Incremental Collection of Dynamic, List-structure Memories

Richard B. Kieburtz

Oregon Graduate Institute
Department of Computer Science and Engineering
20000 NW Walker Rd
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E-85-008
January 23, 1986

Incremental collection of dynamic, list-structure memories

Richard B. Kieburtz
Oregon Graduate Center
19600 N.W. von Neumann Dr.
Beaverton, Oregon 97006

Technical Report No. CS/E-85-008
January 23, 1986

Abstract

With the arrival of parallel, multiprocessor computing systems, the solutions used in the past to implement dynamic, list-structure memories for applicative language evaluators are no longer adequate. Almost all of the techniques in current use are based upon a "stop-and-collect" approach. When only a single processor is applied to program execution, this approach is acceptable, but the relative cost of garbage collection becomes enormous when a concurrent evaluation using a large number of processors must be shut down to re-organize storage.

In this paper we review the suitability of various schemes for concurrent garbage collection, and propose a new scheme that allows incremental, parallel collection. Incremental collection is attractive even for a monolithic memory, but has really compelling advantages when memory is partitioned into disjoint modules. The new strategy is a reference-counting scheme, modified to allow collection of cyclic graphs. It requires little explicit cooperation between evaluator and collector processes.

1. Dynamic, list-structure memory

Since the introduction of the first list-processing languages, every evaluator has made use of memory in a way that emulates a dynamic, list-structure memory. Such a memory consists of individually addressable nodes, each capable of storing a pair (or possibly more) of data, which may be basic values or the addresses of other nodes. Fresh nodes are supplied by the memory upon demand, ideally without limit. Dynamic, list-structure memory is an architectural primitive of all abstract machines that implement list-processing languages.

In the early LISP interpreters, the implementation of dynamic, list-structure memories was solved by design of an emulator that allowed its function to be realized with a finite, linearly addressable physical memory. The emulation initializes linear memory into the form of a list of free nodes, and employs mark-sweep garbage collection to reclaim nodes that had been allocated to the evaluator, but are no longer in use to represent the expression under evaluation.

When most list-processing evaluators were interpreters, the overhead of garbage collection to emulate a dynamic memory was acceptable. As evaluators became faster, however, memory management consumed an ever-growing fraction of total processing time. This is particularly noticeable since the most expensive activity involved in memory management is the in-line execution of graph-marking and a memory sweep, which interrupts the normal activity of evaluation.

Four approaches have subsequently evolved to improve the performance of dynamic, list-structure memory emulations. None is completely satisfactory.

1.1. Parallel mark-sweep collection

The in-line overhead of a mark-sweep collector can be partially overcome by introducing an additional processor to execute the mark-sweep algorithm and to allocate node addresses upon demand of the evaluation processor. Algorithms for such a collector have been proposed and proven correct. Synchronization between a memory management processor and an evaluation processor can be confined to

(i) low-level, hardware-supported synchronization of access to physical memory,

augmented by

(ii) producer-consumer synchronization of allocation requests made by the evaluation process,

and periodically,

(iii) the state of an evaluation process must be communicated to the memory manager in order to reveal the current roots of the active expression graph.

The use of a parallel processor to perform the memory emulation function is a direct approach to support dynamic, list-structure memory by improvement of the best-known techniques used for in-line emulation. It suffers the disadvantages that the processing load required to implement mark-sweep collection in parallel is approximately double that of sequential in-line collection. This is because in the marking phase, the parallel collector must use partially obsolete data locating the roots of the active expression graph. In consequence, a node eligible for collection must be examined in two successive passes of the collector before it can be confirmed that it can be collected.

A recent analysis [HiC84] indicates that parallel mark-sweep garbage collection will yield fairly modest performance gains over a sequential, stop-and-collect algorithm, and will not significantly relieve worst-case congestion of memory. The worst case occurs when the active expression graph nearly fills the available memory.

1.2. Copying compaction

An alternative to the use of mark-sweep garbage collection is a scheme that divides the memory into two or more workspaces. Allocation of new graph nodes occurs by sequential

address incrementation within a current workspace. When the current workspace has been exhausted, the active expression graph is copied into a fresh workspace, and compacted into an initial segment of the address space as it is copied. This strategy with two workspaces "wastes" half of the total available address space, as the active expression graph cannot occupy more than one workspace at a time. However, this is not significant in a virtual memory system.

Just as is the case with mark-sweep collection, the copying-compaction algorithms do not perform well when the active expression graph nearly fills a workspace. In a virtual memory system, workspace size should be expanded dynamically when the active graph grows to occupy more than 70 per cent of a workspace.

Copying-compaction has the advantages that allocation is a relatively cheap in-line operation for the evaluator, and that memory fragmentation is effectively eliminated by compaction. However, copying is approximately twice as expensive as is the marking phase of a mark-sweep collector, and it is less amenable to parallel execution.

A parallel copier requires exclusive access to a node while it is being copied, thus mutual exclusion synchronization is required between the copier and the evaluator, and on a per-node basis. Furthermore, the evaluator will incur the additional overhead of following indirection pointers whenever a reference is made to a node in the old workspace whose contents have already been copied. This requires an implementation mechanism in which the evaluator "faults" on an indirection pointer, and retries its memory access on the node that is pointed to.

1.3. Reference counting

Both the mark-sweep and copying-compaction schemes traverse the active expression graph in order to preserve it during the collection process. Reference-counting attempts to account for the potential use made of a graph node, and allows it to be discarded as soon as no further reference to it is possible. Reference counting does not suffer the disadvantage that its overhead increases with the size of the active expression graph. However, it has several other disadvantages:

i) Cyclic graph structures cannot be reclaimed by reference-counting alone;

ii) In-line overhead is incurred to maintain reference counts, and this effectively increases the cost of each memory reference;

iii) Reference counting requires more memory bits to be devoted to memory management than does graph traversal for marking or copying. Furthermore, reference counts may overflow any fixed field.

Modified reference counting schemes have been proposed to cope with various of these difficulties. One in particular, a modification in which reference counts are used to trigger graph traversals in order to detect cyclic structures, has been proposed by Ashoke Deb [Deb84]. Simulations indicate that his scheme may incur far less overhead than either mark-sweep collection or copying-compaction, provided that the overhead of maintaining the reference counts is not excessive. Other schemes based upon reference counting, yet able to collect cyclic structures, have been proposed by David Brownbridge [Bro85]. Our parallel collection scheme is based upon an extension of Deb's algorithm.

1.4. Adaptive memory management schemes

Many of the newly-designed, high-performance list-processing computer systems have turned to the use of adaptive schemes for memory management. In a typical adaptive scheme, the nodes allocated to the evaluator are partitioned into those whose lifetimes are so long that they can remain allocated for the duration of an entire evaluation, those whose lifetimes are short and which can be reclaimed quickly, and those whose lifetimes are indeterminate, and which must be subjected to a more general storage reclamation policy.

Examples of adaptive schemes include the Lieberman-Hewitt "scavenging" collector [LiH83], and the "one-bit" reference counting scheme of [SCN84]. These have all been shown to improve performance in the cases they were designed to handle. The difficulty with such schemes is that they are all ad-hoc to some degree, and don't suggest any general principles on which to base memory management for functional program evaluators. Also, they do not admit use of parallel processing to reduce in-line overhead.

2. Memory management for a graph-reduction evaluator

In considering the requirements of a dynamic, list-structure memory, it is important to take advantage of any information available about how the memory is likely to be used. For instance, if it were known that a program in execution always allocates an initial set of nodes that are not to be released until execution terminates, then these nodes might be reserved from the dynamic memory management activity. When the evaluator is allowed to use the memory as a random-access store, and can perform explicit pointer assignments dictated by a program (as does a LISP interpreter, or a program written in a language with **ref**-type variables), then little can be said about how memory may be used. Thus memory management must use a very general algorithm to reclaim nodes.

At the other extreme, an evaluator that used list-structure memory only to implement representations of data types composed with cartesian product and discriminated union would form only acyclic graphs. Storage reclamation of acyclic graphs could make use of reference counts just as well as any more general scheme.

A graph reduction evaluator does not form exclusively acyclic graphs; a graph representing an application of fixpoint operator to a recursion scheme may exhibit one or more cycles during its reduction. However, in practice, the incidence of cyclic graphs is very much less than the incidence of acyclic graphs. This suggests that a reference-counting scheme might successfully be augmented by a graph-traversal scheme in order to handle cyclic graphs as the exceptional case. This is a circumstance under which Deb's modified reference counting collector might be used with considerable advantage.

2.1. Modified reference counts: the sequential algorithm

In describing algorithms for garbage collection, we shall revert to the terminology of [DLM78] when referring to the functional-language evaluator, and call it the "mutator" process. The only aspect of its operation that concerns us is that it may mutate the active data graph occupying memory.

In Deb's scheme, reference counts are maintained as the mutator constructs and modifies a graph. Whenever the reference count of a node is decremented, a pointer to that node is put into the "Garbage Can", which is a set of those nodes that might be eligible for collection. Subsequently (it is not crucial just when this action is scheduled), the mutator is stopped and nodes in the Garbage Can are examined to determine if they can be collected.

Any node whose reference count is zero can be collected immediately. As a node is collected, the reference counts of any nodes it points to are decremented, and hence pointers to those nodes are put into the Garbage Can.

Any node N whose reference count is non-zero might still be collectable, if it were determined that N is the root of a strongly-connected subgraph in which

(C) the reference count at each node is equal to the number of edges incident from nodes within the subgraph.

To test condition C, it is sufficient to make two traversals of a spanning tree through the graph rooted at N. An additional reference count field, called the *local reference count*, is maintained for each node. At the start of the first traversal, assume that the local reference count of each node is zero. During the first traversal, the local reference count of each node visited will be

incremented each time a pointer to the node is encountered[1]. The condition tested to determine if a node is being visited for the first time is that its local reference count is zero.

Threshold	local ref. ct.	reference ct.	C	U	R	A	P	data

The memory management fields of a node are shown above. In addition to the reference-count threshold, and the local and total reference counts, the control bits are:

C is collectable -- for private use by the memory manager

U forever uncollectable -- private use by the memory manager

R recently-referenced -- set when a ref. count is modified; explicitly cleared by the memory manager

A allocated/not-allocated -- private use by the memory manager

P persistent -- private use by the memory manager

Table 1 -- Template of a graph node

After the first traversal is complete, the local reference count at each node is exactly the number of edges incident upon the node from within the graph rooted at N. To determine if the graph is collectable, a second traversal is made. During this traversal, the collector runs in one of two states, optimistic or pessimistic. If a node is encountered when the collector is optimistic, and if the node's local reference count equals its total reference count, then the node is marked as potentially collectable and the collector continues to be optimistic. Otherwise, the (potentially) collectable bit is cleared for the node, and the collector becomes pessimistic in visiting all successors of the node. Also during traversal 2, the local reference count of each node is restored to zero the first time the node is encountered. Thus the condition indicating the first visit to a node (other than the root) in traversal 2 is that its local reference count is non-zero.

The collector begins the second traversal in an optimistic state. At the end of the traversal, the local reference count of each node will be zero, and the collectable nodes will be exactly those that are both

i) marked as collectable, and

ii) reachable from N without traversing any node not marked as collectable.

Actual collection of nodes requires yet another traversal of the collectable subgraph rooted at N. It is convenient to allocate yet another bit to record whether a node is currently allocated, or has been collected, in order that a list data structure (allowing duplicate entries) can be used to implement the Garbage Can, without risking multiple collections of the same node.

At first glance, this collection algorithm may appear to be burdened with too much overhead, since it requires two full tours of a graph in order to determine whether or not it is actually collectable. There are two factors that mitigate the apparent overhead. First and most important is that the collection strategy has become *incremental.* Only a subgraph rooted at a candidate node is traversed, rather than the entire active graph, as is the case in mark-sweep or copying-compaction. The average size of subgraphs does not grow in proportion to the total

[1] The local reference count of the root node, N, is not incremented when it is initially visited. Local references are arcs from nodes *within* the graph, and the initial visit to the root is a consequence of a pointer from the Garbage Can, which lies outside the graph.

size of the active graph. Simulations carried out by Deb have shown that for active memories of even moderate size (of the order of 10,000 active nodes), the average incremental collection traverses only a tiny fraction of the nodes that a full graph-marking algorithm does in a mark-sweep collection. The relative benefit of incremental collection can only be expected to improve as the size of the active memory increases. Furthermore, the average cost per node collected remains constant over all ratios of memory occupancy p with the incremental collection strategy, whereas it grows as

$$1/(1-p)$$

for both the mark-sweep and the copying-compaction algorithms.

A second cost-reducting factor is particular to the collection of heap-allocated memory for a programmed graph-reduction evaluator. Only properly-nested (i.e. reducible) cyclic structures are ever formed. In nearly all cases, a compiler can (in principle) determine a bound on the number of edges that might close cycles on the root of a subgraph. The most usual bound is, of course, zero. Cyclic graphs are created only to represent recursively defined ground values, in implementing expressions of the form

$$\textbf{letrec } x = H\,x \textbf{ in } G\,x$$

Assuming that a compiler can successfully identify all node allocations that may be the roots of cyclic graphs, and can in many cases bound the maximum number of cycles that could close on a node, then each allocation request can carry with it the value of a bound on its cyclic closures. It is sufficient in practice to restrict cyclicity bounds to four values, 0, 1, 2 and more. If we provide, in addition to the total and local reference count fields, a third field for this cyclicity bound, then it serves as a threshold reference count to determine whether or not to place a node in the Garbage Can. When the reference count of a node is decremented, it should be put into the Garbage Can only if its reference count has fallen to the level of its threshold or less. This strategy can decrease nearly to zero the probability that a node is put into the Garbage Can when it cannot actually be collected.

2.2. A concurrent algorithm

As might be expected, the sequential, stop-and-traverse algorithm to determine whether the contents of the Garbage Can are collectible breaks down if it is naively extended to execute concurrently with a mutator. It is possible to develop scenarios in which the collection criterion (C) appears to apply, yet there remains a reference to the root node N from outside the strongly-connected subgraph rooted at N. We say the criterion "appears to apply", because it can only be tested by executing a sequential algorithm. The criterion is in fact the correct one; the difficulty is in formulating a reliable way to test it.

The first problem is that it must be possible to detect when an action of the mutator modifies the reference count of a node in a graph being traversed by the collector. This problem is solved by adding a single bit to the set of node tags, the "recently-referenced" tag. This tag will be set when a node is allocated and whenever its reference count is incremented or decremented. The recently-referenced tag will be cleared only by the memory manager when performing the collectability test.

A second problem is to assure that incrementation of local reference counts starts from zero. It is not adequate to zero the local reference counts while inspecting a graph for collectability. If in the course of inspecting a graph that was not, in fact, collectable the graph were modified by action of the mutator, a node whose local reference count had been incremented could become separated from the local graph. It could become attached to some other subgraph while its local reference count is non-zero. The memory manager has no way to locate it in order to restore its local reference count to zero.

To solve the second problem, the concurrent algorithm makes one more traversal of a graph than does the non-concurrent version. This extra traversal precedes the other two. In this

traversal, the local reference count is zeroed and the recently-referenced tag is cleared (in that order) on each node visited. The test for first visit to a node is that one of these values is non-zero. Note that once a node has been visited in this first traversal, it cannot be disconnected from the graph by action of the mutator without causing its recently-referenced tag, or that of one of its ancestors, to be set.

Proposition 1: Following the initial traversal (which we number as 0), every node x in the graph rooted on N will either have zero local reference count, or else every path from N to x will contain at least one node whose recently-referenced tag is set.

Proof: Suppose the graph rooted on N contains a node y whose local reference count is non-zero after Traversal 0. Since the mutator cannot have changed the local reference count, it must be that node y was not visited by the collector during the traversal, but has been appended since the traversal began. If y was added to the graph by writing a pointer to y into a node z already in the graph, then y's recently-written bit will be set. Otherwise, y has an ancestor z in the graph which also was not visited during Traversal 0. The proof follows by induction on the depth of the unvisited node. □

Traversals 1 and 2 follow traversal 0 just as in the non-concurrent algorithm, except that these traversals do not descend below a node whose recently-referenced tag is set.

Proposition 2: At the conclusion of traversal 2,

(a) no node reachable from the root N along a path containing only nodes marked as collectible can be reached from any node of the active graph, and

(b) any node in the active graph having a successor whose local reference count is non-zero either has its own local reference count non-zero, or has its recently-referenced tag set.

Proof: Let G_N be the maximal, strongly-connected graph rooted on N. Condition (a) depends upon the fact that the local reference count of every node visited in Traversal (1) was zero at the start of the traversal, and that no node's reference contributes to the local reference count if any other node unless both were present in G_N at the start of Traversal 1. Thus the local reference count of each node at the end of Traversal 1 is less than or equal to the number of references to the node from within the graph. If no node of the graph has its recently-written tag set, then all will have been visited in Traversal 1, and the local reference counts equal the number of internal references. Thus the nodes of G_N are marked as collectable in Traversal 2 iff condition (C) holds. This establishes (a). The proof of (b) is analogous to the proof of Proposition 1. □

Condition (a) guarantees that only unreachable nodes will be collected. Condition (b) assures that at the conclusion of traversal 0, every node reachable from a root N along a path containing no node whose recently-referenced tag is set will have a local reference count of zero.

2.3. Economizing on reference counts

The G-machine [Joh83,Kie85] is a programmed graph-reduction processor. Because it is programmed, it is frequently able to reduce an applicative expression without building a full application graph in memory. It uses a stack of graph pointers (P-stack) to locate arguments of a function application, and these can often be produced without traversing an application graph. P-stack operations can make copies of pointers or destroy copies of pointers without mutating the stored expression graph. If such references are to be counted by the memory reference-count mechanism, signals must be exchanged between the evaluation processor and the storage manager on nearly every stack operation the processor executes. It would be a considerable economy not to burden either the evaluator or the storage manager with the task of counting node references made from the P-stack.

Many graph nodes are really short-lived and are never connected to the rest of the active expression graph. Such nodes are allocated to the processor, their pointers remain in the P-stack while in use, and then are discarded. If P-stack references were not counted, cells

allocated for temporary nodes such as these would never have their reference counts incremented from zero.

Temporary nodes such as described above could be collected effectively by a stack allocation discipline. Stack deallocation is satisfactory for temporaries, but not for all cells, because some cells allocated in executing a function call may be linked into the graph produced to represent the value of the function application. Such cells persist after return from the call.

Since the functions of storage management and graph reduction can be separated, the mutator is not responsible for maintaining reference counts. That is left to the storage manager. The rules governing reference count managment are as follows:

a) The reference count of a cell is zero upon its allocation.

b) The reference count of a cell is incremented whenever a pointer to the cell is written into the graph. Thus a WRITE command from the G-processor signals the storage manager to increment the reference count of the cell pointed to by the datum, if the datum is a pointer. (Pointers are tagged data.)

c) Whenever a WRITE causes the content of a cell to be erased, and the content was a pointer, the reference count of the cell it had pointed to is decremented.

According to the rules given above, the reference count of a cell accounts only for arcs of the expression graph explicitly represented in G-memory. References held by the G-processor are not counted. This approximate form of reference counting relieves the storage manager of having to track closely the activity of the G-processor. There are ramifications.

* A cell with zero reference count cannot be deallocated so long as the G-processor's state may include a pointer to the cell;

Condition * can be accommodated by a stack discipline for possible deallocation, if

i) no component of the state of the G-processor persists after completing evaluation of a function application, and

ii) no cell allocated during evaluation of an application $f\ e_1 \cdots e_n$ is accessible after return from the function call unless it is reachable in G-memory from the root of the expression graph representing the result of the call.

Condition (i) asserts that the implementation of a function call by the G-processor produces no side effects. Given the above conditions, any cell allocated during a function call can be collected after return from the call, provided that its reference count is zero or it is identified as collectable by the cycle-detection algorithm. A cell which is not collectable at that time will be marked as *persistent*. Any persistent cell can be collected immediately upon detecting that its reference count has become zero (or the cycle-detection algorithm certifies it as collectible). The G-processor must signal the storage manager when it executes a function CALL or RETURN.

The strategy outlined above can be called *stack allocation with persistence*. It has been studied in connection with storage management problems for Algol 68 [Wia75] and similar languages, although not in connection with reference-counted storage management. We propose two levels of modified stack allocation disciplines, that allow for persistence of data.

2.4. Stack allocation from a freelist

Suppose free cells are linked into a one-way list (freelist) from which all new allocations are made. Let us also provide an additional tag bit -- the persistence bit -- with each cell. When a cell is allocated from the freelist, its persistence bit is cleared, and a pointer to the cell is entered into a list of cells allocated (but not yet collected) in the current function call.

When a new function call is made, the list of allocated cells is pushed onto a stack, and a new list is started. When a function returns, the list of cells allocated during its activation is

examined for collectability. Nodes in this list with zero reference counts are immediately collectable. (So are cells found to be collectable by the incremental traversal algorithm.) Such cells are collected, and can be linked back into the freelist for re-issue. Nodes found not to be collectable are marked as persistent.

Once marked as persistent, a cell can be collected on-the-fly whenever its reference count becomes zero (or an incremental traversal finds it to be collectable). This is because, in operation of the G-machine to reduce functional program graphs, we can prove that when a persistent cell is collectable, there can be no reference to it from the P-stack.

Proposition 3: If the P-stack holds a pointer to a node n, then either there is also a pointer to n from the active graph (and hence n is not collectable) or else the function call in which n was allocated has not yet returned.

Proof: Since we must reason about operation of the G-machine, the proof is necessarily informal. A node n is collectable if there is no pointer to n from the active expression graph. If no pointer ever existed in the graph, then the only pointer to n, from the time it was allocated, must be in the P-stack. Since every pointer returned on the P-stack is a pointer to a (former) application node that a function call was made to evaluate, no node allocated in executing a function call is ever returned. Thus if a pointer to n actually exists, then the P-stack frame in which the allocation of n was made still exists.

Suppose on the other hand that a pointer to n once existed in the graph, but all such pointers have been removed. The only operation of the G-machine that deletes a pointer from the graph is the UPDATE of an application node, which occurs immediately preceding a return. Thus the P-stack at a return never holds a pointer that has been removed from the graph. At a return, the current frame of the P-stack holds only a single pointer, that to the evaluated node which, as argued above, was allocated in an as-yet-unreturned function call. □

Here we see the tradeoffs among various strategies for storage management. Full reference counting allows collection of nodes at the earliest time they become inaccessible to the ongoing evaluation. However, close cooperation between the mutator and the storage manager is required to achieve this. The mutator must communicate to the storage manager the references made and deleted by its internal state transitions. The cost of implementing this close cooperation may not be worth the benefit of immediate collectability.

In stack allocation with persistence, if the storage manager maintains detailed accounts of the nodes allocated to each outstanding function call, then garbage nodes can always be collected immediately after return of the function call in which they were allocated. Cooperation between the evaluation process (mutator) and the storage manager is reduced to signaling WRITE operations, and function CALL and RETURN, but does not require signalling of internal state transitions.

2.5. Reducing fragmentation of the free storage pool

As mentioned previously, the copying-compaction algorithm exhibits an advantage that is particularly significant when paged virtual memory is in use; each time collection occurs, the interval of addresses occupied by cells of the active graph shrinks. This prevents the incidence of page faults generated by the running program from increasing over time to degrade performance.

On the other hand, when cells are allocated from a freelist, there is no inexpensive mechanism that systematically improves program locality. Unless sorting by address occurs when a collected cell is re-linked into the freelist, a freelist whose cells were initially sorted by address can be expected to degrade to random order over time, as cells are allocated and reclaimed out of order. This has the bad effect that performance may degrade with the passage of time, because as storage fragmentation increases, so also does the incidence of page faulting.

The scheme we have proposed here does not provide a mechanism for storage compaction, and this is a weakness. A partial remedy may be achieved by using a variant of "buddy system" allocation [Knu68].

For buddy system allocation, storage is logically organized as a tree of nested blocks, each of size a power of two in address space. Each block, save the largest one (which is the entire address space) has a "buddy" whose initial address differs from its own in just a single bit. Buddy blocks are adjacent and of equal size. With each block, down to the level of the smallest allocatable cell, there is associated a boolean flag which we call its *alloction flag.* This is set true if the block is currently allocated (or partially allocated), false if the block is free. When two buddies are free, so also is the block of twice their individual sizes that encloses them. Thus when a block that has been in use is returned to the free storage pool, its allocation flag is cleared and the allocation flag of its immediate ancestor block is given the value of the allocation flag of its buddy. Since the allocation flag of the block being released is false, the conjunction of its allocation flag and that of its buddy equals that of its buddy. This process of freeing ancestor blocks continues up the ancestor tree until some ancestor fails to be freed by the process.

Suppose the buddy system were modified so that the allocation flag of an ancestor block became the logical conjunct of the allocation flags of its progeny. Then the interpretation made of an allocation flag would be that it is true if and only if *every* atomic cell of the block were allocated. This has some advantage if the only storage units allocated are atomic cells, because no freelist needs to be maintained in order to locate the next allocatable cell. In time logarithmic in the size of the memory, one can start from the root of the ancestor tree and locate the lowest-addressed atomic cell that is not currently allocated.

In the average case, allocation is much faster. To find the next unallocated cell, first examine the allocation flag of the father of the last previously allocated cell. If this flag is clear, then the next free cell is the buddy of the cell at which the search started. If not, then examine the allocation flag of the next higher ancestor block. Eventually, an ancestor block is found to be free, assuring that a downward search within that block will find a free cell. On the average, the required depth of this search will be one when the storage is 50 per cent occupied, independent of the total memory size.

Freeing a cell is similarly cheap. The allocation flag of a newly freed cell is cleared, and so also are the allocation flags of all its ancestors. Of course, the explicit clearing of ancestors' allocation flags can stop upon encountering, in a path from a leaf toward the root, an allocation flag that is already cleared.

Note the advantages of this scheme over a freelist:

Freelist pointers do not have to be modified when a cell is allocated or freed (although allocation flags do);

Memory fragmentation is reduced, as the lowest-addressed free cell is always selected for allocation.

Furthermore, in the modified reference-count storage management scheme suggested here, allocation flags are already present in atomic cells (graph nodes). Thus the added storage overhead of buddy system managment is one additional bit per cell. The allocation flags of ancestor blocks need not be stored with the cells themselves.

2.6. Collecting un-reference-counted temporary storage Temporary storage cells that are allocated during the evaluation of a function call and referred to only by pointers from the evaluator's traversal stack are never reference counted and never put into the Garbage Can. It was suggested in Section 3.2 that the storage manager should maintain a list of all pointers that have been allocated during each function call so that this set of cells can be examined for collectability after the call has returned. Maintaining a list of pointers to all cells allocated implies a

substantial burden on storage space, in order to support collection. It is unnecessary if the free storage pool is maintained by the modified buddy system.

Since the modified buddy system permits address-sequential allocation of free storage cells, the set of addresses of cells allocated during any function call will lie in an interval whose starting address is that of the first cell allocated and whose ending address is that of the last cell allocated. In fact, the interval may also contain addresses of some cells allocated during other function calls that have previously terminated, but any such cell will either be marked as persistent or will already have been as collected at the time the interval is swept. Thus, instead of explicitly saving a list of addresses allocated during a function call, it is only necessary to save the boundary addresses of an interval.

Actually, it may be necessary to allocate from a (small) set of intervals, because some storage fragmentation can be introduced by the asynchronous collection scheme. When a function call returns, the address interval it used for allocations is made available to the storage manager to be swept of garbage. However, this interval should not be made available for reallocation until a collection sweep has been completed, for if it were, then attempted new allocations within it might overtake collection from it. Thus, after a function call has returned, new allocations should begin immediately from another interval, while the returned interval is held out of the free storage pool until it has been swept. The division of intervals we have just described may result in a function call allocating from a set of address intervals rather than just a single one.

3. Simulation results

The incremental garbage collection scheme described in the preceding sections has been simulated to obtain an estimate of its performance [Fos85]. The simulation defined separate UNIX processes for

a) the mutator,

b) the storage manager which executes the cycle-detection algorithm,

c) a memory access controller which performs reference-count maintenance and maintains the free storage pool as a linked list.

Two other service processes were provided to simulate the garbage can and a node-preallocation queue. Of these processes, the time consumed by the mutator and the storage manager was of interest. The other processes performed functions that could be supported by hardware or firmware enhancements in a real implementation of the storage manager.

The mutator process executed only memory reference operations. This corresponds to an assumption that performance in graph-reduction is limited by the available bandwidth to the graph-storage memory. The operations, READ, WRITE, ALLOCate and context-switches were generated in a pseudo-random sequence controlled by probabilities for each type of operation.

The rates of node allocation and of context switches were controllable parameters of the simulation. For the data quoted here, context switches (calls and returns) occurred with a probability of 1/15 or 1/30 and the allocation rate was 1/12. READ and WRITE operations were given probabilities that were 1/3 and 2/3, respectively, of the remaining distribution. READ operations are benign, as they cause no mutation of the active data graph.

The size of the graph storage memory as simulated was 2000 nodes. The initial, active graph consisted of approximately 300 nodes. It was first constructed as an almost-balanced binary tree, then approximation 60 additional edges were introduced from leaves or non-branching interior nodes to randomly selected nodes within the active graph. These edges formed shared subgraphs, including some cycles.

After simulation began, the number of active nodes increased as a result of the simulated allocations. Pointers were also removed from the graph as a consequence of WRITE operations

that occurred as the mutator randomly traversed the active graph. Thus the graph was fragmented into a rooted part and unrooted (i.e. inaccessible) parts, as well as mutated in its topology. The collector process located unrooted graph fragments, dissected them and released their storage cells to the free storage pool. The collection process and the mutator process were given equal computation time, in alternating time slices, which fact was confirmed by profiling.

As the simulation ran on, the probability of ALLOC operations was decreased by the ratio of $\frac{1}{1-\alpha}$, where α is the memory occupancy ratio. Thus the allocation rate and the collection rate were brought into equilibrium over a period of time. It is the equilibrium allocation rate that can be sustained that is the measure of performance of a collection algorithm. The ratio of one allocation per 12 mutator instructions that was obtained in the simulation experiments is comparable to the ratio of occurrence of memory allocation instructions in a static examination of comiler-generated code for a graph-reduction processor. The ratio obtained in the simulation is also known to be pessimistic because

a) the storage manager was maintaining an explicit list of allocated in each function call, and

b) there were several sources of overhead in the collection algorithm as it was simulated that can readily be removed. These may have accounted for up to 35 per cent of its execution time.

4. Multi-process parallel collection

Suppose M mutators share access to a common address space with N garbage collectors. With the present algorithm, the N collectors cannot coexist, because they may have read-write and write-write conflicts on shared data structures. A first step would be to ensure mutually exclusive access to the GC and to the free storage pool from which nodes are taken and returned, respectively. Then multiple collectors might coexist safely, provided that they traversed mutually disjoint graphs in the incremental, cycle detection phase. Unfortunately, there is no way to assure that separately rooted subgraphs will be disjoint; in general they won't be. Let us analyze the consequences of conflict in the accesses of several collectors to shared data.

Collectors wouldn't interfere with one another on the primary reference count data that they share with the mutators since they only read these data. Read-read conflicts don't interfere. Other data, the recently-written (R) tags, the local reference counts and the is-collectible (C) tags, may be both read and written by a collector, so multiple collectors would interfere. To ensure that no conflict on these data can occur, each collector might set a lock on each node it touches, while these data are in use, releasing the lock again after it has completed its task. A two-phase locking protocol can ensure exclusive access.

Two-phase locking is possible with a single test-and-set bit associated with each node, provided that a collector maintains a record of the set of nodes it has previously visited. This is necessary so that upon subsequent visits, when the collector finds the test-and-set flag to be set, it must be able to tell that the node belongs to the subgraph for which it has established exclusive (among collectors) access. Mutators, of course, can access nodes reachable from the active graph concurrently with collectors, as before. Thus a second reason to maintain the set of nodes visited is to ensure that each node can be located in the lock-releasing phase, even if the node has been disconnected from the subgraph by action of the mutator.

In fact, it is not possible to avoid interference if an unbounded number of collectors are allowed concurrent access to a common set of nodes. Let's consider a hypothesized "solution" that does not involve two-phase locking. Each collector could maintain a private copy of the local reference counts and the C-tags, rather than using shared data in a common address space. Private data can be associated with shared nodes by maintaining an association table (say, by hash-coding on the node addresses).

But there remains one serious problem with such an approach. Conflicts on the R (recently-written) tag have not yet been accounted for. The accesses of a collector to an R tag only read its value during Traversals 1 and 2, thus no direct interference occurs while collectors are in these traversals. A collector can also write the R tag, clearing it to zero during Traversal 0. The write-write conflict of two collectors is seemingly non-interfering because when a collector writes, it invariably clears the tag. However, the mutators also write the R tag, to set it, and each collector relies upon this tag for warning that a mutator has modified a reference to a node that it visits during Traversals 1 or 2. If collector A has cleared the R tag of some node during Traversal 0, then a mutator subsequently sets it, the value of the tag must remain set for the collector's test during its subsequent visits to the node in Traversals 1 and 2. However, if another collector, B, arrives at the tag in Traversal 0 after the mutator has set it and B clears the tag, collector A will have no way to detect that the mutator has modified the graph since A's Traversal 0, and the collection algorithm may be invalidated. In the worst consequence, collector A might determine that the graph is collectible even though it is still reachable from the active graph!

If we assume that the number of collectors is larger than any bound, then it is easy to prove that no finite-state protocol can alleviate this problem. Suppose P were an N-state protocol proposed as a solution. Since the number of collectors can exceed N, the states of P cannot be used to distinguish collectors from one another, thus the actions of different collectors inducing the same state change cannot be distinguished.

The action of a mutator cannot drive P into a 'dead' state from which an initial state is unreachable, for then a node could be rendered forever uncollectable. Thus any node visited by a mutator and driven from a state r into a state s must eventually be restored to an initial state by the action of collectors executing protocol P. But from an initial state, another collector can reach state r of the collection process as if the node in question had never been visited by the mutator! A clear R tag no longer offers incontrovertible evidence that the reference count of a node has not been modified by the mutator since a collector began its work.

5. Future work

Incremental collection has a very significant potential advantage over stop-and-collect schemes when the storage occupied by a data graph becomes heavily loaded. By "heavily loaded" we mean that either

a) the ratio of occupied address space to available address space exceeds 0.7, or

b) the ratio of occupied virtual memory pages to real memory pages exceeds some critical value (probably 3 or 4), or

c) the storage is physically distributed among the nodes of a message-passing multiprocessor system.

In such cases, the cost of executing a stop-and-collect algorithm over the entire address space becomes excessively high. Incremental collection, on the other hand, should not degrade in performance when storage is heavily loaded.

We have developed an algorithm for concurrent, incremental collection, on the assumption that reference-counting can be made economical by providing direct hardware support for reference count maintenance. Our initial simulation indicates that the computational load imposed to collect possibly cyclic data structures will not exceed the capacity of a processor running concurrently with the primary evaluation processor.

Although the concurrent, incremental collection scheme appears potentially attractive, there are several aspects that must be thoroughly investigated before it can be deemed a practical solution. These include:

1) Design and simulation of a reference-counting memory controller. Critical questions are: Will performance be degraded significantly by contention for memory cycles between

reference count maintenance and the evaluation process? To what extent is the pattern of locality of references degraded by reference-count updating? (The latter question is particularly relevant for virtual or distributed memory systems.)

2) Confirmation of the effectiveness of stack management with persistence. This strategy, adopted in order to avoid counting references from the internal stack of the evaluation process, delays recovery of garbage until the collapse of the stack frame in which a node has been allocated. What is the penalty (in terms of available free storage) paid for this strategy? Simulations can provide useful data.

3) Evaluation of the modified buddy-system for maintaing the free storage pool. We have estimated that the cost of allocation, using this scheme, is constant-time in the average case, independent of the size of the address space. Perhaps this can be verified by simulations or by instrumentation of the allocation process.

4) Extension to multi-process collection. We have described the synchronization requirements for multi-process collection. Such an extension seems highly desirable if incremental collection is to be used to support multiple, concurrent evaluators. Simulations need to be done to evaluate the effect on performance that must be attributed to the additional synchronization.

References

[Bro85] Brownbridge, D. R., "Cyclic reference counting for combinator machines," in *Functional Programming Languages and Computer Architecture*, vol. 201, J. Jouannaud (ed.), Springer-Verlag, Nancy, 1985, pp. 273-288.

[Deb84] Deb, A., *An efficient garbage collector for graph machines*, Oregon Graduate Center, 1984.

[DLM78] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. and Steffens, E. F. M., "On-the-fly garbage collection: an exercise in cooperation," *Comm. ACM*, vol. 21, 11 (1978), pp. 966-975.

[Fos85] Foster, M. H., *Design of a list-structure memory using parallel garbage collection*, M.S. thesis, Oregon Graduate Center, 1985.

[HiC84] Hickey, T. and Cohen, J., "Performance analysis of on-the-fly garbage collection," *Comm. ACM*, vol. 27, 11 (1984), pp. 1143-1154.

[Joh83] Johnsson, T., *The G-machine -- an abstract architecture for graph-reduction*, Dept. of Computer Sciences, Chalmers Univ. of Technology, Gothenburg, 1983.

[Kie85] Kieburtz, R. B., "The G-machine: a fast, graph-reduction evaluator," *Proc. of IFIP Conf. on Functional Prog. Lang. and Computer Arch.*, Nancy, 1985.

[Knu68] Knuth, D. E., in *The Art of Computer Programming, Vol 1 -- Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[LiH83] Lieberman, H. and Hewitt, C., "A real-time garbage collector based on the lifetimes of objects," *Comm. ACM*, vol. 26, 6 (1983), pp. 419-429.

[SCN84] Stoye, W. R., Clarke, T. J. W. and Norman, A. C., "Some practical methods for rapid combinator reduction," *Proc. 1984 ACM Sympos. on Lisp and Functional Programming*, 1984, pp. 159-166.

[Wia75] Wijngarten, A. and al, "Revised report on the algorithmic language Algol 68," *Acta Informatica*, vol. 5, 1-3 (1975), pp. 1-236.