Garbage Collection in the Larchant Persistent Distributed Store

Paulo Ferreira, Marc Shapiro

Garbage Collection in the Larchant Persistent Distributed Shared Store*†

Paulo Ferreira‡ Marc Shapiro
Project SOR - INRIA Rocquencourt
B.P. 105, 78153 Le Chesnay Cedex, France

Abstract

We study tracing garbage collection (GC) for a distributed shared memory (DSM) in order to provide persistence by reachability (PBR), in a large-scale distributed system. Within a general model of DSM, we specify a distributed tracing GC algorithm that scales, collects cycles, and is orthogonal to coherence. Its main features are: (i) piecewise collection of opportunistically-chosen subsets of the memory, (ii) each site collects independently of other sites, (iii) data replicas are collected independently and no coherence operation is needed for GC purposes; and (iv) asynchrony of collection with respect to applications.

1 Introduction

A basic function of an operating system is the sharing of information among its application processes. Larchant supports sharing via a *persistent* virtual memory, shared by applications even if they run at different sites and/or at different times. From the point of view of the application programmer, persistence, memory management, and distribution are all transparent and automatic. Data reachable from a *persistent root* is persistent; unreachable data is not. Unreachable data is removed, and holes compacted.

Such *persistence by reachability* (PBR) requires tracing the pointer graph from the persistent root and garbage collecting unreachable data. Ideally, a GC would be *complete*, *i.e.,* would eventually collect all unreachable data (including cycles). However, scalability and performance conflict with completeness. In our setting, perfect completeness is not economically feasible. We propose an approximate solution that is not provably complete, but which we believe adequate for all real-life situations.

We show how to collect an arbitrary subset of the persistent memory, on a single site, independently of the rest of the memory. To collect cycles, subsets vary in time. The choice of a subset to be collected is heuristic, and should maximize the amount of garbage reclaimed while minimizing the cost. Our current heuristic is locality-based: we collect the subset equal to the data currently in the local cache.

Scanning, moving objects, and pointer patching (for GC purposes), are not observable by applications. The GC is independent from the coherence protocol, *i.e.*, it requires no coherence operation. Thus, the GC algorithm can be used with any coherence protocol.

For safety, GC events must be delivered in mutual causal order; in particular, the events signaling the creation and deletion of remote references.

In this paper we describe the GC algorithm in the context of a persistent distributed shared virtual memory containing ordinary memory pointers. Furthermore, given the diversity of useful coherence models, we make no coherence assumptions. Therefore we believe that our results are generally applicable.

We are currently starting to evaluate a first prototype of Larchant. The first results are encouraging and confirm our expectations in terms of good performance and coherence independence.

The rest of the paper is organized as follows. The upcoming section presents our model of mutators (application programs), memory coherence, and collectors; we make minimal assumptions. Section 3 describes the piecewise collection of dynamically-chosen subsets of the memory without replication. Section 4 presents the collection of replicated bunches. The paper terminates with two sections on related work and conclusions.

2 Model

In this section we propose a general model of the interactions between application programs, the distributed shared memory, and GC. The model is general in the sense that it makes minimal assumptions. Thus, we believe it capable of describing a wide range of distributed shared memory systems and garbage collection techniques. We use the standard vocabulary of the garbage collection literature [7].

*This article appears in the Proceedings of the 5th Workshop on Future Trends in Distributed Computing Systems (FTDCS'95), August 28-30, 1995, Cheju Island, Republic of Korea.

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

‡Full time Ph.D. student at *Université Pierre et Marie Curie (Paris VI)*. Supported by a JNICT Fellowship of Program *Ciência* (Portugal). Tel.: +33 (1) 39-63-53-52, fax.: +33 (1) 39-63-53-30, email: Paulo.Ferreira@inria.fr, www: http://prof.inria.fr/.

The mutator is the application program that dynamically modifies the pointer graph: it creates objects, dereferences pointers, and assigns pointers. An object reachable via some path of pointers from the persistent root is said *live*. As a side-effect of pointer assignment, some live objects become unreachable and are called *garbage*.

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 collects garbage created by the mutator. A collector is composed of a number of threads executing at different sites; we call each one a collector.

Our algorithms are based on an extremely simplified model. Many events traditionally associated with coherence management are not present in the model; some because they are not relevant to GC (for instance, non-pointer writes), and others because our algorithm is independent of them (for instance, reads, tokens, or locks). Furthermore, the model makes minimal assumptions about the coherence protocol (for instance we tolerate arbitrary writes and non-coherent data). All this ensures that the algorithm is applicable to a large variety of shared stores and is very robust.

To simplify the presentation, we assume that all data is replicated at every site. This extends easily to the case where a datum is cached at only some sites, but we do not present the extension, because it would complicate the description without adding any useful new information.

2.1 Objects and Bunches

Our memory is structured at two levels of granularity: (i) The *object* is the unit of allocation, deallocation, and identification. By definition, an object is also the DSM granule of coherence.[1] 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. In order to simplify the notation we assume (unless stated otherwise) that there is a single pointer per object, also written x, y, z. 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_i, x_j, for replicas of x observed respectively at sites i and j.

2.2 Mutator

A mutator running at site i observes object x through the replica currently cached at its site, x_i. Any mutator may write or read an object for which it holds a pointer. (The propagation of an object, to other sites different from the one where it has been written, is addressed in the following section.)

[1] In fact, the size of a coherence granule can include several objects; however, this would complicate the description without any useful new information.

dup (z, x, i)	Atomically, at site i: discard the previous value of x_i; read the value of z_i and write it into x_i.

Table 1: *Relevant event caused unpredictably by mutators.*

ownership (x, i, j)	Pre-condition: i is owner of x, and $x_j = x_i$. Effect: j becomes owner of x.
propagate (x, i, j)	Pre-condition: i is owner of x. Effect: when event received at j, $x_j := x_i$.

Table 2: *Relevant coherence events.*

The event in Table 1 is caused unpredictably by mutators. The left column names the event, and the right one describes its effect as observed by mutators at that site.

The dup event is atomic locally only, *i.e.*, the read and write of the local object replicas are indivisible. There is no need to make this event atomic with respect to remote replicas.

Finally, note that destroying a pointer or creating a new object are special cases of dup.

2.3 Coherence

Table 2 identifies coherence events relevant to GC. In order to accommodate different coherence models, we leave undetermined the times when they occur. (In a practical system, coherence events are caused by mutator activity.) Our minimal assumption is that, at any point in time, each object has a single *owner*. The owner of an object is the only site that is allowed to propagate its value to other sites. Note that this does not preclude concurrent writes (or reads) done on other replicas of x, but simply that only the value from the owner will propagated to other sites. Cache invalidations are modeled as receiving a propagate containing a special value "undefined".

2.4 Garbage collector

To enable a bunch to be collected independently from the rest of the memory, a bunch keeps track of cross-bunch pointers. An outgoing pointer is described by a *stub* and an incoming pointer by a *scion*. A stub identifies its target scion and vice-versa. Each bunch replica has its own replica of stubs and scions. The number of cross-bunch pointers to some object is approximated by the number of scions to that object.

The names stub and scion are inspired by the similar structures found in the SSP (stub-scion pair)

create (B, x, C, y)	Description: a new cross-bunch pointer has appeared; object x of bunch B points to object y of bunch C. Effect: create scion at C.
delete (B, x, C, y)	Description: a cross-bunch pointer has disappeared, that previously had object x of bunch B pointing to object y of bunch C. Effect: delete scion at C.
scan (x, i) $\rightarrow$ y_i, z_i, ...	At site i, list objects y_i, z_i, etc., pointed by object x_i.
move (y, @y')	Move object y to new address @y'.
patch (x, @y', i)	At site i, patch pointer x_i, pointing to y_i, with new address @y'.

Table 3: *Relevant events of a tracing GC algorithm.*

Chain message-passing system [16]. In contrast to SSP Chains, Larchant's stubs and scions are not indirections participating in the mutator computation, but simply auxiliary data structures.

Table 3 lists, without justification, the relevant events of a tracing GC algorithm. This model accommodates both marking and copying collectors [17]. The triggering and the safety of these events will be studied in Sections 3 and 4.

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 deallocates the corresponding stub and deletes the target scion.

The main loop of a tracing GC at some site i uses scan. For some object x_i, scan determines what other objects y_i, z_i, ..., are pointed to by x_i (here we disregard the assumption that x_i contains a single pointer).

A copying collector moves objects (to compact memory and reduce fragmentation) and patches pointers with the new addresses. Objects that, for some reason, cannot be moved are still scanned and collected.

Note that both scan and patch events are local to a site, *i.e.*, they are applied to the local replica and they do not imply a propagate event in spite of the implicit read and write of the concerned object. On the other hand, the move event applies to every replica of an object. However, as described in Section 4, all these GC events do not require any coherence event.

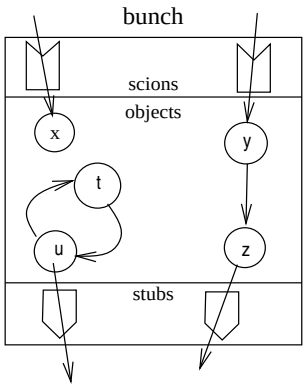

Figure 1: *A bunch containing five objects, three of which are potentially reachable (*x, y *and* z*).*

3 Piecewise GC

Ideally, a GC would be *complete*, *i.e.*, would eventually collect all unreachable data. However, scalability and performance conflict with completeness since the only known provably complete algorithms are based on global tracing and employ a global synchronization. This does not scale and causes a large amount of disk I/O and network traffic.

Thus, in our setting (a large scale network), perfect completeness is not economically feasible and apparently, the problem seems hopeless. But, as we shall show in this section, it is possible to approximate the unfeasible global trace, and to avoid its drawbacks.

Larchant approximates a global trace with a series of non-synchronized, piecewise, local traces. It collects bunch replicas independently of one another. Each bunch is collected at each site where it is cached. For collecting cycles of garbage that span several bunches, groups of bunches mapped at some site are collected simultaneously.

In the following sections, we explain the collection algorithm for piecewise collection, first of a single bunch, then of a group of bunches. To simplify the explanation, we defer the collection of replicated bunches to Section 4. Thus, in this section we assume that there is no replication.

3.1 Collecting a single bunch

A live object is reachable directly or indirectly via a scion into its bunch of residence. By considering its scions as roots, a bunch may be collected independently of others.

Collection of a bunch proceeds as follows (see Figure 1). At the beginning of a run, the collector allocates a new, empty set of stubs; conservatively, the collector considers live those objects that are indicated by a scion. The collector scans live objects. If a live object points to another object inside the same bunch, that object is also considered live. If a live object points to an object outside the bunch, the collector allocates the corresponding stub in the new stub set and does not follow the cross-bunch pointer.

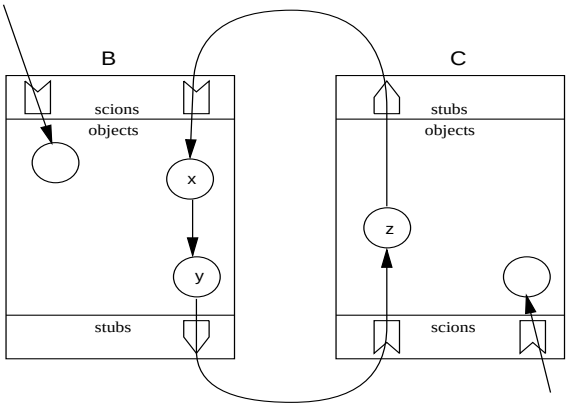

Figure 2: *Cross-bunch cycle of garbage.*

When all live objects of the bunch have been scanned, any objects not live are garbage. The collector compares the new stub set with the old one (resulting from the previous GC). For every stub that has appeared, the collector sends a create message to (the owner of) the target object. For every stub that has disappeared, it issues a delete. These messages are delivered in order but possibly asynchronously. Finally, the collector discards the old stub set; the new set becomes the current one. Collection of this bunch is now finished.

A bunch collection is complete w.r.t. the collected bunch, *i.e.,* it collects all garbage that is entirely within the bunch (for example, the cycle in Figure 1). However, it is conservative w.r.t. other bunches, since it does not collect a cycle of garbage that crosses the bunch boundary.

3.2 Collecting cross-bunch cycles

The same algorithm that collects a single bunch collects any group of bunches as well. The only difference w.r.t. Section 3.1 is that: (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 is complete w.r.t. to the group of bunches being collected. As an example, consider Figure 2: the cycle formed by objects x, y and z is collected because the scions referencing x and z are not members of the root.

The choice of the group to be collected is heuristic, and aims at maximizing the amount of garbage reclaimed while minimizing the cost. In our current prototype [9] we use a very simple locality-based heuristic that favors simplicity (instead of completeness): a group is formed by every bunch mapped on a site. This heuristic avoids extra disk I/O and network traffic due to the GC. However, cycles not included in the group are not collected. We believe that this garbage can be collected by improving the grouping heuristic, for instance by forcing bunches to be mapped on a site at the same time, if that rarely happens due to applications behavior. Thus, collecting such cross-bunch garbage involves I/O costs that need to be balanced against the expected gain. First, we intend to experiment with the locality-based heuristic over a large number of applications. Only then, if experimental results mandate it, we will explore more complex heuristics.

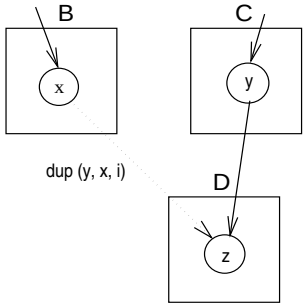

Figure 3: *Example illustrating promptness condition; mutator executes* dup (y, x, i). *Bunches* B, C *and* D *are not replicated in Section 3.3 and are replicated in Section 4.3. (Stubs and scions are elided in the figure.)*

3.3 Promptness (non-replicated case)

A collector must be aware of pointer assignments performed by concurrent mutators [7, 17]. Such assignments may result in the creation of new cross-bunch pointers; the collector must recognize them in order to allocate the corresponding stub and create the target scion. Otherwise, the collector might erroneously collect a reachable object.

However, creates do not need to be sent immediately as the corresponding cross-bunch pointer is created. In fact, delaying creates represents a substantial performance advantage, since this simplifies pointer assignment, might avoid work, and enables message batching. We will now examine how late a create may safely be delayed; we call *promptness* the corresponding safety condition.

Without loss of generality, we consider the example illustrated by Figure 3. Objects x, y, and z are located in bunches B, C and D respectively. Suppose that initially both x and y are reachable, and y points to z. The mutator executes dup (y, x, i) (since we assume no replication for the time being, site argument i is meaningless). We will say object x is *GC-dirty* after being modified by dup; it remains GC-dirty until scanned. We will say a bunch is GC-dirty if it contains any GC-dirty object.

By definition, at the time of the dup, y is reachable and z is reachable via y. As long as D is not collected, it is not necessary to deliver create (B, x, D, z). We can state the promptness condition thus:

> Deliver create (B, x, D, z) before z could otherwise be collected (1)

This condition cannot be implemented easily.[2] So

[2] It implies that immediately before collecting D, every live

we will now replace it with progressively weaker ones. The only pointer we know for sure points to z is the one from y, so, assuming messages are delivered in the order sent, we have:

Send create (B, x, D, z) before sending delete (C, y, D, z) (2)

Recall that we are currently assuming that there is no replication. The delete might be sent either because y changes value, or because it became unreachable; but it cannot be sent until the enclosing bunch C is collected (by the algorithm of Section 3.1). Thus, the promptness condition becomes:

Send create (B, x, D, z) before C collected (3)

To apply Statement 3 effectively in an implementation, we notice that the create message will be sent by collecting the GC-dirty bunch B.

A possible implementation of promptness is thus the following. The collector may run at arbitrary times. When it runs, it collects at least all GC-dirty bunches. (It is allowed, but not obligated, to collect clean bunches at the same time, because, for instance, y may be clean but have become garbage). However, to take into account the 'before' condition above, all create messages of a collector run are sent before any deletes in the same run.

There are numerous ways to implement detecting GC-dirty objects or bunches. The simplest is to conservatively assume all bunches GC-dirty, *i.e.,* to collect all bunches. If the coherence protocol only allows the owner of an object to modify it, then the collector might instead conservatively assume that any object the current site owns, is GC-dirty. Alternatively, a compiler may insert a *write barrier* instrumenting every pointer write, that sets a GC-dirty bit associated with the object [17]; collecting a bunch resets the GC-dirty bits of its objects. Another alternative, assuming objects are protected by locks, is to set the bit when taking a write lock; in this case, the scan resets the GC-dirty bit only if the lock has been released [9].

4 GC of Replicated Bunches

In this section we describe how a replicated bunch is collected focusing on the independence from coherence. A collector runs at each site independently and asynchronously from collectors at other sites.

The main problems we have to consider are synthesized by the following questions: (i) must collectors synchronize with each other ? (ii) does scanning need coherent data ? (iii) when using a copying GC algorithm: (a) which collector decides where to move an object ? (b) is it necessary to acquire the ownership of an object in order to move it or to patch its internal pointers ?

Our GC algorithm does not require synchronization between the collectors, nor any coherence event. As a consequence, the GC does not interfere with applications coherence needs. The price to pay for these features is that the GC is conservative, and some messages need to be delivered in causal order.

4.1 Scanning and Coherence

The scanning of a non-coherent replica evidently does not take into account pointer writes occurring at the owner. Less obviously, observe that scanning the owner's replica alone is not safe: some object might be reachable via a non-coherent replica. Therefore, a delete may be sent only when the target object has become unreachable from *all* replicas of the source object.

We call this the *union rule*: delete is safe only in the union of the stubs of all replicas. A non-owner replica informs the owner of the existence of stubs by a union message. Only after a stub has disappeared in all replicas, the owner sends a delete to deallocate the corresponding scion.

Many coherence protocols impose that only the owner of an object y can write it. Thus, a non-owner replica y_i cannot cause an object unreachable from y_i to become reachable, because to do so requires writing y. Thus, the set of stubs at a non-owner 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.

4.2 Moving, Patching and Coherence

A copying collector may move an object from one location to another. It maintains the invariant that move is not observable by any mutator, by patching pointers to the moved object.[3] In this section, we study move and patch with respect to coherence.

The first problem is to avoid two sites move'ing (their local replicas of) the same object to two different locations concurrently. A simple solution, without interfering with applications coherence needs, is that the owner site of some object decides where to move it. Non-owner sites will move their replicas of an object to the same address after receiving a move message from the owner.

The second problem is the following, must the collector acquire the ownership of an object in order to move it to a new location or to patch one of its pointers with the target's new location? Surprisingly, the answer is no, because these events are visible only at the site where they occurred, or transitively because a patched pointer is dup'ed and propagate'd.

Therefore a live object is moved as follows. (i) Site i, the owner of y, sends a move (y, @y') message to

object in every bunch has to be scanned to find if there is some cross-bunch pointer referencing z (as is the case of x).

[3] We refer to the GC literature [17] on how to implement move and patch, concurrently with mutators, on a single site.

sites with a replica of y or a pointer to y, including to itself. (ii) A site j receiving a move message copies its replica y_j to the new location and patches pointers accordingly; the move and patch actions are atomic locally to site j. Note that move messages are delivered in the background. Thus, they do not disrupt applications functioning.

The collector of a bunch replica may *flip*[4] independently from the collectors of other replicas of that same bunch. Before a GC on a site terminates, the local site must have already received all the move messages regarding live objects not locally owned.

To summarize, move and patch are not observable by mutators and therefore can be carried by writing local replicas with no special precautions. In particular, no coherence event is generated. In addition, there is no synchronization between collectors running on different sites.

4.3 Promptness (replicated case)

In this section we generalize the promptness condition derived in Section 3.3 to the replicated case. In this case, by the union rule of Section 4.1, a delete message can be sent only when the corresponding stub is unreachable at all sites. In particular (see Figure 3), delete (C, y, D, z) cannot be sent until the stub on site i (non-owner), connecting y_i to z, is removed at i, and the corresponding union message sent to the owner site of object y.

Generalizing Section 3.3 to the replicated case, a suitable algorithm that satisfies promptness is the following. The collector at site i may run at arbitrary times. When it runs, it collects at least all GC-dirty bunches. All create messages of a collector run are sent before any union message for suppressed stubs, in the same run.

The detection of GC-dirty objects (or bunches) can be done in the same way as described in Section 3.3.

4.4 Causal delivery of create and delete

We have already seen how promptness is ensured. In other words, how create and delete events are made to occur in a safe order. However, this is not enough to guarantee the correctness of the GC. In addition to promptness we must ensure that create and delete events are also delivered in a safe order. In fact, as seen in the previous section, more than two sites can be involved, which implies the need for *causal delivery* [4].

Considering Figure 3, it is clear that safety depends on the relative delivery ordering of create and delete at site k (see Figure 4). (We ignore the uninteresting case where the dup (nil, y, j) completes and is propagated to site i before the dup (y, x, i), causing z to become garbage.) Thus, there is a causal dependence between the create and delete messages. We require therefore a causal transport protocol that will deliver them in causal order. It's worthy to note that the union rule of Section 4.1 implies a hidden union message from site i to site j that ensures the causal ordering.

[4] A flip in GC terminology designates the moment after which the mutators start observing only the objects already moved.

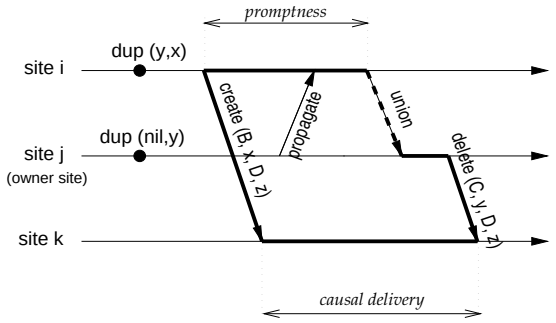

Figure 4: *Promptness and causal delivery of* create *and* delete *messages. Note the implicit* union *message that carries the causal dependency (shown in bold lines).*

In conclusion, an asynchronous communication protocol with causal delivery is necessary to ensure GC safety.

5 Related work

The concept of PBR was first proposed by Atkinson and Morrison [3, 15] in the early 1980's. We have found two PBR+DSM systems in the literature. The specification of the Casper collector [11] is sketchy and seems incapable of collecting a persistent object that has become garbage. EOS [10] has a tracing GC that takes into account user placement hints to improve locality. However, their GC is quite complex and has not been implemented.

Much previous work in distributed garbage collection [5, 16] considers processes communicating by messages (without shared memory), using a hybrid of tracing and reference counting. Each process traces its internal pointers; references across process boundaries are reference-counted as they are sent in messages. Some object-oriented databases use a similar approach [1, 6, 18], *i.e.*, a partition can be collected independently from the rest of the database. In particular, Thor is a research OODB with PBR [14]. In Thor, the data resides at a small number of servers and is cached at workstations for processing. A Thor server counts references contained in objects cached at a client; Thor delays the creation of scions as proposed in Section 3.3.

On top of a hybrid algorithm, Lang, Queinnec and Piquer [12] propose to scan dynamically-changing groups of processes in order to collect cycles of garbage. Lang's groups are distributed, and therefore the mechanism to form and disband a group is quite complex, as is the internal synchronization inside a group. They do not propose any grouping heuristic.

Previous work on garbage collection in shared memory deals either with multiprocessors [2, 8] or with a small-scale DSM [13]. These authors make strong coherence assumptions.

6 Conclusions

We described a GC algorithm in the context of a persistent distributed shared memory containing ordinary memory pointers.

The GC in Larchant is a novel hybrid of tracing and counting. It traces whenever economically feasible, *i.e.*, as long as the trace remains local to a site, and counts references that would cost I/O or network traffic to trace. The reference-counting boundary changes dynamically and seamlessly, and independently at each site, in order to collect cycles of garbage.

Given the diversity of useful coherence models, we chose not to make any coherence assumptions. As a result, the GC is orthogonal to coherence. The collector can work with incoherent objects and therefore there is no interference with applications coherence needs.

For these reasons the GC algorithm scales well, is robust, and should be applicable to many other cases, such as persistent object stores and shared-memory multiprocessors, no matter the coherence protocol being used.

Acknowledgments

We are grateful to Laurent Amsaleg and Christian Queinnec for the interesting discussions regarding the subject of this paper.

References

[1] Laurent Amsaleg and Olivier Gruber. Efficient incremental garbage collection for workstation/server database systems. Technical Report 2409, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1994.

[2] Andrew W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience*, 19(2):171–183, February 1989.

[3] 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.

[4] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[5] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, NC (USA), December 1993.

[6] 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.

[7] E. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Comm. of the ACM*, 21(11):966–975, November 1978.

[8] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, pages 113–123, Charleston SC (USA), January 1993.

[9] 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.

[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.

[11] Bett Koch, Tracy Schunke, Alan Dearle, Francis Vaughan, Chris Marlin, Ruth Fazakerley, and Chris Barter. Cache coherency and storage management in a persistent object system. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 99–109, Martha's Vineyard, MA (USA), September 1990.

[12] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Proc. of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, Albuquerque, New Mexico (USA), January 1992.

[13] T. Le Sergent and B. Berthomieu. Incremental multi-threaded 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), September 1992. Springer-Verlag.

[14] Umesh Maheshwari. Distributed garbage collection in a client-server, transactional, persistent object system. Technical Report MIT/LCS/TM-574, Mass. Inst. of Technology, Lab. for Comp. Sc., Cambridge, MA (USA), October 1993.

[15] R. Morrison, M. P. Atkinson, A. L. Brown, and A. Dearle. Bindings in persistent programming languages. *SIGPLAN Notices*, 23(4):27–34, April 1988.

[16] Marc Shapiro, Peter Dickman, and David 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.

[17] 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.

[18] V. Yong, J. Naughton, and J. Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proc. Data Engineering Int. Conf.*, pages 120–133, Houston TX (USA), February 1994.