Morris's Garbage Compaction Algorithm Restores Reference Counts*

David S. Wise

Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 75
MORRIS'S GARBAGE COMPACTION ALGORITHM
RESTORES REFERENCE COUNTS

DAVID S. WISE
REVISED DECEMBER, 1978

---

*Research reported herein was supported in part by the National Science Foundation under a grant numbered MCS77-22325.

To appear in ACM Transactions on Programming Languages and Systems 1, 1.

Morris's Garbage Compaction Algorithm Restores Reference Counts[*]

David S. Wise[†]

Indiana University

Key Words and Phrases: Reference counting, garbage collection, storage management.

CR Categories: 4.34, 4.1.

Abstract

The two-pass compaction algorithm of F. L. Morris, which follows upon the mark phase in a garbage collector, may be modified to recover reference counts for a hybrid storage management system. By counting the executions of two loops in that algorithm where upward and downward references, respectively, are forwarded to the relocation address of one node, we can initialize a count of active references and then update it but once. The reference count may share space with the mark bit in each node, but it may not share the additional space required in each pointer by Morris's algorithm, space which remains unused outside the garbage collector.

---

[*] Research reported herein was supported in part by the National Science Foundation under a grant numbered MCS77-22325.

[†] Author's address: Computer Science Department, 101 Lindley Hall, Bloomington, IN 47401.

Introduction

The elegant three-pass, genetic-order-preserving [8] garbage-collection/compaction algorithm presented by F. L. Morris [6] may easily be modified to recover accurate reference counts for a hybrid storage management scheme. Deutsch and Bobrow [3] describe such a scheme, in which the garbage collector is invoked only when no space is available from the previous garbage collection *or* from a constrained reference-counting scheme that uses only a few bits in each record and which, therefore, often records its "infinity" as the count for heavily referenced nodes. Clark and Green [2] present practical evidence of the power of such a hybrid. Morris's garbage compactor can restore an accurate count whenever the count of incoming references to a node again drops within the range of such a limited reference-count scheme. Thus, a hybrid storage manager may benefit from the reference counts' postponement of garbage collection, even in instances where the counts of all nodes temporarily exceed the capacity of the counters, as well as from the compaction provided by his algorithm.

Any garbage collection algorithm may be modified to restore reference counts by zeroing all counts in a first phase and then incrementing a count during the "marking" phase every time a node is approached via an active reference. A non-zero reference count thus plays the role of the mark after the mark phase. The disadvantage of this scheme is that a count in memory must be fetched, incremented, and stored for every active reference in the system. We shall see how Morris's algorithm avoids this.

Morris's Algorithm

A brief description of Morris's algorithm follows; the reader is directed to his paper for details [6]. Consider the cell in the center of Figure 1a. (We use the term "cell" to refer to the smallest unit of memory which might be a pointer; generally a node contains several cells.) It is referenced by accessible pointers below it in memory (at the left of the figures) and by accessible pointers above it in memory (at the right). The two passes of Morris's algorithm will compact all this accessible information in the high end of memory, preserving the relative order of everything (assuring that genetic order [8] is maintained.) In the figures a solid rectangle represents a cell which originally contains a particular datum and a dotted rectangle underneath it represents the cell--at the same or at a higher address--to which the compactor moves that datum. A blank rectangle indicates a cell whose contents are irrelevant to this discussion, but which may still contain useful information. It is quite possible that a solid rectangle might coincide with a dotted rectangle to its left in the figures as a single memory word.

The "cell" is emphasized here because a pointer must be large enough to refer to every other pointer in the system with a "shifted" reference, as well as ever other node with an "unshifted" reference. "Shifted" and "unshifted" are motivated by Morris's scheme for tagging pointers, and correspond to dashed and solid edges in the figures, respectively. For a typical LISP system with two cells per node, thrice as many possible shifted/unshifted cell pointers (as possible node references) require an additional four bits in each node.

Figure 1a illustrates the original state of affairs before garbage collection. All edges illustrated are references accessible from the user's variables. The mark phase has been run so all accessible nodes are marked and a census of available space exists. The first pass of Morris's compaction algorithm sweeps from the lower end of memory to the higher and only alters marked, upward-pointing references. (It also organizes the unmarked cells so that the second sweep may skip over contiguous groups of them.) As it proceeds it increments the census for every marked node, determining the relocation address of each, and alters upward references. By the time it reaches the cell at the center of Figure 1a, it therefore will know relocation address of this cell and, moreover, will have inverted all upward references to this into a linear list with dashed ("shifted") edges. Figure 1b illustrates this inversion, which is reminiscent of the transformation used in another garbage collector [4] in a converse manner. The processing of this cell revises all upward references to point to its new location. Although not illustrated, the datum, INFO, would itself be inverted and chained into such a linear list if it, too, were an upward pointer. Figure 1c illustrates the state of affairs at the end of the first sweep.

The second sweep is somewhat different because it relocates all data to new addresses, as well as revising all downward references similarly to the first. (It is simpler in that it need not sweep through chunks of contiguous, inaccessible cells.) The sweep is from the highest memory address down, decrementing a census of all memory for each accessible cell encountered to yield the relocation address of each cell. A "shifted" linearization of each family of downward references like that of Figure 1b is performed, so that by the time the second sweep reaches the cell considered in the figures, the situation is that of Figure 2a.

In processing this cell, all downward references are updated to its new address _and_ its content, INFO, is moved there. Since the destination of the move must be an already-swept address, no useful information is over-written. Again, INFO itself may be altered if it in turn is a downward reference, but we do not illustrate this in Figure 2b; it indicates the situation after processing that cell and after all of garbage collection/compaction.

Restoring Reference Counts

The modifications are fairly simple because there are only two tight loops in the algorithm where all references to a particular node are forwarded to its new address. In each of the two updating sweeps we need only count the number of inverted (or "shifted") references as they are forwarded. According to Morris's code [6, p. 664] we count the number of times that the two compound statements containing the assignment, M[j]:=n, is performed for each value of i with iteration counters on these two while loops. After the first such loop, that count becomes the initial reference count of M[i]; the count on the second loop is added on to the reference count of M[i] later. Reflexive references (pointers from locations to themselves) are not treated; they should be ignored in any reference count scheme [5].

Reference counts are thus reinitialized to the total number of upward references--possibly zero--during the first sweep (In Figure 1 there are five of these), and are incremented but once by the number of downward references during the second sweep (four in Figure 2). The memory access overhead for this scheme is, therefore, at most two stores and one fetch per accessible cell. The same bound is achieved when reference counts are only maintained on nodes when all accessible references point only to such nodes.

The space overhead for reference counting remains the same here as that for the scheme in which the mark phase restores counts. The mark bit may be located within the reference count field, as suggested elsewhere [9], because "accessibility" information can be moved from the mark bit during the first sweep. We need only implement Morris's suggestion that the first sweep pass chain the holes of available space together so that the second pass may traverse the chain instead of scanning for marked nodes. (A hole one pointer long can be marked and linked in the chain, while a larger hole remains unmarked, linked in the chain, and loaded with an explicit measure of its size.) His suggestion speeds up the second sweep and releases the mark bit in accessible nodes for use within the reference count already in the first sweep. A reference count field must be at least two bits long in this scheme, however, in order to provide a distinct zero, an "infinity" and meaningful counts (one, two, etc.) in between, all of which can occur between the two sweep phases. In the following strategy, however, a one-bit count suffices for most nodes.

Deutsch and Bobrow [3] suggest a scheme wherein nodes with reference counts of two or more have limited counts maintained in a short scatter table hashed on the addresses of those nodes. Reference counts of one are efficiently denoted by absence of a node from the table, whose brevity is justified by the plethora of uniquely referenced nodes [2]. Under such a scheme the mark bit itself may be used as a one-bit reference count [9] in order to avoid unsuccessful probing of the hash table outside garbage collection by the following procedure. The scatter table is emptied before the sweeping/compaction phase of garbage collection. If an accessible node with exactly one upward reference is discovered on the first sweep then its reference count is initialized to 1 and nothing else is done. If it has zero or at least two upward references, its one-bit count is initialized to 0; in the latter case an entry is made in the scatter table hashed on its relocation address (n in Morris's notation). On the second sweep a reference count is updated after consultation of the scatter table whenever one or more downward references are uncovered. If the reference count was 1 it is changed to zero and a new entry is inserted into the table for that (relocated) node: its count is the number of downward references plus one. If the reference count was 0 and there already was an entry in the table, the number of downward references is added to that entry. Otherwise (count was 0 and there was no entry) there were no upward references, so the action depends on the number of downward references: if only one, then the count is merely changed to 1; otherwise the number is inserted into the scatter table at the position already consulted. At the end of garbage collection all nodes with reference counts of 0 will have valid counts entered in the scatter table, but the far more numerous nodes which are uniquely referenced (will not have a table entry and) will have a one-bit reference count of 1. Outside the garbage collector the scatter table need only be consulted when that mark-bit/reference-count indicates that an entry exists.

Reducing the Space Overhead for Shifting

In spite of all the extra space required in each pointer by Morris's collector (the bit for his *shift* increment [6]

and the ability to address every pointer, as opposed to every node), there appears to be no way to put it to use within the reference count between garbage collections. The problem is that M[i]--in Morris's notation--may or may not contain a shifted reference to another pointer at the code where we would initialize and increment its reference count in either of the two sweep passes. The garbage collector would break down if this information were confused.
(This space might be used by a Deutsch-Schorr-Waite marking phase [7] which maintains its stack with shifted pointers.)

A reduction is possible, however, when every pointer has a mark bit associated with it. This sort of data structure occurs in ECL or under linearization (sometimes called CDR-coding) [1] where nodes may physically contain subnodes. That mark-bit may be used to play the role of the shift increment during the sweep phases since the accessibility information can be moved away into the inaccessible nodes during the first sweep (as shown above), before a pointer's content must be "shifted". Because the pointers in such systems are already large enough to refer to any other pointer in the system, Morris's compaction may be implemented without any additional space overhead in such an application.