Garbage Collection of Persistent Objects in Distributed Shared Memory

Paulo Ferreira, Marc Shapiro

Garbage Collection of Persistent Objects in Distributed Shared Memory*†

Paulo Ferreira‡
INRIA - Projet SOR
Rocquencourt - France

Marc Shapiro
INRIA - Projet SOR
Rocquencourt - France

Abstract

This paper describes a garbage collection algorithm for distributed persistent objects in a loosely coupled network of workstations. Objects are accessed via a weakly consistent shared distributed virtual memory with recoverable properties. We address the specific problem of garbage collecting a large amount of distributed persistent objects, cached on several nodes for efficient sharing.

For clustering purposes, objects are allocated within segments, and segments are logically grouped into *bunches*. The garbage collection subsystem combines three sub-algorithms: the *bunch garbage collector* that cleans one bunch (possibly multiply-cached) independently of any other, the *scion cleaner* that propagates accessibility information across bunches, and the *group garbage collector* aimed at reclaiming inter-bunch cycles of dead objects.

These three sub-algorithms are highly independent. Thus, the garbage collection subsystem has a high degree of scalability and parallelism. On top of this, it reclaims cycles of garbage, it does not require any particular communication support such as causality or atomicity, and is well suited to large scale networks.

1 Introduction

Garbage collection (GC) is a fundamental component of a platform supporting distributed persistent objects. As a matter of fact, applications are becoming more and more complex, and their object graphs extremely intricate. Thus, manual storage management is increasingly difficult and error-prone. Even in 64-bit address spaces, memory reorganization and address recycling are necessary [12]. Otherwise, garbage ends up filling the secondary storage, and the address space becomes fragmented.

Our GC design is integrated in a platform, called BMX (Bunch Manager Executive) [6], supporting distributed persistent objects, not only via a weakly

*This article appears in the Proceedings of the Sixth International Workshop on Persistent Object Systems, 5 to 9 September 1994, Tarascon, Provence, France.

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

Author's address: Paulo.Ferreira@inria.fr, INRIA, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, FRANCE, Tel: +33 (1) 39 63 52 08, Fax: +33 (1) 39 63 53 30.

‡Supported by a JNICT Fellowship of Program Ciência (Portugal). Also affiliated with Université Pierre et Marie Curie (Paris VI).

consistent distributed shared memory (DSM), but also via remote procedure calls (RPC). The garbage collection design is language independent and can be used in other systems supporting distributed persistent objects.

In this paper, we focus our attention on garbage collecting a single address space (implemented with a DSM mechanism) spanning several machines on a network, including secondary storage (see Plainfossé[17] for GC of objects accessed via RPC). There are two fundamental problems related to GC in such a DSM platform: (i) how to garbage collect the large amount of objects cached on several nodes without stopping the system, and (ii) how to perform a GC without interfering with the consistency protocol. Due to space limitations, in this paper we address only the first issue. The second one is addressed by Ferreira[7].

This paper is organized as follows. The next section presents the most important aspects of the BMX platform. In Section 3 we describe the GC design. Sections 4, 5, and 6 describe the GC sub-algorithms, and Section 7 presents the implementation. The paper ends with sections on related work and conclusion.

2 The BMX Platform

This section describes the most important aspects of the BMX platform that are relevant for the GC design (see Ferreira[6] for more detail). BMX offers a 64-bit single address space spanning all the nodes of a network, including secondary storage.

2.1 Objects, Segments, and Bunches

An object is a contiguous sequence of bytes. The granularity of identification and invocation is the object. An object is identified by its address. We assume that objects are passive and generally small, i.e., the size of most objects is much smaller than a virtual memory page.

For clustering purpose objects are allocated within segments. A *segment* is a set of contiguous virtual pages. The size of a segment is defined at creation time and remains constant. BMX ensures that segments have non-overlapping addresses.

Segments are logically grouped into *bunches* because a single segment is not flexible enough for holding an application's data (for instance, segment overflow could occur). Each bunch has an associated owner and protection attributes (the usual Unix read, write and execute permissions), and a set of manager methods that provide a specific management policy of the enclosed objects.

2.2 Programming Model

The user program (usually called the *mutator* in the GC literature [5]) sees a huge graph of objects allocated within a large number of bunches. The bunch where an object is to be allocated is chosen by the programmer taking into account its protection attributes and the management policy provided by the bunch's set of manager methods.

Bunches are potentially persistent: a bunch enclosing a persistent object becomes a persistent bunch; an object becomes persistent by reachability, i.e., when it becomes reachable from the persistent root.

Bunches are mapped in shared memory, and can be simultaneously cached on several nodes of the network. For consistency purposes, the system supports a weakly consistent DSM with the entry consistency protocol [2]. This protocol supports the traditional model of multiple readers and a single writer. There can be either several read tokens, or (exclusively) a single write token associated with each object. Nodes holding a read token are ensured to be reading a consistent version of the object. The possession of the write token ensures that there is no other consistent copy of the object in any other node of the network. Otherwise, i.e., if a process does not possesses a token, the observed object's state is undefined.

There is a notion of object owner, which is either the node holding the corresponding write token, or the node that last held it. A write token can only be obtained from the owner, and a read token can be obtained from any node already holding a read token. Thus, the copy-set of an object (set of nodes with a read token) is not centralized by its owner; rather, it is distributed among the owner and those nodes holding a read token that have given a read token to other nodes. The token management is done with an algorithm similar to Li's dynamic distributed manager with distributed copy sets [15]. Thus, for each object, there is a forwarding pointer mechanism indicating which node is the current object's owner. We call such a pointer, `ownerPtr`.

In order to support persistent objects capable of surviving workstation crashes, bunches are supported by a recoverable virtual memory mechanism [18]. When there is an object fault (the application is trying to invoke an object not locally mapped) the BMX gets the enclosing bunch and maps it. This mapping automatically initiates a logging mechanism for the range of addresses that will be modified by the application. Thus, every modification performed on that range of addresses has an associated log, and can be recovered in case of a crash, using a mechanism of recoverable virtual memory. Before un-mapping a bunch, the system automatically flushes the associated log (intermediate flushes can be also made).

3 Garbage Collection Design

In this section we describe the main problems the GC design has to deal with, and the guidelines of the corresponding solutions, i.e., the three sub-algorithms that are utilized.

3.1 Main Issues

The specific problems the GC is faced with can be summarized as follows:

- *Amount of objects.* The graph of objects is enormous and widely distributed, therefore it is not feasible to collect them all at the same time.

  The solution is to reclaim groups of related objects independently: each bunch is collected by its own garbage collector, called the *bunch garbage*

collector (BGC), independently of other bunches and of other copies of the same bunch on other nodes.

- *Accessibility propagation.* Grouping objects into bunches raises the problem of propagating accessibility information concerning inter-bunch references.

  The *scion cleaner* is the sub-algorithm that is responsible for propagating accessibility information, and discovering which objects are no longer reachable from other bunches or from other copies of the same bunch.

- *Cycles of garbage.* Independent GC of bunches (BGC) with accessibility propagation (scion cleaner) fails to collect inter-bunch cycles of dead objects.

  The *group garbage collector* (GGC) collects groups of bunches in order to reclaim inter-bunch cycles of dead objects.

- *Performance.* The mutator must not be subjected to long pauses due to the GC.

  Therefore, we use an incremental copying algorithm (for the BGC and GGC), and the GC of different bunches (or of different copies of the same bunch) is done in parallel with no synchronization constraints.

3.2 Outline of the Algorithms

To keep the isolation of a bunch for the purpose of GC, and in addition to the set of outgoing and entering `ownerPtrs`, each cached copy of a bunch has two associated tables (see Figure 1). The *scion table* contains information about which objects are referenced from which other bunches. The *stub table* contains information about which referenced objects are allocated in which other bunches. Thus, for each stub there is a corresponding scion: these two entities form a stub-scion pair (SSP)[1].

There are two kinds of slightly different SSPs: an *inter-bunch* SSP describes inter-bunch references; an *intra-bunch* SSP records relevant dependencies between replicas of a given bunch.

Inter-bunch SSPs have the same direction of the inter-bunch reference they represent (e.g., `O7`→`O5` in Fig. 1). Intra-bunch SSPs have the opposite direction of the corresponding `ownerPtr`, i.e., they start at the owner node. This is due to the purpose they serve: to preserve an object's replica at a node that stores inter-bunch stubs created when the node was previously the owner of the object, but is no longer so. For example, in spite of being unreachable by the mutators at `N1`, object `O3` must be kept alive because it is reachable by a mutator in `N2`, and there is an inter-bunch reference starting from `O3`, for which the corresponding stub is allocated in `N1`.

Each execution of the local BGC reconstructs a new version of the stub table in which each entry points to remote scions still reachable after the collection, and a new list of outgoing `ownerPtrs` (non-owned objects still alive

[1] Our stub-scion pairs are much simpler than the ones used in RPC-based distributed systems [19] because they are not used for indirections, that is, they are just auxiliary data structures that describe relevant references, and do not perform any kind of marshaling/unmarshaling.

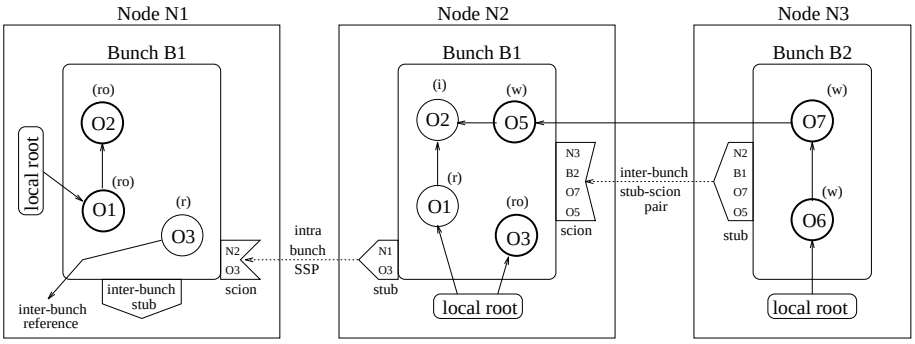

Figure 1: Stub and scion tables contain inter-bunch and intra-bunch SSPs. For each object the state of its token is indicated as follows: letters `r` and `w` indicate that the node has a read or a write token respectively; `o` means that the node is the object's owner (`ownerPtrs` are not represented); `i` is used for inconsistent copies. The `local root` includes the mutators stacks.

locally). When the local BGC terminates, the new stub table and the new list of `ownerPtrs` are sent to other nodes on which the scion cleaner (based on the received information), will delete the unreachable scions and unnecessary entering `ownerPtrs`. Later, those objects that were reachable only from the just deleted scions or entering `ownerPtrs`, will be reclaimed by the local BGC.

When the scion cleaner looks at a scion in order to find if it is still reachable (e.g., `(N3,B2,O7,O5)` in Fig. 1), it scans the stub table of the source bunch indicated by the scion (`B2` in this case). Note that the scion cleaner does not have to scan the stub tables of all cached copies of the source bunch. For example, even if `B2` was cached on nodes `N3` and `N4`, the scion cleaner only has to scan the stub table of the cached copy indicated by the scion being considered (see Section 5.2).

The BGC and scion cleaner do not reclaim cycles of dead objects spanning more than one bunch. To collect them involves scanning every bunch containing an object of such a cycle. For this purpose, the GGC groups bunches in a dynamic way. The heuristic used for grouping bunches does not disrupt the applications working set because it takes into account their locality behavior, as described in Section 6.

The three sub-algorithms above outlined perform complementary tasks: local collection of a bunch (BGC), find which objects are no longer reachable from other bunches (scion cleaner), and reclaim inter-bunch cycles of garbage. Together, these sub-algorithms support an integral GC solution for DSM.

4 Bunch Garbage Collector

In order to maintain the bunch isolation for the purpose of GC, the system must keep track of exported and imported references (with respect to each bunch), without reducing applications performance. This reference tracking is similar to the inter-generation pointer tracking performed in generational garbage collectors [20]. Thus, we use the same kind of write-barrier [8]: every

pointer assignment is tested (by code generated by the compiler) to see if the new pointer crosses a bunch boundary (from a source bunch to a target bunch). If so, then: (i) create a stub in the source bunch's stub table, and (ii) create the corresponding scion in the target bunch's scion table (see inter-bunch reference 07→05 in Fig. 1). Section 5.2 describes the creation of such an SSP in detail.

For the local BGC, every object referenced (either directly or indirectly) from the root is considered live. The root of a local BGC is the union of: local mutators stacks, intra-bunch and inter-bunch scions, and those objects for which there is at least one entering `ownerPtr`. The set of entering `ownerPtrs` ensure that a shared object locally owned and no longer reachable by local mutators, is kept alive as long as there are `ownerPtrs` coming from other nodes. As described in Section 5.1, a shared object is detected as being dead by the local BGC executed at its owner node. A necessary condition for such death is the unexistence of entering `ownerPtrs` for that object.

Each bunch is cleaned by its own garbage collector: the BGC incrementally copies live objects from from-space to to-space within the same bunch. When a bunch has several copies, the BGC is composed of a garbage collector per copy (local BGC). The local collection of a bunch can proceed independently (i.e. asynchronously, with no synchronization) from the collection of other bunches, and from the collection of copies of the same bunch at other nodes.

The BGC is an incremental copying collector. We use a copying algorithm because this solution seems to be more efficient than others [20, 21]; in particular, it improves locality. However, any other algorithm could be used as well. Currently, the BGC is based on Nettles's algorithm [16]. We use this technique because it is is well suited to systems in which there is already a log (for making objects tolerant to crashes), as is the case of BMX (see Section 2.2).

It would seem that copying live objects would interfere with consistency because copying a live object invalidates all its cached copies. However, our solution avoids such a negative effect. The main idea consists of: (i) a live object is copied by the local BGC only at its owner node, and (ii) the scanning of a live object can be done on an inconsistent copy (this results in a more conservative approach w.r.t. scanning a consistent copy). This solution is based on the observation that GC needs in terms of consistency, are less strict than applications'. See Ferreira[7] for more detail.

When the BGC copies a live object from from-space to to-space (in the same bunch), a forwarding pointer is written in the object's header (left in place of the copied object). Other references to that same object will be updated accordingly. In particular, remote references (from objects in other bunches) must be updated because inter-bunch references are direct (i.e., SSPs do not provide any indirection). Such an update is done lazily, and does not imply sending extra messages. In fact, such information (object's new address in to-space segment) is piggy-backed on messages either used by the consistency protocol, or containing new stub tables. Thus, applications are not disrupted.

5 Scion Cleaner

The scion cleaner propagates accessibility information, and discovers which scions are no longer reachable from any stub and which entering `ownerPtrs` are no longer necessary. In this section we first describe how intra-bunch SSPs are created and deleted. Then, inter-bunch SSPs are considered.

5.1 Intra-bunch SSPs

Intra-bunch scions are created when the ownership of an object moves from one node to another and the old owner holds an inter-bunch stub for this object. Thus, the intra-bunch SSP takes care of creating a forwarding link between the new owner and the inter-bunch stub at the old owner. For example, in Figure 1, when the ownership of `O3` goes from `N1` (where the inter-bunch reference was created) to `N2`, the corresponding intra-bunch SSP from `N2` to `N1` is created.

For each shared object there is a forwarding pointer used for the purpose of token management, i.e., an `ownerPtr` pointing to the object's owner node. As long as such an object is alive, there is an intra-bunch SSP going from the object's owner to every node sharing that object (opposite direction of `ownerPtrs`) in which there is an associated inter-bunch stub. Now, suppose that a shared object is no longer reachable by any mutator in any node. Thus, this object is in fact dead. The GC subsystem will detect such death as explained now.

A local BGC starts scanning those objects that are referenced from: local mutator stacks, inter-bunch scions, and entering `ownerPtrs` (objects locally owned still alive on other nodes). While the local BGC is executing, not only a new stub table is created (containing inter-bunch and intra-bunch stubs), but also a new set of exiting `ownerPtrs` (indicating the owner node of each not locally owned live object). An object not locally owned that is not locally reachable, is not found by the local BGC. Therefore, the corresponding `ownerPtr` will not be part of the new set of outgoing `ownerPtrs`.

Then, the local BGC scans those objects that are reachable from intra-bunch scions. Contrary to the previous scanning (done from the other elements of the root), the resulting outgoing `ownerPtrs` are not created. As a matter of fact, such objects are alive only because they are reachable from some mutator at some other node and there is at least an associated inter-bunch stub in the local node. Only the resulting stubs are inserted in the new stub table being created by the local BGC (see `O3` at `N1` in Fig. 1).

Finally, the new intra-bunch stubs constructed by the local BGC (due to locally owned live objects), are sent to other nodes sharing the corresponding objects, and the new set of exiting `ownerPtrs` (not locally owned live objects) is sent to the nodes they point to. Once received by the destination nodes, the intra-bunch stubs are scanned by the scion cleaner for the purpose of finding the unreachable intra-bunch scions. The same reasoning applies to entering `ownerPtrs`. Both unreachable scions and unused entering `ownerPtrs` are deleted.

Hence, a shared object no longer reachable by any mutator at any node, will end up not being referenced by any entering `ownerPtr` in its owner node. Thus, when the local BGC at the owner node is executed, the mentioned object will not be *seen*, and the corresponding intra-bunch stub(s) will not be part of the

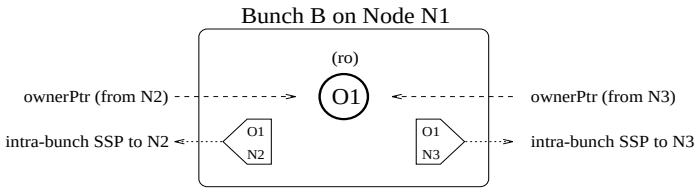

Figure 2: Reclamation of a shared object: O1 is cached on nodes N1, N2, and N3. In nodes N2 and N3 there is an inter-bunch stub due to O1 (that is why there is an intra-bunch SSP from N1 to each one of those nodes).

new stub table. The scion cleaner running at other nodes, after receiving the new intra-bunch stubs from the owner, will delete the unreachable intra-bunch scion(s), and the referenced object will be reclaimed by the next local BGC.

Figure 2 illustrates how a shared object is reclaimed: O1 is no longer reachable by any mutator at any node, and the local BGC has not started yet in any node. Now, suppose that the local BGC in N2 is executed. The new set of exiting `ownerPtrs` from N2 will no longer contain the `ownerPtr` corresponding to O1. This fact will be communicated to N1, and the local scion cleaner will delete the entering `ownerPtr` from N2. Because there is still an entering `ownerPtr` from N3, O1 will be kept alive by the local BGC in N1, and the intra-bunch SSPs ensure that O1 remains alive in N2 and N3. After the local BGC in N3 has executed and the new set of exiting `ownerPtrs` has been communicated to N1, there will be no entering `ownerPtrs` in N1, and the local BGC will consider O1 to be dead. Thus, the intra-bunch stubs in the figure will no longer exist in the new stub table, and this information is communicated to N2 and N3. The scion cleaner and the local BGC in these two nodes will delete the unreachable intra-bunch scions and reclaim object O1, respectively.

Finally, note that a new stub table and/or a new set of exiting `ownerPtrs`, can be piggy-backed in messages used by the consistency protocol, or exchanged in the background. Thus, they do not disrupt applications functioning.

5.2 Inter-bunch SSPs

In DSM there is only one way for creating an inter-bunch reference (see Fig. 3): assignment operation that reads a reference from an object cached on several nodes (e.g., O1), and writes it into another object (O2). Thus, for instance, the reference to O3 has been passed to node N2, through O1 that is cached on N1 and N2 (references only travel between nodes inside objects).

When creating an inter-bunch reference, either both source and target bunches are already mapped on the local node, or only the target bunch is not yet locally mapped. In the first case, the corresponding SSP is created locally. The second case, requires sending a message to the node where the target bunch is mapped. This message is called *scion-message* and is used to inform the target bunch about the new (inter-node) inter-bunch reference that has been created, and the necessity of creating the corresponding scion. Figure 3 illustrates both cases: locally created SSP corresponding to O1→O4 on N1, and inter-bunch SSP due to O2→O3 (from N2 to N3).

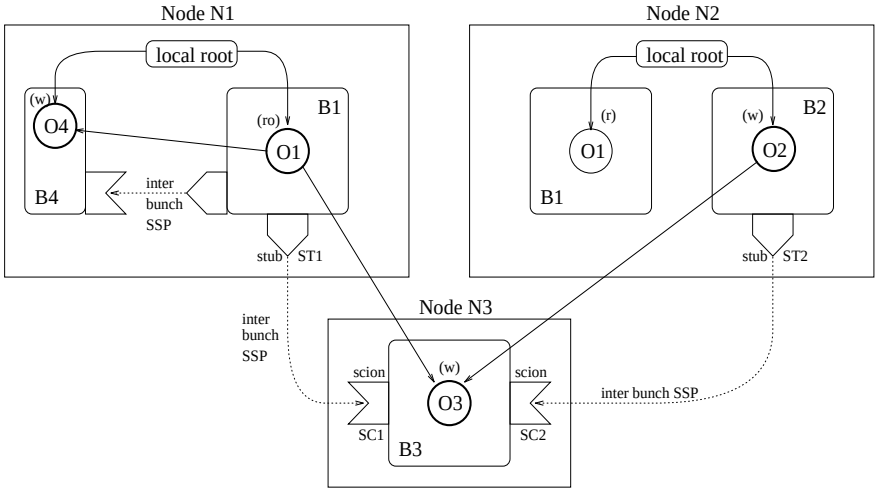

Figure 3: Bunch `B1` is cached on nodes `N1`, and `N2`. Bunch `B3` is mapped only on `N3`. Object `O1` was initially mapped only on `N1`. Then, as the result of an `acquireRead`, it becomes mapped also on `N2`. Object `O3` is not reachable from the mutators in `N3` (no references from the `local root`).

When an object becomes cached on multiple nodes, the inter-bunch stubs that are due to its references do no have to be replicated. In fact, inter-bunch stubs and scions are not exactly the same in every copy of the same bunch. This is not problematic because a single SSP is enough to keep the target object alive. Instead of replicating inter-bunch SSPs, we use intra-bunch SSPs because no scion message is necessary and the amount of memory consumed for GC purposes is smaller. Figure 3 illustrates such a situation: in spite of the fact that `O1` is cached on `N1` and `N2`, there is only one inter-bunch stub due to `O1`→`O3` that is kept at `N1` (node where the inter-bunch reference was created). If `O1` becomes reachable only from the local root at `N2`, `O1` will be kept alive at `N1` by the corresponding entering `ownerPtr` that comes from `N2`.

As already mentioned, each time a local BGC is executed, a new version of the bunch's stub table is constructed. A bunch only receives new versions of stub tables from other bunches that are mapped either on the same node, or on a node from which a scion-message has already been received. Thus, `B3` must receive a scion-message from `N1` when the inter-bunch reference `O1`→`O3` is created. Otherwise, the scion cleaner in `N3` is not aware of such inter-bunch reference. The same reasoning applies to the reference `O2`→`O3`.

The main advantage of sending stub table messages over sending increment/decrement messages [3] is that the former are idempotent. In case of lose they can be retried, and is not necessary a reliable communication protocol. However, scion messages and stub tables must be received in FIFO order. Otherwise, the scion cleaner may use an old stub table that is not consistent with the scions being considered. Such inconsistency could result in the erroneous deletion of a scion. For example (see Figure 4), if a scion message is sent after a stub table message, and they are received in the opposite order,

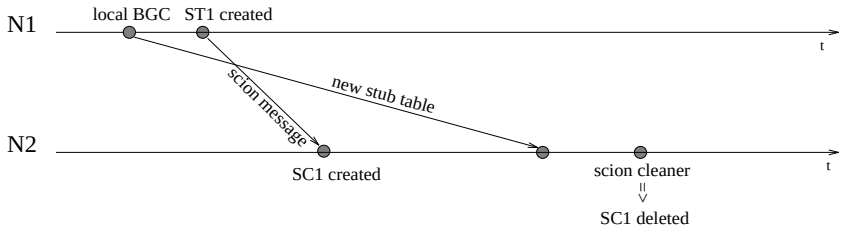

Figure 4: A new inter-bunch reference is created between nodes N1 and N2: situation in which the scion-message and a stub table message are received in the opposite order w.r.t. their sending.

the scion cleaner in the receiving node will not find the stub corresponding to the scion just created (SC1), in the new stub table. Since these messages are exchanged between a pair of nodes (point-to-point communication), FIFO ordering is easily guaranteed by numbering them.

5.3 Race Conditions

A fundamental issue that arises in the context of distributed GC is that of races. A race situation occurs when, due to the different amount of time taken by related messages sent from different nodes to reach the same target node, one or more objects might become erroneously unreachable from the point of view of the GC algorithm. In this section, with the help of a few examples, we describe how such race problems are solved in our GC subsystem. Neither extra messages, nor causal communication support is necessary.

Consider Figure 3 at the moment when the inter-bunch reference O2→O3 has not been created yet. Now the system evolves as follows (see Figure 5):

1. The mutator in N2 reads the pointer to O3 from O1, and stores it into O2. This assignment creates an (inter-node) inter-bunch reference. Thus, stub ST2 is locally created, and a scion-message is sent to B3 on N3, in order to create the corresponding scion SC2.

2. A mutator in N1 acquires the write token of O1 and deletes the reference to O3.

3. The local BGC of B1 is executed at N1 which results in the creation of a new stub table that does not contain stub ST1.

4. Bunch B3 receives B1's new stub table from N1, and the scion cleaner finds out that scion SC1 is no longer reachable. Thus, this scion is deleted.

5. The local BGC of B3 is executed and O3 is (erroneously) collected.

6. The scion-message sent in the first step arrives at node N3 but it is too late.

This race problem is solved by ensuring that the scion-message sent from N2 is received by N3 and the corresponding scion SC2 created in B3, before

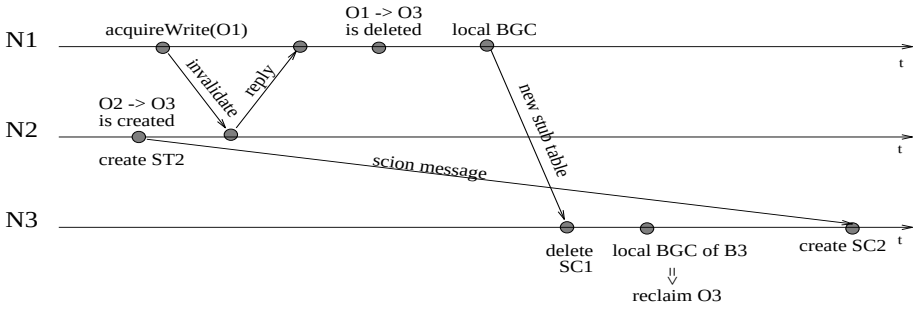

Figure 5: Time diagram showing a race problem.

the new B1's stub table. This can be done without any extra communication overhead by taking advantage of consistency protocol dedicated messages. As the acquisition of the write token of O1 by node N1 implies the invalidation of every copy of O1 (copy-set is in N1), the scion-message is piggy-backed in the reply (to the invalidation message) sent from N2 to N1. When the new B1's stub table is sent to N3, it will carry along the scion-message. Thus, the scion cleaner in N3 will first create scion SC2, and only then deletes scion SC1. Hence, the local BGC of B3 will not reclaim O3.

Another race situation in which O3 could be erroneously collected is the following (consider the same initial condition of the previous example and its first step already done):

1. The mutators on nodes N1 and N2 delete their references from the local root to O1.

2. One or more local BGCs are executed at nodes N1 and N2, collecting bunch B1. Because O1 is dead, the local BGC of B1 at N1 will create a stub table without stub ST1. This stub table is then propagated to N3 and results in the deletion of scion SC1 by the scion cleaner.

3. The local BGC of B3 is executed and O3 is (erroneously) collected.

4. The scion-message sent in the first step arrives at node N3 but it is too late.

In this case, as opposed to the previous one, there is no communication between nodes N1 and N2 on behalf of the consistency protocol. Thus, we can not piggy-back the scion-message as in the previous solution. To solve this race problem we must ensure that object O1 is kept alive until the scion-message is received by N3, and the corresponding scion SC2 created. Obviously, we maintain the requirement of generating a minimum number of extra messages. Hence, when a reference to O3 is stored into O2, to ensure that O1 is kept alive, it is conservatively created a special SSP on node N2 from B2 to B1. When the reply to the scion-message is received at N2, the special SSP is deleted. Clearly, mutators in N2 do not have to wait for the reply to the scion-message, i.e., they may continue to execute freely as if no (inter-node) inter-bunch reference has been created.

Note that, in the two examples presented above, a scion-message is sent to `N3` because bunch `B3` is not cached on `N2`. If `B3` was also cached on `N2`, the scion-messages would not be necessary because every inter-bunch SSP would be locally created (in `N2`).

A possible optimization concerning the accessibility propagation, consists of avoiding to send a scion-message per each new (inter-node) inter-bunch reference created. Thus, only the first scion-message must be sent. Such message informs the destination node that there are some inter-bunch references for which the corresponding scions may not have been created yet. Since such scions are only necessary for the purpose of the local BGC on the destination node, the *missing* scion-messages can be postponed until explicitly asked (before starting a local BGC).

6 Group Garbage Collector

The GGC has the goal of reclaiming inter-bunch cycles of garbage. The algorithm is basically the same used by the BGC, only that it operates on a group of bunches, rather than on a single bunch. The root of the GGC includes: mutators stacks, intra-bunch scions, entering `ownerPtrs`, and those inter-bunch scions identifying source bunches that are *not* members of the group being collected. The inter-bunch scions corresponding to SSPs that originate within the group that is being collected are not part of the root. Therefore, objects in the group, which are not reachable from any sources other than these SSPs, will be collected. In particular, objects that form an inter-bunch cycle, but are non-reachable from bunches outside the group or the mutator's stack, will be collected, because they are not artificially held over by SSPs from within the group.

For grouping bunches, the GGC uses a very simple heuristic: a group is constituted by bunches that are already mapped on the local node. We expect this heuristic to be effective on collecting cycles of garbage because bunches will be traveling through the network from node to node. Thus, this heuristic relies on applications locality.

This locality-based heuristic does not collect cycles of garbage that partially reside on disk, i.e., cycles with objects in bunches not currently mapped in memory. Collecting such a cycle involves input-output costs that need to be balanced against the expected gain. In addition, if an application does not move bunches around the nodes there is a possibility that some dead cycles may not ever be collected at all. We believe that some of these cycles can be removed by improving the heuristic. However, we intend to do that only after having experimented with the locality-based heuristic. If experimental results mandate it, we will explore more complex heuristics.

Grouping was first proposed by Lang[13]. However, our solution is much simpler because a group collection occurs at a single site. Therefore, we expect our solution to be more scalable.

7 Implementation

The BMX prototype is being implemented on a local network comprising DEC Alpha workstations running OSF/1.

Our implementation results from a straightforward transposition of the three sub-algorithms of the GC subsystem (BGC, scion cleaner, and GGC) to a loosely coupled network of Unix based workstations. Each application is supported by one or more processes (each one possibly multi-threaded) executing at several nodes of the network. A local BGC is supported by a thread inside a process accessing the bunch being collected. The scion cleaner and the GGC are each one supported by a privileged process that can access any bunch, on each node of the network.

Bunches are mapped in shared memory. There is no more than one cached copy of a bunch per node. Every bunch can be freely accessed by any process that does not violate the bunch protection attributes.

On each node, a single *BMX-server* process provides basic system services such as allocating non-overlapping segments. Executing this server in a separate process protects it from misbehaving or malfunctioning applications. A library called *BMX-client* is linked with each application process. This library acts as a proxy that interacts with the system internals (the BMX-server in particular). Some sites on the network manage the secondary storage for persistent bunches. Each one of these nodes executes an Object Repository (OBR) process that performs such managing task.

For persistence support we use the RVM (Recoverable Virtual Memory) subsystem [18]. There is a log where segment modifications are registered (a segment is implemented as a file). Segment modifications are done in *flush mode* in order to guarantee that the log reflects such modifications accurately. However, at this stage, the segment on disk does not contain yet the changes done since it was mapped in memory. Only after the *truncation* of the log is the segment on disk guaranteed to reflect the modifications done.

In BMX, the node where a segment is first mapped (i.e., the one that initiated the current set of modifications) behaves as a coordinator for other nodes where the segment is also cached. For example, suppose that segment `S1` is first mapped on node `N1` (a log is automatically created). In order to map `S1`, node `N1` asked the segment to the OBR responsible for managing the secondary storage where `S1` is stored. This server ensures that, later when another node (say, `N2`) tries to map `S1`, it will receive a reply from OBR with the identification of `N1`. From this point on, node `N2` becomes a subordinate of `N1`, i.e., `S1` modifications done on node `N2` are stored in a log kept on `N2`, and when the truncation is made, the log that has to be considered is the union of each local log on every node where the segment has been cached (`N1` and `N2`).

8 Related Work

To our knowledge little work has been done on garbage collecting objects in a large DSM. On the contrary, a large amount of work has been done in the domain of GC of objects either in multiprocessors [1, 4], or in RPC-based distributed systems (see Plainfossé[17] for a survey).

For the purpose of GC, the fundamental difference between our system and

a multiprocessor, is that of scale and synchronization overhead. This difference implies that, if we apply a GC algorithm designed for multiprocessors to our case (for instance, Appel[1]), 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 strongly consistent objects. In fact, communication and synchronization overhead arises because of the necessity of providing strongly consistent objects and the interference with applications' consistency needs.

Furthermore, our GC problem is more difficult than in distributed RPC-based systems (e.g., [9]); in such systems there is only one copy of each object, which is remotely invoked by those nodes that are sharing it. Thus, objects are modified by explicit messages that can be easily intercepted for the purpose of reference control by the GC algorithm. Additionally, since there is no memory sharing, there is no such problem as interference with the consistency protocol.

A GC system for DSM that we are aware of is due to Le Sergent[14]. His garbage collector was first developed for a multiprocessor machine, then extended to a loosely coupled network with DSM. The algorithm is of the copying type, is incremental, and uses virtual memory protection traps to synchronize the mutator and the garbage collector, as suggested by Appel[1]. The heap is seen as a contiguous set of pages, and the shared memory is strongly consistent. With respect to our design, the main differences are the following: (i) we divide the virtual heap into bunches, in order to deal with the huge amount of objects, and allow parallel collection of bunches (we flip each bunch separately, not the whole memory as in Le Sergent's design), (ii) our GC subsystem is composed of three sub-algorithms with complementary functionality, (iii) we use a write-barrier rather than virtual memory protection for synchronization purposes, and (iv) our shared memory is weakly consistent. All these differences contribute to make our design much more scalable and efficient.

Another collector for DSM was developed by Kordale[11]. His design is very complex and relies on a large amount of auxiliary information in the form of tables. These tables are used basically to control inter-node references and the algorithm is a variation of mark-and-sweep.

Casper [10] is a system that supports a single persistent address space shared by several clients in a network. Objects are cached by clients that are served by a central server where persistent objects are stored. Each client creates its objects within a separate area of the persistent address space, called local heap. A local heap can be garbage collected independently from the rest of the heap by maintaining the information of which objects inside a local heap (not yet persistent) are referenced from the persistent store. This pointer tracking is done just like in generational based garbage collectors [20], i.e., with a write barrier. With respect to our GC design, we may see the local heap as a bunch. Both systems use the same kind of reference tracking; however, in BMX every inter-bunch reference is tracked, i.e., not only the ones that point from the persistent store to objects inside a mapped bunch, but also the references between mapped bunches not yet stored on disk.

Our GC design borrows some ideas from current distributed GC techniques: (i) stubs and scions describing inter-bunch references are based on the SSP Chain mechanism developed by Shapiro[19]; however, our stubs and scions neither introduce any indirection, nor perform marshaling/un-marshaling, (ii) we collect inter-bunch cycles of garbage similarly to Lang[13] but his groups

are distributed, which is much more complex and less scalable.

Our original contributions are: (i) independent GC (i.e. asynchronously, with no synchronization) of different groups of objects (done by the BGC), (ii) the set of local group collections done by the GGC approximates a global collection without the synchronization costs of the latter, (iii) use of locality-based heuristics for group collection by the GGC, and (iv) taking advantage of consistency protocol messages to reduce the number and ensure the causality of messages used for the purpose of distributed GC (done by the scion cleaner).

9 Conclusion

We have presented a GC design for persistent objects accessed via DSM, in a loosely coupled network. Objects are allocated in bunches supported by recoverable non-overlapping segments in a single 64-bit address space spanning the whole network, including secondary storage. Objects are kept weakly consistent by the entry consistency protocol.

The GC subsystem combines three sub-algorithms with complementary functionality that run without synchronization constraints: GC of a bunch (BGC), accessibility propagation (scion cleaner), and reclamation of inter-bunch cycles of garbage (GGC). This solution is scalable, allows parallel collection of different bunches, collects cyclic garbage, and does not need special communication support such as atomicity or causality. Furthermore, messages for the purpose of distributed GC are reduced to a minimum, and are exchanged in the background. Thus, applications are never disrupted.

Acknowledgments: We thank the anonymous referees for their comments on an initial version of this paper.

References

[1] Andrew W. Appel, John R. Ellis, and Kai 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] Brian N. Bershad and Matthew J. Zekauskas. The Midway distributed shared memory system. In *Proceedings of the COMPCON'93 Conference*, pages 528–537, February 1993.

[3] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE'87—Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, pages 117–187, Eindhoven (the Netherlands), June 1987. Springer-Verlag.

[4] Hans-J. Boehm, Alan J. Demers, and Scott 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.

[5] 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.

[6] Paulo Ferreira and Marc Shapiro. Distribution and persistence in multiple and heterogeneous address spaces. In *Proc. of the International Workshop on Object Orientation in Operating Systems*, Ashville, North Carolina, (USA), December 1993. IEEE Comp. Society Press.

[7] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, California (USA), November 1994. ACM.

[8] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanoviè. A comparative performance evaluation of write barrier implementations. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*, volume 27 of *SIGPLAN Notices*, pages 92–109, Vancouver (Canada), October 1992. ACM Press.

[9] Niels C. Juul. *Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System Emerald.* PhD thesis, Dept. of Computer Science, Univ. of Copenhagen, Denmark, February 1993.

[10] 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.

[11] 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), December 1993. IEEE Comp. Society Press.

[12] David Kotz and Preston Crow. The expected lifetime of single-address-space operating systems. In *Proceedings of SIGMETRICS'94*, Nashville, Tennessee, (USA), May 1994. ACM Press.

[13] 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.

[14] 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.

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

[16] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 217–226, Albuquerque, N. Mexico, June 1993. ACM-SIGPLAN.

[17] David Plainfossé. *Distributed Garbage Collection and Reference Management in the Soul Object Support System.* PhD thesis, Université Paris-6, Pierre-et-Marie-Curie, Paris (France), June 1994. Available from INRIA as TU-281, ISBN-2-7261-0849-0.

[18] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 146–160, Asheville, NC (USA), December 1993.

[19] 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.

[20] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, 19(5):157–167, 1984.

[21] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, pages 87–98, Nice, France, June 1990.