BIT 17 (1977), 351–359

THE ONE-BIT REFERENCE COUNT

DAVID S. WISE and DANIEL P. FRIEDMAN*

Abstract.

Deutsch and Bobrow propose a storage reclamation scheme for a heap which is a hybrid of garbage collection and reference counting. The point of the hybrid scheme is to keep track of very low reference counts between necessary invocation of garbage collection so that nodes which are allocated and rather quickly abandoned can be returned to available space, delaying necessity for garbage collection. We show how such a scheme may be implemented using the mark bit already required in every node by the garbage collector. Between garbage collections that bit is used to distinguish nodes with a reference count known to be one. A significant feature of our scheme is a small cache of references to nodes whose implemented counts "ought to be higher" which prevents the loss of logical count information in simple manipulations of uniquely referenced structures.

Key word and phrases: garbage collection, storage management, assignment statement.
CR categories: 4.19, 4.34.

Introduction.

Deutsch and Bobrow [4] have recently proposed a hybrid storage management scheme which combines the features of garbage collection and reference counts in order to overcome the problems of each. Garbage collection [8 §2.3.5] is a sufficient strategy for recovering any unused nodes in a heap. One invocation requires time proportional to the heap size which often disrupts real-time performance of the program using it since the processor usually is dedicated to garbage collection at the indeterminate times when it becomes necessary. In instances where freshly allocated nodes are formed into simple structures and then quickly released, the overhead per node for recovery through garbage collection seems expensive in comparison to the trivial nature of such structures. Reference counts [3] are a scheme for recovering such space directly and cheaply as soon as it is released. They require extra work at every assignment statement (discussed below) and they cannot handle generalized circular structures. Furthermore, they require a count field in every node which may be smaller than a pointer only if we are willing to allow the counts to reach a maximum whence they cannot be decreased, with the result that some nodes enter a state in which they become unrecoverable.

We propose a hybrid system inspired by Deutsch and Bobrow which uses a very limited reference count scheme to delay invocation of garbage collection.

* Research reported herein was supported in part by the National Science Foundation under grants numbered MCS 75-08145 and DCR 75-06678.

Received Nov. 24, 1976. Revised June 29, 1977.

Ours, however, is motivated by the constraints of a heap maintained in main memory. A suitable application of such a heap is within a classic *LISP* [10] implementation for a mini-computer or for an unpaged multiprocessor with a word size that restricts a node to two flags and two pointers. It applies as well to storage management in other languages, just like the garbage collector of Schorr and Waite [11] which was conceived under identical constraints. Our hybrid is designed to be conservative with regard to space without consideration for the problems of data structures which may be partially resident in secondary memory. While the transaction file and hashing scheme of Deutsch and Bobrow [4] may be applied in our scheme, their treatment of secondary storage is inapplicable with the environment envisioned here.

Reference Counts.

The main idea of our system is a one-bit reference count field which is used between garbage collections to distinguish between nodes with reference counts known to be one and other nodes in the heap. We describe relevant algorithms with the idea that this bit will coincide with the mark bit required in every node by the garbage collector. This assumption implies that some reference count information may be lost when garbage collection finally occurs. In compacting the reference count field to the minimum, however, we reduce the space overhead per node to that of garbage collection schemes, and in making that bit coincide with the mark bit of garbage collection we achieve the double use of a bit just as Schorr and Waite [11] used the *ATOM* bit [8 §2.3.5] in two ways between and during garbage collections.

Since one bit plays two roles we shall allude to its value in two non-conflicting ways. As the mark bit within garbage collection its value determines whether the node containing it is "marked" or "unmarked" as being in use. As the reference count between garbage collections its value will determine whether the number of references to this node is "one" or "unknown/greater-than-one." It is convenient in implementations to have the values of the bit for "unmarked" and "one" coincide in order to simplify the first allocation of unused nodes, but the equivalence of these values is immaterial for this discussion.

In any reference count system the nodes with only one reference are of the most interest to the storage management scheme; they are the only ones which might be returned to available space. On creation a node is in this class. Clark and Green point out something else interesting for an approximation to this class of nodes in the *LISP* system they studied [2]: shortly after creation a node tends to get "nailed down" by more references or (less frequently) it tends to be

"abandoned".* Like the scheme of Deutsch and Bobrow, ours keeps track of nodes allocated between garbage collections to detect such early abandonment. Nodes which carry a reference count of more than one can only be recovered through eventual garbage collection. Immediately abandoned nodes, however, will be recovered directly and returned to available space.

Reference Assignments.

Under any reference count scheme the expensive operation is assignment to a reference variable. The node whose address is the value being assigned gains a new reference, and so its reference count must be incremented. The node referred to by the former value loses a reference, (as a side-effect of the assignment statement) and so its reference count must be decremented. If the decremented count is zero then the node (and possibly others indirectly referenced through it) should be returned to available space. The expense of an assignment under this regime should be correlated with other works which demonstrate the dangers of assignment statements [5, 7, 15]. Barth [1] has shown how some of this expense may be avoided.

The one-bit reference count system can only discriminate between nodes with a reference count of one which are candidates for immediate recovery and those with unknown/larger counts which can only be recovered through garbage collection. In order to make the reference count scheme more effective in delaying garbage collection we propose using a cache to avoid temporary increments of a node's reference count above one. Such behavior may occur, for instance, as a

* Almost all the development of storage management algorithms is indirectly due to problems of implementing list processing languages, notably LISP. In this footnote we deal briefly with problems specific to LISP implementation while the body of the paper is written independently of any specific application; a PASCAL heap manager might use it as well.

Fundamental reference material for this paper [2,4] deals specifically with the problems of a "standard" LISP which uses a sequential stack and "shallow bindings". References from program variables and the stack are distinguished from user-created structure. Only the latter are included in the implemented reference counts; the former are handled separately [4]. The term, "nailed down", there refers to the creation of a first reference of the latter sort to a node hitherto only accessible through the former kind of reference [1]. System behavior or user style results in the fact that nodes are not always nailed down.

Our idea of a LISP system [5] does not differentiate between such references; all logical references are included in some implemented reference count, either in each node or in the cache. The stack is used in an entirely different manner because of outputdriven evaluation [6] and the required "deep bindings" [5]. We anticipate that careful use of continuations [12] will avoid multiple references to argument structures, so that logical reference counts of one will indeed be frequent where the "standard" interpreter would have multiple (redundant) references from the stack. The operation analagous to "nailing down" a node here is adding a second reference, and we have used that term consistently with the analogy. In the one-bit reference count scheme, moreover, any second reference (no longer cached) truely nails down a node since it thereby becomes irretrievable through reference counting.

result of the assignment $r:=f(r)$ where r refers to a node whose reference count is one and f is an elementary function (e.g. a field accessing function) which returns a new reference to **part** of the structure hitherto uniquely referenced from r. Certainly the substructure is irretrievably lost if the structure referenced by r is dereferenced and destroyed before the reference count of its substructure, $f(r)$, is incremented. If the reference count of $f(r)$ is incremented first, however, this reference count is raised above one and the ability to decrement it back to one during the subsequent destruction of r is lost.

To avoid the incrementation-decrementation difficultly, we propose a cache of N references to nodes whose implemented reference count "ought to be two" but are still set to one. We define exactly two operations on the cache: **hit**(r) and **insert**(r). The predicate **hit**(r) succeeds precisely when r is a reference already in the cache; a side-effect of the success of **hit**(r) is the removal of r from the cache.

An invocation of **insert**(r) immediately tests **hit**(r). On success (with its side-effect) the reference count in the node referenced by r is incremented. (The "ought to be two" becomes "in fact three;" there is little hope of decrementing three to one before being bumped from the cache by two "hits.") On failure, r is added to the cache; if insertion results in an overfull cache, then some previous entry, p, must be bumped* and p's reference count must at last be incremented to "in fact" two. (Henceforth node p can only be recovered through garbage collection.)

Since the contents of the cache are therefore ephemeral, it is important to release references as soon as possible; after the last use of a reference it is likely to remain in the cache. If the compiler can determine, say through an immediate predominator analysis [9, 1], or if the programmer indicates, say by the assignment of a special constant, that the value of a temporary reference variable, t, is unnecessary beyond a certain point in the program, then the reference count effects of releasing that reference should be initiated there. These may be effected by assigning the null pointer to t, $t:=NIL$, at that point, but the assignment itself need not occur if the reference count operations implied by it are compiled there and just before all immediately following assignments to t.

These operations are now specified. Consider the evaluation of the reference assignment $r:=e$, where $r\uparrow$ is the node to which the variable r refers just prior to the evaluation of the assignment and $e\uparrow$ is the node to which the value of the expression e points. First, the expression e is evaluated to that value. If $e\uparrow$ has a reference count unknown/greater-than-one or if it has been freshly allocated from available space then nothing else is done at this first step. If, however, $e\uparrow$ is a pre-existing node with reference count of one, **insert**(e) is invoked implicitly incrementing the reference count of $e\uparrow$ in anticipation of the new reference to be created by the assignment. The second step releases the reference to $r\uparrow$ by sending it and its infrastructure to the "recycling center" described below. If $r=NIL$ this second step is trivial. The third and final step is to store the value of e in r.

* A reasonable strategy for choosing p is least-recently-used.

Recycling.

The recycling center is an in-place traversal of the node $r\uparrow$ and its infrastructure as long as reference counts are found to be one. No node with unknown/greater-than-one reference count is processed. If the cache is empty then all of the uniquely referenced structure will be returned to available space. However, as long as one reference remains in the cache every recyclable (reference count one) node is handled individually as it is encountered. Before such a node is treated at all, however, **hit** is invoked. When **hit** is successful the cache entry is removed by side-effect, the traversal goes no deeper, and the node is not returned to available space. That node's implemented reference count "ought to have been two but its discovery in the recycling center cancels its first reference. Its actual reference count of one therefore becomes correct when the "owed" count, the cache entry, is cancelled. In this way, actual setting of reference counts above one, whence they cannot be decremented, is avoided.

The necessity to apply the **hit** function repeatedly for each recyclable node requires that the entire cache be cheaply accessible. We intend it to be maintained in high-speed registers with the understanding that N will therefore be small. On microprogrammable machines **hit, insert**, and perhaps the reference assignment should be performed close to the hardware level.

Implementation Options.

This completes the explanation of a reference assignment in our system. Some observations on implementation are appropriate. The first is the impact of different values of N and the second is a possible curtailing of the recycling traversal when the cache becomes empty.

The value of N depends on the system. N being one is sufficient for avoiding reference count increments over the assignments like $r:=f(r)$ as discussed before; such assignments are common in compiled images of purely applicative code [12]. In hand written iterative code statement sequences like $r:=f(s); s:=g(t); t:=h(u)$ are common [8 p. 257]. N should be two in interpreting such code in order to avoid premature increments of the reference counts in s and t. In the midst of the second assignment the cache would contain the values of $f(s)$ and $g(t)$; completion of that assignment would leave only $g(t)$ in the cache.

The choice of N indirectly affects another design option. When the cache is empty it is possible to return a dereferenced structure to the available space list without traversing it entirely using a trick due to Weizenbaum [13] which allows the available space list to have infrastructure. The traversal is no longer necessary since there are no "owed' references in the cache to be reconciled. The choice of a small N, like one, increases the likelihood that the cache will be empty and increases the usefulness of the trick described below.

Let us first accept the convention that the available space list is linked via the "A-field" of every node (at its top level). Nodes returned individually to this list

will have *NIL* in all fields other than the *A*-field and as any node is removed from this list (to be allocated) even its *A*-field will be set to *NIL*. Under this convention no spurious references remain in a freshly allocated node when its various fields are initialized.

If the cache is empty, then a structure to be recycled may be spliced onto the available space list at the first non-unique reference encountered on a linear traversal via *A*-fields. That is, it is sufficient to traverse only the *A*-fields in the recycled structure until a node is encountered with reference count unknown/greater-than-one. The reference to available space is exchanged with the reference to that node and then with the reference to the recycled structure itself. The structure thereby spliced to the front of available space may have non-null fields aside from the *A*-field, so the available space list may have infrastructure. When such nodes are allocated (into a new incarnation) only the *A*-field will certainly be *NIL*; other archaic references will travel along with it. When these other fields are subjected to an initializing assignment, however, their former contents (from their last incarnation) will be inspected and, if found to have reference count one, will be released for recycling.

If Weizenbaum's trick is not used then the explicit "cleansing" of a node individually returned to available space, which guarantees that all fields will be *NIL* at the start of its next incarnation, is not necessary. In this case, the initial assignments within a new incarnation are not treated like general assignments. Former values in such fields must be ignored on these initial assignments and the contents of any node on the linear available space list is immaterial except for the *A*-field used to link it.

The Garbage Collector.

The garbage collector is invoked when available space is exhausted [11]. Before it traverses all structures in use it must initialize all mark bits in the system to their "unmarked" setting. If these bits are the same as the one-bit reference counts, then reference counts are lost. The loss of information due to the obliteration of "one" reference counts is not ultimately fatal since *all* useless nodes are recoverable in a garbage collection. On the assumption that the garbage collection is relatively successful at reclaiming space there will be plenty of opportunity for the reference count scheme to get going again and plenty of "abandoned" nodes for it to recycle before garbage collection is needed again.

The garbage collection uses a standard marking algorithm [8 §2.3.5] followed by a sweep phase. If a two-pass compaction algorithm [8 §2.3.5–9] replaces the sweep phase then "one" reference counts may be easily restored in the nodes which are moved. Nodes which are relocated in the first phase of the compacting algorithm must have *all* references to their former addresses reconciled in the second phase, a traversal of all active references in the system. In the relocation phase the contents of the former address (previously two pointers and two flags)

are established as a forwarding address and as a zeroed two-bit reference count, and the repositioned node is established with its reference count initially set to one. During the traversal of active references a count of forwarded references (zero, eventually one, perhaps greater-than-one) is kept in the former location of a relocated node. If and when a second reference is forwarded, the reference count in the new location is reset to greater-than-one; such a reset is necessary at most once for each relocated node. Reference counts of one may thus be *restored* even though these nodes being relocated by compacting garbage collection had a far higher logical reference count temporarily in the past. The garbage collector can, therefore, assist the reference count scheme by restoring accuracy after a count had reached its "infinity."

Regardless of whether or not compaction is used, the cache contains information for restoring reference counts of "one," which "ought to be two" in at most N nodes. Nodes referenced from the cache should have their reference counts restored to "one."

If a linked list of available space is built during the sweep phase of garbage collection then it must have the required form of a linear available space list. If Weizenbaum's trick for rapid recycling is used then all fields but that linking the available space list should be *NIL*. It is convenient to preset the reference counts in these nodes to "one" in anticipation of their ultimate allocation. Since these are the nodes "unmarked" after the marking phase, the equivalence of the one-bit values for "unmarked" and "one" becomes a convenience for implementing our scheme.

Conclusion.

While the problems of storage management originated in list processing languages like *LISP* which use the heap for nearly every user and program structure, they are also relevant to compilable languages like *PASCAL* [14,1] whose programs use the heap rarely and explicitly. We have described a hybrid storage management scheme which is useful for implementations of either type of language. We assume only that there is one bit per node for use by the storage management scheme and that the entire heap is resident in main memory. In general the scheme is a garbage collection scheme with the "bells and whistles" of a limited reference count scheme working between garbage collections to delay the exhaustion of available space like Deutsch's and Bobrow's [4].

Aside from the space efficiency gained by the dual use of the one-bit mark/count field in every node, the subtlety in our system rests in the cache which maintains virtual increments to reference counts, delaying the increment of physical counts to their maximum, non-reducible value. Since that maximum is so low, i.e. two, our system would collapse without it. With the cache we have the opportunity to maintain accurate reference counts of one across common sorts of pointer assignment statements. This allows a good opportunity of following a

uniquely referenced node from its allocation to its abandonment through many changes in the source of its reference.

Since the cache contents are ephemeral it is necessary to traverse abandoned structures as soon as possible in order to make the best use of information in the cache which has local relevance, before it is displaced as a result of more remote — and less relevant — reference assignments. If the reference count field were a bit larger, then temporary second and third references to a node would not overflow the reference count so that the cache and the immediate traversal of released structures would be unnecessary.

The hybrid scheme described by Deutsch and Bobrow [4], which in fact inspired our scheme, can recover all the space that ours can without the immediate and thorough traversals required by the cache. Their design is motivated by the rigors of a paged system where traversals must be controlled and space for their transaction file is no problem. Our scheme is motivated by the constraints of a heap resident in main memory, like *PASCAL*'s or as in the classic "deep binding" implementations of *LISP*. It is particularly suited to a microprocessor environment where the register size is a constraint on the address space. If the memory size is 2^{16} words of 16 bits, then our system allows a hybrid storage management scheme for a *LISP* system of 2^{15} nodes containing two references: **CAR** and **CDR**, and two bits: the *ATOM*/Schorr-Waite bit and the reference-count/*MARK* bit. Such constraints are typical in general purpose minicomputers and in special purpose desk-top calculators.

Postscript.

In a recent paper [5] we made the observation that a type of frequently created node, called a suspension, is always referenced uniquely. It has the additional property that every use of such a node results in its abandonment. The implication of that observation was that after use of a suspension it should be returned to available space. Had we not made that observation, or had it been ignored in implementing the system described there, this hybrid storage manager would bring the impact of that (obscure, as colleagues later told us) observation into the system implicitly. We were attempting to anticipate the garbage collector by recovering these nodes immediately after their use — just as the hybrid storage management scheme does automatically.

Acknowledgment.

We thank L. Peter Deutsch for his critical reading of the manuscript.

REFERENCES

1. J. M. Barth, *Shifting garbage collection overhead to compile time*, Comm. ACM. 20, 7 (July, 1977), 513–518.
2. D. W. Clark and C. C. Green, *An empirical study of list structure in LISP*, Comm. ACM. 20, 2 (February, 1977), 78–87.
3. G. E. Collins, *A method for overlapping and erasure of lists*, Comm. ACM. 3, 12 (December, 1960), 655–657.
4. L. P. Deutsch and D. G. Bobrow, *An efficient, incremental, automatic garbage collector*, Comm. ACM. 19, 9 (September, 1976), 522–526.
5. D. P. Friedman and D. S. Wise, *CONS should not evaluate its arguments*, In S. Michaelson and R. Milner (eds.), *Automata Languages and Programming*, Edinburgh University Press, Edinburgh (1976), 257–284.
6. D. P. Friedman and D. S. Wise, *Output driven interpretation of recursive programs, or writing creates and destroys data structures*, Information Processing Lett. 5, 6 (December, 1976), 155–160.
7. C. A. R. Hoare, *Towards a theory of parallel processing*, In C. A. R. Hoare and R. H. Perrott (eds.), *Operating System Techniques*, Academic Press, London (1972), 61–71.
8. D. E. Knuth, *Fundamental Algorithms*, Addison-Wesley, Reading, MA (1975).
9. E. S. Lowry and C. W. Medlock, *Object code optimization*, Comm. ACM. 12, 1 (January, 1969), 13–22.
10. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart and M. I. Levin, *LISP* 1.5 *Programmer's Manual*, MIT Press, Cambridge, MA (1962).
11. H. Schorr and W. M. Waite, *An efficient machine-independent procedure for garbage collection in various list structures*, Comm. ACM. 10, 8 (August, 1967), 501–506.
12. M. Wand and D. P. Friedman, *Compiling lambda expressions using continuations*, Technical Report 55, Computer Science Department, Indiana University (1976).
13. J. Weizenbaum, *Symmetric list processor*, Comm. ACM. 6, 9 (September, 1963), 524–544.
14. N. Wirth, *The programming language PASCAL*, Acta Informatica 1, 1 (1971), 35–63.
15. W. Wulf and M. Shaw, *Global variables considered harmful*, ACM SIGPLAN Notices 8, 2 (February, 1973), 28–34.

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
BLOOMINGTON, INDIANA 47401