.g.*, a name server). In a system with PBR an object reachable from the persistent root is *persistent*, thus it should persist on secondary storage. An object transitively reachable from the persistent root is persistent also. Unreachable objects are not needed and can be reclaimed (and memory compacted). Such objects are said to be *garbage*.

Reachability is accessed by tracing the pointer graph, starting from the persistent root, and reclaiming unreachable objects. This can be done either via manual memory-management, or automatically via *Garbage Collection* (GC).

Manual memory-management is extremely error-prone (*e.g.*, dangling pointers and storage leaks) frequently resulting in the violation of the fundamental requirement of *Referential Integrity*: following a pointer should always work, *i.e.*, if persistent object x points to object y, then y must be persistent also.

GC was until recently thought to be intractable in a large-scale system, due to problems of scale, incoherence, asynchrony, and performance. This paper presents the solutions that Larchant proposes to these problems.

The GC algorithm in Larchant combines tracing and reference-counting. It traces whenever economically feasible, *i.e.*, as long as the memory subset being collected remains local to a site, and counts references that would cost I/O traffic to trace. The reference-counting boundary changes dynamically and seamlessly, and independently at each site, in order to collect cycles of unreachable objects. Most importantly, replicated memory subsets are collected

*This article appears in the Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS), May 27-30, 1996, Hong Kong.

†This work has been done within the framework of the ESPRIT Basic Research Action Broadcast 6360, and was partially supported by Digital Equipment Corporation.

‡Work done while Ph.D. student at *Université Pierre et Marie Curie (Paris VI)*. Supported by a JNICT Fellowship of Program *PRAXIS XXI* (Portugal). Email: Paulo.Ferreira@inria.fr. Tel: +33 (1) 39 63 52 08. Fax: +33 (1) 39 63 53 30. Address: INRIA - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex, FRANCE.

independently without interfering with applications coherence needs.

The novelty of this work resides on being the first to propose a GC algorithm for a PDS which is *(i)* scalable, *(ii)* orthogonal to coherence, *i.e.*, makes progress even if only incoherent replicas are locally available, *(iii)* completely asynchronous to applications, and *(iv)* reclaims cycles of garbage by dynamically changing the borders of the distributed reference-counting mechanism.

This paper is organized as follows. The upcoming section presents our model of application programs, coherence protocols, and collectors. Section 3 gives an overview of the GC design. Sections 4 and 5 describe the tracing and reference-counting algorithms, respectively. We compare our proposal with the literature in Section 6. In Section 7 we briefly present the implementation and show some performance results. Section 8 concludes with a summary of the most important ideas.

2. Model

We use the standard vocabulary of the garbage collection literature [14]. The *mutator* is the application program that dynamically modifies the pointer graph: it creates objects, dereferences pointers, and assigns pointers. As a side-effect of pointer assignment, some reachable objects become unreachable.

In a distributed system, the mutator is actually composed of multiple independent threads running at different sites; by extension, we call each of these threads a mutator.

The collector is the system component that identifies and reclaims garbage created by the mutator. Our collector is composed of a number of threads executing at different sites; we call each one a collector.

Our GC algorithm is based on a very simplified model. Thus, many operations traditionally associated with coherence management are not present in the model; some because they are not relevant to GC (for instance, non-pointer read or writes) and others because our algorithm is independent of them (for instance, DSM locks). In particular, the GC algorithm tolerates arbitrary writes and incoherent data. The simplicity of the model ensures that the GC algorithm is applicable to a large variety of shared stores.

2.1. Memory Structures

Memory is structured at two levels of granularity. *(i)* The *object* is the unit of allocation, deallocation, and identification. It is also the unit of coherence and update propagation (*i.e.*, dissemination of the most up-to-date replica of an object). A pointer is either null or points to an object. An object may contain any number of pointers. *(ii)* A *bunch* is the unit of caching and of collection. It contains any number of objects.

Hereafter, objects are noted x, y, z, etc. The address of object x is noted @x. Bunches are noted uppercase B, C, etc. Since any structure may be replicated (cached) at multiple sites, we distinguish between replicas with a per-site subscript, *e.g.*, x_1, x_2, for replicas of x observed respectively at sites 1 and 2.

A pointer variable ptr inside an object x is noted x.ptr. In order to simplify the notation we make the simplification that objects have only one pointer inside and identify x.ptr with x.

2.2. Mutators

A mutator running at site 1 observes object x through the replica currently cached at its site, x_1. Any mutator may write or read an object x for which it caches a pointer.

For GC purposes, the relevant operation executed by mutators is the *assignment* of a pointer variable inside an object. An assign operation executed at site 1, resulting from a read of object y and a write of x, is noted $<x := y>_1$. This operation is atomic locally only, *i.e.*, the read and write of the local objects replicas are indivisible. There is no need to make this operation atomic with respect to remote replicas.

Note that the assignment operation may result both in the destruction of a pointer and in the creation of a new one. This is done unpredictably by mutators and modifies objects reachability.

2.3. Coherence

This section identifies the coherence operations that are relevant for GC purposes. In order to accommodate different coherence models, we leave undetermined the times when such operations occur. (In a practical system, coherence operations are caused by mutator activity.)

Our minimal assumption is that, at any point in time, each object has a single *owner*. We define owner of an object as the only site that is allowed to disseminate its value to other sites. This dissemination is done by a *propagate* message.

The effect of a propagation of x from its owner site 1 to site 2 is that x_2 becomes equal to the propagated x_1. Note that this operation does not preclude concurrent writes (or reads) done on other replicas of x.

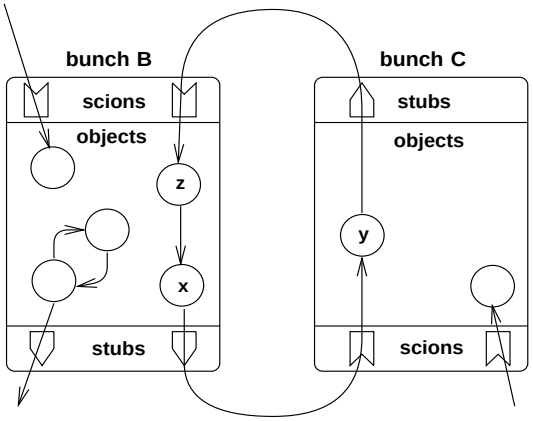

Figure 1. Stubs and scions.

The owner of an object may change. This is done by the *ownership* message: the transfer of x's ownership from site 1 to site 2 has the effect of making site 2 the new owner of x.

Other coherence operations can be modeled as special cases of the above ones. For example, cache invalidation is modeled as receiving a propagate message containing the value "nil".

2.4. Garbage Collection

A bunch keeps track of cross-bunch pointers, in order to support its independent collection. An outgoing pointer is described by a *stub* and an incoming pointer by a *scion*.[1] (See Figure 1.) A stub identifies its matching scion and vice-versa. Each bunch replica has its own set of stubs and scions.

We list here, without justification, the relevant operations of our GC algorithm (both mark-and-sweep and copying techniques [14]):

- create(Bx, Cy): create scion in C describing the cross-bunch pointer from x in B to y in C.
- delete(Bx, Cy): delete scion in C describing the cross-bunch pointer from x in B to y in C.
- scan(x_1) scan x_1 to find the objects pointed directly from it.
- move(y, @y'): move object y to new address @y'.
- patch(x_1, @y'): patch x_1 with y's new address @y'.

The triggering and the safety of these operations will be studied in Sections 3, 4 and 5.

Sometime after a mutator has created a new cross-bunch pointer, the collector allocates a stub at the source bunch and *creates* a scion at the target. Similarly, after a cross-bunch pointer disappears, the collector eventually deallocates the corresponding stub and *deletes* the target scion. The number of cross-bunch pointers to some object is approximated by the number of scions to that object.

The main loop of a tracing GC uses the *scan* operation. For some object replica x_1, scan determines what other objects are pointed to by x_1. A copy collector *moves* objects (to compact memory and reduce fragmentation) and *patches* pointers with the new addresses.

Note that both scan and patch operations are local to a site, *i.e.*, they are applied to a local replica and they do not necessitate any propagate operation in spite of the implicit read and write of the concerned object (thus, local replicas may be incoherent). On the other hand, the move operation applies to every replica of an object. As described in Section 4, all these GC operations do not require coherent data.

In GC terminology, a *flip* designates the moment during which the mutator is halted; when flipping, the collector (either mark-and-sweep or copy algorithms) performs some operations in order to finish the collection. For example, when flipping, a mark-and-sweep collector re-scans those objects that, after having been scanned, were modified due to concurrent mutator activity.

Finally, an object is *GC-dirty* after being modified by a (mutator) assign operation; it remains GC-dirty until scanned. A bunch is GC-dirty if it contains a GC-dirty object.

3. GC Overview

Ideally, a GC algorithm would be *complete*, *i.e.*, would eventually reclaim *all* unreachable data. The only known provably complete algorithms are based on tracing and employ a global synchronization. This does not scale, generates a large amount of I/O traffic, and disrupts applications. On the other hand, reference-counting algorithms are scalable but are incomplete as they do not reclaim cycles of garbage. Apparently, the problem is hopeless. But, as we will show, it is possible to avoid the unfeasible global trace, and to avoid its drawbacks.

We claim that perfect completeness is not feasible in a large scale system. Thus, we propose an approximate

[1] The names stub and scion are inspired by the similar structures found in the SSP (stub-scion pair) Chain message-passing system [13]. In contrast to SSP Chains, Larchant's stubs and scions are not indirections participating in the mutator computation, but simply auxiliary data structures describing cross-bunch pointers.

solution that is not provably complete, but which we believe adequate for all real-life situations. It works by combining tracing and reference-counting as described now.

GC in Larchant approximates a global trace with a series of non-synchronized, piecewise, local traces. Each bunch is collected at the site where it is cached, with a tracing algorithm, independently from the rest of the memory. In addition, if a bunch is replicated, each one of its replicas is also collected independently with the same algorithm.

By considering a bunch scions as interim roots, a bunch can be collected independently of others. The collection of a bunch replica (*intra-bunch collector*) proceeds as follows. Any object pointed at directly from a scion is considered reachable and is scanned for pointers. If a reachable object points to another object inside the same bunch, the intra-bunch collector transitively considers reachable the pointed-to object. If it points outside the enclosing bunch, the collector allocates a stub. Thus, the result of collecting a bunch is a set of reachable objects and a new set of stubs. Objects not in the set set are garbage.

When an intra-bunch collection is finished, the *cross-bunch collector* (reference-counting algorithm) compares the new stub set with the old one (resulting from the previous intra-bunch GC). Stubs that did not previously exist indicate that a new outgoing cross-bunch pointer has been created by the mutator. Stubs that have disappeared (*i.e.*, which are not in the new set of stubs) indicate that an outgoing cross-bunch pointer no longer exists.

For every new stub that has been allocated (by the intra-bunch collector), the cross-bunch collector issues a create. For every stub that no longer belongs to the new set, the cross-bunch collector issues a delete concerning the target scion.

Thus, the cross-bunch collector deletes scions therefore enabling future intra-bunch collections to reclaim those objects that were reachable only because they were pointed from the deleted scions. (As we will see in Section 5, create and delete operations are asynchronous to mutators.)

4. Tracing

The main difficulty of the intra-bunch collection algorithm is to collect a bunch (concurrently with mutators) without requiring any coherence operation, in order to avoid disrupting applications. For example, the intra-bunch collector must be able to progress without requiring an updated replica of the object(s) to be scanned. A similar reasoning applies to the operations move and patch.

Each site traces its bunches (locally cached) independently of all other sites, even though its bunches may be replicated elsewhere. This raises the following questions:

- Must collectors synchronize with each other?
- Does scanning need coherent data?
- When using a copy GC algorithm:
  - Is it necessary to synchronize the collectors to decide where to move an object?
  - Is it necessary to perform some coherence operation before (or after) moving an object or patching its internal pointers?
  - Is it necessary to synchronize the flip?

A "yes" answer to any of these questions would impact scalability and efficiency. In the rest of this section we will show that, surprisingly, the answers are all "no" under the right conditions. As a consequence, the intra-bunch collector does not compete with applications for coherent data, there is no synchronization between collectors and mutators or between different collectors, and the GC messages are asynchronous and exchanged in the background. The price to pay for these features is some degree of conservativeness, and some messages need to be delivered in causal order [4].

4.1. Scan

The scanning of an incoherent replica evidently does not take into account pointer writes occurring at any other site. However, this is not a problem. In fact, scanning an out-of-date replica simply results in making a more conservative decision about the pointed objects reachability.

On the other hand, scanning only the most up-to-date replica of an object is not safe. An object z may no longer be referenced from the most up-to-date replica of an object y but it may still be reachable from an incoherent replica of y (both y and z in the same bunch). Thus, an object can be reclaimed only after becoming unreachable from the union of all replicas of its source objects. We call this the *union rule*. We address this issue with more detail in Section 5, when describing the cross-bunch collector.

4.2. Move and Patch

In this section, we study the move and patch operations w.r.t. coherence. The first problem is to avoid two sites moving (their local replicas of) the same object to two different locations concurrently. One obvious solution to this problem would be: the site that wants to move an object would acquire the object's ownership before moving it. However, this solution is clearly undesirable, since it interferes with applications coherence needs.

A simple solution that does not interfere with applications coherence needs is as follows: the owner site of x decides where to move it; non-owner sites will move their replicas of x to the same address after receiving a move message from the owner. Therefore a reachable object is moved as follows. *(i)* Site 1, the owner of x, sends a move(x, @x') message to sites caching a pointer to x, including to itself. *(ii)* A site 2 receiving the above message moves its replica x_2 to the new location and patches pointers accordingly. Note that move messages are delivered in the background. Thus, they do not disrupt applications.

The second problem is the following: must the collector acquire the ownership of an object in order to patch one of its pointers with the target's new location? Surprisingly, the answer is no, because this operation is visible only at the site where it occurred. (Note that, even with coherence protocols that require the ownership of an object to write into it (*e.g.*, entry-consistency [3]) the patch operation can be done without holding the object's ownership.)

Finally, the collector of a bunch replica may flip independently (without synchronization) from the collectors of other replicas of the same bunch. The intra-bunch collector can flip even before having received all the move messages regarding reachable objects not locally owned. (The moving of such objects can be delayed until the next GC run.)

4.3. Reclamation of Cross-Bunch Cycles

The intra-bunch GC algorithm is complete w.r.t. the collected bunch, *i.e.*, it reclaims all garbage that is entirely within the bunch. However, it is incomplete w.r.t. other bunches, since it does not reclaim a cycle of garbage that crosses the bunch boundary (*e.g.*, cross-bunch cycle in Figure 1: objects x, y, and z).

The same algorithm that collects a single bunch can collect any group of bunches. The only difference w.r.t. to the collection of a single bunch, is that to trace a group: *(i)* scions for cross-bunch pointers internal to the group are not considered as roots, and *(ii)* tracing continues across bunch boundaries internal to the group. This algorithm reclaims a cross-bunch cycle unreachable from bunches outside the group. For example, in Figure 1, a group collection including bunches B and C would not consider the scions pointing to y and to z as members of the root. The cross-bunch garbage cycle constituted by objects x, y and z would therefore be reclaimed. Thus, a group collection is complete w.r.t. the group being reclaimed.

The significance of group collection is that any arbitrary subset of the memory can be collected, on a single site, independently of the rest of the memory. The choice of the group to be collected is heuristic, and should maximize the amount of garbage reclaimed and minimize the cost.

To form such groups, Larchant uses a locality-based heuristic. A group contains all the bunches currently cached in the site. This heuristic avoids extra I/O costs. However, it does not enable the reclamation of cross-bunch cycles enclosed in bunches that reside partially on disk. This garbage might be reclaimed with a more aggressive grouping heuristic. This extra I/O cost needs to be balanced against the expected gain. We intend to experiment with the locality-based heuristic over a wide number of applications, and to do some simulation studies. Then, if experimental results mandate it, more complex heuristics will be the topic of future research.

5. Reference-counting

We start by observing that scanning the owner's replica of an object y alone, is not safe: for example, some object z might be reachable from an incoherent replica y_2 and not from the most up-to-date replica y_1. Therefore, a delete may be issued only when the target object z has become unreachable from *all* replicas of the source object y.

As already mentioned in Section 4.1, we call this the union rule: delete is safe only in the union of the stubs of all replicas. A non-owner site of object y informs y's owner of the existence of stubs by a *union* message. After a stub (due to an outgoing pointer from y) has disappeared in all sites caching a replica of y, the owner sends a delete concerning the corresponding scion.

Many coherence protocols impose that only the owner of an object y can write it (*e.g.*, entry-consistency [3]). In this case, a non-owner replica y_2 cannot cause an object unreachable from y_2 to become reachable, because to do so requires writing y on site 2. Thus, the set of stubs at a non-owner site is monotonically decreasing, and therefore union messages may be delivered asynchronously, in FIFO order. In short, the union rule can be implemented cheaply when only the owner of an object can write into it.

5.1. Asynchrony

The cross-bunch collector must be aware of pointer assignments (performed by concurrent mutators) because they may result in the creation of new cross-bunch pointers. Such cross-bunch pointers must be tracked in order to allocate the corresponding stub and create the target scion. Otherwise, the intra-bunch collector might unsafely reclaim a reachable object.

An important observation is that creates do not need to be issued immediately as the corresponding cross-bunch

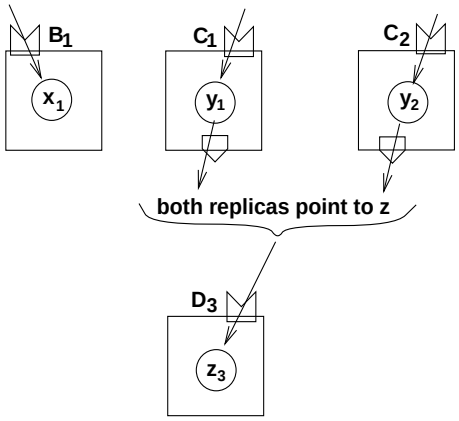

Figure 2. Initial situation.

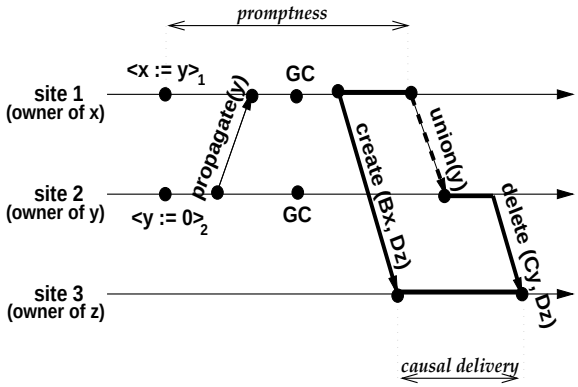

Figure 3. The union message carries the causal dependency (shown in bold lines).

pointer appears (resulting from an assign operation). Thus, creates can be issued asynchronously. This represents a substantial performance and portability advantage, since mutator is not halted, it might avoid work, and enables message batching.

Without loss of generality, consider the following example. Objects x, y, and z are allocated in bunches B, C and D, respectively. Object x is owned by site 1, y is owned by site 2, and z is owned by site 3. Suppose that initially x_1, y_1, and y_2 are reachable, both replicas of y are equal (y was propagated from site 2 to site 1), and x_1 is nil. This initial situation is illustrated by Figure 2.

Now, consider the following sequence (see Figure 3). The mutator executes $<x := y>_1$, therefore creating a cross-bunch pointer from x to z. Then, both replicas of y are modified, such that they no longer point to z (*e.g.*, $<y := 0>_2$ and propagation of y from 2 to 1). Then, bunches B and C are collected on sites 1 and 2.

Clearly, safety depends on the relative delivery order of create and delete at site 3: *(i)* the create message must be sent from site 1 before the delete message is sent from site 2, and *(ii)* the create message must be delivered at site 3 before the delete. To ensure the first condition all create messages of an intra-bunch collector run must be sent before any union or delete message in the same run. The second condition is ensured by delivering the create and delete messages in causal order. (Note that if there is no replication, safety requires that all create messages of an intra-bunch collector run must be sent before any delete message in the same run.)

To conclude, the create operation may be safely issued during all the *promptness* time period indicated in Figure 3 (at the latest before the union message). An asynchronous communication protocol with causal delivery is necessary.

6. Related work

A large amount of literature exists in the area of GC either for multiprocessors [1, 5], or for client-server distributed systems (see Plainfossé[12] for a survey). On the contrary, to our knowledge, little work has been done on GC in a loosely coupled network with weakly consistent DSM.

The fundamental difference between GC in Larchant and GC in a multiprocessor is that of scale and synchronization overhead: if we apply a GC algorithm designed for multiprocessors (*e.g.*, Appel[1]) to our case, the overhead will be unacceptable due to communication and synchronization costs. These costs are due to the fact that current multiprocessor GC algorithms implicitly assume the existence of coherent objects.

Much previous work in distributed GC [10, 12] considers processes communicating by messages (without shared memory), using a hybrid of tracing and counting. Each process traces its internal pointers; references across process boundaries are reference-counted as they are sent in messages. However, none of these previous work supports a DSM mechanism on which multiple replicas of the same object are concurrently accessed by applications running on different sites, as Larchant does.

Previous work on garbage collection in DSM is rare and does not solve our problems as they all assume coherent objects [7, 8].

7. Implementation and Performance

The Larchant prototype implements the coherence protocol *entry-consistency* [3]. This protocol provides the traditional model of multiple readers and a single writer: there can either be several read tokens, or one exclusive

write token associated with each object. Sites holding a read token are ensured to be reading a coherent replica of the corresponding object. Every object has a owner, which is either the site currently holding the object's write token, or the site that last held the write token. A write token can only be obtained from the object's owner, while a read token can be obtained from any site already holding a read token. A token is obtained by performing a read or write *acquire* operation and is freed by the corresponding *release*. The acquisition of a write token for object x implies the *invalidation* of every readable x replica. (See Bershad[3] for more details.)

We implemented two intra-bunch algorithms: mark-and-sweep and copy. Both run concurrently w.r.t. mutators and are based on the replication-based technique from O'Toole et al. [11]. (Obviously, both collectors can also run in non-concurrent mode, *i.e.*, the mutator is halted while the collector runs.)

In the rest of this section we focus on the asynchrony of the cross-bunch collector w.r.t. mutators, and show some performance results of intra-bunch collection.

7.1. Asynchrony

We start by observing that a cross-bunch pointer can only appear or disappear by writing into an object x (assign operation). This requires that the site where the mutator is running holds the write token of x. So, when the mutator acquires the write token of x, Larchant logs x's address (in a log called *GC-log*). The GC-log contains the GC-dirty objects.

Later, when the intra-bunch collector runs, it scans every GC-dirty object locally cached (objects in the GC-log). For each new cross-bunch pointer found, the collector allocates the corresponding stub.

Then, when the cross-bunch collector runs: for each new stub, it issues a create; for each disappeared stub, either issues a union message (local site does not own the corresponding object) or issues a delete after having applied the union rule (owner of the corresponding object). This ensures that creates are always issued before union or delete messages. However, safety also requires causal ordering. For this purpose, create messages are piggy-backed on union and delete messages.

Consider the case illustrated in Figure 3 restricted to the entry-consistency protocol. Note that the operation $<y := 0>_2$ can be done on site 2 because this is the owner of y. The propagation of y from 2 to 1 is done via an acquire-read (entry-consistency operation) performed by site 1. In our implementation, to ensure causality, create(Bx, Cy) is piggy-backed both on the union message (from 1 to 2) and on delete(Cy, Dz) (from 2 to 3).

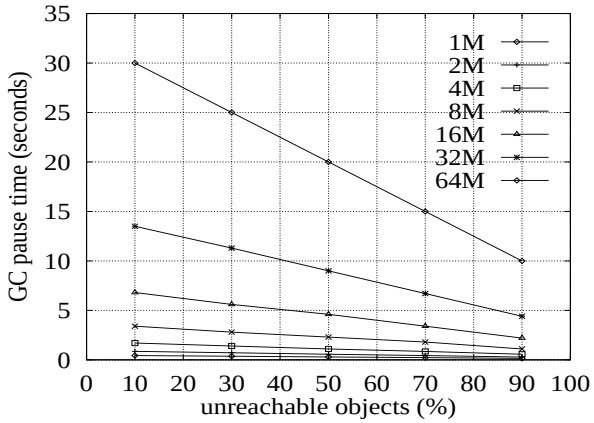

Figure 4. Non-concurrent mark-and-sweep.

7.2. Performance

Note that both the intra-bunch and cross-bunch collectors run concurrently to mutators; thus, GC pause time is only due to intra-bunch collection flips.

To obtain performance results concerning the GC pause time we ran the following benchmark. On each site caching the bunch being collected (maximum of 6 sites in our experiment) the mutator keeps creating objects and inserts some of them in a list; objects in the list are reachable, objects not in the list are garbage.[2] We experimented with bunch sizes from 1 Mbyte to 64 Mbytes. The mean value of the GC pause time both for the mark-and-sweep and copy collectors is 40 milliseconds independently of the bunch size, the number of reachable objects, and the number of sites caching the bunch being collected.

The bunch size and number of reachable objects have no impact on the GC pause time because most of the collector work is done concurrently to the mutator. When flipping, only those objects that were modified by the mutator after being scanned by the collector, are re-scanned (and moved again in the case of a copy collector).

The number of sites caching the bunch being replicated does not impact the GC pause time because each site collects independently and asynchronously from other sites. In particular, when flipping there is no communication among the sites caching replicas of the same bunch.

[2] We run our benchmarks on a network of DEC Alpha workstations. Each has 64 Mb of main memory, 1 Gb of disk, an 8 Kb data cache, a 512 Kb secondary cache, and a 150 MHz clock. They are connected by FDDI.

For comparison, we show in Figure 4 the results obtained with the same benchmark for the mark-and-sweep collector in non-concurrent mode, *i.e*, the mutator is halted while the collector runs. The GC pause time is still independent of the number of sites caching a replica of the bunch being collected. However, for large bunches the GC pause time is very disruptive.

To obtain more performance results, we also implemented a cooperative text editing simulator, called TX1. A TX1 document is a hierarchical structure containing objects for sections, subsections, paragraphs, lines, etc. The simulator performs no work aside from allocating objects, and assigning pointers. We measured the GC pause times of concurrent intra-bunch collection with TX1 running on one, two, and three sites, with a 4 Mb bunch and various amounts of garbage. The GC pause times are never superior to 40 milliseconds. The mean GC pause time is 20 milliseconds and the time between flips varies between 1 and 4 seconds.

We have also ported the well-known OO7 benchmark [6], widely used to measure the performance of object-oriented databases. Our main goal was to test the reliability of the Larchant prototype. The standard OO7 benchmark runs on a single site and does not generate a large amount of garbage. We have modified it to generate extra garbage by mutating the pointer graph in a larger amount. We ran OO7 in a single site and GC pause times are all less than 15 milliseconds.

8. Conclusion

We described the Larchant model, which provides transparent sharing and distribution, persistence by reachability, and automatic memory-management. Persistence by reachability requires tracing the pointer graph from the persistent root and reclaiming unreachable data. This is the task of GC.

The collector in Larchant opportunistically collects local groups of bunches: it traces as long as the trace remains local to a site, and counts references that would cost I/O traffic to trace. There is no coordination or synchronization between applications and collectors. All collector messages are asynchronous; however, for safety, causally-ordered delivery must be ensured.

Given the diversity of useful coherence models, we chose to make only minimal coherence assumptions. Each site collects its bunch replicas independently from any other site without requiring locally available coherent memory. Thus, we believe that our results are generally applicable.

References

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *SIGPLAN'88 - Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta (USA), June 1988.

[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] B. N. Bershad and M. J. Zekauskas. The Midway distributed shared memory system. In *Proceedings of the COMPCON'93 Conference*, pages 528–537, Feb. 1993.

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[5] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proc. of the SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pages 157–164, Toronto (Canada), June 1991. ACM.

[6] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proc. Conf. on the Management of Data*, Washington D.C. (USA), May 1993. ACM SIGMOD.

[7] R. Kordale, M. Ahamad, and J. Shilling. Distributed/concurrent garbage collection in distributed shared memory systems. In *Proc. of the International Workshop on Object Orientation and Operating Systems*, Ashville, North Carolina (USA), Dec. 1993. IEEE Comp. Society Press.

[8] T. Le Sergent and B. Berthomieu. Incremental multithreaded garbage collection on virtually shared memory architectures. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 179–199, Saint-Malo (France), Sept. 1992. Springer-Verlag.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[10] U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server, object database. In *Proceedings of the Parallel and Distributed Information Systems*, pages 239–248, Austin, Texas (USA), Sept. 1994.

[11] J. O'Toole, S. Nettles, and D. 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), Dec. 1993.

[12] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), Sept. 1995.

[13] M. Shapiro, P. Dickman, and D. Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), nov 1992. Also available as Broadcast Technical Report #1.

[14] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), Sept. 1992. Springer-Verlag.