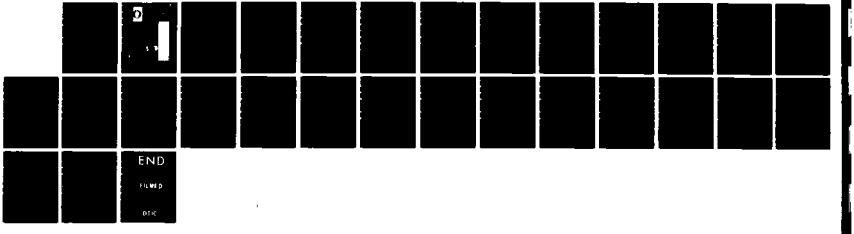
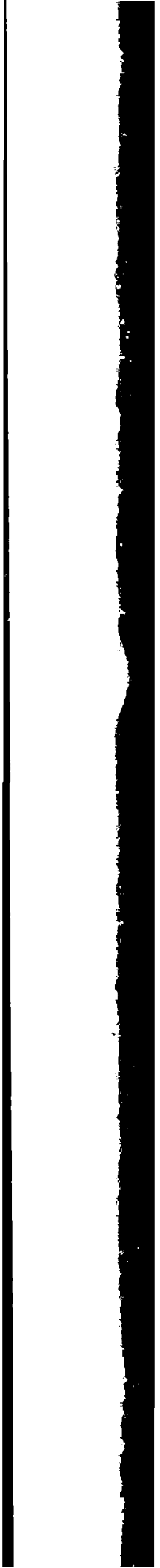
AD-A156 558 ON THE GARBAGE COLLECTION OF BLOCK STRUCTURED MEMORIES 1/1
(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN
(ENGLAND) S R WISEMAN MAR 85 RSRE-85006 DRIC-BR-95552
UNCLASSIFIED F/G 9/2 NL





Report No. 85006

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

ON THE GARBAGE COLLECTION OF BLOCK STRUCTURED MEMORIES

Author: S R Wiseman

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

March 1985

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report No 85006

TITLE: ON THE GARBAGE COLLECTION OF BLOCK STRUCTURED MEMORIES

AUTHOR: S R Wiseman

SUMMARY

A survey of existing garbage collection methods is presented. These are considered as candidates for use in a system where the memory address space is shared by all users of a computer. The special problems that arise from this requirement are described and it is shown that no one garbage collector solves all the problems.

List of Contents

1. Introduction

This paper surveys the various techniques that have been proposed and used for garbage collection. The various solutions are described and their suitability for use on a system wide basis, in a computer with a block structured memory, is discussed. Computers with capability based addressing [Fabry74] are examples of such systems.

A comprehensive review of early garbage collection techniques is given in [Knuth73], while [Cohen81] provides a more recent survey. This paper differs in that it considers the usefulness of the algorithms when applied system wide to a computer with a block structured memory.

Most garbage collectors have been designed for use in applications running on time shared computers. Here the various users of the system are quite independent, so the garbage collection of one user does not affect the others. Also the computer's virtual memory system provides separate memories for each user. Therefore if one user allocates all the free memory available to him, this will not affect the free storage of another user.

However, in a computer that uses a block structured memory system wide, the circumstances are quite different. A garbage collector that operates on the entire memory at once will affect all users. So, for example, if one user's activities require his data to be garbage collected frequently, the performance experienced by the other users will be unfairly degraded. Also if one user allocates excessive amounts of memory, then other users will unfairly be denied more memory. However, some form of rationing or quota system could be introduced to alleviate this problem.

Also, the system as a whole has to remain responsive enough to service peripherals adequately. A garbage collector that precludes normal operation for long periods will limit the usefulness of the computer.

Another requirement is that the garbage collector handles large cyclic structures efficiently. Cyclic structures are often assumed to occur infrequently and to be quite small. This is not the case, however, when the operating system kernel shares the memory with user processes. For example, cyclic structures involving many blocks arise if two processes deadlock. Each process is suspended on a semaphore queue which is accessible only from the other. This creates a cyclic structure, involving all the blocks in each process, which is garbage because it is inaccessible elsewhere.

We are therefore looking for a garbage collector that prevents users from denying others access to memory and cpu resources, that can operate largely in parallel with normal processing and is otherwise suitable for managing a system wide block structured memory. The purpose of this survey is to find suitable candidates or to identify techniques that could be used to construct a new garbage collector which is suitable .

2. Block Structured Memories

A block structured memory is one in which storage is viewed as many individual blocks, rather than as one large linear array. The blocks can contain a mixture of ordinary numeric data, such as integers and characters, and pointer data. Pointers refer to blocks as a whole, not to some word within a block, and are the sole means for addressing memory. To reference an individual word, a pointer to the block containing the word and an offset within the block to the word must both be specified.

A block structured memory is similar to a segmented memory. The difference is that pointers, which are used to reference blocks, can be identified by a general purpose garbage collector. Segments are referenced by numeric addresses which are indistinguishable from other numeric data. This means that a general purpose garbage collector is not possible, because it could not identify which segments are required.

It is necessary to distinguish between pointer and numeric data. This can be achieved by partitioning blocks into those that can contain pointers only and those that can never contain pointers. However, this presents difficulties when structuring data in high level languages. An alternative is to allow pointer and numeric data to be freely mixed together in a block. Then the most effective way of distinguishing between them is by tagging each word in memory with an extra bit. This 'tag_bit' is set if the word contains a pointer, otherwise it is clear. It is necessary to constrain pointers to being stored on word boundaries and to ensure that the tag bit is cleared if part of a word is overwritten. The tag bits are used and maintained only by the machine and are not accessible by the user.

It is also necessary to be able to determine the size of a block. This is required for storage management functions and for protection checks during runtime. The block size could be stored with the block in, say, the first word. This may be hidden from the users of the block or simply just protected from alteration by them.

When a new block is created, storage for it is found from a pool of free storage and the creator is returned a pointer to the new block. The free pool may be one large area of free storage or it may be a list of many smaller free areas. When a block becomes inaccessible, which is when all pointers to it have been overwritten or are in inaccessible blocks, the storage it occupies may be safely returned to the free pool.

The free pool may become fragmented such that, although it contains sufficient free storage in total to satisfy an allocation request, no individual free area is large enough to accomodate the new block. In this case it is usually appropriate to compact the memory, moving allocated blocks about to produce larger free areas.

3. Finding Inaccessible Blocks

Garbage collection involves detecting inaccessible storage blocks and returning the storage occupied by the block to the free pool. The first stage of garbage collection is, then, to discover which blocks are inaccessible. This can be done either by detecting when a block becomes inaccessible or by following all accessible pointers to find all accessible blocks.

3.1 Reference Counting

The reference count method, first introduced by [Collins60], attempts to detect when a block becomes inaccessible. For each block, a count is maintained of the number of pointers to the block. If the count ever drops to zero, then the block has become inaccessible and so its storage can be freed. However, before the storage can be put into the free pool, it must be scanned for pointers, so that the reference counts of the blocks they refer to can be decremented. This may recursively cause further blocks to be freed.

The count must be maintained when pointers are created and destroyed. When a block is created, a pointer to it is returned to the creator, therefore the reference count is initially set to one. Further pointers are created simply by copying, so it is necessary to check all memory writes to see if the data being written is a pointer. If so, it is necessary to increment the reference count of the block referred to by that pointer. A pointer is destroyed if it is overwritten, in which case the reference count of the block that it refers to must be decremented. Since pointers can generally be stored anywhere in memory, it is necessary to check whether a word contains a pointer before writing to that word.

Reference counting does not detect all inaccessible blocks. This is because a block can become inaccessible by the destruction of a pointer which is not the last pointer to the block. The simplest case is where a block contains a pointer to itself and is referred to by one other pointer stored in an accessible block. If the latter pointer is destroyed, then the block's reference count decreases from two to one. The block is therefore not freed even though it has become inaccessible. This is in general true for an arbitrarily complex cyclic structure.

3.1.1 Recovering Inaccessible Cyclic Structures

The inability to handle circular structures correctly is the greatest disadvantage of reference counting. This is especially so since the use of a block structured memory as the main memory of a computer, is likely to cause large cyclic structures to be created. The problem can be overcome either by detecting when a circularity is produced and then not incrementing the reference count, or by ignoring the problem and using another technique to release inaccessible cyclic structures.

The first solution was proposed by [Weizenbaum62&63] though, as pointed out by [McBeth63], detecting when a circular structure is formed involves a search of potentially the entire memory. There are special cases where it is known when circularities are produced, such as the use of the Y-combinator in combinator based systems. Systems which take advantage of this are described in [Friedman&Wise79] and [Brownbridge84]. However, these are not applicable to general purpose block structured memories.

Another, scheme is proposed by [Bobrow80]. In this, all the blocks of a circular structure are treated as a single group for deallocation purposes. A reference count to the group as a whole is maintained, but no counts within a group are kept. This scheme is not particularly suitable for a block structured memory system as memory usage is not divided into convenient groups and cyclic structures can be quite large.

The second solution, that of using another technique to find the inaccessible cyclic structures not detected by reference counting, is proposed by [Deutsch&Bobrow76] and [Christopher84]. In these hybrid schemes, reference counting is used to recover storage, except for inaccessible cyclic structures, until there is no free storage left. Then a scanning technique, see section 3.2, is used to recover any inaccessible cyclic structures.

3.1.2 Recursively Releasing Storage

The procedure for freeing a block is recursive, since a freed block may contain pointers to other blocks which consequently become free. Therefore a stack of some sort is required to control the recursion. Since the number of blocks freed recursively is potentially all the blocks in the memory, the stack size must be the maximum number of blocks in the memory. This is half the size of the memory, assuming each block has one overhead word per block to hold the block size.

The size of this stack is such that using a separate memory for it is prohibitively expensive. The use of a transaction file on disc, as in [Deutsch&Bobrow76], or a virtual memory system would be possible, but would be slow. The stack could be limited to some affordable size as long as stack overflow can be handled and does not happen very often. Finding blocks that require scanning, but are not on the stack because of overflow, involves visiting all the blocks in memory to find those with a reference count of zero.

The use of a separate stack can be avoided altogether by utilizing the storage of the blocks themselves. If a block is no longer referenced it is scanned for pointers. If one if found then the block it points to has its reference count decremented. If the count of this block falls to zero, then scanning of the first block is suspended and the new block is scanned. The word occupied by the pointer is used as the link word in the chain of blocks that have not been completely scanned. When scanning of a block is completed, a block is removed from this chain and the scanning of it continues. The word forming the link is of course the place that was scanned last. This scheme requires that the end of the block can be identified, since there is no room to store how much more scanning is required. If the start of a block is distinguishable then the scan can progress backwards instead.

3.1.3 Storing the Reference Count

An obvious requirement for a reference counting system is that a reference count for each block must be stored somewhere. The maximum number of pointers to a block at any one time is the size of the memory. Therefore the reference count field must be large enough to hold this number. This is potentially wasteful of memory, since most blocks are referred to by very few pointers. Two schemes have been proposed by [Deutsch&Bobrow76] to reduce this overhead.

Firstly the reference count field is made much smaller than the maximum required. Whenever the count reaches its maximum it is assumed to be 'infinite' and is never subsequently incremented or decremented. The count can therefore never reach zero so the block will never be freed. The scanning garbage collector which is used to release inaccessible cyclic structures also releases inaccessible blocks which have an infinite reference count. It is hoped that most blocks are referenced very few times, so that reference counts becoming infinite is a rare event.

Secondly it is assumed that the vast majority of reference counts are one. A hash table is used to record the reference count of all blocks whose reference count is greater than one. If a block is not in the hash table then a count of one is implied. The problem of hash table space overflow is not addressed, but it does make this approach unattractive for a block structured main memory system.

[Wise&Friedman77] propose the use of a single bit as a reference count. This indicates whether the count is one or greater than one. A simple cache memory is used to record some of those blocks whose reference count is two. This is on the assumption that most blocks have a count of one, but that they often increase to two temporarily, during pointer manipulation operations. Wise and Friedman suggest using the mark bit, required for the scanning garbage collector, as the reference count field. It is necessary to clear all the bits before scanning, but the reference count is easily restored afterwards.

A method for recomputing infinite reference counts during scanning garbage collection is proposed by [Wise79]. In this method the number of references is computed for each block at a time, so only one access is made to the reference count field. This contrasts with the more obvious technique of incrementing the reference counts of blocks whenever a pointer is found during scanning, which requires many increments to be performed for each reference count.

3.1.4 Accessing the Reference Count Field

During normal processing many pointers are created and destroyed, which means there is a lot of incrementing and decrementing of reference counts to be performed. The use of a transaction file is proposed by [Deutsch&Bobrow76] to save all increment and decrement actions. These are then processed by the system some time later, during a slack period. Techniques for reducing the number of increments and decrements are given by [Barth77], but these are compile time optimizations meant for language run-time systems.

3.1.5 Reading Before All Writes

The requirement to read a location before writing to it, in order that the overwriting of pointers can be detected, is a serious handicap to performance. This read could, of course, be avoided if it is known that no pointer could possibly be stored in that location, though on the whole this is not so.

This problem is much less serious if numeric and pointer data cannot be freely mixed in the same block. In this case the read check need only be performed when a pointer is written to memory. Also, computers with this partitioned type of block structured memory, tend to manipulate pointers less often than those with the general type.

The Plessey PP250 [England75], Cambridge CAP [Needham&Walker77] and Intel iAPX432 [Tyner81], are capability computers that have partitioned memory. Even so, none of these computers actually use reference counting for garbage collection, presumably because it is thought to be too inefficient.

3.2 Scanning for Accessible Storage

If each allocated block has a mark bit, and initially all these are clear, then by tracing all the accessible blocks and setting their mark bit, it is possible to discover all inaccessible blocks. This method, first proposed by [McCarthy60], is recursive.

McCarthy's algorithm uses a stack to control the recursion, but since this stack would need to be at least half the size of the memory, this is impractical. A more practical proposal is made by [Hanson77] which suggests the use of a spare pointer sized field per block. This is used to link together the blocks which have yet to be scanned.

A further variation of McCarthy's garbage collector is given by [Baecker72]. This is intended for use in virtual memory systems and has one mark bit per page as well as one per block. A single recursive scan is made to determine which pages contain accessible blocks. Pages that only contain inaccessible blocks are then freed along with their page table entry. The advantage of this system is that compaction is unnecessary, but the disadvantage is that pages are not freed until they are completely inaccessible.

3.2.1 Pointer Reversal

A method which does not require an auxiliary stack is given by [Schorr&Waite67]. The algorithm scans a block looking for pointers to blocks which have not been marked. When one is found the position of the pointer is remembered on a list, using the location itself as the link word. Scanning then continues in the new block. When scanning a block is completed, the list is popped to find the location of the pointer which led to the block. The pointer's value is restored and scanning then continues at the location after the pointer. For a formal description of the algorithm, see [Broy&Pepper82].

It is necessary to detect when scanning a block has been completed. Since Schorr and Waite are dealing with LISP structures they only require a single bit per node to indicate which word has been scanned; the reversed pointers actually point to the start of the node. In a system with variable size blocks this approach would require an extra field the size of the block size field.

An alternative method is possible if the start of a block can be distinguished from the rest of it. By scanning the blocks from the end towards the start, the end of the scan is given simply by detecting the start of the block.

The Schorr/Waite algorithm is given as a way to mark accessible blocks. However, if the step which restores the reversed pointers is omitted then the algorithm can rearrange the memory so that the first word of each accessible block contains the head of a list of all the pointers to the block. The list is contained in the locations of the pointers themselves and ends with the original contents of the first word of the block. This structure can then be used to update the pointers ready for compaction. This is discussed in section 4.1.2 of this paper.

3.2.2 Non Recursive Scanning

The stack required to control the scanning operation can be avoided altogether. This is achieved by making repeated scans of the memory to find accessible pointers to unmarked blocks. This is the method adopted by [Dijkstra.et.al.78]. Two bits per block are required, one for marking whether a block is accessible, the other for marking whether it has been scanned or not. Blocks that are marked as accessible are scanned to completion and then marked as scanned. Any blocks for which pointers are found are marked as accessible. This method is less efficient than pointer reversal, because repeated visits to each block in memory are required to find accessible unscanned blocks. The advantage, however is that the memory remains useable whilst scanning is in progress. For this reason it is suitable for the on the fly garbage collector described by Dijkstra.

3.3 Memory Copying

Garbage collection of a memory can be achieved easily if a spare memory is available. All accessible blocks are copied from the memory into the spare, where they are placed compactly. This leaves one area of free storage from which blocks can easily be allocated. The roles of the two memories are then reversed, the first becomes the spare whilst the second becomes active.

3.3.1 Two Memory Copying

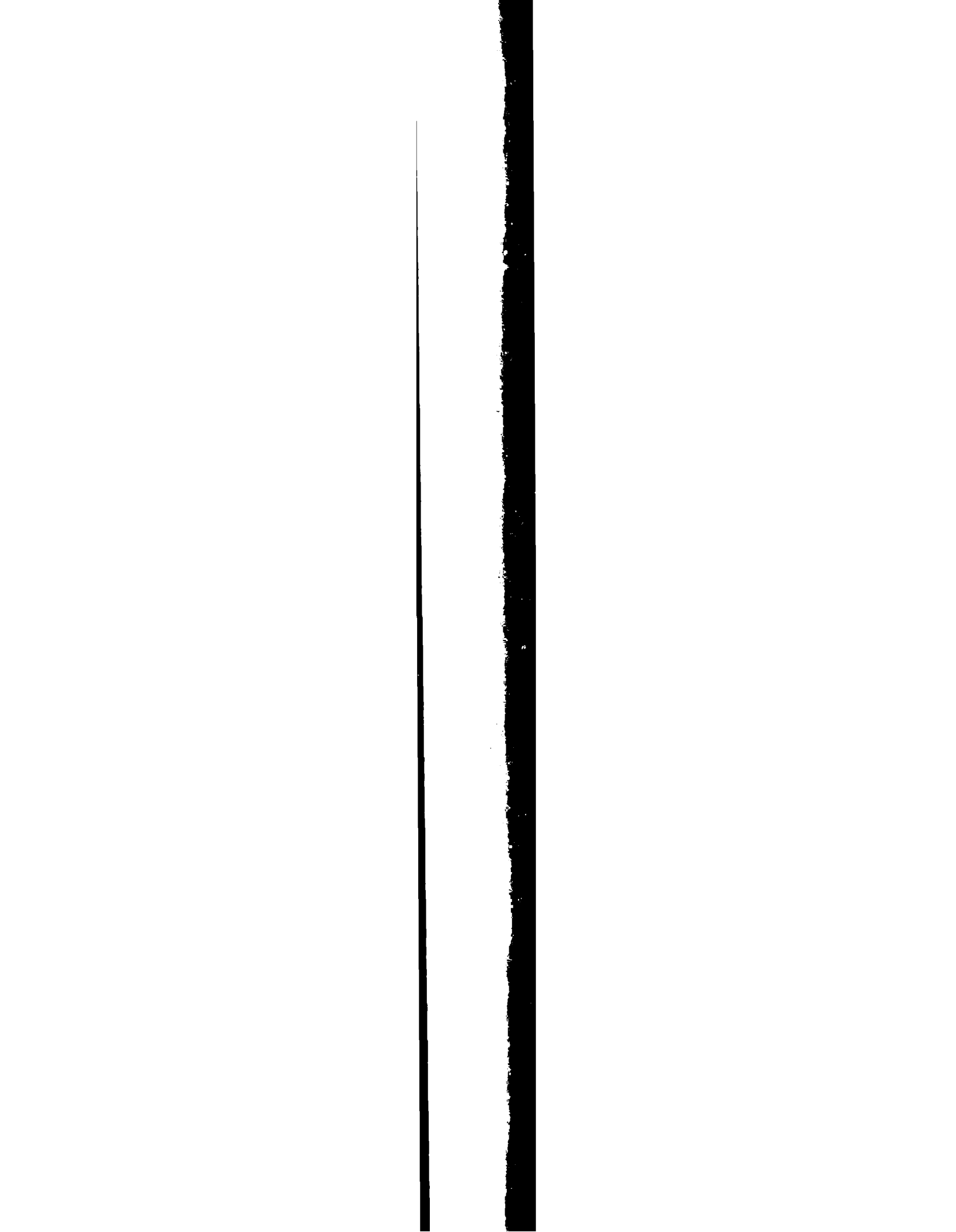
The copying process is recursive in nature and, since the memory is also compacted, it is necessary to update all pointers that are copied. The algorithm proposed by [Hansen69] is explicitly recursive and hence will require a stack to control the recursion.

Hansen's algorithm makes two recursively intertwined passes across the memory. First all the pointers in a block are found and the algorithm is applied recursively to the blocks to which they refer. This gives the new location of those blocks. The pointers are then updated and the updated block is copied to its new location. Two bits are used to mark the blocks. One indicates that the block has been found and is being updated. When a block has been moved, its new address is stored in the old copy and the second mark bit is set. A fixup table is used to cope with circularities.

A similar algorithm for LISP is given by [Fenichel&Yochelson]69]. Whilst their algorithm is recursive, they suggest that the [Schorr&Waite67] pointer reversal method could be applied. This is where the space occupied by the pointers themselves is used to control the recursion, thus eliminating the need for a separate stack.

Another scheme is given by [Cheney70]. The pointers themselves are used to control the recursion, though in a much simpler fashion than in Schorr/Waite pointer reversal. A version that does use Schorr/Waite pointer reversal is given by [Reingold73]. Some improvements to this are suggested in [Clark76].

In Cheney's algorithm, two index variables, "next" and "scan", which point into the spare memory, are used. "next" indicates where the next block to be copied is to be placed, "scan" indicates the progress of a single scan of the copied blocks. Initially both are set to zero, then any blocks known to be accessible are copied, with "next" being suitably incremented. The blocks are copied without modification so any pointers point back into the active memory.



to search for pointers. When
d to is copied to "next", if
. When a block is copied a
in the block so if further
can readily be updated. The
nt to the new location of the
The scan finishes when "scan"

needed in each pointer to
red to, it must be possible to
moved or not. This can easily
it on the block size field.

g a two-memory copying
e systems. Cheney's algorithm
reversal techniques, the
ate whilst copying is in
the active memory is accessed
the block has moved to the
er is updated and the access
stead.

rther developed by
oose dividing the memory up
two, which are kept in order
region and from a younger to
d normally. However, each
through which all pointers
gion must pass.

arted by copying the
o a spare region, as in
e sure that no pointers into
ary to scan all younger
that region. Also, blocks
indirection table are
re copied. Only when the
torage occupied by the region

der regions contain
ittle garbage whilst the
rbage. It is therefore
he younger regions more
the older ones since much
efore the rate of garbage
e varied according to their
the garbage collector.

Cyclic structures can
boundaries. If these become
algorithm fails to recover t
a region and all those young
is possible to recover any i
cross region boundaries but
copied regions. If a cyclic
oldest region then it is nec
memory in order to recover i

Lieberman and Hewitt
from younger to older region
are rare. This may be true o
algorithm is intended, but i
computer with a block struct

3.3.3 Multiple Independent R

The garbage collector
described by [Bishop77] also
style copying garbage collec
and Hewitt, but is performed
individual regions.

For each region two l
pointers that leave a regior
enter a region. Therefore ea
two lists. The list of incon
blocks that are immediately
garbage collection. The list
remove the pointers from the
after the region has been co

Bishop's garbage coll
regions, in an attempt to in
of pointers. This is mainly
performance in a paged virtu
program's working set, but a
inaccessible cyclic structur
one region. Once this happer
recovered by normal garbage
a satisfactory solution for
structures, because they wil

Inter region pointers
three word link block. Two
the incoming and outgoing li
address of the actual block.
when a pointer is copied fr
block in another region. Th
on copying pointers.

When a pointer is copied between blocks in the same region, no extra overhead is imposed. If copies of an inter region pointer are made, then they will use the same link block. However if a pointer is copied from another region it is not possible to tell, without searching the lists, whether a link block already exists for that pointer. Either time must be spent searching the lists or a link block is allocated regardless of any duplication. The link blocks could therefore become a large overhead if pointers are often copied across region boundaries.

To avoid the need for storing and maintaining link blocks for pointers from rapidly changing regions, such as the temporary store of a process, to relatively more stable regions, such as the operating system, Bishop proposes the Cable. If a region A is Cabled to region B, then pointers in A can point directly to blocks in B. A consequence of this is that when region B is garbage collected, region A must be as well. However the garbage collection of region A can still occur independently of region B.

The problem with cables is that it is difficult to decide when to use them and that if they are used indiscriminately garbage collection can no longer be performed independently on each region.

3.3.4 Infrequent Garbage Collection

An interesting variation on the copying method of garbage collection is suggested by [White80]. This is to perform garbage collection very infrequently, say once a year, and in the meantime rely on a vast virtual memory system to supply new space. When space really does get low a large physical memory is used as a spare memory, into which the virtual memory is copied. White suggests this large memory could be hired from a garbage collection contractor, just for the duration of the garbage collection.

Whilst this approach seems attractive, especially with the advent of large density write-once laser discs, it has two drawbacks. Firstly a virtual memory may be undesirable for the application, for example where real-time response is required. Secondly a computer that needs a large virtual memory to operate, will always cost more than one that does not.

3.4 Explicit Deallocation

Although the idea of automatic garbage collection is to relieve the user of the responsibility for deallocating unwanted blocks, there are circumstances where explicit deallocation could be useful. For example, on return from a procedure call the activation record could be discarded.

The problem with this, however, is that "dangling references" may be left. This is where a pointer to a block remains after the block has been deallocated. If that pointer is subsequently used then the result will be unpredictable.

[Lomet75] describes a scheme for invalidating all the pointers to a block which is explicitly deallocated. This prevents any dangling reference problems. Lomet's solution is to place a "tombstone" at the site of the deallocated block. Whenever a pointer is used to access a block a check is made to see if the block has been replaced by a tombstone. If so the pointer is set to nil and the access fails.

The tombstone may occupy the first word of the block, in which case the rest of its storage can be returned to the free pool. This however will cause the memory to be heavily fragmented and so may be ineffective at reusing storage. If an indirection table is used to implement pointers, then the tombstone may be placed in the indirection entry. This would allow the entire block to be deallocated, reducing fragmentation.

4. Reallocation of Storage

Once the inaccessible blocks have been found, the storage they occupy can be returned to the free pool where it can be used to allocate new blocks. There are two types of free pool, those where there is only one area of free store and those where there is more than one.

Storage allocation from a free pool which consists of just one free area is easy. The block is allocated from the start of the area and the area is made smaller. Garbage collection is required when the size of the free area is less than that of the block requested. Returning inaccessible blocks to such a free pool is more difficult, since they are dispersed between the accessible blocks. It is necessary to compact the accessible storage to one end of store, leaving one free area at the other end.

A free pool which consists of many areas of free store can be constructed in several ways, the simplest being a linked list. Inaccessible blocks are returned to the pool by adding them to the list. To allocate a new block, the list is searched for a free area which can accomodate it. It is then allocated from this area with any remaining free space staying on the free list. If no free area is large enough then garbage collection is necessary. However this may not result in an area large enough being found. The free pool may actually contain enough store, but be fragmented into many smaller pieces. In this case it will probably be worth compacting the memory to produce larger free areas so that the allocation request can succeed.

Compaction is not required for systems which have a fixed block size, such as LISP. However, it is sometimes used to reduce fragmentation in virtual memory systems.

4.1 Compaction

Storage compaction involves moving some blocks and updating all the pointers to those blocks to reflect their new location. This may be done as a final pass of compacting garbage collection or may be a separate affair. Compaction is inherent in the copying style garbage collectors described in 3.3.

4.1.1 Indirection Tables

The use of an indirection table to implement pointers greatly eases the problems of compaction. The table contains the addresses of all the blocks in memory, whilst each pointer contains the index, within the table, of the entry for the block it points to. Whenever the block referred to by a pointer is to be accessed, the entry for that block must be read from the indirection table to discover the block's address.

If a block is moved as the result of compaction then by altering the address in the indirection table, all pointers to the block are simultaneously updated. It is not necessary to find all the pointers to the block and update them individually. The disadvantages of the indirection table approach are that space must be found for the table and that going through the indirection table to reach a block takes time. The latter problem, however, is greatly reduced by using a simple cache memory or other special mapping hardware.

If the maximum number of blocks are allocated then the indirection table would be one third the size of memory. This is assuming a one word size field, one word indirection table entries and one data word per block. Preallocating a table of this size is too wasteful of memory to be considered viable. Choosing a smaller size is a compromise between wasting memory and having enough entries available for peak demands.

Dynamically altering the space occupied by the table is possible, though it becomes necessary to be able to compact the table space as well as the memory space. However, this is much simpler since the entries are all the same size.

The use of indirection tables has not been given very much consideration is past literature. This is because previous work has centred on LISP systems, in which the block size is always two. The use of an indirection table would therefore impose a serious overhead in time and space.

Indirection has been used in some capability computers, such as the Cambridge CAP computer and the Plessey PP250. Notably the Intel iAPX432 uses a two level indirection table to avoid the problem of preallocating enough table space.

4.1.2 Pointer Updating

If pointers contain the address of the block they point to, then when a block is moved all the pointers to it must be found and updated.

An algorithm for compacting store in this way was first given by [Haddon&Waite67]. While the accessible blocks are moved, a table is constructed which gives the new location of each set of consecutive accessible blocks. When all the blocks have been moved a linear scan is made of the accessible storage. Any pointers are found and updated using the table.

Haddon and Waite show that no extra storage is required for the table, because it can always fit in the available free space. However, as compaction proceeds it becomes necessary to relocate the table. Improvements to this algorithm are proposed by [Fitch&Norman78] which speed up the accesses to the relocation table. [Berry&Sorkin78] show how the algorithm can be modified to give improved performance when the blocks are allocated and discarded in a stack like fashion, as is usual for procedure activation records.

[Wegbreit72] gives an algorithm which updates all the pointers before moving any blocks. The free block located before a consecutive set of accessible blocks is used to hold their new address. To update a pointer it is necessary to find the first free block preceeding the block referred to, since this gives the new address. This is accomplished by searching from the start of store until the free block is found, though this search can be speeded up by constructing a directory.

The use of an extra address field in each pointer is suggested by [Zave75]. This field is used to link together all pointers, sorted in order of the address of the block they point to. The pointers are then updated in one pass by following this list. The store is then compacted.

The method of using reversed pointer chains, which link all pointers to a block together, to facilitate compaction was first suggested by [Fisher74]. Fisher's algorithm however, only works for systems in which the pointers all run in the same direction. [Morris78] gives a more general scheme. This uses two separate passes, one forwards and one backwards, in order to process both forward and backward pointers. A similar algorithm, which makes two forward passes, is given by [Jonkers79]. [Martin82] gives a faster version of Fisher's algorithm.

The algorithms of Haddon and Waite, Morris and Jonkers are compared in [Cohen&Nicolau83] using results obtained from a PDP10.

The compacting garbage collector proposed by [Thorelli76] also uses reversed pointers. However an extra word per block is used to control the recursive scanning. The algorithm used in the Flex computer, [Foster et.al.79], avoids the use of this extra word. This is done by adding all pointers, except the first, to the reversed list after the first pointer. By ensuring that the first pointer on the reversed list is the first that was found, it can be used to control the recursive scan, as originally proposed by Schorr and Waite. The pointers are updated in a separate pass, before a final pass compacts the blocks. This makes the restrictions imposed by Fisher unnecessary.

4.2 Storage Allocation

The allocation of new blocks from a free pool consisting of one free area is straightforward. However with multiple free areas there are several possible allocation strategies. The free areas are chained together on a linked list or in a tree structure so that they can be searched.

In the first fit strategy, the list is searched and the block is placed in the first area found which is large enough to contain it. For the best fit strategy, the block is placed in the smallest area which is large enough to contain the block.

The cyclic placement strategy is similar to first fit, except that the search continues in a round robin fashion, rather than starting at the beginning each time a block is allocated.

The different schemes are a compromise between the time taken to place an inaccessible block in the free pool, the time taken to allocate a new block and the storage utilisation gained. Which scheme is best will depend on the pattern of storage usage in the computer.

It may be possible to tailor the placement strategy dynamically by using special hardware. A cache technique could be used to give the best free area for the most recently used block sizes. This would obviate the need for searching the free area list except on a cache miss.

5. Multi-Processor Garbage Collection

The use of two processors, one for list processing the other for garbage collection, was first suggested by [Steele75] as a way of avoiding the pause in list processing experienced when using most garbage collectors.

In Steele's system, the garbage collection processor operates continually. It scans the memory marking accessible blocks, using a stack to control recursion, and then returns any newly freed blocks to a free list. The system is intended to run LISP in a virtual memory environment and so compaction is also performed, to reduce the size of the working set.

An analysis of a two processor system is provided by [Wadler76]. Conditions are given which ensure that the free list is never exhausted. This allows the list processor to run uninterrupted. Wadler concludes that this type of garbage collection requires twice as much processing power as the regular type.

A two processor garbage collector was taken by [Dijkstra.et.el.78] as an example in proving the correctness of a multiprocess program. Steele's original proposal used many semaphores to synchronize the two processors. Dijkstra attempts to limit the amount of synchronization required, thus keeping the list processor's overhead to a minimum. The flags used to control the marking of nodes is described in terms of colours. This has become "standard" notation for parallel garbage collectors.

The algorithm is extended by [Lamport76] to allow more than one list processor and more than one garbage collector processor. A correctness proof for this is given.

Another two processor garbage collector is described by [Kung&Song77]. This avoids the use of critical sections by relying on the mutual exclusion inherent in accessing the memory. A special queue is also used to hold pointers to all the nodes that have yet to be scanned.

The results of a study of Wadler's and Lamport's algorithms are presented in [Newman.et.al.83]. Some improvements to both are suggested which give significant speed increases.

All the multi-processor garbage collectors that have been presented are basically the same and they all suffer from the same serious fault. This is due to the fact that list processing systems use memory intensively. This is also true of high level language oriented computers such as Flex. The use of multiple processors connected to a single shared memory in these circumstances is more likely to produce a drop in performance than any gain. This is due to the extra time required for memory contention.

6. Conclusions

There are three main classes of garbage collector; scanning, copying and reference counting. They differ in the number of memory accesses required to perform garbage collection, in the utilization of the memory and in whether they require normal list processing to be suspended while they operate.

The scanning class consists of the serial type, perfected in the Flex computer, and the parallel type, first introduced by Steele. Whilst the serial version is much faster, it does require the suspension of normal operation while it operates. Therefore, a parallel system may be preferred, despite its inefficiency, because list processing can normally continue uninterrupted.

Cheney's algorithm for a copying type garbage collector is fast, but because a spare memory is required, its utilization is poor. However, as shown by Baker, it is possible to adapt the algorithm for real time use.

The reference counting schemes do not recover inaccessible cyclic structures, which makes them useless for general purpose systems. Several authors propose using a hybrid scheme in which reference counting is used to recover as much storage as possible. When necessary, a scanning garbage collector is used to recover cyclic structures. Reference counting is, however, very expensive in its usage of memory and also requires extra space for the reference counts. It is therefore impractical as a system for the garbage collection of main memory.

It seems that none of the garbage collectors surveyed are entirely suitable for use on a system wide basis. Although solutions to each of the specific problems are to be found, no one system is entirely satisfactory. However, the proposal of Bishop is perhaps the closest. This system solves the major problems but is likely to be inefficient, both in terms of processing overhead and memory utilization, and does not cope well with large cyclic structures.

All the garbage collectors examined are concerned with single linear memories. However, large system wide memories, especially in a distributed computing environment, are likely to be hierachical in nature. A suitable garbage collector may well be able to exploit this structure.

The aim of further research must be to produce a garbage collector which is suitable for use on a system wide basis. Bishop's algorithm seems an appropriate starting point. This could be developed in a hierachical fashion to give a garbage collector suitable for use in distributed systems.

7. References

H.D.Baecker
Garbage Collection for Virtual Memory Computer Systems.
Comms. of the ACM
Vol 15, Num 11, Nov 1972, pp981..986

H.G.Baker
List Processing in Real Time on a Serial Computer.
Comms. of the ACM
Vol 21, Num 4, April 1978, pp280..294

J.M.Barth
Shifting Garbage Collection Overhead to Compile Time.
Comms. of the ACM
Vol 20, Num 7, July 1977, pp513..518

D.M.Berry & A.Sorkin
Time Required for Garbage Collection in Retention Block-Structured Languages.
Int. Jour. of Computer and Information Sciences
Vol 7, Num 4, 1978, pp361..404

P.B.Bishop
Computer systems with a Very Large Address Space and Garbage Collection.
PhD Thesis
Massachusetts Institute of Technology, May 1977

D.G.Bobrow
Managing Reentrant Structures Using Reference Counts.
ACM Trans. on Programming Languages and Systems
Vol 2, Num 3, July 1980, pp269..273

D.R.Brownbridge
Recursive Structure in Computer Systems.
PhD Thesis
University of Newcastle upon Tyne, July 1984

M.Broy & P.Pepper
Combining Algebraic and Algorithmic Reasoning: An Approach to the Schorr-Waite Algorithm.
ACM Trans on Programming Languages and Systems
Vol 4, Num 3, July 1982, pp362..381

C.J.Cheney
A Nonrecursive List Compacting Algorithm.
Comms. of the ACM
Vol 13, Num 11, Nov 1970, pp677..678

T.W.Christopher
Reference Count Garbage Collection.
Software - Practice and Experience
Vol 14, Num 6, June 1984, pp503..507

D.W.Clark
An Efficient List-Moving Algorithm using Constant Workspace.
Comms. of the ACM
Vol 19, Num 6, June 1976, pp352..354

J.Cohen & A.Nicolau
Comparison of Compacting Algorithms for Garbage Collection.
ACM Trans. on Programming Languages and Systems
Vol 5, Num 4, Oct 1983, pp532..553

J.Cohen
Garbage Collection of Linked Data Structures.
ACM Computing Surveys
Vol 13, Num 3, Sept 1981, pp341..367

G.E.Collins
A Method for Overlapping and Erasure of Lists.
Comms. of the ACM
Vol 3, 1960, pp655..657

L.P.Deutsch & D.G.Bobrow
An Efficient, Incremental, Automatic Garbage Collector.
Comms. of the ACM
Vol 19, Num 9, Sept 1976, pp522..526

E.W.Dijkstra, L.Lamport, A.J.Martin, C.S.Scholten & E.F.M.Steffens
On-the-Fly Garbagae Collection: An Exercise in Cooperation.
Comms. of the ACM
Vol 21, Num 11, Nov 1978, pp966..975

D.M.England
Capability Concept Mechanisms and Structure in System 250.
Rev. Fr. Autom. Inf. Rech. Oper. (France)
Vol 9, Sept 75, pp47..62

R.S.Fabry
Capability Based Addressing.
Comms. of the ACM
Vol 19, July 1974, pp403..412

R.R.Fenichel & J.C.Yochelson
A LISP Garbage-Collector for Virtual-Memory Computer Systems.
Comms. of the ACM
Vol 12, Num 11, Nov 1969, pp611..612

D.A.Fisher
Bounded Workspace Garbage Collection in an Address-Order Preserving List Processing Environment.
Information Processing Letters
Vol 3, Num 1, July 1974, pp29..32

J.P.Fitch & A.C.Norman
A Note on Compacting Garbage Collection.
The Computer Journal
Vol 21, Num 1, Feb 1978, pp31..34

J.M.Foster, C.I.Moir, I.F.Currie, J.A.McDermid, P.W.Edwards, J.D.Morison & C.H.Pygott
An Introduction to the Flex Computer System.
RSRE Report 79016
Oct 1979

D.P.Friedman & D.S.Wise
Reference Counting Can Manage the Circular Environments of Mutual Recursion.
Information Processing Letters
Vol 8, Num 1, Jan 1979, pp41..45

B.K.Haddon & W.M.Waite
A Compaction Procedure for Variable-Length Storage Elements.
The Computer Journal
Vol 10, Aug 1967, pp162..165

W.J.Hansen
Compact List Representation: Definition, Garbage Collection, and System Implementation.
Comms. of the ACM
Vol 12, Num 9, Sept 1969, pp499..507

D.R.Hanson
Storage Management for an Implementation of SNOBOL4.
Software - Practice and Experience
Vol 7, 1977, pp179..192

H.B.M.Jonkers
A Fast Garbage Compaction Algorithm.
Information Processing Letters
Vol 9, Num 1, Jul7 1979, pp26..30

D.E.Knuth
The Art of Computer Programming, Vol 1.
Addison-Wesley, 1973

H.T.Kung & S.W.Song
An Efficient Parallel Garbage Collection System and its Correctness Proof.
Procs. 18th Ann. Symp. on Foundation of Computer Science, 1977
pp120..131

L.Lamport
Garbage Collection with Multiple Processes: An Exercisein Parallelism.
Procs. 1976 Int. Conf. on Parallel Processing
pp50..54

H.Lieberman & C.Hewitt
A Real Time Garbage Collector Based on the Lifetimes of Objects.
Comms of the ACM
Vol 26, Num 6, June 1983, pp419..429

D.B.Lomet
Scheme for Invalidating References to Freed Storage.
IBM Journal of Research and Development
Jan 1975

J.H.McBeth
On the Reference Counter Method.
Comms. of the ACM
Vol 6, Num 9, Sept 1963, p575

J.McCarthy
Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1.
Comms. of the ACM
Vol 3, 1960, pp181..195

J.J.Martin
An Efficient Garbage Compaction Algorithm.
Comms. of the ACM
Vol 25, Num 8, Aug 1982, pp571..581

F.L.Morris
A Time- and Space- Efficient Garbage Compaction Algorithm.
Comms. of the ACM
Vol 21, Num 8, Aug 1978, pp662..665

R.M.Needham & R.D.H.Walker
The Cambridge CAP Computer and its Protection System.
Operating System Reviews
Vol 11, Num 5, Nov 77, pp1..10

I.A.Newman, R.P.Stallard & M.C.Woodward
Improved Multiprocessor Garbage Collection Algorithms.
Procs. 1983 IEEE Int. Conf. on Parallel Processing
Aug 1983, Columbus, Ohio, pp367..368

E.M.Reingold
A Nonrecursive List Moving Algorithm.
Comms. of the ACM
Vol 16, Num 5, May 1973, pp305..307

H.Schorr & W.M.Waite
An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures.
Comms. of the ACM
Vol 10, Num 8, Aug 1967, pp501..506

G.L.Steele
Multiprocessing Compactifying Garbage Collection.
Comms. of the ACM
Vol 18, Num 9, Sept 1975, pp495..508

L.Thorelli
A Fast Compactifying Garbage Collector.
BIT
Vol 16, Num 4, 1976, pp426..441

P.Tyner
iAPX-432 General Purpose Data Processor: Architecture Reference Manual.
Intel Corp. Jan 1981

P.L.Wadler
Analysis of an Algorithm for Real Time Garbage Collection.
Comms. of the ACM
Vol 19, Num 9, Sept 1976, pp491..500

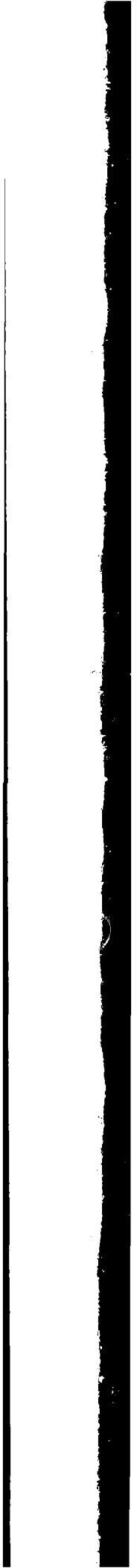
B.Wegbreit
A Generalised Compactifying Garbage Collector.
The Computer Journal
Vol 15, Num 3, Aug 1972, pp204..208

J.Weizenbaum
Knotted List Structures.
Comms. of the ACM
Vol 5, Num 3, March 1962, pp161..165

J.Weizenbaum
Symmetric List Processor.
Comms. of the ACM
Vol 6, Num 9, Sept 1963, pp524..543

J.L.White
Address/Memory Management for a Gigantic LISP Environment or, GC Considered Harmful.
Procs. LISP 80 Conference
July 1980, pp119..127

D.S.Wise
Morris's Garbage Compaction Algorithm Restores Reference Counts.
ACM Trans. on Programming Languages and Systems
Vol 1, Num 1, July 1979, pp115..120



Count.

59

age Collector.
Letters
5, pp167..169

DOCU

Overall security classification of sheet ..UNCL

(As far as possible this sheet should contain onl
classified information, the box concerned must be

1. DRIC Reference (if known)	2. Originator's R Report 850(
5. Originator's Code (if known)	6. Originator (Co Royal Sign
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Ag

7. Title
ON THE GARBAGE COLLECTION (

7a. Title in Foreign Language (in the case of tr

7b. Presented at (for conference papers) Titl

8. Author 1 Surname, initials Wiseman S R	9(a) Author 2
11. Contract Number	12. Perio

15. Distribution statement
UNLIMITED

Descriptors (or keywords)

Abstract

A survey of existing garbage colle
considered as candidates for use i
shared by all users of a computer.
requirement are described and it i
all the problems.

END

FILMED

8-85

DTIC