D. B. Lomet

Scheme for Invalidating References to Freed Storage

Abstract: A storage management scheme is described that supports the invalidation of addresses to freed storage and thus, in that sense, provides a secure system. Unlike previous virtual memory techniques, the allocated areas of our scheme can vary from the very large, requiring multiple pages of storage, to the very small, in which several can be contained on a single page. Special treatment is accorded procedure activation storage so as to provide increased effectiveness for this important case. The interaction of this deletion scheme with garbage collection techniques is also examined. Finally, the relative advantages of retention and deletion strategies of storage management are considered.

Introduction

In modern high-level languages, e.g., ALGOL 68 [1] and PL/I [2], in which references (or pointers) are data values and in which storage can be allocated dynamically and, more specifically, deleted or freed, the problem arises of determining the validity of a reference. This is the so-called dangling reference problem. Consider the PL/I program fragment of Table 1. The BASED declaration X on line (1) is first used to allocate a FIXED point variable at line (2). At line (4) this FIXED point variable is freed, i.e., destroyed, its storage being returned to a free list. At line (5) an attempt is made to use this variable, referenced via P. This reference is dangling because the variable no longer exists. The dangling reference problem exists in other variants as well. Labels and procedures, when treated as data values, can give rise to the same difficulty. The problem here consists of determining when a procedure activation exists. Dangling references, in all their variants, arise when references or values that include references can persist longer than the storage objects to which they refer.

The following two approaches to coping with this problem have evolved, both representing attempts to avoid the creation of dangling references:

1. *Scope rules* Scope rules, introduced in ALGOL 68, prevent the assignment of references (or procedures and labels that implicitly contain them) to variables that can persist longer than the objects to which they refer. Such scope rules have been instituted only for references to procedure activations and for labels and procedures that implicitly reference activations, because the lifetime of activations is specified, i.e., the activations follow a stack discipline. Assignment of references from higher in the stack (the stack growing down) to variables lower in the stack is permitted, but not the reverse. The chief virtue of scope rules is that a compile time check is frequently sufficient to assure that the rules have not been violated. This is not always the case, however, and full enforcement of the rules requires run time interpretation. Such thorough enforcement is not really anticipated for ALGOL 68, it being considered too expensive [1]. Further, of course, scope rules reduce function, although how seriously is a subject of some debate.

2. *Retention* Even if scope rules were instituted to control the persistence of references involving procedure activations, this would not cope with based or heap storage where storage objects have no implicit lifetime. The dangling reference problem can be eliminated here by adopting the so-called retention strategy, i.e., no explicit FREE or destroy operator is provided. Thus objects exist for as long as any references to them exist. What happens to an object when there are no outstanding references is not part of the semantics of the language being supported. However, the viability of the retention strategy depends on some form of storage reclamation, i.e., "garbage collection." The retention strategy can also be applied to procedure activation storage [3, 4, 5]. Activations then no longer follow a stack discipline. Several implementation methods have been described recently to realize the retention strategy for procedure activations in some reasonable fashion [6, 7].

Unlike scope rules, there seems to be little that can be done at compile time to reduce the cost of the retention. This cost is extra storage consumed by the no-longer referenced storage objects, extra time for garbage collection of this storage, or some combination

of the two. The retention strategy, at least for based or heap storage, also reduces function. It is no longer possible for a program to specify that a storage object is to be freed when it no longer serves some sensible purpose. As a consequence, large amounts of data (perhaps entire files) continue to exist until their last accessible references are destroyed.

In contrast to these two solutions there is the approach that ignores the problem, i.e., dangling references are created, and no effort is made to detect them. The advantage of this approach is that there is no performance loss in run time checking or garbage collection. Further, there is no loss of function. Retention for heap storage can be supported by simply never freeing storage. What is lost, however, is error detection capability, and the integrity of programs and storage may be compromised.

This paper presents a method, not for eliminating dangling references, but rather for automatically invalidating all such references. Thus, an attempt to access storage using a dangling reference is detected, and an exception is indicated. This approach requires run time interpretation. To make this interpretation reasonably efficient requires hardware assistance, which greatly reduces time and space penalties. The cost is primarily in terms of extra logic, which, given current and projected technologies, seems likely to be quite reasonable.

Tombstones

To invalidate dangling references, an indicator, called a tombstone, must persist after an object is destroyed. The tombstone might be located in any of three places.

1. *In the references* This requires that a list of all references to an object be maintained, so that it is possible to find and invalidate each reference when the storage is freed. This approach is untenable because of the enormous overhead.
2. *In the storage area* The idea here is to reserve part of each storage area for its tombstone(s). Each reference to the area must check a tombstone to determine whether the reference is valid or dangling. There are two problems with this approach:
   a. One cannot reclaim the tombstone part of the storage area when the area is freed. This part must persist to trap dangling references. This creates a serious storage fragmentation problem.
   b. Selection of a subarea of a major area leads to difficulties. Each potential subarea could have its own tombstone, a very costly solution, if it is a solution at all. Freeing the area would require the resetting of all the subarea tombstones, and none of them could be reclaimed. Alternatively, all pointers to subareas could retain knowledge of the location of the tombstone for the major area. If one were to do this, however, it would seem more desirable to use alternative three, which avoids the storage fragmentation problem.
3. *In a separate location* As indicated above, this requires that each address not only contain the location of the storage area it references but, in additon, the location for its tombstone. This approach results in an increase in the size of an address and requires a separate access to the tombstone each time the address is used. However, it is this approach that holds the most promise, given the right kind of hardware assistance. The remainder of the paper elaborates on this approach and the hardware assistance needed to realize it.

Table 1 A PL/I procedure fragment illustrating dynamic storage allocation and freeing. The use of P at (3) is valid whereas its use at (5) is dangling because the storage referenced by P has just been freed at (4).

```
Q: PROC;
   DCL Y FIXED;
   DCL P POINTER;
(1) DCL X BASED FIXED;
    .
    .
    .
(2) ALLOCATE X SET (P);
    .
    .
    .
(3) Y = P → X;
    .
    .
    .
(4) FREE P → X;
(5) Y = P → X;
    .
    .
    .
    END;
```

The area machine

There are more potential difficulties with addresses than just the dangling reference problem. When implementing a high-level interface, it is necessary to keep track of the form of data referenced by each address in order to check that such data are correctly interpreted. Several techniques, or combinations of them, can be used for this purpose. If the language is sufficiently constrained, most of this checking can be done at compile time. Run time interpretation is only needed when checking at compile time is not feasible.

Similarly, a low-level interface, when coupled with a virtual memory, e.g. [8, 9 10], provides a mechanism for checking that addresses are used correctly though this checking is at a more primitive level. More specifi-

cally, when an address is used or address computation is performed, the resulting address is checked to assure that it references a valid segment and is within the bounds specified by this segment. This checking assures that derived addresses remain within the boundaries of the original areas.

Here we describe a method of realizing an *area machine*. This machine combines an essentially low-level interface with a memory very much like a segmented virtual memory. The difference is a pragmatic one. Generally one cannot afford to create a new segment every time a new area of storage is allocated. Segment creation is usually too expensive to be used every time a procedure activation is created or BASED storage is allocated. Further, segments are typically paged, and hence each is associated with its own page table. This means that the amount of actual storage consumed by a segment is some multiple of the page size, thus discouraging the allocation of small segments. Some Burroughs Machines [10] do not page segments, but this leads to a reduction in maximum segment size and to storage fragmentation [11].

The principal idea for the area machine, in a segmented, paged, virtual memory framework, is to associate an area table with each segment, in addition to a page table. Each address includes both the location of its storage area (indirectly via a page table) and an index identifying an area table entry. The area table entry, at a minimum, contains the tombstone for the area. This makes it possible to do multiple area allocations within a segment in a secure fashion, i.e., all addresses to an area can be checked to determine if they are dangling by examining the tombstone in the indexed area table entry. The segment identifier becomes, in essence, merely the name of a page table. Another way of viewing this is to consider a segment identifier as consisting of two parts, a page table identifier and a suffix indicating a segment within the storage described by the page table. Viewed in this way, the area machine is an implementation that permits both multiple pages per segment and multiple segments per page. The latter has been suggested by Fabry [12] among others.

Bounds information for an area can be maintained either in the references or in the area table entry. Two conflicting pressures must be balanced. On the one hand, there is generally more than one address to an area, and thus it is desirable to keep the size of addresses small. On the other hand, an area table entry is more or less permanent and must often exist even if its area has been freed and no dangling references exist. Hence, keeping the area table entry small is also important.

There is yet another way in which addresses may be validated. This is to rigorously control how they are generated. The idea is to make it impossible to manufacture an address using ordinary data processing operations. This is done as follows: All addresses to major areas originate as a result of a storage allocation operation. Addresses to subareas are generated by means of selection operations, in which one argument is an address to an area (or subarea) and the other is a displacement or index. These operations must check that their first argument is a valid address and that the result of adding the displacement or doing the indexing yields a resulting address that is within the bounds of the area. The final requirement is to make sure that ordinary data in storage cannot be misinterpreted as an address. Thus addresses must be distinguished from all other data, i.e., they must be tagged. The concept of a tagged architecture and of protected addresses have been advocated by several authors, e.g., [12, 13, 14].

The detailed mechanism presented below and shown in Fig. 1 depends on the fact that addresses are protected. In particular, this makes it possible to perform address bounds checking using only the end-of-area bound. The starting offset of an area within a segment need not be explicitly maintained. (As we will see, however, it can be useful to maintain it.) The end offset of an area is kept with the tombstone in the area table, reflecting the hypothesis that keeping addresses small is more important than keeping the area table entry small. It should be noted, however, that the ability to subset areas is restricted by this decision. Only the low order storage of an area can be removed in forming a subarea. Eliminating access to the high end of an area is no longer possible.

An address (which is protected by some form of tag, here assumed to be in a part of the memory inaccessible to the area machine user) consists of the following components:

1. *A segment identifier* (SID), which names a page table and an area table to be used in interpreting this address.
2. *An area identifier* (AID), which names the area table entry in the area table of the above named segment; it describes the area to which this address refers.
3. *A starting offset* (S. OFF), which indicates the starting location of the area or subarea addressed, relative to the start of the segment. It consists of two parts:
   a. *A page identifier* (PID), which names the page table entry used to locate the storage for the addressed area or subarea.
   b. *A displacement* (D), which is the quantity to be added to the contents of the page table entry named by the PID in order to identify the precise starting location for the area or subarea.
4. *Control information*, which indicates the capabilities associated with the address and otherwise describes it. One component of it must be a destroy capability

indicator. This determines whether, by means of this address, the area can be freed. Of course, additional capabilities might also be included in order to support protection mechanisms.

The segment table consists of entries that contain the addresses of the page table (P.TAB) and the area table (A.TAB). Each of these tables begins with descriptor information concerning the table itself. The page table descriptor consists of a field P.NO that indicates the number of pages in the segment. The area table descriptor consists of two fields. Field A.NO contains an index to the last entry in the area table, whereas A.ALOC contains the number of currently allocated areas, i.e., those that have been allocated but not yet freed. A page table entry contains the real storage address of the page or the location of the page on secondary storage. An area table entry contains:

1. *A tombstone* (TS), which has two possible values, *allocated*, indicating that the area still exists, and *freed*, indicating that it does not and hence that all outstanding references to the area are invalid (dangling).
2. *An end offset* (E.OFF), which indicates the end location of the area relative to the start of the segment. It consists of a PID and a D, just as does the S.OFF in each address.

Figure 1 illustrates how these components fit together and how an address is decoded. The actions involving the page table are omitted, since they are unchanged from the usual virtual memory mechanisms.

Each access to storage requires that the segment table entry be examined to validate the SID. Given a valid segment table entry, the area table for it must be accessed (via the A.TAB) and the appropriate entry located by means of the AID. The tombstone field TS of that entry is checked to be sure that the area is allocated. An access to storage also must contain an indication of the length of the data involved. The data must all fall within the area. For this to be true, the following condition must hold:

S.OFF + length of data − 1 ≤ E.OFF.

If these conditions are met, either the contents are delivered to the accessor or data are written into the area.

To specify a subarea of an area, the operation

SELECT (address, displacement)

must be executed. A SELECT operation, is checked in a fashion similar to an access. In this case, the bounds check requires the following:

S.OFF + displacement ≤ E.OFF.

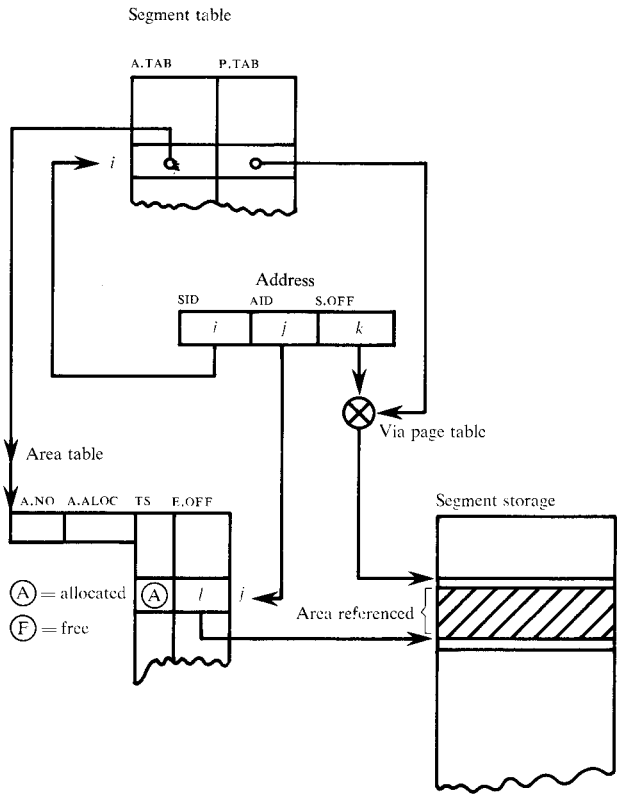

Figure 1 Addressing an area within a segment. The area is determined via the S.OFF of the address and the E.OFF of the area table entry.

The result of a valid selection operation is formed by adding the displacement to the S.OFF of the argument address. The destroy capability is removed from the resulting address.

To create a new area of a specified size in an existing segment requires that an area table entry be permanently dedicated to the new area. The address returned as a result of allocation consists of:

1. SID ← the SID of the segment of which the area is a part.
2. AID ← A.NO + 1 (the next unused area table entry).
3. Destroy capability is granted.
4. S.OFF is set according to where free storage for the area is found.

The area table entry A.NO + 1 consists of:

1. TS ← *allocated*
2. E.OFF ← S.OFF + size of area − 1.

Finally, in the area table descriptor, A.NO ← A.NO + 1, and the number of allocated areas is increased by 1, i.e., A.ALOC ← A.ALOC + 1.

The freeing of an area requires the execution of a FREE operation, which consists of performing the following actions:

1. The address designating the area is checked to assure that it has destroy capability.
2. The storage between the address's S.OFF and the area table entry's E.OFF is returned to the free list.
3. The tombstone TS of the area table entry is set to *free*.
4. In the area table descriptor, the number of allocated areas is reduced by one, i.e., A.ALOC ← A.ALOC − 1.

If A.ALOC = 0, then it becomes possible to invalidate the entire segment by nulling the area and page table addresses of the segment table entry. This would permit reclamation of the area and page table storage. Dangling references would be trapped by means of the nulled segment table entry. This might be done if a significant number of area table entries have been consumed, i.e., A.NO ≥ some threshold.

Procedure activation storage

One of the unfortunate consequences of the area machine as described is that to securely manage procedure activations requires that a new area table entry be provided for every activation. These area table entries must persist even after the procedure activation has been freed, whether or not a dangling reference to the activation exists. This, of course, is true of all separately allocated (major) areas. However, procedure activations follow a stack discipline, and hence the lifetime of an activation, relative to other activations, is known. Further, it is comparatively unusual for references to an activation to persist after the activation has been freed. The conjunction of these two conditions plus the great frequency of procedure activation creation and destruction are compelling reasons for providing a special mechanism for procedure activations.

One would like to assign an area table entry to a procedure activation only when a reference to that activation can persist longer than the activation, and hence become dangling. An approach to doing this is to provide two classes of addresses to procedure activations, one called *temporary* and the other *permanent*. Temporary addresses do not require an area table entry to be permanently consumed, and these addresses are utilized as much as possible. Permanent addresses for procedure activations, which do require a permanently assigned area table entry, are only generated when the storage management system cannot guarantee that a dangling reference will not arise, i.e., when the ALGOL 68 type of scope rules are violated [1].

In order to support temporary addresses, a second area table is associated with each segment that provides storage for procedure activations. This is called the *temporary area table*. Entries in this table, like the activations themselves, follow a stack discipline and hence are not permanently consumed on the creation of each new activation. The entries in this table differ from those in the regular area table and consist of:

1. An offset (E.OFF), which indicates the end of the storage for the activation within the segment
2. An index (PI), possibly null or zero, which indicates which, if any, permanent (regular) area table entry identifies the same activation.

In order to access the temporary area table, its address is included in the segment table entry. The descriptor at the start of the temporary area table consists of a single field, T.NO, which indicates the last active entry in the table.

Finally, a temporary address must be distinguished from a permanent address. This is done by means of a component in the control information of an address. This component, called the T/P indicator, is set to T if the address is a temporary one and to P if it is permanent.

When a procedure is called, activation storage for it must be allocated. The storage management system needs only the size of the activation storage. This storage is allocated from the activation stack, necessitating the use of temporary area table entry T.NO + 1, which is specified as follows:

1. E.OFF ← $\text{E.OFF}_{T.NO}$ + size
2. PI ← 0, i.e., there is no permanent area table entry

Further, the temporary area table descriptor is updated, i.e., T.NO ← T.NO + 1.

Since activation storage utilizes a stack discipline, there is no free list to be searched. The only other action might be to assure that the newly allocated storage is set to some undefined value so that attempts to use it before it is initialized can be detected. (This same consideration applies to ordinary allocation as well.) The address returned consists of the following:

1. SID ← the segment ID of the segment providing activation storage.
2. AID ← T.NO (the new value after the temporary area table descriptor has been updated).
3. S.OFF ← $\text{E.OFF}_{T.NO-1}$ + 1.
4. Control information in which T/P ← T.

How the above tables are organized is illustrated in Fig. 2. Again, the page table has been omitted.

The requirement to convert temporary addresses to permanent ones arises only in certain cases involving the moving of data containing addresses. In these cases, a temporary address in the source area must have an

equivalent permanent address substituted for it in the target area. The derivation of this equivalent permanent address is accomplished thusly. If there is not currently a permanent area table entry associated with the activation storage, i.e., if the temporary area table entry associated with the address has a PI component equal to 0, then the equivalent permanent address is formed as follows:

1. The SID, S.OFF, and control information, except for the T/P component, are the same as those of the temporary address.
2. T/P ← P.
3. AID ← A.NO + 1 (the index to the next unused regular area table entry).

The area tables are updated as follows:

1. A.NO ← A.NO + 1 (the number of area table entries is increased by one).
2. A.ALOC ← A.ALOC + 1 (the number of currently allocated permanent area table entries is increased by one).
3. The temporary area table entry indexed by the AID of the temporary address is updated so that
   PI_{AID} ← A.NO.
   (A.NO is the index for the new permanent area table entry. AID is the area identifier of the temporary address.)
4. For permanent area table entry A.NO
   TS ← *allocated*
   E.OFF ← $E.OFF_{AID}$
   (the end offset of the corresponding temporary area table entry).

If $PI_{AID} \neq 0$, then a permanent area table entry for the activation already exists. No new one is created. Rather, this regular area table entry is used, i.e., the AID for the equivalent permanent address is set to the contents of PI_{AID}.

The circumstances that govern whether address conversion must be performed during a data move are as follows:

1. No conversion is required for any addresses if:
   a. The size of the data being moved is smaller than the size of an address, since an address cannot be moved if this is true. Movement of a partial address may indicate an exception.
   b. The source address refers to an area in a segment that does not contain activation storage, since no temporary addresses can exist in such a segment.
   c. Both source and target addresses refer to areas in the same segment and that segment contains activation storage and the source persists at least as long as the target. Clearly, any addresses in the source area then must be prepared to persist as long as the source persists and thus as long as the target persists. This is indicated under the following circumstances:
      i. S.OFF for the source is less than or equal to S.OFF for the target.
      ii. Both source and target addresses are permanent or both are temporary and their AIDs are equal.
      iii. One address is permanent and one is temporary, and the temporary address has a temporary area table entry with a permanent area index (PI) that equals the AID of the permanent address.
2. Conversion from temporary to permanent addresses may be required otherwise. This conversion applies only to certain addresses contained in the data to be moved from source to target. There are two cases to consider.
   a. The target is not in the same segment as the source. Here, all temporary addresses in the source area must be converted.
   b. The target is in the same segment as the source. Here, only the temporary addresses that fail the tests described in 1c) above (where source ad-

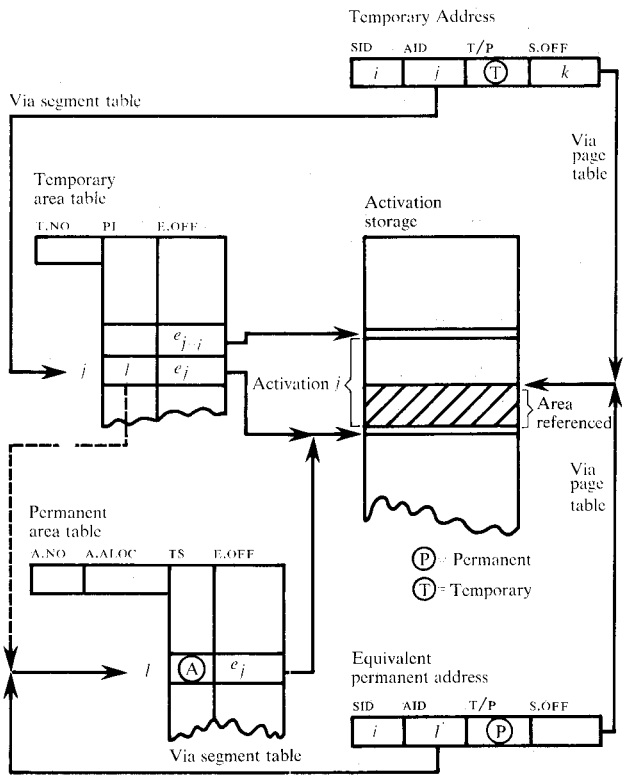

Figure 2 Addressing activation storage by means of a temporary address and its equivalent permanent address.

dress is now interpreted to mean address in the source data being moved) need be converted.

Activation storage is only freed when a RETURN or GOTO out of a block is encountered. Either implicitly for the RETURN or explicitly for the GOTO, the argument for the operation is the destination address where execution is to proceed. All activations from the current activation up to but not including the activation referenced by the destination address must be freed when using the deletion strategy. This requires that the following search for the destination activation be executed starting with the last activation, i.e., the activation indexed by T.NO.

1. If the destination address is temporary and its AID equals T.NO, then we say that the destination has been *reached*. Otherwise it is *not reached*.
2. If the destination address is permanent and its AID equals $PI_{T.NO}$ (the permanent area index associated with the T.NOth activation), then the destination has been *reached*.
3. If the destination has *not* been *reached*, then
   a. If $PI_{T.NO} \neq 0$, a permanent tombstone has been associated with the activation. This tombstone must be set to *free*.
   b. The activation storage must be reclaimed. This is accomplished by merely setting T.NO ← T.NO − 1.
   c. Steps 1, 2, and 3 are repeated until the destination activation is *reached*.
4. If the destination has been *reached*, then control is transferred to the destination address.
5. If T.NO has been reset to zero without *reaching* the destination activation, an exception must be indicated. This can only occur if the destination address is permanent and its permanent tombstone equals *free*. This condition can be detected prior to the search.

Hardware assistance

There is, of course, a cost associated with secure storage management. This cost can appear in three forms, increased time, increased space, or increased logic. The area table consumes extra space and further, each address must be wider, because it must also include an AID. Additional time may be required on every access in order to check the validity of the address. Finally, increased logic must be present in order to implement the address validation and to keep the extra tables updated.

Not much can be done about the increased space required. However, there is considerable flexibility in choosing a time vs logic trade-off. Given the current low cost of logic and the continuing decrease in its cost, there is very good reason for doing as much as possible in logic (additional hardware) so as to minimize the time penalty. This sort of trade-off has occurred before and has resulted, for instance, in associative memories being used to increase the speed of dynamic address translation in virtual memory computers.

The addressing mechanism of the area machine needs, in fact, the same kind of hardware assistance as does virtual memory. Like virtual memory, always accessing the required tables results in too many storage accesses. Without special assistance, the following additional storage accesses would be required every time an address were to be used:

1. To the segment table entry to find the location of area and page tables.
2. To the page table entry.
3. To the area table entry.

In order to circumvent this process at least most of the time, associative memories can be used. For virtual memory, and for the segment and page table accesses above, this associative memory retains the real storage address of an SID/PID pair. The area table accesses can be circumvented in most cases via a second associative memory that retains the tombstone and E.OFF field of an SID/AID pair. The same locality of reference that causes this technique to work reasonably well for virtual memories should make it work well for the area machine. Without this kind of hardware assistance, the time penalty for the extra accesses is simply too large.

Role of garbage collection

Clearly one of the primary purposes of using a deletion strategy is, hopefully, to reduce the need for general purpose garbage collection. The stack discipline of procedure activations ensures that such garbage collection need never be used for activation storage. Further, languages with explicit FREE commands, e.g., PL/I, traditionally do not provide garbage collection to those structures that can no longer be referenced. It then becomes the responsibility of the user to ensure that storage is reclaimed by his issuing of FREE commands.

It should be emphasized, however, that none of the mechanisms introduced thus far preclude garbage collection. Further, it is our view that garbage collection can be a valuable *supplementary* tool for achieving a balance between user control, performance, and ease of use. Whereas a user should be able to control both the allocation and freeing of storage, with assurance that dangling references will be detected, he should not be required to always exercise such precise control. To realize this flexibility, a garbage collection capability must be present.

It is beyond the scope of this paper to discuss the many ways in which garbage collection might be implemented. What is discussed is the way in which most garbage collection techniques can be reconciled with, indeed, can take advantage of, the area machine organiza-

tion. In particular, two aspects of garbage collection are elaborated. One is the so-called marking phase in which referenced (accessible) storage is distinguished from inaccessible storage. The second is the way in which the reclaimed area table entries can be reused.

The marking phase of garbage collection requires an initial set of addresses to immediately accessible areas within the domain that is to be collected. For these addresses, the associated storage must be examined so as to discover addresses to additional areas that are accessible, etc. Since addresses are distinguished from other data in the area machine, these additional addresses can be identified. Marking these areas requires that a one bit MARK field be associated with each area to indicate which areas are accessible. If area table entries of already freed areas are to be reclaimed, the MARK field should also be associated with each of these. This is a fairly compelling reason for including the MARK field as part of the area table entry, thus marking both the area and the entry. Finally, if there is a convenient way of invalidating references without recourse to their tombstones, the area table entries for all freed areas can be reclaimed during garbage collection. One way to do this might be to set all the bits of each such address to zero.

Provision must be made for determining the starting offset of each accessible area so that accessible storage can be distinguished from "garbage." Only that part of each area that can be accessed need be saved. This part of each area is merely the storage from the lowest S.OFF of any address still referencing the area to the end E.OFF of the area. However, there are advantages to maintaining the full extents of all areas. In particular, it becomes possible

1. For the destroy capability to be associated with any reference to the area, not simply those that reference the entire area. (When storage is freed, one must know the extent of the area if all its storage is to be reclaimed.)
2. To support operators that can provide, given an address, its displacement within the area.
3. To use the boundary tag method of storage reclamation to handle the maintenance of the free list. This ingenious method, due to Knuth [15], provides for the consolidation of adjacent free storage areas without any list searching.

Thus we illustrate a realization in which, for each area, an A.STR field is maintained that contains its starting offset so as to indicate its entire extent. Further, two fields, called S.TAG and E.TAG, are provided to hold the tags required by the boundary tag storage reclamation algorithm.

The one section of adjacent storage that is accessible via all addresses to an area is that storage immediately

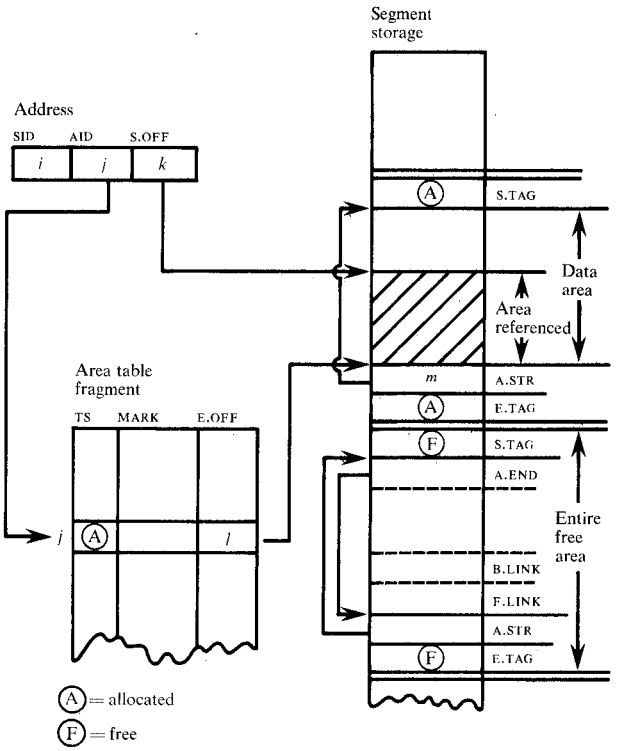

Figure 3 The lay-out of storage for an allocated area and for a freed area, using the boundary tag storage management scheme and marking areas for garbage collection via area table entries.

following the end of the area, i.e., at E.OFF + 1, since E.OFF is maintained in each area table entry and each address contains an AID that designates its area table entry. The range checking of addresses (to insure that addressing past E.OFF does not occur) will protect this storage. Putting control information here is desirable since the information is accessible without increasing the size of every address or area table entry. Because this control information is exploited only by a few operators and occassionally by the storage management system, and hence is not needed for most storage accesses, there is little incentive for placing it in the area table. Figure 3 illustrates one way of organizing storage that reflects these considerations. This section of adjacent storage can, of course, also be exploited to hold additional kinds of information, e.g., locks, authorization information, ownership, etc.

Figure 3 also shows the format of a free (or available) area adjacent to the addressed area. Such a free area must maintain forward and backward links (F.LINK and B.LINK) to other areas on the free list. It is the maintenance of a symmetric list that makes the updating of the free list possible without list searching. Finally, an A.END field must be provided at the start of the free area to indicate the location of the end of the area. Storage

for these last three fields is obtained from the storage originally provided for user data. For more details on the boundary tag method, the reader should consult Knuth [15].

The second aspect to be discussed is how to reuse area table entries. This involves maintaining a free list of these entries. This can be done by using the E.OFF component as a link field to the next reclaimed area table entry. The first entry in the area table, i.e., entry 0, can be used for the free list header.

Other control mechanisms and retention

Because the security of the storage management system can be assured using a deletion strategy as well as with a retention strategy, choosing between them requires an assessment of how the system will be used. One of the prime factors in this context, for favoring a retention strategy, is its ability to readily support more sophisticated (or at any rate, additional) control mechanisms. Some of these are coroutines, multiple tasks, upward "fun-args," and back-tracking. All of these mechanisms require that some of the storage for a procedure activation be retained after the activation exits.

Systems currently exist, of course, that use a deletion strategy and yet support some of these capabilities. What is usually required is that the programmer make some explicit indication of his intention to use a given control mechanism at the time a procedure is called and its activation constructed. This is the case for multitasking in, e.g., the Burroughs B6700 [16] control program and in the PL/I systems for IBM 360/370 computers [2]. Each task creation requires that a separate stack be dedicated to it. A deletion strategy is used on each stack, whereas the relative persistence of at least some of the activations on the various stacks is unspecified. This kind of strategy we call user-planned retention, whereas we call the earlier retention strategy general retention. Whether user-planned or general retention should be chosen depends primarily on:

1. The relative costs of the two strategies
2. Whether user planning for retention is considered to be either an unnecessary burden or a valuable discipline and
3. How frequently retention will be employed.

The earlier sections of this paper provide a qualitative analysis of the cost of using a deletion strategy, with some comments on how it compares to general retention. User-planned retention does not change this analysis very much. It probably, however, does require an additional segment for each separate procedure activation stack. The significance of this requirement depends on the frequency with which retention is used.

User planning for retention does not seem to be a great burden. In fact, for most of the control mechanisms mentioned above, the use of the mechanism is naturally indicated at the initiation of a procedure. Coroutines, separate tasks, and back-tracking are all naturally indicated at precisely the points in the program where the planning for retention must be done. Upward fun-args, i.e., procedures that are returned as values by other procedures and that depend on variables local to these *returning* procedures, do not fall into this category. A user might substitute for variables local to the returning procedure either

1. STATIC (*own*) variables, or
2. BASED (*heap*) variables.

Neither of these classes of variables is deleted when a procedure returns. Both of these substitutes do require user planning of a more burdensome form than that associated with coroutines, tasks, or back-tracking. Such planning does, however, eliminate the possibility that references in the upward fun-arg *mistakenly* refer to the local variables of the returning procedure.

Particularly critical is how frequently retention is used. Even some advocates of retention [7] not only concede but emphasize that most programs are "well-stacked," i.e., do not require retention. It must be acknowledged, however, that some applications, particularly where back-tracking is naturally used, require rather extensive use of retention. For these applications, a good scheme for implementing retention, e.g., the so-called spaghetti stack [6], is probably to be preferred to the separate stack approach. Even here, however, deletion of ordinary activations does not seem at all unreasonable. In this connection, the combination of the spaghetti stack with the tombstone notion does not seem to present any large difficulties. The use of temporary addresses no longer seems very attractive, however.

Conclusion

A new addressing mechanism has been presented that invalidates dangling references, rather than avoiding them. This mechanism makes a secure realization of the so-called deletion strategy of storage management possible. When contrasted with the existing alternatives, such a strategy seems desirable. Whereas the addressing mechanism requires hardware assistance to be feasible, such assistance greatly reduces the performance penalty without enormous cost. The same set of cost/performance trade-offs that apply to virtual memory seem to apply with equal force to the addressing mechanism of the area machine. The bonus is that the area machine permits the extension of the address validation mechanism to small areas, several of which might appear on a single page.

Acknowledgments

In addition to the referenced papers, the author expresses his debt for ideas developed over a period of years of close association with M. A. Auslander, P. H. Oden, R. Goldberg, M. Hopkins, W. H. Harrison, and C. Lewis. It can safely be said that this paper would not have been written without this period of association.

References

1. C. H. Lindsey and S. G. van der Meulen, *Informal Introduction to ALGOL 68*, North Holland Publishing Company, Amsterdam, 1971.
2. *PL/1 Language Specifications*, Form No. GY33-6003-2, IBM Corporation, White Plains, NY, 1970.
3. D. M. Berry, "Introduction to Oregano," *SIGPLAN Notices* (ACM) **6,** 171 (1971).
4. J. B. Johnston, "The Contour Model of Block Structured Processes," *SIGPLAN Notices* (ACM) **6,** 55 (1971).
5. P. Wegner, "Data Structure Models for Programming Languages," *SIGPLAN Notices* (ACM) **6,** 1 (1971).
6. D. G. Bobrow and B. Wegbreit, "A Model and Stack Implementation of Multiple Environments," *Commun. ACM* **16,** 591 (1973).
7. D. M. Berry, L. Chirica, J. B. Johnston, D. F. Martin, and A. Sorkin, "On the Time Required for Retention," *SIGPLAN Notices* (ACM) **8,** 165 (1973).
8. A. Bensoussan, C. T. Clingen, and R. C. Daley, "The Multics Virtual Memory: Concepts and Design," *Commun. ACM* **15,** 308 (1972).
9. M. A. Auslander and J. F. Jaffe, "Functional Structure of IBM Virtual Storage Operating Systems, Part I: Influences of Dynamic Address Translation on Operating System Technology," *IBM Syst. J.* **12,** 386 (1973).
10. *Burroughs B5500 Information Processing System Reference Manual*, Burroughs Corporation, Detroit, MI, 1964.
11. B. Randell, "A Note on Storage Fragmentation and Program Segmentation," *Commun. ACM* **12,** 365 (1969).
12. R. S. Fabry, "Capability-Based Addressing," *Commun. ACM* **17,** 403 (1974).
13. J. K. Illiffe, *Basic Machine Principles*, 2nd Edition, MacDonald/American Elsevier Inc., New York, 1972.
14. E. A. Feustel, "On the Advantages of Tagged Architecture," *IEEE Trans. Comput.* **C-22,** 644 (1973).
15. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1968.
16. E. I. Organick and J. G. Cleary, "A Data Structure Model of the B6700 Computer System," *SIGPLAN Notices* (ACM) **6,** 83 (1971).

Received April 24, 1974; revised August 29, 1974

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.