Garbage Collection and DSM Consistency

Paulo Ferreira, Marc Shapiro

Garbage Collection and DSM Consistency*†

Paulo Ferreira‡ *and* Marc Shapiro
INRIA - Projet SOR

Abstract

This paper presents the design of a copying garbage collector for persistent distributed shared objects in a loosely coupled network with weakly consistent distributed shared memory (DSM).

The main goal of the design for this garbage collector is to minimize the communication overhead due to collection between nodes of the system, and to avoid any interference with the DSM memory consistency protocol.

Our design is based on the observation that, in a weakly consistent DSM system, the memory consistency requirements of the garbage collector are less strict than those of the applications. Thus, the garbage collector reclaims objects independently of other copies of the same objects without interfering with the DSM consistency protocol. Furthermore, our design does not require reliable communication support, and is capable of reclaiming distributed cycles of dead objects.

1 Introduction

Garbage collection (GC) is a fundamental component for supporting persistent objects in distributed systems. The importance of garbage collection in such systems is twofold: first, the object graphs of applications, like financial or design databases, cooperative work and exploratory tools similar to the World-Wide-Web, are very intricate, which makes manual storage management increasingly difficult and error-prone, often resulting in dangling pointers and storage leaks. Second, garbage collection is necessary to support the property of persistence by reachability [2]; this property states that only objects reachable from the persistent root should be persistent. In other words, objects that are no longer reachable from the persistent root should not be stored on disk. Even in a persistent 64-bit address space, there is a need for memory reorganization and address recycling, otherwise the address space gets severely fragmented and secondary storage fills with garbage.

In addition, distributed shared memory systems have become popular because they support a simpler programming model for distributed applications than RPC-based systems [3]. Furthermore, weak consistency protocols seem to offer the best performance when compared to sequential consistency [4].

For these reasons, we have designed and implemented a platform, called BMX [8], that provides persistent weakly consistent shared distributed virtual memory and copying garbage collection. We chose a copying garbage collector because it can improve an application's locality [13], it contributes to reduce memory fragmentation, and it provides sufficient support to reclaim cycles of unreachable objects.

The paper focuses on the issue of how the garbage collector copies shared objects without interfering with the DSM consistency protocol. This is an important systems problem, since interference between the garbage collector and the consistency protocol could potentially nullify the advantages of using a weakly consistent DSM system. For example, when updating a reference inside an object, to reflect the new location of a live descendent that has already been copied, the garbage collector should not require exclusive write-access to modify the object. If exclusive write-access was needed, read-access to all other replicas of the object would have to be invalidated, therefore nullifying the advantage of using weak consistent DSM. Current distributed GC algorithms do not handle this problem; they implicitly assume the existence of a single object copy, which is not the case in a DSM system.

*This article appears in the Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI), November 14-17, 1994, Monterey, California, USA.

†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). 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.

Besides the problem of avoiding interference with the DSM consistency protocol, the other two problems that need to be addressed in the design of a garbage collector for distributed shared objects are: (i) collection of acyclic distributed dead objects, and (ii) collection of distributed cycles of garbage. Due to space limitations, we only present a general overview of our solution to these two issues (see Ferreira[9] for more detail). As mentioned above, this paper focuses on the techniques used by the garbage collector to avoid interference between the collector and the DSM consistency protocol.

Our GC algorithm is orthogonal to DSM consistency, that is, it tolerates inconsistent objects, therefore it is generally applicable to other consistency protocols. Furthermore, our design is application- and language-independent. However, the compiler must be instrumented in order to interface with the algorithm support layer.

The paper is organized as follows. The next section presents the basic aspects of the BMX platform. Section 3 gives a global overview of the GC design. The collection of shared objects and its relation to the DSM consistency protocol is described in Sections 4 and 5. Sections 6 and 7 briefly describe acyclic and cyclic distributed garbage collection. Section 8 gives an overview of the implementation, and finally, the paper ends with related work and some conclusions.

2 BMX Overview

This section presents the aspects of the BMX platform that are relevant to the GC design described in this paper.

2.1 Objects, Segments, and Bunches

BMX offers a 64-bit single address space spanning all the nodes of a network, including secondary storage. The object, which consists of a contiguous sequence of bytes, is the basic unit of identification and invocation. An object is represented by its address; object references are therefore ordinary pointers. Each object has an header that precedes the object's data, which includes system information such as the object's size. We assume that objects are passive and generally small, that is, the size of most objects is much smaller than a virtual memory page.

Objects can become persistent by reachability, that is, they are persistent if reachable from the persistent root. Once mapped in main memory, such objects are shared through a DSM mechanism, just like any other non-persistent object.

For clustering purposes, objects are allocated within segments. A *segment* is a set of contiguous virtual memory pages with a constant size. BMX ensures that segments have non-overlapping addresses.

Segments are logically grouped into *bunches* because a single segment is not flexible enough to support solutions for situations like segment overflow. Each bunch has an associated owner, and protection attributes like the usual Unix read, write, and execute permissions.

BMX supports recovery for operations on bunches. Recovery is based on the recoverable virtual memory techniques proposed by Satyanarayanan et al. [19]. Therefore, after a bunch is mapped into memory, every modification performed on the bunch's range of addresses has an associated log entry and can be recovered after a system failure.

In summary, the user program, called the mutator in the GC literature [7], operates on a single, shared, persistent, possibly large graph of objects allocated from a number of bunches. These bunches can be simultaneously replicated on several nodes in the system and are kept weakly consistent by the DSM system described below.

2.2 DSM Support

The BMX system supports weakly consistent distributed shared memory based on the entry consistency protocol [4]. Applications do not send explicit messages. They only use the distributed shared memory paradigm for communication.

The entry consistency protocol provides the traditional model of multiple readers and a single writer. Thus, there can either be several read tokens, or one exclusive write token associated with each object. Nodes holding a read token are ensured to be reading a consistent version of the corresponding object. The possession of the write token means that there is no other consistent copy of the object at any other node of the network. The entry consistency protocol therefore guarantees that an object is consistent, with respect to previous operations on the object, as long as a node holds the corresponding read or write token. Otherwise the observed state of the object is undefined.

Every object has an *owner*, which is either the node currently holding the object's write token, or the node 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 node already holding a read token. A token is obtained by performing a read or write token *acquire* operation and is freed by the corresponding *release*.

Tokens are managed with an algorithm similar to Li's dynamic distributed manager with distributed copy sets [16]. Thus, the copy-set of an object (list of nodes with a read token) is not centralized by its owner; rather, it is distributed among the owner and

those nodes that have transitively granted a read token to other nodes.

In addition, there is a forwarding pointer mechanism indicating which node is the current object's owner. We call such a forwarding pointer an `ownerPtr`. Therefore, for each bunch there is a set of entering `ownerPtrs` that originate at nodes with non-owned replicas for the objects in this bunch, and a set of exiting `ownerPtrs` pointing to the owner node of each of the objects from this bunch.

3 Garbage Collection Design

The premise of our garbage collection algorithm is that an application's object graph can be enormous and widely distributed among the nodes of the system. It would therefore not be feasible to collect all objects of an application at the same time. Our algorithm collects each bunch of objects independently of any other bunch.

To be able to collect a bunch independently, it must be isolated from all other bunches. In other words, every copy of a bunch has to contain enough local information to independently make all reachability decisions for its objects, that is, without requesting information from any other bunch, nor from other copies of the same bunch. For this purpose each cached copy of a bunch holds two tables (in addition to the tables of entering and exiting `ownerPtrs`): the stub table and the scion table. The *stub table* contains information about outgoing links, that is, which objects referenced from within the bunch are allocated in some other bunch and the bunches to which they correspond. The *scion table* contains information about incoming references, that is, which local objects are referenced from other bunches and from where these references originate. Thus, for each stub there is a corresponding scion, these two entities form a stub-scion pair (SSP).

The bunch isolation provided by stubs and scions affects how inter-bunch reachability is propagated, and therefore how inter-bunch garbage is collected. For this reason, our GC design is based on three subalgorithms that perform complementary tasks: the first component, called the *bunch garbage collector* (BGC), executes the collection on a local replica of a bunch, independently from the collection of any other bunch and other replicas of the same bunch; the second component, called the *scion cleaner*, uses information generated by the bunch garbage collectors of other bunches (stub tables and lists of outgoing `ownerPtrs`) to recognize which objects are no longer reachable from remote bunches or remote copies of the same bunch; and finally, the last component is called the *group garbage collector* (GGC), which is in charge of reclaiming inter-bunch cycles of garbage. Together, these subalgorithms support an integral GC solution for a DSM system providing a high degree of scalability and parallelism.

This paper focuses on the garbage collection of a replicated bunch and how the BGC interacts with DSM consistency. However, for the sake of completeness, we will include an overview of both the scion cleaner and the group garbage collector (Sections 6 and 7).

3.1 Stubs and Scions

Figure 1 illustrates the use of stubs and scions. These stub-scion pairs are simpler than the ones used in RPC-based distributed systems [20], because: (i) they are not used for indirections, that is, they are just auxiliary data structures that describe relevant references, and (ii) stubs/scions do not perform any kind of marshaling/un-marshaling.

There are two kinds of slightly different SSPs: an *inter-bunch* SSP describes references that cross bunch boundaries, while an *intra-bunch* SSP records relevant dependencies between copies of the same bunch.

Inter-bunch SSPs have the same direction of the cross-bunch reference they represent (for example, `O3`→`O5` in Figure 1). When an object becomes cached on multiple nodes, the inter-bunch stubs that represent the object's links to objects in other bunches do not have to be replicated. This is not problematic because a single SSP is enough to keep the target object alive in the whole system, as will be described further on. Figure 1 illustrates this situation: in spite of the fact that `O3` is cached on `N1` and `N2`, there is only one inter-bunch stub due to `O3`→`O5` that is kept at `N2`.

Intra-bunch SSPs have the opposite direction of the corresponding `ownerPtr`, and are used 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. Intra-bunch SSPs are necessary because **inter**-bunch SSPs do not move with the ownership of an object, instead the **intra**-bunch SSP serves as a forwarding link. For example, in spite of being unreachable by the mutator at `N2`, object `O3` must be kept alive at this node because: (i) there is an inter-bunch reference in `O3` for which the corresponding stub is allocated at `N2`, and (ii) a copy of `O3` is reachable by a mutator at `N1`. Object `O3` can be collected in `N2` only after becoming unreachable in `N1`.

3.2 Creation of Stub-Scion Pairs

An inter-bunch SSP is automatically constructed immediately after detecting the creation of the corresponding inter-bunch reference. This detection is done with a write-barrier [10] associated with every write performed by an application.

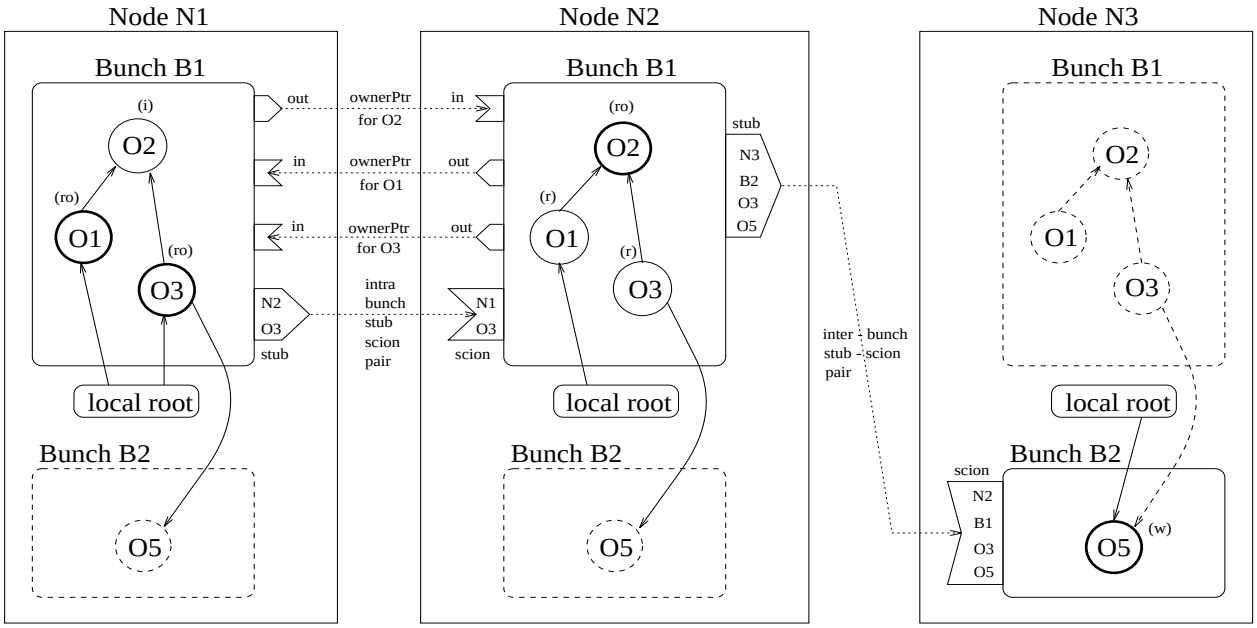


Figure 1: Bunch `B1` is mapped on nodes `N1` and `N2`, and bunch `B2` is mapped only on `N3`; mapped bunches are represented with a solid line, unmapped bunches are represented with a dashed line. Stub and scion tables contain inter-bunch and intra-bunch SSPs. For each object, the state of its token 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 (thicker objects); `i` is used for inconsistent copies. The `local root` includes mutator stacks. No GC has taken place on any node.

When an inter-bunch reference is created, either both the source and target bunches are already mapped on the local node, or only the target bunch has not yet been locally mapped. In the first case, the corresponding stub and scion get created locally. The second case requires that a message be sent to a node where the target bunch is mapped. This message is called a *scion-message* and is used to inform the target bunch about the necessity of creating the scion corresponding to the the new cross-node inter-bunch reference.

Figure 1 shows a cross-node inter-bunch SSP required by the link `O3`→`O5` from `N2` to `N3`. Node `N3` gets a scion-message from `N2` when the inter-bunch reference `O3`→`O5` is created and creates the matching scion for the stub on `N2`.

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. In other words, the old owner created a link to an object in a different bunch and the information required by the garbage collector for this purpose, that is the inter-bunch stub, needs to be preserved after the object's ownership is transferred. The intra-bunch SSP takes care of creating a forwarding link between the new owner and the inter-bunch stub at the old owner. Therefore, in the example illustrated by Figure 1, when `O3`'s write token goes from `N2`, where the inter-bunch reference was created, to `N1`, the corresponding intra-bunch SSP from `N1` to `N2` is created.

We decided to use intra-bunch SSPs, instead of replicating inter-bunch SSPs, in order to reduce the number of scion messages and the amount of memory consumed for GC purposes. In fact, if inter-bunch SSPs were replicated, each time object ownership changes, a new inter-bunch SSP would have to be created, which would imply sending the corresponding scion-message. By using intra-bunch SSPs, no extra messages are needed, because the information is piggy-backed onto consistency protocol messages. In addition, an inter-bunch SSP occupies more memory than an intra-bunch SSP.

4 Bunch Garbage Collection

This section describes the garbage collection of a bunch with multiple, possibly inconsistent, cached copies on several nodes. We first present the main aspects of the algorithm; then, we describe the algorithm in more detail, focusing on how live objects are copied and scanned, how reachability information is regenerated by the BGC, how references are updated and how the from-space is reused. Each of these issues is discussed in light of what is needed for the collector not to interfere with the DSM consistency protocol.

4.1 Outline

The BGC is based on the algorithm by O'Toole et. al [17] for three main reasons: (i) the time to *flip*[1] is very small and therefore not disruptive to applications, (ii) portability (no virtual memory manipulations), and (iii) objects are non-destructively copied (suitable for recovery purposes). However, any other copying algorithm could be used.

When a bunch has several copies on different nodes, a separate local BGC operates on each copy. The local collection of a bunch proceeds independently of the collection of other bunches, and of the collection of copies of the same bunch at other nodes. The roots of the BGC are located in the mutators stack, intra-bunch and inter-bunch scions, and list of entering `ownerPtrs`.

Each time the local BGC is executed, it reconstructs a new version of the bunch's stub table, and a new set of exiting `ownerPtrs`. The new stubs and exiting `ownerPtrs` will later be sent to the scion cleaner of all nodes that either have cached copies of the same bunch or contain the scions corresponding to the stubs of both old and reconstructed stub tables. On those nodes, the scion cleaner discovers and removes all local intra-bunch and inter-bunch scions that are no longer reachable from any stub, and all incoming `ownerPtrs` for local copies of objects that are no longer live remotely.

To avoid synchronizing all nodes in an object's copy-set, a local BGC only copies the objects that are locally owned. Non-locally owned objects are simply scanned. This asynchronous collection of different copies of the same bunch can cause inconsistent views of object's addresses across cached replicas. However, as explained in the following sections, the asynchronous collection of object replicas is not a problem, because addresses on other nodes can be updated when these nodes synchronize on behalf of application's DSM consistency needs. Thus, applications designed for weakly consistent DSM systems will work correctly.

4.2 Copying/Scanning Live Objects

A local BGC copies live objects from the from-space segment to the to-space segment, independently of other BGCs of the same bunch. The first challenge of such an algorithm is to avoid that two BGCs, simultaneously executing on different replicas of a bunch, move the same live object to different memory locations. One obvious solution to this problem would be to acquire the write token of every live object before copying it. However, this solution is undesirable, since it would trigger memory consistency actions that could disrupt the application's working-set. For example, each readable copy would be invalidated.

Our solution avoids this drawback by only copying those objects that are locally owned. Thus, if the node holds the write token or was the last one to hold it, the corresponding object is copied to to-space and scanned, and a forwarding pointer is written into the object's header, which is left in from-space. This header modification is strictly local and does not imply acquiring the object's write token because, at this time, only the local node has to be aware of the object's new location. As explained later in this section, other nodes will eventually be informed of the object's new location.

Other live objects, that is, those not locally owned, are simply scanned. An important aspect of this design is that these objects can be scanned, even though their copy might not be consistent with the owner's. In fact, an inconsistent copy of the object is sufficient, because scanning an old version results in making a more conservative decision about the referenced objects reachability, ensuring that they will not be erroneously collected if not dead.

The header with the forwarding pointer, left in place of an object copied to to-space, is deleted only when every reference to that object has been updated with the new address. These references can be local or remote, as shown in Figure 2: object `O2` has been copied to the to-space segment by the BGC in `N2`; thus, pointers inside `O1` and `O3` must be updated accordingly, on both nodes.

After updating the local references to a copied object, and before performing the same operation on remote references to the object, the copied object will be at different addresses on different nodes. However, this is not a problem. The data inside the copied object is kept consistent from the applications point of view, as guaranteed by the consistency protocol. Thus, remote references can be updated lazily. As an example, consider Figure 2: after updating the references to `O2` in `N2`, and before performing the same operation in `N1`, object `O2` will exist in different addresses on these two nodes. However, mutators in both nodes continue to work correctly.

Applications perform correctly because they are implemented for a weakly consistent DSM. Remember that applications do not send explicit messages that could, for example, reference objects with a new address not known to the receiver. Furthermore, to account for the existence of forwarding pointers, a special operation is provided to perform pointer comparison. The conditions that need to be upheld at DSM synchronization points (read or write token acquires) to ensure the proper execution of applications are described in Section 5.

[1] Time during which an application is stopped due to garbage collection.

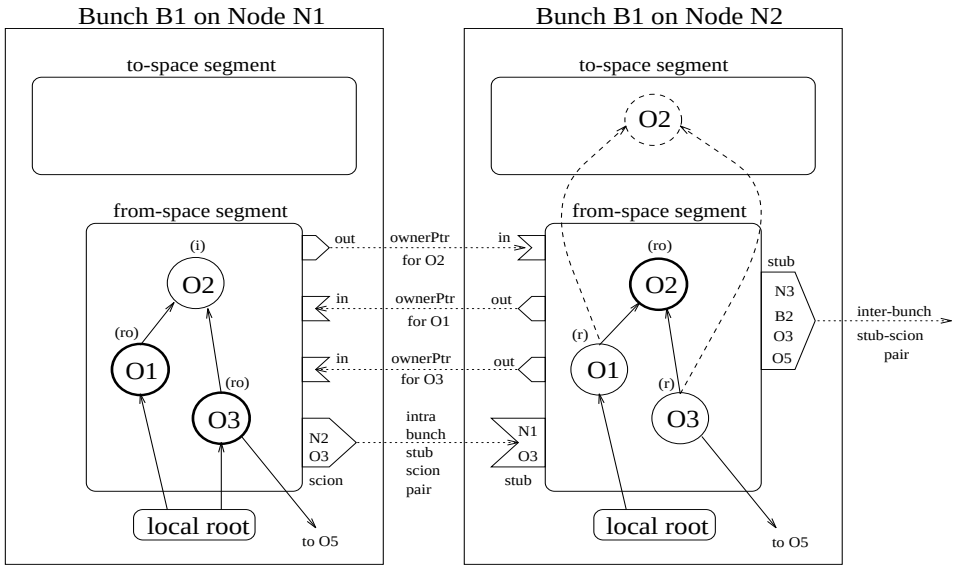


Figure 2: Zooming into Figure 1, we show more detail of bunch `B1` on nodes `N1` and `N2`. The BGC on `N2` only copies locally-owned live objects, that is, `O2`. The update of pointers to `O2` is represented by dashed arrows. Node `N1` has not yet been informed of the `O2`'s new address, and the local BGC of `B1` has not been executed.

4.3 Creating new Stub and Outgoing `ownerPtr` Tables

As mentioned earlier, a local execution of the BGC scans all objects that are reachable from the mutators stacks, scions, and entering `ownerPtrs`, and creates a new table of inter-bunch and intra-bunch stubs, as well as a new list of outgoing `ownerPtrs`. The new stub table and list of outgoing `ownerPtrs` represents all objects in other bunches or in other nodes that are accessible from the local copy of the bunch. An object that has been locally garbage collected will neither add a stub nor an outgoing `ownerPtr` to the new tables. An inter-bunch stub will not be added to the new stub table if the corresponding local object no longer includes the inter-bunch reference associated with the stub. When the BGC scans a live object containing an inter-bunch reference, three actions may be taken (see Figure 1):

- if the inter-bunch reference has been created at the local node (as is the case of O3 at N2), than the corresponding inter-bunch stub is added to the new stub table,

- if the inter-bunch reference has not been created locally, but the scanned object is locally owned (as is the case of O3 at N1), than the corresponding intra-bunch stub is added to the new stub list, and

- if neither the inter-bunch reference has been created locally, nor the local node is the object's owner, than no stub is added to the new stub list.

Furthermore, for objects that can only be accessed via an intra-bunch scion the exiting `ownerPtr` will be omitted from the new outgoing list (see Section 6.2).

The new intra-bunch stubs and the new set of exiting `ownerPtrs` constructed by the local BGC are eventually sent to other nodes where the corresponding objects are mapped.

4.4 Updating References

The next challenge in designing our garbage collector is how to update all local and remote references to an object that has been copied to to-space, without interfering with the DSM consistency protocol and without incurring in high communication overhead between nodes.

At first, it may seem impossible to update a reference without interfering with the consistency protocol, because it implies updating the object that contains the reference. Normally to update an object it is necessary to acquire the write token for the object, which would in turn make outstanding readable copies stale. However, the same way it was possible to copy an object without exclusive-access, the object can also be modified without acquiring the corresponding write token, because the modification is visible only to the

local node, and does not affect the applications behavior on other nodes, which might be accessing another copy of the object. Hence, reference updating can be done without interfering with the consistency protocol. For example, in Figure 2, updating O1 in N2 with O2's new address can be done without acquiring O1's write token. In spite of the difference between O1 on both nodes, mutators at N1 and at N2 still *see* a copy of O1 that is consistent with the application's requirements.

For the purpose of updating remote references to locally copied objects we want to avoid sending an explicit message from the node where an object was copied to the node holding the remote reference. Furthermore, we want to avoid blocking an application while such an update is taking place.

Because remote references can be updated lazily, an object's new address can be communicated to other nodes by piggy-backing such information onto messages due to the consistency protocol, which are performed on behalf of applications. Thus, no extra message is used. For instance (see Figure 2), O2's new address can be sent from N2 to N1 in a message due to the consistency protocol exchanged between these two nodes. After N1 receives O2's new address, O2 is copied to the indicated address, and all the local references are updated accordingly without requiring any token.

Reference updating does not have to be done immediately after receiving the corresponding message with an object's new location. In fact, it can be postponed until a bunch garbage collection takes place at the local node. Therefore, there is a tradeoff on how consistent the addresses are going to be and the overhead of immediately executing the updates at the remote nodes.

It is important to note, that if there is no communication between nodes on behalf of applications, then there is no need for updating references unless the from-space needs to be reused. In this case, as explained in the next section, explicit messages must be used.

4.5 Reusing From-Space

Since during bunch garbage collection only locally owned objects are copied to to-space, it may happen that some live objects and forwarding pointers remain in the from-space after the local BGC has completed. Thus, immediately after a local bunch garbage collection, the from-space segment might not be fully reused nor freed. This is the case of B1's from-space segment after the BGC on N2 has finished (see Figure 2): objects O1 and O3 are live and remain in the from-space segment. These objects will eventually be copied by B1's local BGC running on node N1. Before reusing or freeing a from-space segment, it is therefore necessary to ensure that no live objects remain in the segment and that the forwarding pointers are no longer necessary.

It is worthy to note that a from-space segment will be reused for allocating new objects only after the corresponding to-space segment becomes full. Until then, non-owned objects remaining in the from-space segment may either die or be copied by their owners. Furthermore, all the space that was occupied by dead objects could be completely reused.

Nevertheless, suppose that we want to fully reuse or discard a from-space segment. In that case, we must ensure that it contains neither forwarding pointers to already copied objects nor non-locally owned live objects. Both conditions are guaranteed by informing all other nodes affected by the address changes in this segment about these changes, and by asking the owner nodes to copy those live objects still allocated in the from-space segment. Once the local node receives the replies to the above messages, the from-space segment can be fully reused or freed.

Since the address-change messages are exchanged in the background, applications can make progress without having to process them immediately. From the point of view of the application, processing these messages is part of the garbage collection overhead, and is no more disruptive than garbage collection itself.

For example, consider Figure 2: node N2 informs N1 of O2's new address, asks the BGC in N1 to copy its locally owned live objects (O1 and O3), and updates its local references to O1's and O3's new location. After that, B1's from-space segment can be freed or reused in its entirety.

Note that the list of nodes where an object's reference must be updated is already kept in the object's owner node for the purpose of the DSM mechanism: nodes from where the set of entering `ownerPtrs` originate. This implies that there is no extra memory overhead due to the GC.

5 DSM Acquires and the BGC

As explained before, since remote object references are updated lazily, different replicas of the same object can be located at different addresses on different nodes after the execution of a BGC. We also mentioned that mutators on these nodes will operate correctly, because the data held by the objects is consistent with respect to the guarantees made by the DSM consistency protocol. However, after synchronization points in the application, that is, read or write token acquires, the system has to ensure that the synchronizing nodes reference objects with the same addresses. In this section we describe three invariants that the garbage collector has to ensure for this purpose.

1. *The acquisition of a read or write token for an object can complete only after ensuring that the object's address and all references inside it are valid at the acquiring node.*

   This invariant avoids erroneous situations such as the following: suppose that object O1 points to O2, both objects are allocated from bunch B and nodes N1 and N2 hold replicas of the objects.

   (a) Node N1 is the owner of both objects.

   (b) Bunch B is collected at N1, therefore O1's copy at N1 will point to the new location of object O2.

   (c) Node N2 issues a read or write token acquire request for O1 to N1.

   (d) If invariant 1 was not maintained, the new copy of O1 at N2 could point to the new address of O2, while O2 was still at the old address.

   This first invariant prevents the situation described above, by ensuring that N2 is aware of O2's new location before the acquire request completes. The invariant is easily maintained by piggy-backing information with the new locations of the object being acquired and of every object directly referenced from it, onto the reply to the acquire message.

2. *A node that receives a message with the new location for an object forwards this information to all the nodes that are in the local copy-set for the object, that is, to which it has granted a read token.*

   This second invariant is necessary because a distributed copy-set algorithm is used for token management. Remember that a read token for an object can be obtained from a node already holding a read token. Therefore, a node with a read token for an object, that is not the object's owner, is responsible for forwarding messages containing an object's new location. The mechanism, to forward new location information to all nodes with a read token, is similar to the one used to invalidate all read copies of an object when some node acquires the object's write token.

3. *The acquisition of a write token for an object completes only after all necessary intra-bunch SSPs have been created.*

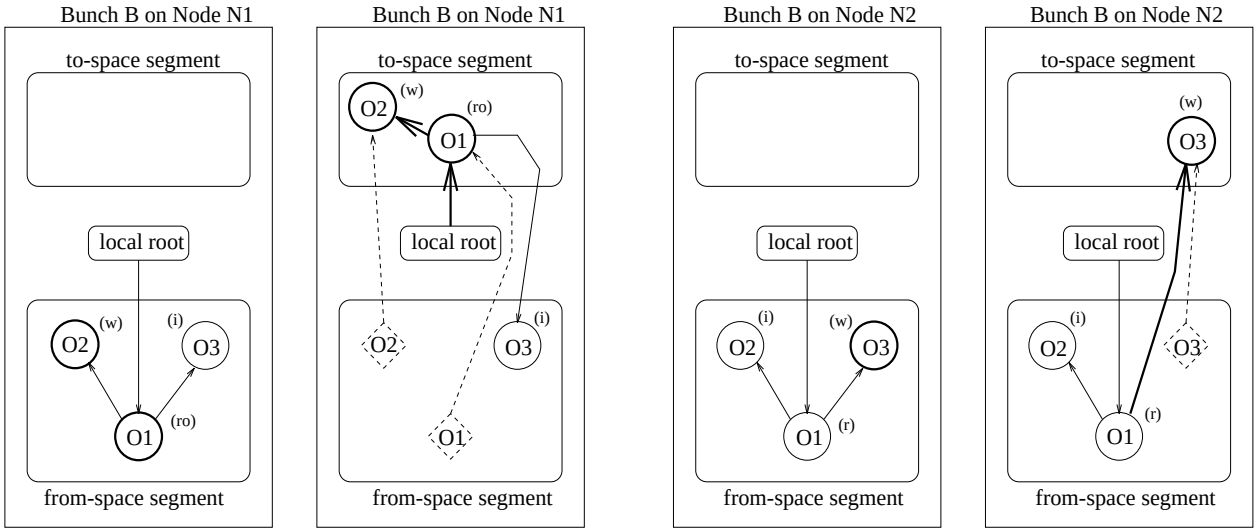
   This invariant ensures that the appropriate intra-bunch SSP is created between the new owner of the object and an old owner, when ownership is transferred from a node holding either inter-bunch stubs or an intra-bunch stub for the object. Remember that the intra-bunch SSPs serve as forwarding pointers from the new owner to inter-bunch stubs located at previous owners of the object.

The three invariants can be maintained without incurring in extra communication overhead, by taking advantage of the messages transferred by the DSM system on behalf of the applications at synchronization points, that is, replies to acquire messages. For a DSM system that does not require applications to synchronize on accesses to shared objects, the invariants can be guaranteed by ensuring that whenever a node faults on the access to an object O, the node that supplies O also sends all the necessary location updates and intra-bunch SSP information.

Now, let us present a detailed example of the specific operations that are executed to satisfy the invariants presented above before a write token acquire is completed (see Figure 3). Suppose that a write token for object O1 is requested by node N2 from node N1:

- If neither O1 nor any of the objects referenced directly by O1 have been copied to to-space at either node (situation represented by case (a) and (c)), no special operation has to be performed.
- On the other hand, if either O1 or an object pointed at by O1 have been copied to to-space at N1 (situation represented by case (b) and (c)), their new locations are piggy-backed in the message granting the token to N2 and are processed at N2 before the application returns from the token request.

  In addition, the new locations of O1 and of the objects directly referenced by O1 are forwarded by N2 to all nodes to which it had granted a read token for those objects. Nodes to which N2 granted a read token are listed in the corresponding object's copy-set.
- If any of the objects pointed at by O1 gets copied to to-space at N2 prior to the write token acquire operation (situation represented by case (d)), when N2 receives the valid copy of O1 from N1 with the token, it updates all references in O1 pointing to forwarding pointers in from-space, to point to the new addresses in to-space directly.
- If any inter-bunch stubs exist for O1 at N1 (not represented in Figure 3), before the write token acquire can complete, an intra-bunch SSP for O1 has to be created, pointing from N2 to N1. N1 creates the intra-bunch scion before it replies with the token-grant message, and piggy-backs a request for N2 to create the appropriate intra-bunch stub upon reception of the message.



(a) before B's BGC at N1. (b) after B's BGC at N1. (c) before B's BGC at N2. (d) after B's BGC at N2.

Figure 3: Bunch `B` is mapped on nodes `N1` and `N2` (before and after the execution of the BGC). Thicker arrows represent references locally updated by the BGC. Dashed lines are used to represent forwarding pointers left in place of copied objects.

This set of operations, executed when a write token is acquired, ensures that, after synchronizing, a pair of nodes sees consistent addresses for objects in the DSM system, by guaranteeing the three invariants described earlier.

6 Scion Cleaner

This section presents an overview of the scion cleaner and then describes in detail how intra-bunch SSPs are deleted when no longer necessary.

6.1 Overview

The scion cleaner locally processes the reachability information (stubs and outgoing `ownerPtrs`) that has been constructed by the execution of a BGC on other bunches. This process is identical for information received from a bunch with or without a local replica. There is a scion cleaner service per node that operates on all bunches of that node. The cleaner recognizes and removes all scions that are no longer reachable from any stub and all entering `ownerPtrs` that correspond to remote replicas that have been garbage collected. In essence, the scion cleaner, updates the roots for the next execution of the local BGC.

The main advantage of sending messages with tables containing all the reachability information, over sending increment/decrement messages [5], is that the former are idempotent. In case of message loss they can be resent without the need for a reliable communication protocol. However, messages with reachability information must be received in FIFO order. Otherwise, the scion cleaner may use an old stub table that does not match the scions being considered. Such an inconsistency could result in the erroneous deletion of a scion. Since the messages with stub and `ownerPtr` information are exchanged between a pair of nodes (point-to-point communication), FIFO ordering is easily guaranteed by numbering the messages. In Ferreira[9] we describe some race situations that can occur between stub table messages and scion-messages; however, the description of these race conditions is beyond the scope of this paper.

Messages with the reachability information can be piggy-backed onto messages used by the DSM consistency protocol, or exchanged in the background. Thus, they are not disruptive in the sense that applications do not depend on the scion cleaner having finished to continue their execution. Note that the scion cleaner does not have to process each new message it receives immediately; messages can be accumulated and their processing can be postponed until the start of the next local BGC.

6.2 Deletion of Intra-bunch SSPs

This section describes how an object with copies on several nodes is collected and the corresponding intra-bunch SSPs and `ownerPtrs` are deleted, once all replicas become unreachable.

An object that becomes unreachable from all mutators at all nodes will end up not being referenced by any inter-bunch scion at any node and by any entering

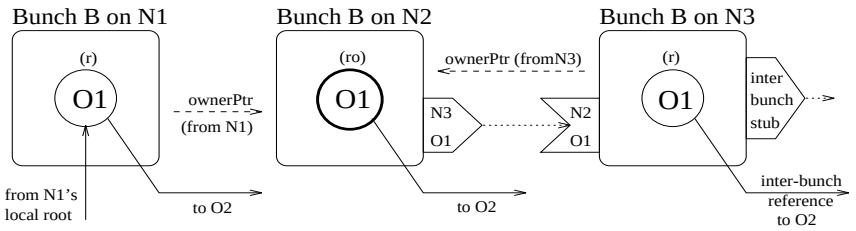


Figure 4: O1 is cached on nodes N1, N2, and N3 and is reachable from a single mutator in N1.

ownerPtr at its owner node. Thus, when the BGC is executed on the owner node, the mentioned object will not be *seen*. Should the object have any intra-bunch stub(s) on the owner node, these will not become part of the new stub table. Transitively, the scion cleaner running on the other nodes will delete the unreachable intra-bunch scion(s) and the object will be reclaimed by the next BGC at those nodes as well.

For example, consider Figure 4 and suppose that the BGC of B is executed at N3; the new set of exiting ownerPtrs will not include the one from N3 to N2, because O1 is not reachable from the mutator at N3. However, O1 remains alive at N3 due to the intra-bunch scion. Note that if the ownerPtr from N3 to N2 was included in the list of outgoing pointers, O1 could never be reclaimed because of the following cycle: O1 on N2 → intra-bunch SSP → O1 on N3 → ownerPtr from N3 to N2 → O1 on N2. By not including the outgoing ownerPtr from N3 the cycle is broken and the scion-cleaner at N2 deletes the entering ownerPtr for N3.

The BGC running on N2 considers O1 alive because of the entering ownerPtr, which originates at N1. Now, imagine that O1 becomes unreachable at N1, that is, the reference to O1 is deleted from the local root of the mutator at N1. Then, a BGC is executed for B on N1. Object O1 can be reclaimed at N1, and the ownerPtr pointing from N1 to N2 will not be part of the new set of ownerPtrs exiting B on N1.

Finally, when N2 receives the new information generated by the BGC at N1, the scion cleaner at N2 deletes the last entering ownerPtr for O1. Therefore, during the next execution of B's BGC at N2, object O1 is no longer reachable, which in turn will drop the intra-bunch stub pointing to O1 at N3 from the new stub table. Thereafter, when N3 receives this new information from N2 and runs its own BGC on B, object O1 will no longer be reachable on N3 either, and will also be garbage collected there.

7 Group Garbage Collector

The GGC is used to reclaim inter-bunch cycles of garbage. There is one GGC per node and it operates on groups of bunches local to that node. The algorithm used by the GGC is identical to the one used by the BGC, only that it operates on a group of bunches, rather than on one bunch at a time.

The root of the GGC includes: mutator stacks, intra-bunch scions, entering ownerPtrs, and 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.

The significance of the group garbage collector is that it can collect an arbitrary subset of the distributed and persistent objects on a single site, independently of the rest of the address space. Bunches are grouped based on a heuristic that maximizes the amount of inter-bunch garbage that is collected and minimizes the cost of performing the collection. Currently, we use a locality-based heuristic, that is, we collect all bunches that are in memory at the site where the GGC is going to run. This heuristic avoids disk input-output overheads.

This locality-based heuristic does not collect cycles of garbage that partially reside on disk, that is, cycles with objects allocated 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 removed at all. We believe that some of these cycles can be collected by improving the grouping 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. Dynamic grouping of collection spaces was first proposed by Lang[14]. However, our solution is much simpler because a group collection occurs at a single site, instead of spanning multiple nodes. Therefore, we expect our solution to be more scalable.

8 Implementation

The current prototype of BMX is implemented in C++ for a network of DEC Alpha workstations. The prototype implementation was simplified by placing the following constraints on the system: first, a bunch is shared only by processes on different nodes, in other words, there is only one process per node accessing a bunch; second, the copy-set of an object is centralized at the object's owner node, instead of being distributed among those nodes that have transitively granted a read token to other nodes; and finally, persistence is supported by associating each segment with a Unix file.

The current prototype is based on BMX-servers and BMX-clients. A BMX-server runs on every node in the system and provides basic services, such as allocation of non-overlapping segments. The BMX-client is a library that is linked with each application and is used to interact with the BMX system internals, in particular with the BMX-server.

Bunches are mapped into shared virtual memory. The BGC runs as a thread inside the process that is accessing the corresponding bunch. This particular implementation of the BGC is facilitated by our simplification of allowing only one process per node to have access to a bunch. The scion cleaner and the GGC each run as a privileged process on each node. Because these processes are privileged they have access to all bunches local to the node.

Recovery is based on the recoverable virtual memory (RVM) techniques proposed by Satyanarayanan et al. [19]. RVM provides simple recoverable transactions with no support for nesting, distribution, or concurrency control. Recovery in RVM is implemented with a disk-based log. In our prototype we use the approach proposed by O'Toole et al. [17], in which the from-space and the to-space are each supported by a file. Changes to mapped segments are atomically transferred to disk by RVM.

As previously mentioned, inter-bunch pointers are described by inter-bunch SSPs that are automatically allocated whenever an inter-bunch reference is created. We detect the creation of cross-bunch references using a write barrier technique, that is, write barriers are inserted into applications by instrumenting every write with a C++ macro. Another macro is provided to perform pointer comparison. This macro is necessary to account for the use of forwarding pointers left by the execution of a garbage collection. Currently, the programmer must include these macros explicitly. In the future, we expect to modify the pre-processor to insert these macros automatically.

The contents of a bunch are described by two special data structures that contain information needed for garbage collection: an *object-map*, which describes the location of objects inside the bunch, and a *reference-map*, which indicates where pointers are located inside each object. These data structures are implemented as bit arrays; each bit describes the contents of a 4-byte address range inside the bunch. A set bit in the object-map means that at the corresponding address is an object; a set bit in the reference-map means that at the corresponding address is a pointer to some object.

One of the performance goals of our design is to support replication of bunches on several nodes without increasing the cost of the bunch garbage collection, when compared to a system without support for bunch replication. From the point of view of the application, the cost of the BGC should be the same whether the bunch is replicated or not. We believe we can ensure this cost property by avoiding the interference between the garbage collector and the DSM mechanisms. This expectation is based on two observations: (i) the BGC never acquires a token for any object, and consequently does not interfere with the DSM consistency protocol, and (ii) information exchanged among nodes is either piggy-backed onto messages due to the consistency protocol, or exchanged in the background.

9 Related Work

A large amount of literature exists in the area of concurrent GC either for multiprocessors [1, 6], or for RPC-based distributed systems (see Plainfossé[18] for a survey). On the contrary, to our knowledge, little work has been done on garbage collecting objects in a loosely coupled network with weakly consistent DSM.

The fundamental difference between our system and a multiprocessor system, with respect to GC, is that of scale and synchronization overhead. This difference implies that, if we apply a GC algorithm designed for multiprocessors (for instance, 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 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. Note that the overhead is not due to the synchronization between the mutator and the GC algorithm, as is usually the case in non-distributed settings.

Furthermore, GC in DSM is more difficult than in distributed RPC-based systems (for example, Juul[11]) due to the existence of multiple copies of the same object on several nodes, and the problem of consistency interference.

Le Sergent[15] describes an extension of a copying garbage collector first developed for a multiprocessor,

to a DSM system. Objects are kept strongly consistent, the entire address space is collected at the same time, which is not scalable, and the garbage collector locks pages while scanning, which interferes with the consistency protocol.

Kordale's GC design for distributed shared memory [12] is very complex and relies on a large amount of auxiliary information. This GC algorithm is based on the mark & sweep technique, and objects are kept strongly consistent.

10 Conclusion

We have presented a copying garbage collector algorithm for objects accessed via DSM in a loosely coupled network. Objects are allocated from bunches of 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.

Our design goals were that the garbage collector neither interfere with the consistency protocol, nor introduce high communication overheads. Therefore, in our garbage collection design: (i) a cached copy of a bunch can be collected independently of any other copy of that same bunch on other nodes, (ii) only locally-owned live objects are copied by a bunch garbage collector; not owned live objects are simply scanned, and (iii) references to copied objects are lazily updated, either by taking advantage of messages sent on behalf of the consistency protocol (piggy-backing), or in the background. In any circumstance, the garbage collector acquires neither a read nor a write token.

The fundamental observation that guided our design is that GC consistency needs are less strict than applications'. Thus, the garbage collector can work with objects that are inconsistent from the point of view of the consistency protocol. This allows the GC to be performed without interfering with applications' consistency needs and requiring very little synchronization or communication.

We are currently in the process of evaluating the performance of BMX. In future work, we hope to generalize our design to other consistency protocols and other GC algorithms, in addition, to evaluating the impact of the consistency granularity on our approach. We are also extending the current GC design to incorporate a weakly consistent distributed shared memory system with full support for transactions.

Acknowledgments: We are grateful to our shepherd, Karin Petersen, and to the anonymous referees for their help with improving 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] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *Computer*, 26(4):360–365, 1983.

[3] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. 2nd Annual Symp. on Principles and Practice of Parallel Programming*, Seattle, WA (USA), March 1990. ACM SIGPLAN. In SIGPLAN Notices 25(3).

[4] 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.

[5] 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.

[6] 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.

[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] 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.

[9] Paulo Ferreira and Marc Shapiro. Garbage collection of persistent objects in distributed shared memory. In *Proc. of the 6th International Workshop on Persistent Object Systems*, Tarascon (France), September 1994. Springer-Verlag.

[10] 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.

[11] 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.

[12] 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.

[13] M. S. Lam, P. R. Wilson, and T. G. Moher. Object type directed garbage collection to improve locality. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 404–425, Saint-Malo (France), September 1992.

[14] 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.

[15] 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), September 1992. Springer-Verlag.

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

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

[18] 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.

[19] 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.

[20] 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.