Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme

Benjamin Goldberg
Department of Computer Science
New York University
251 Mercer Street
New York, NY 10012

Abstract

This paper describes **generational reference counting**, a new distributed storage reclamation scheme for loosely-coupled multiprocessors. It has a significantly lower communication overhead than distributed versions of conventional reference counting. Although generational reference counting has greater computational and space requirements than ordinary reference counting, it may provide a significant saving in overall execution time on machines in which message passing is expensive.

The communication overhead for generational reference counting is one message for each copy of an interprocessor reference (pointer). Unlike conventional reference counting, when a reference to an object is copied no message is sent to the processor on which the object lies. A message is sent only when a reference is discarded. Unfortunately, generational reference counting shares conventional reference counting's inability to reclaim cyclical structures.

In this paper, we present the generational reference counting algorithm, prove it correct, and discuss some refinements that make it more efficient. We also compare it with weighted reference counting, another distributed reference counting scheme described in the literature.

1 Introduction

The emergence of LISP and functional language implementations for multiprocessors has encouraged the study of efficient distributed methods for storage reclamation. New reclamation algorithms are especially important for loosely-coupled (distributed memory) multiprocessors in which each processor is responsible for allocating and reclaiming structures residing in its local memory. The reclamation schemes that were suitable for uniprocessors must be altered to work on loosely-coupled multiprocessors. Additional message passing and synchronization is generally required to maintain the correctness of these schemes.

Many current loosely-coupled multiprocessors have a very large communication cost associated with sending a message. Each message may consume hundreds or even thousands of processor cycles. For many distributed algorithms, it is worth sacrificing space and computation in order to avoid communication.

In this paper we present a distributed storage reclamation algorithm that makes such a sacrifice. The algorithm, called **generational reference counting** or **GRC**, is based on the conventional reference counting scheme. Although it has somewhat greater storage and computational requirements then reference counting, GRC has one-third the communication overhead of distributed reference counting.

The GRC algorithm was developed independently of the work on **weighted reference counts**[3,17] and has similar properties, namely a communication overhead of one message per interprocessor reference and extra space associated with each reference.

The relative merits of reference counting versus garbage collection have been well argued (see [5] for a survey of the field). A method based on reference counting is of interest to us for the usual reasons:

- It is incremental: Storage reclamation occurs throughout the computation. The computation does not have to be interrupted for a significant

A preliminary draft of this paper was presented to the Fourth Conference on Hypercubes, Concurrent Computers, and Applications.

period of time for storage reclamation to proceed. This is especially valuable in real time applications.

- The storage reclamation overhead depends on the amount of garbage in the system. This is not the case with mark/sweep garbage collection. If memory is nearly full (with useful data), garbage collection will have to be invoked often and at great expense.
- *Reference counting, even in loosely-coupled multiprocessors,* is relatively simple. This makes storage reclamation easier to implement and prove correct.

A number of interesting algorithms based on mark/sweep garbage collection have been proposed to solve the problems listed above. In particular, these algorithms have striven to make mark/sweep garbage collection incremental, both for uniprocessor and multiprocessor architectures [8,1,5,12,2,13]. These algorithms, however, have tended to be rather complex and require elaborate proofs of correctness. In addition, some incremental mark/sweep garbage collection schemes require at least two concurrent processes, a mutator and a collector. In current loosely-coupled multiprocessors, both of these processes would have to be implemented (via context-switching) by each processor in the system.

We find distributed reference counting especially attractive for its simplicity. The disadvantage of reference counting is that it cannot collect cyclical structures. Although the algorithm we present in this paper has the same problem, the modifications that have been proposed [9,4,6] for reclaiming cyclical structures using uniprocessor reference counting are also applicable to generational reference counting (with the concomitant loss of simplicity).

2 Distributed Reference Counting

Distributed reference counting is a simple extension to uniprocessor reference counting in which the each object o keeps count of the number of outstanding references (pointers) to it. When a reference to o (which we will refer to as an *o-reference* after [16]) is copied, o's reference count is incremented. When an o-reference is discarded, o's reference count is decremented. If o's reference count becomes zero, then o is garbage and can be reclaimed.

On a loosely-coupled system, the creation of a new o-reference on a remote processor requires that a message be sent to o in order to increment o's reference count. Likewise, if a remote reference is discarded then a decrement message must be sent. However, this simple extension of reference counting does not preserve the correctness of uniprocessor reference counting. It is possible for an object to be reclaimed while references to it still exist.

Suppose an object o resides on a processor P_o and the only o-reference p resides on a remote processor P_p. In this case o's reference count will be one. Now suppose that p is copied to create a new o-reference q that resides on a third processor P_q. When p is copied, P_p sends an increment message, call it I_p, to P_o. At some point after q arrives at P_q, q will be discarded. When q is discarded, P_q sends a decrement message D_q to P_o. Likewise, a decrement message D_p will eventually be sent to P_o when p is discarded. The following two scenarios result in o's reference count being prematurely set to zero.

- If q is discarded soon after being created, D_q may arrive at P_o before I_p, even though I_p was sent first. A decrement is performed on o's reference count making its value zero even though p still exists.
- If p is discarded soon after q is created, D_p could arrive before I_p. This can only happen in systems in which messages do not necessarily arrive in the order in which they are sent. This would result in o's reference count being set to zero even though q still exists.

A solution to these problems was published in [16]. The protocol described in that paper is as follows:

- In order to solve the second problem listed above, the protocol assumes that the architecture preserves the order of messages sent by each processor. That is, if two messages are sent from processor P_i to processor P_j, they will arrive at P_j in the order in which P_i sent them. This assumption can be enforced either if the system provides fixed routing of messages or if messages have tags that indicate the order in which they are sent.
- When p is copied, P_p sends a *create* message to P_o indicating that a new o-reference has been created. When q is created on P_q, P_q sends an *acknowledge* message to P_o. Later, when q is discarded, P_q sends a *delete* message to P_o (notice that the delete message cannot be received before the acknowledge message).

  The object o keeps track of the number of create, acknowledge, and delete messages it receives and is only reclaimed when an equal number of each has been received.

While this protocol does provide a correct distributed reference counting scheme, it requires an overhead of three messages per interprocessor reference.

2.1 Weighted Reference Counts

Weighted Reference Counting (WRC) is a distributed reference counting scheme recently presented in [3] and [17] (although Watson&Watson attribute it to [18]) in which only one message is required for each interprocessor reference. The WRC scheme associates a weight with each reference such that the sum of the weights of all references to an object is a constant. Each reference must have an extra field that contains the weight of the reference. The scheme works as follows:

- When an object o is created, its reference count is initially set to some value w. The weight field of the original o-reference also is set to w.
- Each time a reference is copied, the new reference takes half of the weight of the old reference. That is, when a reference p with a weight m is copied to form a reference q, then the weight of each of p and q becomes $m/2$. In this way the sum of the weights of all references to a given object is w.
- When an o-reference is discarded, a *delete* message containing the weight of the deleted reference is sent to o. This weight is subtracted from o's reference count. When o's reference count becomes zero, o can be reclaimed.

2.2 Implementation Issues

A significant improvement (described in[3],[17])in the space efficiency of the WRC scheme comes from observing that a reference's weight is always a power of two. Thus, instead of storing the whole weight of a reference, the log of the weight can be stored. When a delete message containing such a value is received by the object, it must be converted to the actual value (by a simple shift) before being subtracted from the object's reference count. This provides an important reduction in the space on each reference required to store its weight.

In the WRC scheme, the weight of a reference can underflow if it has a weight of one and is copied. When a reference a, which points to an object o, has a weight of one and is copied to form a reference b, an indirection pointer i is created to point to the o. This indirection pointer includes a reference count field (just like other objects) initialized to w. Both a and b are modified to point to i and are assigned weights of $w/2$ each. Thus, a and b (and any of their descendants) access o through i, using an extra level of indirection. When any of these references are discarded, the delete messages are sent to i and i's reference count is reduced appropriately. When i's reference count becomes zero a delete message containing a weight of one (a's original weight) is sent to the object.

An unfortunate aspect of the use of an indirection pointer is that a reference, its indirection pointer, and the object may reside on different processors. If so, accessing the object would require an extra message. In section 4.2 we describe how this can be avoided (but only at the price of extra space).

3 Generational Reference Counting

Generational reference counting (GRC) was developed independently of WRC and has similar communication performance. GRC is a reference counting scheme in which each each reference has a *generation* associated with it. The original reference to an object o is a zero generation reference, any reference copied from the original reference is a first generation reference, and so on. In general, any copy of an ith generation reference is an $(i+1)$ generation reference. (We use the notation G_i as shorthand for the phrase "ith generation")

Each object o contains a table, called a *ledger*, which keeps track of the number of outstanding o-references of each generation. If o's ledger indicates that there are no outstanding o-references of any generation then o can be reclaimed.

3.1 The GRC Algorithm

Each o-reference contains two additional fields: a *generation* field and a *count* field. The generation field identifies the generation of the reference and the count field records the number of references copied from this particular reference.

The ledger contained in each object o is an array of integers. The ith element of the array contains information about the number of outstanding G_i o-references.

Generational reference counting is performed as follows:

- When an object o is created on processor P_o, the initial o-reference p is initialized as follows:

  ```
  p.generation := 0
  p.count := 0
  ```

  and o's ledger is initialized as follows:

```
o.ledger[0] := 1
o.ledger[i] := 0,   for all i ≥ 1
```

- When an o-reference p is copied the new reference q is initialized as follows:

  ```
  q.generation := p.generation + 1
  q.count := 0
  ```

  In addition, p.count is incremented:

  ```
  p.count := p.count + 1
  ```

- When an o-reference p is discarded, a *discard* message is sent to o containing the values of p.**generation** and p.**count**.
- When a discard message to o is received, in which the generation field has the value i and the count field has the value c, the following actions are performed:

  ```
  o.ledger[i] := o.ledger[i] - 1
  o.ledger[i + 1] := o.ledger[i + 1] + c
  ```

  When every element of the ledger becomes zero (for all $i \geq 0$, o.ledger[i] = 0), o can be reclaimed.

Notice that some elements of the ledger may hold negative values. This can occur when discard messages for $G_{(i+1)}$ references are received before discard messages for the G_i references. In this case the $(i+1)$th element of the ledger might be negative. We prove below that GRC is correct for any order in which discard messages are received.

3.2 Correctness Proof

In order to show that the GRC algorithm is correct, we need to prove two things:

- If o.ledger[i] = 0, for all $i \geq 0$, then o can be reclaimed.
- If a discard message for every o-reference has been received, then o.ledger[i] = 0 for all $i \geq 0$.

These properties must be independent of the order in which the discard messages are received.

Lemma 1 *After all G_i o-references have been discarded, the sum of the counts in their discard messages is equal to the total number of $G_{(i+1)}$ o-references created.*

This follows easily from the following observations about each G_i reference p.

- Each time p is copied in order to create a $G_{(i+1)}$ reference, p's count is incremented. Since p's count was initialized to 0, p's count represents the number of G_{i+1} references created from p.
- A discard message for p is only sent when p is discarded and no more copies can be made from p. Therefore the count in p's discard message gives an accurate count of the $G_{(i+1)}$ references copied from p.
- A $G_{(i+1)}$ reference can only be created by copying a G_i reference.

Lemma 2 *When a discard message for every o-reference has been received, o can be reclaimed.*

A discard message for an o-reference p is sent only when p is discarded. Therefore, if a discard message for every reference has been received then all references must have been discarded and o can be safely reclaimed.

Lemma 3 *At any point during the lifetime of o, for each $i \geq 0$*

$$o.\mathbf{ledger}[i] = C_{(i-1)} - N_i$$

where $C_{(i-1)}$ is the sum of the counts in the discard messages received for $G_{(i-1)}$ o-references and N_i is the number of discard messages received for G_i o-references.

This follows directly from the GRC algorithm. Each time a discard message for a $G_{(i-1)}$ o-reference is received, o.ledger[i] is increased by the count contained in the message. Each time a discard message for a G_i o-reference is received, o.ledger[i] is decremented by one.

Theorem 1 *If o.ledger[i] = 0 for all $i \geq 0$ then a discard message has been received from every o-reference.*

We proceed by induction on the generation i of each o-reference:

- A discard message must have been received from the G_0 (original) o-reference in order for o.ledger[0] to be zero. A discard message from any other o-reference cannot affect the value of o.ledger[0] and the value of o.ledger[0] was initially one.
- Assuming that a discard message for each G_i reference, $0 \leq i \leq k$, has been received, we must show that a discard message for every $G_{(k+1)}$ reference has also been received.

  Since o.ledger[k+1] = 0, the number of discard messages received for $G_{(k+1)}$ references is equal to the sum of the counts received in discard messages for G_k references (by lemma 3). And, since discard messages for all G_k references have been received, the sum of their counts is the total number of generation $G_{(k+1)}$ created (by lemma 1).

Therefore, discard messages for all $G_{(k+1)}$ references have been received.

Theorem 2 *If discard messages for all references have been received, then o.ledger[i] = 0 for all $i \geq 0$.*

We prove the contrapositive: If there is a $j \geq 0$ such that o.ledger[j]$\neq$ 0 then there is a o-reference for which a discard message has not been received.

If o.ledger[0]$\neq$ 0 then a discard message from the G_0 reference has not been received. The effect of receiving a discard message from the G_0 reference is to decrement o.ledger[0] to zero.

Otherwise, if there exists a $j > 0$ such that o.ledger[j]$\neq$ 0 then there two possibilities:

- If o.ledger[j]$<$ 0 then (by lemma 3) the number of discard messages received for G_j references is greater then the sum of the counts in discard messages for $G_{(j-1)}$ references. By lemma 1, there must be at least one $G_{(j-1)}$ reference for which a discard message has not been received.
- If o.ledger[j]$>$ 0 then (by lemma 3) the sum of the counts in discard messages for $G_{(j-1)}$ references is greater then the number of discard messages for G_j references. By lemma 1, there must be at least one G_j reference for which a discard message has not been received.

Theorems 1 and 2, along with lemma 2, prove the correctness of the GRC algorithm.

3.3 Implementation Issues

Just as weighted references are susceptible to underflow in the WRC scheme, references and ledgers can overflow in the GRC scheme. In this section, we describe how such overflow can be handled.

In a ledger, the field associated with a particular generation may overflow. In addition, the number of generations of references may exceed the number of fields in the ledger. In either case, more space must be found to increase the number of generation fields or to increase the size of each generation field. Several alternatives are possible, including:

- If the ledger is sufficiently large, it can be overwritten with the address of a new, larger ledger. When a discard message arrives, the new ledger is accessed through the old ledger using a level of indirection.
- Rather than allocate a whole new ledger, new space is allocated to serve as an extension of the old ledger. If the number of reference generations has overflowed, then the new extension will contain additional generation fields of the same size. If the count field for a generation has overflowed, the ledger will have to be restructured so that together the ledger and the extension contain the same number of fields as before, with each field being larger. The extension would reside in a hash table and be found by hashing the address of the object (similar to the Deutsch & Bobrow scheme [7] described in section 5.2).

In both cases, the new ledger would need a tag to indicate how the old ledger was extended (either number or size of generation fields). Likewise, the object must contain a bit to indicate that its ledger had been extended.

A similar method can be used to handle overflow of the generation or count fields of a reference. The reference can be overwritten to point to an indirection pointer with larger generation and/or count fields. Unlike WRC, this indirection pointer will always be on the same processor as the reference, thus adding no extra communication overhead. Alternatively, a larger generation field and/or count field can be stored in a hash table, hashed on the address of the reference.

4 Comparing WRC and GRC

4.1 Space Efficiency

Weighted reference counting appears significantly more space-efficient than generational reference counting. We examine the number of references that can be supported if n bits are associated with an object in each scheme.

In WRC, a n bit reference count allows a total weight of 2^n. This allows there to be at most 2^n references (each with a weight of 1). It would be extremely unlikely, however, that the reference weights would be distributed so evenly (An analytical study of the average number of reference copies that will be made before an indirection pointer is required is beyond the scope of this paper). In the worst case, a single pointer would be copied many times, limiting the number of references to n. A reference's weight field always occupies log n bits.

It is much more difficult to analyze the number of references that can be created using an n bit ledger in the GRC scheme. There are several reasons for this difficulty:

- Whether a particular field of the ledger overflows or not may depend on the order of arrival of discard messages. By lemma 3, even if the number of ith generation references created is greater than can be stored in the ith field of the ledger, overflow will occur only if the difference between

the sum of the counts in the $(i-1)$ generation discard messages and the number of ith generation discard messages is too large to store in the ith field of the ledger.

- The size of the count field in a reference does not necessarily depend on the size of the fields in the ledger. It may be reasonable for the count field in a reference to be smaller than each field of the ledger because the ledger stores the total number of known references of each generation. On the other hand, a discard message containing a count too large for the corresponding ledger field may not necessarily overflow the ledger, for the reason mentioned above. In this case, it may be reasonable to give a reference a larger count field than in the ledger.
- The number of generation fields may be chosen according to the language, program, or application being executed. If there are g fields, then each field will be n/g bits wide (Again, an analytical study of the expected number of generations and number of references per generation is beyond the scope of this paper). The choice of g depends on the expected width (i.e. number of times each reference is copied) and the expected depth (i.e. length of a chain of copies) of reference copying.

Assuming that discard messages will arrive in the worst possible order, a ledger containing g fields of n/g bits each (with each field able to store values between $-2^{(n/g)-1}$ and $2^{(n/g)-1}$) can record the creation of approximately $g * 2^{(n/g)-1}$ references at best without overflowing. Although dependent on g, this is substantially less than the best (but unlikely) case for WRC. In the worst case the ledger will overflow when MIN(g, $2^{(n/g)-1}$) references are created, when either the number of generations or the count within a generation overflows. Again, this is smaller than the worst case for WRC.

The space efficiency advantage that WRC holds over GRC is perhaps the strongest argument for using WRC. If, however, communication costs is of much greater concern, GRC may be preferable for reasons discussed in the next section.

4.2 Communication Efficiency

The GRC scheme always has a communication overhead of at most one message per interprocessor reference. This is independent of whether or not a generation or count field had to be extended in a reference or ledger. In the WRC scheme, unless indirection pointers are required to handle underflow, the communication overhead is also one message per interprocessor reference.

If indirection pointers are used in the WRC scheme, two messages may be required to access an object, one to the indirection pointer containing the address of the object and one to the object itself. If reducing the communication overhead of storage reclamation is the overriding concern of the implementor, the GRC scheme may prove preferable. However, given the space efficiency advantage of WRC, it may be sufficient to assign each object a large enough weight such that the chances of requiring indirection pointers are slim.

Notice that reference weight underflow in WRC can be handled in the same manner as overflow in GRC (although this hasn't been discussed in the WRC literature). Extra space can be allocated for a reference's weight when underflow occurs. A reference's weight will still be 2^n, but now negative values of n can be stored. When the object receives a delete message containing a negative exponent, the reference count field must be extended to the appropriate precision. This would avoid the extra communication of the indirection pointer scheme.

5 Refinements to the GRC Scheme

The following modifications to the GRC algorithm can be used to reduce its communication and memory requirements. These refinements can just as easily be applied to the WRC scheme.

5.1 Using escape information

To further reduce the communication requirements of GRC, we enlist the support of the compiler in order to determine when a discard message can be avoided. The compiler technique that we will rely on to improve the performance of GRC is called *escape analysis*. There have been several papers [14,10,15] written on the techniques for performing escape analysis and thus we will concentrate on applying it to generational reference counting. Our use of escape analysis for storage reclamation is related to (although more general than) the *call-graph reclamation scheme* presented in [11,10].

Traditionally, escape analysis has been used to determine when an object is returned ("escapes") from the procedure in which it was created. For example, escape analysis has been used [15] to determine if the extent (lifetime) of a closure (e.g. the result of a lambda expression in LISP) exceeds the extent of its

defining procedure. This is useful in deciding whether the closure must be allocated in the heap or may be allocated on the stack.

This same analysis can be applied to determining if the lifetime of a reference exceeds the lifetime of the reference it was copied from. We first need to define some terms:

A reference p is a *descendant* of a reference q if p was copied from q or if p was copied from another descendant of q. If so, then q is said to be an *ancestor* of p. A reference q is said to have a greater *extent*, or *lifetime*, then a descendant p if and only if q is discarded after p. If a reference p has a greater lifetime then all of its descendants, then we say p is *dominant*.

The compiler, via escape analysis, can in many cases determine which references are dominant. We are interested in this information for the following reason: If a reference p is dominant, then no discard message should be sent for any descendant of p. This is because p's descendants will have already been discarded by the time p is. The referenced object can remain unaware of the fact that p's descendants were ever created and discarded. Of course, if p is dominated by another reference, then no discard message should be sent for p.

We modify the GRC algorithm to use the escape information provided by the compiler. The modification consists of initializing a dominated reference's generation field to -1 to indicate that it is dominated by another reference. When a dominated reference is discarded, no discard message is sent.

When a reference p is copied to create a new reference q, one of the following occurs:

- If the compiler has determined that p is dominant, then the generation field of q is initialized to -1 and the count field is left uninitialized (to save an instruction).
- If p's generation field is -1 then q's generation field is initialized to -1. By definition, a descendant of a dominated reference p is also dominated.
- Otherwise, p is not (detectably) dominated and q's generation and count fields are initialized according to the GRC algorithm presented in section 3.1. The count field of p is incremented as usual.

When a reference p is discarded, one of the following actions are performed:

- If p's generation field is -1, then no action is performed.
- Otherwise, a discard message is sent as specified by the GRC algorithm. If p happened to be dominant (and not dominated) then its count field will have remained at zero.

In this way, no discard message is sent by any (detectably) dominated reference. While we lack any empirical evidence on the savings gained from this refinement of the GRC algorithm, we suspect it may be substantial.

Escape information can be exploited in WRC in a similar manner. A dominating reference would simply not share its weight with any of its copies. The dominated copies (and their descendants) would be given a zero weight and would not report their discards to the object. In order to give a zero weight to a reference, an extra bit is required (since a reference's weight normally must be a power of two).

We are still engaged in finding an effective algorithm for performing escape analysis to determine which references are dominant. We hope to describe such an algorithm in a future paper.

5.2 Using Multiple Reference Tables

In order to reduce the space requirements of uniprocessor reference counting, Deutsch & Bobrow[7] described a scheme using hash tables to store the reference counts of only those objects that have multiple outstanding references. The reference count for each multiply referenced object resides in a *multiple reference table* or MRT. Those objects that do not have reference counts in the MRT are assumed to have a reference count of one.

Each object contains an extra bit that is used to indicate whether a reference count for that object is contained in the MRT. Initially, that bit is set to zero to indicate that no reference count exists for the new object. When a reference to an object o is copied, one of the following actions is performed:

- If no reference count for o exists in the MRT then a new element, hashed on the address of o, is placed in the MRT. This new element contains a reference count whose value is two.
- If o's reference count does exist in the MRT, then that reference count is incremented.

When a reference to o is discarded, one of the following actions is performed:

- If o's reference count exists in the MRT then the reference count is decremented. If the new value of the reference count is one then the reference count is removed from the MRT.
- If no reference count exists for o in the MRT then o can be reclaimed.

This scheme can be modified to work with generational reference counting. An MRT is used to hold ledgers for those objects whose original (0th generation) reference was copied. Once an object's ledger is created in the MRT, it remains there until the object is reclaimed. This is because there is no way to tell if only one remaining reference to an object exists.

The algorithm is as follows. Each object is created with a single extra bit that indicates whether a ledger exists for that object in the MRT (initially this bit is zero). When a discard message arrives for an object o one of the following actions are performed:

- If no ledger exists for o in the MRT and the generation and count fields of the discard message are both zero (indicating that the original reference was discarded without being copied) then o is reclaimed.
- If no ledger exists for o in the MRT and either the generation or count field of the discard message is non-zero, then a ledger is created in the MRT for o and the appropriate fields of the ledger are initialized according to the GRC algorithm. The extra bit in o is modified to indicate that a ledger for o exists in the MRT.
- Otherwise, if a ledger exists for o in the MRT, then the ledger is modified according to the GRC algorithm. If all of the elements of the ledger are zero then o can be reclaimed.

In this way, only those objects that have had multiple references will have a ledger.

It may be worthwhile to use a similar method for references. Those references that have been copied and need to support generational reference counting could have their generation and count fields contained in a hash table similar to the MRT. Thus each reference p would contain a single bit that would be set if the p is copied or if p had been copied from another reference. When a reference is discarded, if the bit is zero then a discard message with generation and count fields set to zero is sent. Otherwise, the appropriate generation and count values for the reference are found in the reference table and are sent in a discard message.

The Deutsch and Bobrow method can be used in the WRC scheme by allocating an object's weighted reference count in the MRT only if a discard message is received containing a weight less than the original weight w. An object with no reference count field is assumed to have a reference count of w.

6 Conclusion

We have presented a distributed storage reclamation scheme based on reference counting that has significantly lower communication overhead than the multiprocessor extension of conventional reference counting. A price is paid, both in computational and space complexity, for the reduction in communication. Whether this price is worth paying depends on the communication behavior of a particular machine. Experimental evidence is needed to determine if generational reference counting is feasible.

In addition, it is not clear if there is any advantage to using the generational reference counting scheme instead of the weighted reference counting scheme. Weighted reference counting appears to be more space efficient, although further analytical and experimental study is required to fully understand the behavior of each scheme.

References

[1] H.G. Baker, Jr. List processing in real time on a serial computer. *CACM*, 21(4):280–294, April 1978.

[2] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *Trans. on Prog. Lang. and Sys.*, 6(3):333–344, July 1974.

[3] D.I. Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe*, pages 176–187, Springer-Verlag LNCS 259, June 1987.

[4] D.R. Brownbridge. Cyclic reference counting for combinator machines. In *Functional Programming Languages and Computer Architecture*, pages 273–288, Springer-Verlag LNCS 201, September 1985.

[5] J. Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[6] A. Deb. Parallel garbage collection for graph machines. In *Proceedings of the Santa Fe Workshop on Graph Reduction*, pages 252–264, Springer-Verlag LNCS 279, September 1986.

[7] L.P. Deutsch and D.G. Bobrow. An efficient incremental automatic garbage collector. *CACM*, 19(9):522–526, September 1976.

[8] E.W. Dijkstra, L. Lamport, A.J. Martin, and E.M.F. Steffens. On-the-fly garbage collection:

an exercise in cooperation. *Commun. ACM*, 21(11):966–975, November 1978.

[9] D.P. Friedman and D.S. Wise. Reference counting can manage the circular environments of mutual recursion. *Inf. Process. Lett.*, 8(1):921–930, January 1979.

[10] P. Hudak. *Call-graph reclamation: an alternative storage reclamation scheme.* AMPS Technical Memorandum 4, Dept. of Computer Science, University of Utah, August 1981.

[11] P. Hudak. *Object and Task Reclamation in Distributed Applicative Processing Systems.* PhD thesis, University of Utah, July 1982.

[12] P. Hudak and R.M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Proc. 1982 ACM Conf. on LISP and Functional Prog.*, pages 168–178, ACM, August 1982.

[13] J. Hughes. A distributed garbage collection algorithm. In *Functional Programming Languages and Computer Architecture*, pages 256–272, Springer-Verlag LNCS 201, September 1985.

[14] R.J.M. Hughes. *Backward Analysis of Functional Programs.* Technical Report, Glasgow University, 1988.

[15] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: an optimizing compiler for Scheme. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, ACM, June 1986. Published as SIGPLAN Notices Vol. 21, No. 7, July 1986.

[16] Claus-Werner Lermen and Dieter Maurer. A protocol for distributed reference counting. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 343–350, ACM SIGPLAN/SIGACT/SIGART, Cambridge, Massachusetts, August 1986.

[17] Paul Watson and Ian Watson. An effieient garbage collection scheme for parallel computer architectures. In *PARLE Parallel Architectures and Languages Europe*, pages 432–443, Springer-Verlag LNCS 259, June 1987.

[18] K-S. Weng. *An Abstract Implementation for a Generalized Dataflow Language.* MIT/LCS/TR 228, Massachusetts Institute of Technology, Laboratory for Computer Science, 1979.