Collecting Distributed Garbage Cycles by Back Tracing

Umesh Maheshwari Barbara Liskov

MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139
{umesh,liskov}@lcs.mit.edu

Abstract

Systems that store objects at a large number of sites require fault-tolerant and timely garbage collection. A popular technique is to trace each site independently using inter-site references as roots. However, this fails to collect cyclic garbage spread across sites. We present an algorithm that collects cyclic garbage by involving only the sites containing it.

Our algorithm is based on finding objects highly likely to be cyclic garbage and tracing backward from them to check if they are reachable from any root. We present efficient techniques that make conducting such traces practical. The algorithm collects all distributed cyclic garbage, is safe in the presence of concurrent mutations, and has low space and time overhead.

1 Introduction

Emerging distributed systems will use objects stored at a large number of sites. The scale of such systems poses new challenges to reclaiming the storage of objects unreachable by applications. Such objects are known as *garbage*. A simple way to collect garbage is to trace the graph of reachable objects and then collect objects not visited by the trace [HK82]. However, a global trace requires the cooperation of all sites before it can collect any garbage.

Timely and fault tolerant collection requires that each site trace local objects and collect garbage independently of other sites. However, for a local trace to be safe, object references from other sites must be treated as roots. Thus, many distributed systems use local tracing in combination with some variant of inter-site reference counting to track inter-site references [Bis77, Ali85, Bev87, SDP92, JJ92, BEN+93, ML94].

Local tracing has the desirable *locality property* that collecting a garbage object requires the cooperation of only the sites it is reachable from. Locality results in good fault tolerance and timely collection because it avoids unnecessary dependencies; if a site is crashed, partitioned from others, or otherwise slow, it will delay the collection of only the garbage reachable from its objects.

However, treating inter-site references as roots leads to a problem: it does not collect mutually reachable garbage objects located on different sites; such garbage is said to be *cyclic*. Inter-site cycles are relatively uncommon, but they do occur in practice. For example, hypertext documents often form large, complex cycles. Collection of such cycles is particularly important in long-lived systems because even small amounts of uncollected garbage can accumulate over time to cause a significant storage loss.

The challenge in collecting an inter-site garbage cycle is to preserve locality, that is, to involve only the sites containing the cycle. This has proven surprisingly difficult. Most previous schemes do not preserve locality. For example, some conduct complementary global traces in addition to local tracing [Ali85, JJ92]. The drawbacks of global tracing can be alleviated by tracing within groups of selected sites [LQP92, MKI+95, RJ96], but inter-group cycles may never be collected.

Few schemes for collecting inter-site cycles have the locality property. The most prominent among these is based on *migrating* objects so that cyclic garbage ends up in a single site and is collected by local tracing [Bis77, SGP90, ML95]. However, migration is expensive and must deal with updating references to migrated objects; moreover, some systems do not support migration due to security or autonomy constraints. Other local schemes are prohibitively costly or complex [Sch89, LC97].

We present a practical scheme that has locality. It has two parts. The first part identifies objects that are highly likely to be cyclic garbage—the *suspects*. We have previously designed a suitable technique for finding suspects using the distance heuristic [ML95]. The second part checks if the suspects are in fact garbage. This part has the luxury of using techniques that are too costly if applied to all objects but are acceptable if applied only to suspects.

This paper describes a technique for checking suspects by tracing *back* from a suspect to see if it is reachable from any root. This approach preserves locality and scalability. Back tracing was proposed earlier by Fuchs [Fuc95]. However, this proposal assumed that inverse information was available for references, and it ignored problems due to concurrent

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research, contract N00014-91-J-4136.

mutations and forward local traces. We present efficient techniques for conducting back tracing that handle these and other practical problems. Our scheme computes the information required to back trace, controls when to start a back trace, and accounts for concurrent mutations and forward traces. We show that the scheme is safe and collects all inter-site garbage cycles. Its space and time overheads are low and limited to suspected objects.

The rest of this paper is organized as follows. Section 2 describes the system model and local tracing. Section 3 summarizes the distance heuristic for finding suspects. Section 4 presents basic techniques for back tracing, Section 5 describes how the information required for back tracing is computed, and Section 6 describes how concurrent mutations and traces are handled. Section 7 summarizes related work, and Section 8 contains our conclusions.

2 The Problem Context

Our scheme is useful in systems that store persistent objects over a large number of sites. Objects contain references to other objects, which may reside at other sites. Objects are clustered within sites so that inter-site object references are relatively uncommon.

Certain objects, designated as *persistent roots*, serve as entry points into the object store. For example, a name server or a directory object may be a persistent root. User applications begin by accessing a persistent root and then traversing references to access other objects. An application may mutate the object graph by creating objects and by inserting and deleting references; therefore, it is called the *mutator*. A mutator may traverse an inter-site reference by passing the reference in a message from the source site to the target site.

A mutator may store a reference in a local variable outside the object store and retrieve the reference later; these references constitute the *application roots*. In practice, application roots are handled using techniques specific to the application model. For simplicity, in this paper we will treat application roots like persistent roots; Section 6.3 discusses some related issues.

Mutations may cause some objects to become unreachable from any persistent root; these are of no use to applications and are said to be *garbage*, while other objects are said to be *live*. The job of the garbage collector is to detect garbage objects and reclaim their storage.

Local Tracing

Each site conducts a local trace independently of other sites. For a local trace to be safe, it must not collect objects reachable from other sites. Therefore, it treats incoming inter-site references as roots. Different methods may be employed to record inter-site references; e.g., one-bit reference counts [Ali85, JJ92], weighted reference counts [Bev87], and reference lists [Bis77, SDP92, BEN+93]. We use inter-site reference listing because it handles site failures and provides better fault-tolerance for messages [ML94]. It works as follows.

Each site keeps a table of incoming references, called *inrefs*. Each entry in the table, called an *inref*, stores a reference and a list of source sites known to contain that reference. For example, in Figure 1, site *R* has an inref *c*, which indicates the source sites *P* and *Q*. (An inref is identified by the reference it contains, so we say "inref *c*" to denote the inref containing *c*.) The local collector traces from local persistent roots and the inrefs. For simplicity, we treat a local persistent root as a permanent inref.

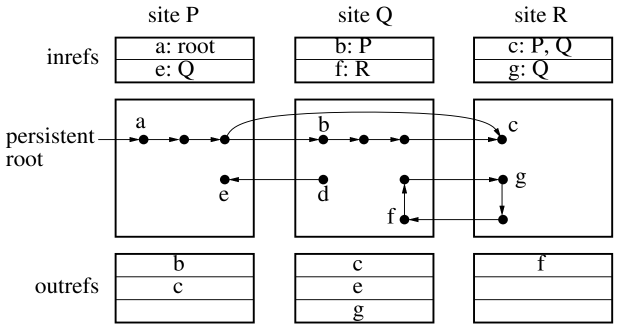

Figure 1: Recording inter-site references.

Each site also keeps a table of outgoing references, called *outrefs*. The outrefs are used in inserting and removing inrefs as follows. Suppose a site *Q* sends a reference to site *P* and the reference points to an object *c* in site *R*. The recipient, *P*, checks whether *c* exists in its outrefs. If not, it enters *c* in the outrefs and sends an *insert message* to the owner, *R*; when *R* receives the message, it inserts *P* in the source list of inref *c*. For safe execution, the sender *Q* retains its outref for *c* until *R* is known to have received the insert message. There are various protocols for sending, deferring, or avoiding insert messages while ensuring safety [SDP92, BEN+93, ML94]. We assume that a safe insert protocol exists and that the full source list of an inref can be found when needed.

A site *P* trims its outrefs during each local trace. After the trace, *P* removes untraced outrefs and reports them to their target sites in *update messages*; the target sites remove *P* from the source lists in the inrefs for the given references. An inref with an empty source list is removed. In the example in Figure 1, when *Q* does its next local trace, it will collect *d*, drop its outref for *e*, and send an update message to *P*. Then *P* will drop its inref for *e* and collect *e* during the next local trace. This illustrates the locality property: collecting a garbage object involves only the sites it is reachable from.

The problem with local tracing is that it fails to collect cycles of garbage objects spread over multiple sites. For example, in Figure 1, *P* and *Q* will never collect *f* and *g* and all objects reachable from them. Therefore, a separate scheme is necessary to collect inter-site cycles. Such a scheme should have the following properties:

Safety: do not collect a live object.
Completeness: collect all garbage cycles eventually.
Locality: minimize inter-site dependence.

3 Heuristic for Finding Suspects

The heuristic for finding objects likely to be cyclic garbage may be unsafe: it may suspect some live objects, but good performance requires that few suspects are live. The heuristic must be complete in identifying all cyclic garbage. Furthermore, it must have little time and space overhead per object since it must inspect a large number of objects.

A suitable technique for finding suspects is the distance heuristic [ML95]. The *distance* of an object is the minimum number of inter-site references in any path from a persistent root to that object. The distance of garbage is infinity. In Figure 1, c is reachable from root a through two paths: one with two inter-site references and another with one; therefore, its distance is one.

Suspects are found by *estimating* distances. A distance field is associated with each source site in an inref, and the distance of the inref as a whole is the smallest such distance. When a new source is added to in an inref, its distance is conservatively set to one. A persistent root is modelled as an inref with zero distance. The local trace propagates distances from inrefs to outrefs. To do this, inrefs are traced in the increasing order of their distances. When the trace first reaches an outref, its distance is set to one plus that of the inref being traced. Finally, changes in the distances of outrefs are sent to target sites in update messages, where they are reflected in the corresponding inrefs.

As distances are propagated through local traces and update messages, the estimated distances of cyclic garbage keep increasing without bound. The following theorem holds for arbitrarily complex cycles: If all sites containing a cycle do at least one local trace in a certain period of time, called a *round*, then n rounds after the cycle became garbage, the estimated distances of all objects in the cycle will be at least n.

Therefore, we select a *suspicion threshold* distance, D, and regard objects with greater estimated distances as highly likely to be garbage. Inrefs with distances less than or equal to the threshold —and objects and outrefs traced from them— are said to be *clean*. The remaining are said to be *suspected*. The higher the threshold, the smaller the chance that suspected objects might be live, but the longer it takes to detect them.

The distance heuristic is complete because all cyclic garbage is eventually suspected. Unlike previous heuristics, its accuracy can be controlled arbitrarily. Heuristics that suspect the inrefs not accessed recently are not suitable for persistent stores since live objects might not be accessed for long periods.

Since suspects might be live, they cannot be reclaimed directly. Instead, another technique must confirm suspected garbage before reclaiming it. The outcome of the this technique may be used to tune the suspicion threshold. For example, if too many suspects are found live, the threshold should be increased.

In an earlier paper, we suggested migrating the suspects [ML95]. This paper presents a technique that does not migrate objects.

4 Back Tracing

The key insight behind back tracing is that whether an object is reachable from a root is equivalent to whether a root is reachable from the object if all references are reversed. Thus, the idea is to trace backwards from a suspect: if the trace encounters a persistent root, the suspect is live, otherwise it is garbage.

The virtue of back tracing is that it has the locality property. For example, a back trace started in a two-site cycle will involve only those two sites. This is in contrast to a forward trace from the persistent roots, which is a global task. Indeed, a global forward trace would identify *all* garbage in the system, while a back trace checks only whether a particular object is live. Thus, back tracing is not suitable as the primary means of collecting garbage. We expect most garbage to be collected by local tracing and update messages. Back tracing is a complementary technique to detect uncollected garbage, and we use it for objects suspected to be on distributed garbage cycles.

Back tracing was proposed earlier by Fuchs [Fuc95]. However, this proposal assumed that each object was contained in a site by itself and inverse information was available for references, and it ignored problems due to concurrent mutations and forward traces. We present practical techniques for conducting back tracing. This section describes the basic technique, and Section 6 extends it to handle the problems due to concurrent mutations and forward traces.

4.1 Back Steps

In practice, tracing back over individual references would be prohibitively expensive—both in the time required to conduct it and in the space to store the required information. Therefore, a back trace leaps between outrefs and inrefs. For brevity, we refer to inrefs and outrefs collectively as *iorefs*. A back trace takes two kinds of steps between iorefs:

Remote steps that go from an inref to the corresponding outrefs on the source sites.

Local steps that go from an outref to the inrefs it is locally reachable from[1].

The information needed to take remote steps from an inref is already present in its source list. We do need extra infor-

[1] A reference is *locally* reachable from another if there is a path of zero or more local references from the object referenced by the first to an object containing the second.

mation to take local steps. We define the *inset* of a reference as the set of inrefs from which it is locally reachable. For example, in Figure 2, in site Q, the inset of outref c is $\{a, b\}$. We compute and store the insets of outrefs so that back traces may use them when required. Section 5 describes techniques for computing insets efficiently.

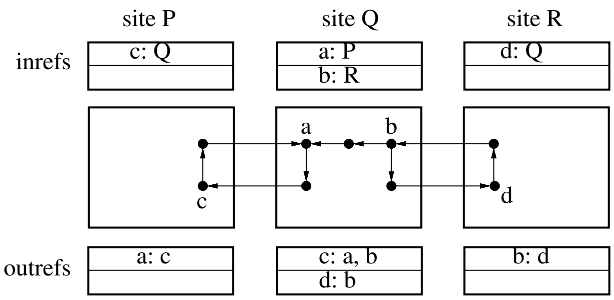

Figure 2: Insets of suspected outrefs.

A back trace consists of taking local and remote steps alternately. For example, a back trace at outref c in Q will take local steps to inrefs a and b. From there, it will take remote steps to outrefs a and b and so on.

In general, a back trace may be started from any suspected ioref. However, a back trace started from an inref a will not find paths to object a from other inrefs on the same site. For example, in Figure 2, a back trace started from inref a will miss the path from inref b to object a. On the other hand, a back trace started from an outref c that is locally reachable from a will find all paths to a because the set of inrefs that c is locally reachable from must include all inrefs that a is locally reachable from. Thus, if a back trace started from an outref does not encounter a persistent root, all inrefs visited by the trace must be to garbage objects. Therefore, we start a back trace from an outref rather than an inref.

4.2 How far to go

A practical requirement on a back trace is to limit its spread to *suspected* iorefs. Thus, rather than going all the way back in the search for a persistent root, a back trace returns "Live" as soon as it reaches a clean ioref. This rule limits the cost of back tracing to the suspected parts of the object graph in two ways:

- We need to compute insets for suspected outrefs only.
- A back trace from a live suspect does not spread to the clean parts of the object graph.

The rule has the disadvantage that back tracing will fail to identify a garbage cycle until all objects on it are suspected. This is not a serious problem, however, because the distance heuristic ensures that all cyclic garbage objects are eventually suspected. The next section describes the suitable time to start a back trace.

4.3 When to Start

A site starts a back trace from a suspected ioref based on its distance. There is a tradeoff here. A back trace started soon after an ioref crosses the suspicion threshold, D, might run into a garbage ioref that is clean because its distance has not yet crossed D and return Live unnecessarily. On the other hand, a back trace started too late delays collection.

Here, we estimate a suitable *back threshold* D_2 to trigger a back trace. Ideally, by the time the distance of any ioref on a cycle is above D_2, those of other iorefs on the cycle should be above D. If the distance of an ioref y is D_2 and the distance from x to y is C, then the distance of x should be at least $D_2 - C$. Therefore, an appropriate value for D_2 is $D + C$, where C is a conservatively estimated (large) cycle length.

The use of back threshold is an optimization and does not compromise completeness. If a back trace is started prematurely in a garbage cycle, the trace might return Live unnecessarily, but a future trace would confirm the garbage. To this end, each ioref has a back threshold field initially set to D_2. When a back trace visits an ioref x, the back threshold of x is incremented by, say, C. Thus, the next back trace from x, if any, is triggered only when its distance crosses the increased threshold. This has the desirable property that live suspects will stop generating back traces once their back thresholds are above their distances. Garbage objects, on the other hand, will generate periodic back traces until they are collected.

4.4 Back Tracing Algorithm

Back tracing can be formulated as two mutually recursive procedures: BackStepRemote, which takes remote steps, and BackStepLocal, which takes local steps. Both are similar to a standard graph search algorithm.

```
BackStepRemote (site P, reference i)
  if i is not in Inrefs return Garbage
  if Inrefs[i] is clean return Live
  if Inrefs[i] is visited by this trace return Garbage
  mark Inrefs[i] visited by this trace
  for each site Q in Inrefs[i].Sources do
      if BackStepLocal(Q, i) is Live return Live
  return Garbage
end
BackStepLocal (site P, reference o)
  if o is not in Outrefs return Garbage
  if Outrefs[o] is clean return Live
  if Outrefs[o] is visited by this trace return Garbage
  mark Outrefs[o] visited by this trace
  for each reference i in Outrefs[o].Inset do
      if BackStepRemote(P, i) is Live return Live
  return Garbage
end
```

If the reference being traced is not found among the recorded iorefs, its ioref must have been deleted by the garbage collector; so the call returns Garbage. To avoid revisiting iorefs and to avoid looping, an ioref remembers that a trace has visited it until the trace completes; if the trace makes another call on the ioref, the call returns Garbage immediately. Note that the calls within both for-loops can be made in parallel. For example, in Figure 3, a call at inref *c* in *R* will fork off two branches to *P* and *Q*. One branch will visit inref *a* first and go further back, while the other will return Garbage.

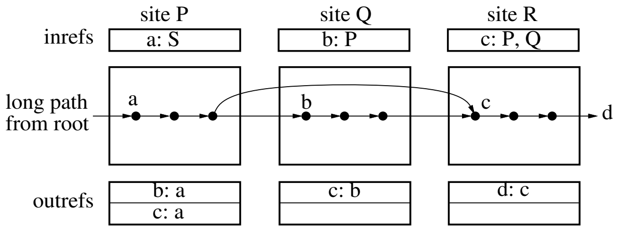

Figure 3: A back trace from *d* will branch.

An activation frame is created for each call. A frame contains the identity of the frame to return to (including the caller site, etc.), the ioref it is active on, a count of pending inner calls to BackStep, and a result value to return when the count becomes zero. We say that a trace is *active* at an ioref if it has a call pending there; we say that a trace has *visited* an ioref if the ioref is marked visited by the trace.

This algorithm is simpler than Fuchs's [Fuc95]. This is because Fuchs's algorithm is designed to detect garbage objects "on the way" even if the back trace was started from a live object. We start back traces from objects highly likely to be garbage and therefore do not incur the complexity of Fuchs's optimization.

4.5 Collecting Garbage

If the outer-most call to BackStep returns Garbage, all inrefs visited by the trace are garbage. However, when an *intermediate* call returns Garbage, it cannot be inferred that the corresponding ioref is garbage, because that call would not have visited iorefs that have already been visited by other branches in the same trace. For example, in Figure 2, the call at inref *b* may return Garbage because inref *a* has been visited, although *b* is not garbage. Therefore, no inref is removed until the outer-most call returns.

When the outer-most call returns, the site that initiated the back trace reports the outcome to all sites reached during the trace, called the *participants*. We call this the *reporting phase*. For the initiator to know the set of participants, each participant appends its id to the response of a call. If the outcome is Garbage, each participant flags the inrefs visited by the trace as garbage. If the outcome is Live, each participant clears the visited marks for that trace. Note that the outcome is likely to be Garbage since the suspected objects are highly likely to be garbage.

An inref flagged as garbage is not used as a root in the local trace. Such an inref is not removed immediately in order to maintain referential integrity between outrefs and inrefs. Flagging the inrefs visited by the trace causes the cycle to be deleted the next time the containing sites do a local trace. The flagged inrefs are then removed through regular update messages.

4.6 Message Complexity

Back tracing involves two messages for each inter-site reference it traverses—one for the call and another for its response. Finally, the report phase involves a message to each participant. Thus, if a cycle resides on N sites and has E inter-site references, $2E + N$ messages are sent. These messages are small and can be piggybacked on other messages.

Loss of messages can be handled by suitably long timeouts (possibly, after repeated query messages to find the status). If a site waiting for a response to a call times out, it can safely assume that the call returned Live. Similarly, if a site waiting for the final outcome times out, it can assume that the outcome is Live.

4.7 Multiple Back Traces

Several back traces may be triggered concurrently at the same or different sites. The site starting a trace assigns it a unique id. Thus, the visited field of an ioref stores a set of trace ids.

Multiple traces may be active for objects on the same cycle, but this is not likely for the following reason. The distances of various iorefs in a cycle are likely to be different such that one of them will cross the threshold D_2 first. Even if several iorefs have the same distance, there will be differences in real time when they cross D_2 due to randomness in when local traces complete. The time between successive local traces at a site is long—on the order of minutes or more—compared to the little amount of processing and messaging involved in a back trace—on the order of milliseconds at each site (or tenths of a second if messages are deferred and piggybacked). Therefore, the first back trace started in a cycle is likely to visit all other iorefs in the cycle before they cross D_2.

There is no problem if one trace confirms garbage and results in the deletion of an ioref when another trace has visited it. The second trace can ignore the deletion, even if its call is active there, because activation frames provide the necessary return information.

5 Computing Back Information

Back information comprises the source sites of inrefs (for remote steps) and the insets of outrefs (for local steps). The source sites are maintained by the underlying scheme as described in Section 2. Here, we describe the computation of the insets of outrefs. This information is computed and stored such that it is ready for use by back traces when they arrive.

We compute insets of outrefs by first computing their inverse, namely, the *outsets* of suspected inrefs. We define the outset of a reference as the set of suspected outrefs locally reachable from it. Outsets and insets are simply two different representations of reachability information from inrefs to outrefs.

Ideally, we want to compute outsets during the local forward trace from suspected inrefs. However, a trace does not provide full reachability from inrefs to outrefs. This happens because a trace scans a reachable object only once, which is crucial for its linear performance. For example, in Figure 4, if a is traced before b, then the trace from a will visit z first. Therefore, the trace from b will stop at z and will not discover the outref c.

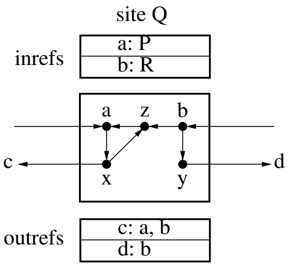

Figure 4: Tracing does not compute reachability.

Therefore, we need to modify the trace from suspected inrefs to compute outsets. We describe two techniques below: the first is straightforward but may retrace objects, while the second traces each object only once.

5.1 Independent Tracing from Inrefs

The straightforward technique is to trace from each suspected inref ignoring the traces from other suspected inrefs. Conceptually, each trace from a suspected inref uses a different color to mark visited objects. Thus, objects visited by one such trace may be revisited by another trace. However, objects traced from clean inrefs are never revisited. They may be considered marked with a special color, black, that is never revisited.

Independent tracing from each suspected inref reaches the complete set of suspected outrefs locally reachable from it. The problem with this technique is that objects may be traced multiple times, and tracing objects is expensive in practice. If there are n_i suspected inrefs, n suspected objects, and e references in the suspected objects, the complexity of this scheme is $O(n_i \times (n + e))$ instead of the usual $O(n + e)$.

5.2 Bottom-up Computation

Outsets of suspected inrefs may be found by computing the outsets of suspected objects bottom up during the forward trace. Outsets of suspected objects are remembered in an *outset table*; once the outset of a suspect z is computed, it is available when tracing from various inrefs without having to retrace z.

The following is a first cut at the solution:

```
TraceSuspected(reference x)
  if x is marked return
  mark x
  Outset[x] := null
  for each reference z in x do
    if z is clean continue loop
    if z is remote add z to Outset[x] and continue loop
    TraceSuspected(z)
    Outset[x] := Outset[x] ∪ Outset[z]
  endfor
end
```

The above solution does not work because it does not account for backward edges in the depth first tree. For example, in Figure 4, if inref a is traced first, the outset of z would be erroneously set to null instead of $\{c\}$. Since the outset of inref b uses that of z, it would miss c as well. In general, a backward edge introduces a strongly connected component, and the outsets of objects in a strongly connected component should all be equal. Fortunately, strongly connected components can be computed efficiently during a depth first traversal with linear performance [Tar72]. For each object, the algorithm finds the first object visited in its component, called its *leader*. The algorithm uses a counter to mark objects in the order they are visited. An auxiliary stack is used to find the objects in a component.

The following algorithm combines tracing, finding strongly connected components, and computing outsets. The algorithm sets the outset of each object to that of its leader.

```
TraceSuspected(reference x)
  if x is marked return
  Mark[x] := Counter
  Counter := Counter+1
  push x on Stack
  Outset[x] := null
  Leader[x] := Mark[x]
  for each reference z in x do
    if z is clean continue loop
    if z is remote add z to Outset[x] and continue loop
    TraceSuspected(z)
    Outset[x] := Outset[x] ∪ Outset[z]
    Leader[x] := min(Leader[x], Leader[z])
  endfor
  if Leader[x] = Mark[x] % x is a leader
    repeat
      z := pop from Stack % z is in the component of x
      Outset[z] := Outset[x]
      Leader[z] := infinity % so that later objects ignore z
    until z = x
end
```

This algorithm traces each object only once. In fact, it uses $O(n+e)$ time and $O(n)$ space except for the union of outsets. In the worst case, if there are n_o suspected outrefs, the union of outsets may take $O(n_o \times (n+e))$ time and $O(n_o \times n)$ space. Below we describe efficient methods for storing outsets and for implementing the union operation; these methods provide near-linear performance in the expected case and thus make bottom-up computation attractive.

First, the outset table maps a suspect to an *outset id* and the outset itself is stored separately in a canonical form. Thus, suspected objects that have the same outset share storage. If objects are well clustered on sites, there will be many fewer distinct outsets than suspected objects. This is because there are expected to be many fewer outrefs than objects; further, objects arranged in a chain or a strongly connected component have the same outset.

Second, the results of uniting outsets are memoized. A hash table maps pairs of outset ids to the outset id for their union. Thus, redoing memoized unions takes constant time. If a pair is not found in the table, we compute the union of the two outsets. Another table maps existing outsets (in canonical form) to their ids. If the computed outset is found there, we use the existing id.

The various data structures used while tracing from suspected inrefs are deleted after the trace. Only the outsets of the suspected inrefs are retained. As mentioned, these outsets are equivalent to keeping the insets of the suspected outrefs, since one may be computed from the other. The space occupied by insets or outsets is $O(n_i \times n_o)$, where n_i and n_o are the number of suspected inrefs and suspected outrefs.

6 Concurrency

So far, we assumed that back traces used the information computed during previous local traces and there were no intervening mutations. In practice, a mutator may change the object graph such that the computed back information is no longer accurate. Further, mutators, local forward tracing, and back tracing may execute concurrently. This section presents techniques to preserve safety and completeness in the presence of concurrency. We divide the problem into several parts:

1. Keeping back information up to date assuming that mutators, local traces, and back traces execute atomically, that is, happen instantaneously without overlap.
2. Accounting for a non-atomic local trace: While the local trace is computing back information, a mutator may change the object graph, or a back trace may visit the site.
3. Accounting for a non-atomic mutator: the mutator may store a reference in a variable and retrieve it later without starting at a persistent root.
4. Accounting for a non-atomic back trace: Even if back information is kept up to date at each site, a back trace might see an inconsistent distributed view because it reaches different sites at different times.

6.1 Keeping Back Information up to Date

This section presents techniques for updating back information conservatively while assuming that mutators, local tracing, and back tracing execute instantaneously.

Back information may change due to reference creation and deletion. We ignore deletions since doing so does not violate safety; also, it does not affect completeness because deletions are reflected in the computation of back information during the next local trace. On the other hand, reference creations must be handled such that a back trace does not miss a new path to a suspect. For example, Figure 5 shows the creation of a reference to *z* followed by the deletion of a reference in the old path to *z*. If site *Q* does not update its back information to reflect the new reference, but site *S* does a local trace to reflect the deletion, a subsequent back trace from *g* might return Garbage.

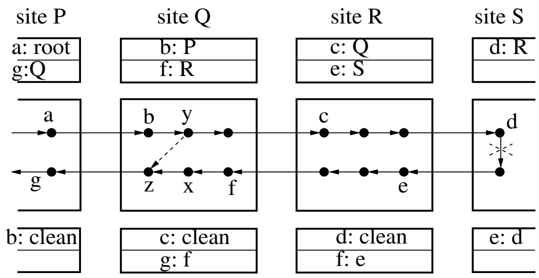

Figure 5: Reference mutations (dotted lines).

In general, creating a reference involves copying a reference *z* contained in object *x* into object *y*.[2] Suppose *x*, *y*, and *z* are in sites *X*, *Y*, and *Z* respectively, some or all of which may be the same. If *X* is the same as *Y*, we say that the copy is local; otherwise, we say it is remote. We discuss these situations separately below.

6.1.1 Local Copy

A local copy is tricky to handle because it may change the reachability from inrefs to outrefs although none of the objects involved might be associated with iorefs. In Figure 5, creating a reference to *z* makes outref *g* reachable from inref *b*. We maintain the following safety invariant.

Local Safety Invariant For any suspected outref *o*, *o.inset* includes all inrefs *o* is locally reachable from.

The key to maintaining this invariant is that in order to create a new path to a suspect *z*, the mutator must have traversed an old path to it. This traversal must include traversing an inter-site reference to a suspected object on the same site since *z* must be reachable only through a suspected inref. In the example shown, the mutator must have traversed the reference to *f*. This provides an opportunity to patch back information as follows.

[2]The creation of a reference to a *new* object *z* may be modeled as copying a reference to *z* from a special persistent root.

Transfer Barrier When a mutator transfers (or traverses) a reference i to site Q, if Q has a suspected inref for i, it *cleans* inref i and the outrefs in *i.outset*.

We show below that the transfer barrier preserves the local safety invariant. We use an auxiliary invariant: For any suspected outref o, *o.inset* does not include any clean inref. This invariant holds right after a local trace, and the transfer barrier preserves it because whenever it cleans any inref i, it cleans all outrefs in *i.outset*. (Insets and outsets are consistent since they are different views of the same information.)

Proof of Local Safety Suppose a reference from x to z is copied into y. This affects only the outrefs reachable from z: any such outref o may now be reachable from more inrefs than listed in *o.inset*. (If z is an outref itself, then o is identical to z.) We show that all outrefs such as o must be clean after the mutation.

Object x must have been reachable from some inref i before the mutation. Since x pointed to z, outref o was reachable from inref i as well. Therefore, if the local safety invariant held before the mutation, either o was clean or *o.inset* included i. Suppose *o.inset* included i. If x was reachable from a *clean* inref i, the auxiliary invariant implies that o must be clean. Otherwise, the transfer barrier must have been applied to some suspected inref i that x was reachable from. The barrier would have cleaned all outrefs in *i.outset*, which includes o since *o.inset* includes i. In either case, o must be clean after the mutation.
□

Outrefs cleaned by the transfer barrier remain clean until the site does the next local trace. If a back trace visits such an outref before then, it will return Live. The back information computed in the next local trace will reflect the paths created due to the new reference. Therefore, if a back trace visits later, it will find these paths.

It is important not to clean outrefs unnecessarily in order that cyclic garbage is collected eventually. We ensure the following condition for completeness.

Completeness Invariant An outref is clean only if it is reachable from a clean inref or if it was live at the last local trace at the site.

Proof of Completeness The invariant is valid right after a local trace. Thereafter, we clean outrefs only due to the transfer barrier. Suppose applying the barrier on inref i cleans outref o. Then i must be live. Since o was reachable from i when the last local trace was performed, o must be live at that time.
□

In RPC-based systems, a reference may be transferred to a site as the target, argument, or result of a remote call. Thus, the transfer barrier may be implemented by checking such references. In client-caching systems where objects from multiple servers may be fetched into a client cache [LAC+96], the barrier may be implemented by checking the transaction's read-write log at commit time.

6.1.2 Remote Copy

If a reference to z is copied into y at another site Y, we handle it in one of the following ways depending on z:

1. Object z is in site Y:
   Since Y received a reference to z from another site, it must have an inref for z, so the transfer barrier applies to z.
2. Object z is not in site Y and Y has a clean outref for z:
   No update is necessary.
3. Object z is not in site Y and Y has a suspected outref for z:
   Clean the outref for z.
4. Object z is not in site Y and Y has no outref for z:
   Y creates a clean outref for z and sends an insert message to Z, which enters Y in the source list of inref z. (Also, the transfer barrier applies to inref z.)

Each of these cases preserves the local safety and completeness invariants. Only the last case results in the creation of a new inter-site reference. Here, a potential problem is that the insert message may arrive too late. We preserve the following safety invariant.

Remote Safety Invariant For any suspected inref i, either *i.sources* includes all remote sites containing i, or at least one of its corresponding outrefs is clean.

We ensure this invariant by using the following rule:

Insert Barrier If site X sends a reference z at Z to site Y and Y does not have an outref for z, then X retains a *clean* outref for z until Z has received an insert message from Y.

If a back trace visits inref z before the insert message reaches there, it will return Live from the clean outref. Otherwise, it will find all outrefs for z.

The insert barrier is a small modification to the insert protocol described in Section 2. In systems that send the insert message synchronously, site X is informed when the insert message reaches Z [ML94]. Also, the insert barrier can be modified to suit schemes that avoid insert messages; in fact, little change is needed for systems using "indirect protection" [SDP92].

6.2 Non-atomic Local Tracing

Mutations may change the object graph while the local trace is computing back information. The computed information must account for these mutations safely. Further, a back trace may visit a site while it is computing back information.

During a local trace, we keep two copies of the back information: the old copy retains the information from the previous local trace, while the current trace prepares the new copy. When this trace completes, the new copy replaces the old atomically. A back trace visiting the site in the meantime uses the old copy. If a transfer barrier is applied on an inref i in this time, we clean the outrefs in *i.outset* in the old copy; we also remember i and clean the outrefs in *i.outset* in the new copy when it is ready. This preserves the safety invariant and preserves the following completeness invariant:

Completeness Invariant An outref is clean only if it is reachable from a clean inref or if it was live when the last local trace *began*.

6.3 Non-atomic Mutator

A mutator may store a reference in a variable outside the object store and retrieve the reference later. This raises the following potential problem. In the context of Figure 5, the mutator may traverse the remote reference *f* and store a reference to, say, *x* in a local variable. The transfer barrier would then clean the outrefs in *f.outset*. If site *Q* does a local trace now, it will revert inref *f* to its suspected status. Later, the mutator may use the stored reference to copy *z* into *y* without invoking the barrier.

The local safety invariant is preserved, however, because local tracing views variables as application roots, which are treated just like persistent roots. As a result, all outrefs reachable from them are cleaned. Thus, all outrefs that could be affected due to copying a reference reachable from a variable remain clean.

6.4 Non-atomic Back Tracing

So far we assumed that a back trace happens instantaneously such that it sees consistent back information at various sites. In practice, a back trace may overlap with mutations, making it unsafe even if back information is kept up to date at each site. For example, in Figure 5, a back trace from *g* may visit site *Q* *before* the mutator creates the reference, so that it does not see the updated back information at *Q*. Yet the trace may visit *S* *after* the mutator has deleted the old path and *S* has done a local trace to reflect that. We use the following rule to ensure a safe view.

Clean Rule When an ioref is cleaned while a trace is active there, the return value of the trace is set to Live.

This rule implies that if there is any overlap in the periods when an ioref is clean and when a trace is active there, the trace will return Live. We show below that it ensures safety.

Proof of Safety

Suppose a mutator creates a new reference to a suspected object *z*. To do this, the mutator must traverse an old path to *z*; we can represent this path as a sequence:

$$\rightarrow f_0 \rightarrow f_1 \rightarrow \cdots \rightarrow f_n \rightarrow z$$

where f_k is an inter-site reference to an object in site Q_k: f_0 is a clean object and the rest are suspected objects.

The new reference may affect back traces visiting an outref o_n reachable from *z*. Such a trace views the path above as a sequence of iorefs:

$$\rightarrow i_0 \rightarrow o_0 \rightarrow i_1 \rightarrow o_1 \rightarrow \cdots \rightarrow i_n \rightarrow z \rightarrow o_n$$

where i_k and o_k are the inref and the outref on site Q_k; i_0 and o_0 are clean, while the rest are suspected. (Note that an intersite reference f_k corresponds to $o_{k-1} \rightarrow i_k$.) We show that such a back trace is safe even if the mutator deletes the existing path to *z*.

Part I

If a back trace reaches o_n *after* the mutator has traversed f_n, it will find the updated back information as described in Section 6.1. A potential problem occurs if the mutation is a remote copy; that is, *z* is a remote reference (corresponding to $o_n \rightarrow i_{n+1}$) and is copied to another remote site. A back trace might visit i_{n+1} before the insert message reaches there and visit o_n after the outref has been reverted to its suspected status. However, i_{n+1} must have received an insert message in the meanwhile due to the insert barrier, so the clean rule will set the return value of the trace to Live.

Part II

If a back trace reaches o_n *before* the mutator has traversed f_n, we show that the trace will return Live. The back trace traverses the path to *z* backwards. Intuitively, either the back trace will reach o_0, or the mutator and the back trace will cross each other and the resultant overlap will cause the trace to return Live. However, the proof is non-trivial because a back trace visiting o_n need not traverse the path in the reverse order. As mentioned in Section 4.4, a back trace may fork parallel branches to other iorefs and some branches may overtake others. It is therefore not obvious that there will be an ioref where the clean period due to the transfer barrier will overlap with the period when the back trace is active there.

For example, in Figure 6, a back trace from *g* will fork a branch to site *Q* and another to site *R*. In one scenario, the first branch might miss the mutator if it visits *R* before the mutator reaches there; it might then return Garbage if inref *e* is already visited by the second branch. In another scenario, the second branch might miss the mutator if the mutator has already traversed past the inter-site reference to *f* and site *R* does a local trace immediately after to revert the cleaned iorefs; the second branch might then find the path deleted and return Garbage. The two scenario are not possible simultaneously, however, because the first requires the second branch to visit *R* before the mutator, and the second requires it to visit *R* later.

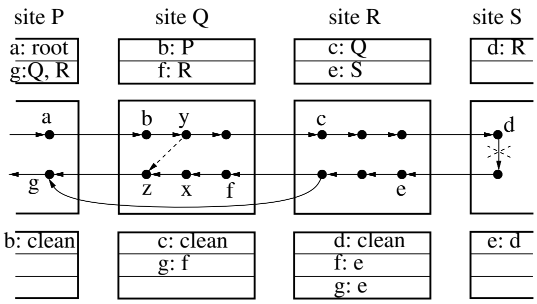

Figure 6: A problem case.

Below we prove that there must be an overlap in general.

The intuition is that if a branch of the back trace reaches o_k before the mutator reaches i_k, then either some branch must reach o_{k-1} before the mutator reaches i_{k-1}, or this branch must return Live. We use the following notation:

m_k The mutator message transferring f_k during its traversal.
b_k The message for the back call from i_{k+1} to o_k.
r_k The message from o_k to i_{k+1} for returning the back call.
$m.sent$ The time when message m was sent.
$m.received$ The time when message m was received.
$i_k.visited$ The time when i_k is first visited by the back trace.

The following relations hold:

R1 $m.sent < n.sent \Leftrightarrow m.received < n.received$, *i.e.*, we assume in-order delivery between a pair of sites.
R2 $m_{k-1}.received < m_k.sent$, for $1 \le k \le n$.
R3 $b_k.received < r_k.sent$, for all o_k visited by the back trace.
R4 $i_k.visited < r_k.sent$, for $1 \le k \le n$, provided $r_k.sent < m_k.received$. Before $m_k.received$, the mutator could not have deleted the path $i_k \to o_k$, so a back trace will return from o_k only after i_k has been visited.

Very little processing is associated with receiving a message before other messages may be received: The handler of a mutator message must atomically apply the transfer barrier. The handler of a back call must check if the ioref is clean. Thus, the critical sections involved are very short. We show that the following lemmas hold. Lemmas 1 and 2 specify periods when an inref or an outref must be clean. Lemmas 3 and 4 specify periods when a back trace must be active at an inref or an outref.

Lemma1 Inref i_k is clean right after $m_k.received$.
Proof The transfer barrier cleans i_k upon receiving m_k. □

Lemma2 Outref o_k is clean during $[m_k.received, m_{k+1}.sent]$.
Proof The transfer barrier cleans o_k upon receiving m_k. The outref remains clean until Q_k does the next local trace or until the mutator holds a variable v from where f_{k+1} is reachable. Thus, o_k must remain clean until the mutator transfers f_{k+1} to Q_{k+1}. □

Lemma3 If the back trace visits i_k before $m_k.received$, it must be active there during $[i_k.visited, min(m_k.received, r_{k-1}.received)]$.
Proof Before $m_k.received$, the mutator could not have deleted the link $o_{k-1} \to i_k$. Therefore, when the trace first visits i_k, it will be active there until it has received a response from o_{k-1}. □

Lemma4 If the back trace visits o_k, it must be active there during $[b_k.received, r_k.sent]$.
Proof True by definition. □

Theorem If a back trace visits o_k before the mutator visits i_k, then either the back trace will return Live or it will visit o_{k-1} before the mutator visits i_{k-1}, for $1 \le k \le n$.

Proof
Given $b_k.received < m_k.received$
if $r_k.sent > m_k.received$
  ⇒ $[b_k.received, r_k.sent]$ overlaps $m_k.received$
  ⇒ [trace active at o_k] overlaps [o_k clean] (Lemma 4, 2)
  ⇒ decide Live (Clean rule)
else
⇒ $r_k.sent < m_k.received$
⇒ $i_k.visited < m_k.received$ (R4)
if $r_{k-1}.received > m_k.received$
  ⇒ $[i_k.visited, r_{k-1}.received]$ overlaps $m_k.received$
  ⇒ [trace active at i_k] overlaps [i_k clean] (Lemma 3, 1)
  ⇒ decide Live (Clean rule)
else
⇒ $r_{k-1}.received < m_k.received$
⇒ $r_{k-1}.sent < m_k.sent$ (R1)
if $r_{k-1}.sent > m_{k-1}.received$
  ⇒ $r_{k-1}.sent$ overlaps $[m_{k-1}.received, m.sent]$
  ⇒ [trace active at o_{k-1}] overlaps [o_{k-1} clean] (Lemma 4, 2)
  ⇒ decide Live (Clean rule)
else
⇒ $r_{k-1}.sent < m_{k-1}.received$
⇒ $b_{k-1}.received < m_{k-1}.received$ (R3)
□

From the theorem above, if the back trace visits o_n before the mutator visited i_n, it must return Live or visit o_0. Since o_0 is clean, the trace will return Live.

We showed above that safety is ensured upon a single mutation. We claim (without giving proof) that safety is ensured upon multiple concurrent mutations as well.

7 Related Work

Previous schemes for collecting inter-site garbage cycles may be categorized as follows.

Global Tracing
A complementary global trace is conducted periodically to collect cyclic garbage, while other garbage is collected more quickly by local tracing [Ali85, JJ92]. The drawback of global tracing is that it may not complete in a system with a large number of faulty sites.

Hughes's algorithm propagates *timestamps* from inrefs to outrefs and collects objects timestamped below a certain global threshold [Hug85]. The persistent roots always have the current time, and a global algorithm is used to compute the threshold. The advantage of using timestamps over mark bits is that, in effect, multiple marking phases can proceed concurrently. However, a single site can hold down the global threshold, prohibiting garbage collection in the entire system.

Central Service
Beckerle and Ekanadham proposed that each site send inref-outref reachability information to a fixed site, which uses the information to detect inter-site garbage cycles [BE86]. However, the fixed site becomes a performance

and fault tolerance bottleneck.

Ladin and Liskov proposed a logically central but physically replicated service that tracks inter-site references and uses Hughes's algorithm to collect cycles [LL92]. The central service avoids the need for a distributed algorithm to compute the global threshold. However, cycle collection still depends on timely correspondence between the service and *all* sites in the system.

Subgraph Tracing

The drawbacks of global tracing can be alleviated by first delineating a *subgraph* of objects reachable from an object suspected to be cyclic garbage. Another distributed trace is then conducted within this subgraph; this trace treats all objects referenced from outside the subgraph as roots. All subgraph objects not visited by this trace are collected. Note that a garbage cycle might point to live objects, and the associated subgraph would include all such objects. Thus, the scheme does not possess the locality property.

Lins et al. proposed such a scheme as *cyclic reference counting* in a system that used reference counting for local collection instead of local tracing [LJ93]. This scheme requires two distributed traces over objects in a subgraph. Jones and Lins improved the scheme such that multiple sites could conduct traces in parallel, but it required global synchronization between sites [JL92].

Group Tracing

Another method to alleviate the drawbacks of global tracing is to trace within a *group* of selected sites, thus collecting garbage cycles within the group. A group trace treats all references from outside the group as roots.

The problem with group tracing is configuring groups in order to collect all inter-site cycles. Lang et al. proposed using a tree-like hierarchy of embedded groups [LQP92]. This ensures that each cycle is covered by some group, but the smallest group covering, say, a two-site cycle may contain many more sites. Further, the policy for forming and disbanding groups dynamically is unclear.

Maeda et al. proposed forming groups using subgraph tracing [MKI+95]. A group consists of sites reached transitively from some objects suspected to be cyclic garbage. This work was done in the context of local tracing and inter-site weighted reference counting. Rodrigues and Jones proposed an improved scheme in the context of inter-site reference listing [RJ96]. One drawback of this approach is that multiple sites on the same cycle may initiate separate groups simultaneously, which would fail to collect the cycle. Conversely, a group may include more sites than necessary because a garbage cycle may point to chains of garbage or live objects. Another problem is that group-wide tracing might never collect all cycles. Since a group-wide trace is a relatively long operation involving multiple local traces, it is not feasible to cover all garbage cycles in a system with many sites.

Schemes with Locality

Few schemes for collecting cyclic garbage have the locality property. The oldest among these is migration. The idea is to converge a suspected distributed garbage cycle to a single site: if it is indeed a garbage cycle, it will be collected by local tracing [Bis77]. Since migration is expensive, it is crucial to use a good heuristic for finding suspects; we proposed the distance heuristic in this context earlier [ML95]. However, some systems do not support migration due to security or autonomy constraints or due to heterogeneous architecture. Those that do must patch references to migrated objects. Shapiro et al. suggested *virtual* migration [SGP90]. Here, an object changes its logical space without migrating physically. However, a logical space may span a number of sites, so local tracing must involve inter-site tracing messages.

Schelvis proposed forwarding local-reachability information along outgoing inter-site references [Sch89]. This algorithm is intricate and difficult to understand; however, some of its problems are apparent. The algorithm requires full reachability information between all inrefs and outrefs (not just suspected ones). An inref i contains a set of *paths* instead of source sites; each path indicates a sequence of inrefs leading to i. Collecting a cycle located on N sites might take $O(N^3)$ messages. Recently, Louboutin presented an improved scheme that sends only $O(N)$ messages [LC97]. However, it too requires full inref-outref reachability information, and its space overhead is larger: each inref i stores a set of vector timestamps; each vector corresponds to a path i is reachable from.

Back tracing was proposed earlier by Fuchs [Fuc95]. However, this proposal assumed that inverse information was available for references, and it ignored problems due to concurrent mutations and forward local traces. A discussion on back tracing, conducted independently of our work, is found in the archives of the mailing list gclist@iecc.com at ftp://iecc.com/pub/gclist/gclist-0596.

8 Summary and Conclusions

We have presented a scheme for collecting distributed garbage cycles. The scheme has two parts: The first uses the distance heuristic to find objects highly suspected to be cyclic garbage. The second traces back from such an object to check if it is reachable from a clean object.

A back trace spreads quickly by traversing inrefs and outrefs rather than individual references. This requires the use of reachability information between suspected inrefs and outrefs. We presented an efficient technique to compute this information during local forward tracing without tracing objects multiple times. Storing this information requires $O(n_i \times n_o)$ space, where n_i and n_o are the number of suspected inrefs and suspected outrefs.

A site starts a back trace from a suspected outref when the back trace is highly likely to confirm it as garbage. We proposed a technique that reduces abortive attempts by waiting until other inrefs and outrefs on the cycle are likely to have been suspected as well. After a back trace completes, the initiator site sends the outcome to other participating sites. The trace requires very little processing at each site, and it sends $2E + N$ small messages where E is the number of inter-site references traversed and N is the number of participants. Its message complexity is lower than that of any previous scheme with locality. Multiple back traces may be triggered concurrently; however, back traces spread quickly enough that overlap is not likely.

Back traces are conducted concurrently with mutators and forward local traces. We use two barriers to keep the back information conservatively safe, yet we ensure that the barriers do not prohibit the collection of garbage cycles. These barriers are applied only when a mutator transfers references between sites and are inexpensive. Keeping the back information up to date at each site is not sufficient because a back trace may see an inconsistent distributed view. We ensure that back traces see a safe view using short critical sections and without sending additional messages.

Back tracing has some drawbacks. First, it is more complex than schemes based on migrating the suspects. Second, it requires full reachability information between suspected inrefs and suspected outrefs. Computing this information requires a modified depth-first traversal of the suspected objects. Therefore, breadth-first copying collectors would need to perform a separate traversal for these objects.

Despite these drawbacks, back tracing is an attractive scheme because it preserves the locality property: the collection of a garbage cycle involves only the sites containing it. It does not migrate objects and its overheads are lower than other schemes with locality. Furthermore, it collects all distributed garbage cycles. We designed this scheme for implementation in a large, distributed object database [LAC+96]. It is suitable for emerging distributed object systems that must scale to a large number of sites.

Acknowledgements

We are grateful to Chandrasekhar Boyapati and Andrew Myers for proofreading this paper. Boyapati pointed out a problem with our handling of iorefs deleted by one back trace when another trace is active there. We thank the anonymous referees for their comments.

References

[Ali85] K. A. M. Ali. Garbage collection schemes for distributed storage systems. In *Proc. Workshop on Implementation of Functional Languages*, pages 422–428, 1985.

[BE86] M. J. Beckerle and K. Ekanadham. Distributed garbage collection with no global synchronisation. Research Report RC 11667 (#52377), IBM, 1986.

[BEN+93] A. Birrell, D. Evers, G. Nelson, S. Owicki, and E. Wobber. Distributed garbage collection for network objects. Technical Report 116, Digital Systems Research Center, 1993.

[Bev87] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187. Springer-Verlag, 1987.

[Bis77] P. B. Bishop. Computer systems with a very large address space and garbage collection. Technical Report MIT/LCS/TR–178, MIT, 1977.

[Fuc95] M. Fuchs. Garbage collection on an open network. In H. Baker, editor, *Proc. IWMM*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[HK82] P. R. Hudak and R. M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 168–178. ACM Press, 1982.

[Hug85] R. J. M. Hughes. A distributed garbage collection algorithm. In *Proc. 1985 FPCA*, volume 201 of *Lecture Notes in Computer Science*, pages 256–272. Springer-Verlag, 1985.

[JJ92] N.-C. Juul and E. Jul. Comprehensive and robust garbage collection in a distributed system. In *Proc. IWMM*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[JL92] R. E. Jones and R. D. Lins. Cyclic weighted reference counting without delay. Technical Report 28–92, Computing Laboratory, The University of Kent at Canterbury, 1992.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. 1996 SIGMOD*, pages 318–329. ACM Press, 1996.

[LC97] S. Louboutin and V. Cahill. Comprehensive distributed garbage collection by tracking the causal dependencies of relevant mutator events. In *Proc. ICDCS*. IEEE Press, 1997.

[LJ93] R. D. Lins and R. E. Jones. Cyclic weighted reference counting. In K. Boyanov, editor, *Proc. Workshop on Parallel and Distributed Processing*. North Holland, 1993.

[LL92] R. Ladin and B. Liskov. Garbage collection of a distributed heap. In *Proc. ICDCS*. IEEE Press, 1992.

[LQP92] B. Lang, C. Queinniec, and J. Piquer. Garbage collecting the world. In *Proc. POPL '92*, pages 39–50. ACM Press, 1992.

[MKI+95] M. Maeda, H. Konaka, Y. Ishikawa, T. T. iyo, A. Hori, and J. Nolte. On-the-fly global garbage collection based on partly mark-sweep. In H. Baker, editor, *Proc. IWMM*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[ML94] U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server object-oriented database. In *Proc. PDIS*. IEEE Press, 1994.

[ML95] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proc. PODC*, pages 57–63, 1995.

[RJ96] H. Rodrigues and R. Jones. A cyclic distributed garbage collector for network objects. In *Proc. 10th Workshop on Distributed Algorithms*, 1996.

[Sch89] M. Schelvis. Incremental distribution of timestamp packets — a new approach to distributed garbage collection. *ACM SIGPLAN Notices*, 24(10):37–48, 1989.

[SDP92] M. Shapiro, P. Dickman, and D. Plainfossé. Robust, distributed references and acyclic garbage collection. In *Proc. PODC*, 1992.

[SGP90] M. Shapiro, O. Gruber, and D. Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapports de Recherche 1320, INRIA-Rocquencourt, 1990.

[Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2), 1972.