SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 7, 179–192 (1977)

Storage Management for an Implementation of SNOBOL4*

DAVID R. HANSON

Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.

SUMMARY

The implementation of the SNOBOL4 programming language requires some scheme for dynamic allocation and reclamation of storage. This paper describes the method used in a machine-dependent implementation of SNOBOL4 called SITBOL. Available storage is divided into two regions: one for the allocation of storage for permanent or immovable objects and one for objects that are subject to reclamation. Reclamation is accomplished by a four-phase algorithm, which includes marking, address adjustment and compaction. The reclamation process is supplemented by several heuristics that reduce the cost of reclamation and increase the amount of storage reclaimed. The storage overhead of the technique is analysed and compared with an alternate method. Some empirical measurements of actual SNOBOL4 programs are given that substantiate the results of the analysis.

KEY WORDS Storage management Garbage collection SNOBOL4 Marking algorithms List processing

INTRODUCTION

SNOBOL4[1] is a high-level programming language designed for advanced string and list processing applications. It is perhaps best known for its extensive pattern-matching facilities, but also includes several other features that, with the pattern-matching facility, make it a very powerful language. Some of these additional features are run-time compilation, run-time definition and creation of programmer-defined data objects, an associative storage facility (tables), dynamic array creation and dynamic creation of variables.

The original version of SNOBOL4, referred to as the macro implementation[2], is a machine-independent implementation. It has been installed on nearly all major large-scale machines. As is sometimes the case with machine-independent software, the macro implementation requires a large amount of storage and is rather slow. Motivated by the power and popularity of SNOBOL4 and by a desire for a more efficient version, a number of *machine-dependent* implementations have either been completed[3–7] or designed[8,9] in recent years. Unlike the macro implementation, these are tailored for their particular machine and, as a result, require substantially less storage in which to run and, in some cases, have achieved an order of magnitude increase in speed.

The implementation of SNOBOL4 poses some difficult problems for the designer; not the least of which is the choice of a storage management scheme. The nature of SNOBOL4 virtually demands that the chosen scheme include a method for dynamic allocation and reclamation of storage. Although there are many techniques for dynamic storage allocation

* This work was supported in part by the National Science Foundation under Grant DCR75-01307.

Received 3 March 1976

and reclamation,[10–15] not all of them are suitable for SNOBOL4. For example, reclamation is complicated by language features that permit the programmer to create structures representing arbitrary graphs. It is easy and quite common to have many instances of circular data structures during the normal course of program execution, e.g. an element of an array can be (i.e. point to) the array itself. This situation renders reclamation techniques based on 'use counts'[10] or schemes designed for non-circular structures useless. The reclamation method used in an implementation of SNOBOL4 must be capable of handling arbitrary list structures. Thus the choice of a storage management method is important, and can have a substantial impact on the resulting system.

The purpose of this paper is to describe the storage management subsystem of SITBOL,[3–5] a machine-dependent implementation for the DECsystem-10. The method used in SITBOL is simple, reasonably fast, requires minimal programming and satisfies the above requirements. A novel two-level layout of dynamic storage permits the allocation of storage for permanent or immovable objects and for objects subject to reclamation. The reclamation algorithm is similar to the one used in SPITBOL,[6] which is a modification of a marking technique for arbitrary list structures,[10, 13, 14] and includes several heuristics to speed up the reclamation process and to increase the amount of storage reclaimed.

The following sections include detailed explanations of allocation and reclamation, a discussion of the reclamation heuristics and an analysis of the storage overhead required to support the method. The storage overhead of another method[15] is analysed and compared to the present method and some empirical results from actual SITBOL programs are given.

ENVIRONMENT

The dynamic storage area is a single contiguous segment partitioned into two regions as shown in Figure 1. The *growing impure data* region, the GRID, is located at the lower part of storage and houses objects that are not subject to destruction or that cannot be moved. For example, input and output buffers, executable code for external functions, symbol table entries and program constants are placed in the GRID region.

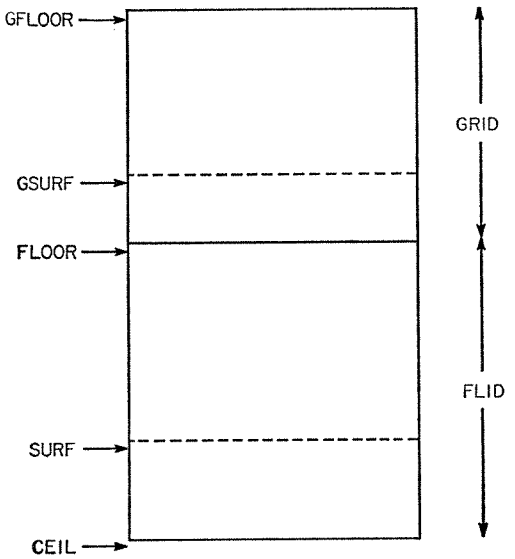

Figure 1. Storage layout

Objects that are transitory in nature are placed in the *floating impure data* region, the FLID. The FLID occupies storage locations numerically above the GRID. The storage elements in the FLID are subject to reclamation; those in the GRID are not. One advantage of this two-level structure is that valuable processing time is not wasted during the reclamation process on objects that are known to be permanent. Another advantage is that the GRID provides an area for objects that cannot be moved such as I/O buffers. In some systems, the movement of i/o buffers is prohibited. Therefore, they cannot be allocated in the FLID region.

The allocated portion of the GRID is defined by the contents of GFLOOR and GSURF. GSURF contains the address of the first free location in the GRID region. The allocated portion of the FLID is defined by the contents of FLOOR and SURF in the same fashion. The value of CEIL indicates the end of the segment.

ALLOCATION

Since there are no declarations in SNOBOL4, a variable can have a value of any datatype at any time during program execution. In most implementations of SNOBOL4, a value is represented by a fixed-size *descriptor*. The descriptor is often thought of as the storage element of the SNOBOL4 virtual machine.[2] Data structures are constructed by various combinations of variable-size *blocks*, which are allocated in both the GRID and FLID regions. A block contains several fields as illustrated in Figure 2. It is composed of four header fields used for storage management and by an arbitrary number of two-word descriptors. The SIZE field contains the size of the block, including the header fields, in words. The size of this field determines the maximum allowable size of a block. The TYPE field is used to indicate the type of information contained in the block. The BREF and MARK fields are used by the reclamation process and must be large enough to contain an address.

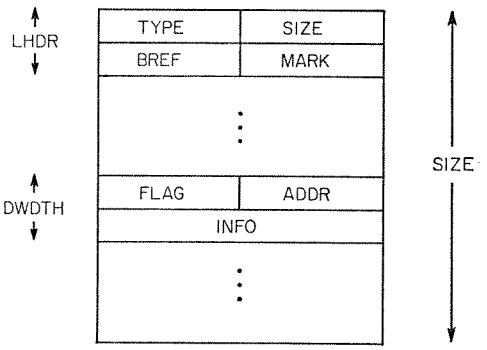

Figure 2. Block layout

The interior of a block is composed of an arbitrary number of two-word descriptors each having three fields. The ADDR field may contain an address. If the FLAG field is 1, the contents of the ADDR field are treated as an address. If the FLAG field is 0, ADDR is ignored. The INFO field contains data that is never regarded as an address. In list processing terminology, the descriptor is an *atom* if the FLAG field contains zero. If the ADDR field contains an address that points into the floating region, it is called a *floating address*. All floating addresses must point to the head of a block. The INFO field is sometimes used to indicate an offset to a specific descriptor within a block.

Allocation in the GRID and FLID regions is trivial. The current value of GSURF or SURF defines the beginning of the new block. To allocate the storage, GSURF or SURF is simply incremented by the amount of the request. Available storage is exhausted if GSURF exceeds FLOOR or if SURF exceeds CEIL. As long as these conditions do not arise, allocation is very fast. On the DECsystem-10, this method of allocation can be performed in as few as three instructions.

If there is insufficient storage to satisfy the request, reclamation is attempted. If the request is for a block in the FLID region, the reclamation process identifies all the 'good' blocks in the FLID and compacts them into the lower portion of the floating region. Hopefully, this action lowers the surface, defined by the contents of SURF, enough to satisfy the request. If not, additional storage is requested from the DECsystem-10 Operating System in order to enlarge the memory segment containing the GRID and the FLID regions.

If storage is exhausted in the GRID region, the FLID is compacted and moved upwards in order to expand the GRID. GRID expansion is performed in large units to reduce the number of times the FLID must be moved. In SITBOL, GRID expansion always obtains 500 decimal words in excess of the amount needed. Allocation in either region depends upon successful reclamation in the FLID region.

RECLAMATION

Reclamation of storage is accomplished by compacting accessible blocks in the floating region. This process is divided into four phases. The first phase involves marking all accessible blocks in the FLID. This is done using a modified version of an algorithm described in References 10, 13 and 14. In the second phase, all addresses that point to accessible blocks are modified to point to the new locations of the blocks. The third phase is the compaction phase; accessible blocks are moved to the lower part of the FLID. The fourth and final phase is performed only for a GRID expansion. It consists of moving the entire FLID upwards to provide additional GRID storage.

Each of these phases is relatively simple. Phase one, however, uncovers a major problem in dynamic storage management–locating pointers into the floating region. Many of the difficulties encountered in systems involving dynamic storage management can be traced to a 'dangling reference' or the improper handling of a floating address. This problem is handled by the definition of a *basic block*[2] containing the descriptors that provide a path to all accessible blocks. The basic block is referred to as the *tended* region, since it is used to begin the marking process in the first phase. In SITBOL, the tended region consists of a basic block, several save areas, the stacks and the entire GRID region. The save areas are used to save floating addresses across an 'unstable' subroutine call, *viz.* a call that may result in the allocation of storage. Programming conventions[3,16] are used within SITBOL to govern the use of floating addresses. Almost all the bugs discovered in the storage management subsystem of SITBOL were caused by the violation of these conventions.

For the enumeration of each phase of the reclamation process, assume that the basic block resides at the location BASIC and is of the form shown in Figure 2. The identifiers TYPE, SIZE, BREF and MARK are field functions that may be applied to a block. A field reference such as SIZE(P) means the contents of the SIZE field of the block pointed to by the value of P. Likewise, the identifiers FLAG, ADDR and INFO are field functions that may be applied to a descriptor. The identifiers LHDR and DWDTH are integers that indicate the length, in words, of the block header information and the size of a descriptor, respectively.

Figures 3(a)–(d) depict the result of each phase for a simple example. Inspection of Figure 3(a) shows that blocks 1 and 3 are inaccessible from the basic block. The example given in Figure 3 does not include GRID expansion.

Phase I. Marking

The marking phase begins by marking all blocks directly accessible from the basic block. It continues in an *iterative* fashion to mark all blocks along any path beginning with a descriptor in the basic block. The BREF (back reference) field of a block is used in a stack-like fashion to facilitate the marking of nested blocks.

The objective of Phase I is to create a list starting at the MARK field of each accessible block containing all of the addresses pointing to that block. This list is constructed using the ADDR field of the descriptors pointing to the block.

The modification of the marking technique described in References 10, 13 and 14 is that the original value of the ADDR field of each descriptor is not restored during the marking phase. The net result is that, upon completion, a non-zero MARK field indicates that the block is accessible and heads a list of addresses that point to it. The algorithm, Algorithm A, is as follows.

A1. [Initialize.] Let P point to the block being processed and Q point to a particular descriptor within that block. Set P to the address of the basic block and $Q \leftarrow P + LHDR - DWDTH$.

A2. [Process next descriptor.] Set $Q \leftarrow Q + DWDTH$. If $Q \geq P + SIZE(P)$ then all descriptors in this block have been processed; go to A5.

A3. [Check for a floating address.] If $FLAG(Q) = 0$, ADDR(Q) is not a floating address; go to A2. If $FLAG(Q) = 1$ but $ADDR(Q) < FLOOR$ or $ADDR(Q) \geq SURF$, the address is outside the floating region; go to A2.

A4. [Process a floating address.] Set $T1 \leftarrow ADDR(Q)$ and $T2 \leftarrow MARK(T1)$. Add this address to the head of the list by setting $MARK(T1) \leftarrow Q$ and $ADDR(Q) \leftarrow T2$. If $T2 \neq 0$, this block has already been marked; go to A2. Otherwise, save the current block address by setting $BREF(T1) \leftarrow P$ and then set $P \leftarrow T1$. Begin marking this block: set $Q \leftarrow P + LHDR - DWDTH$ and go to A2.

A5. [Recovery procedure.] If P points to the basic block, Phase I is completed; otherwise set $Q \leftarrow MARK(P)$.

A6. [Find end of list to restore Q.] If $ADDR(Q) = 0$ the end of the list has been reached; go to A7. Otherwise set $Q \leftarrow ADDR(Q)$ and repeat this step.

A7. [Restore previous block address.] Set $P \leftarrow BREF(P)$ and go to A2.

The use of the BREF field as an element of a stack is evident in steps A4 and A7. Notice that the recovery procedure, steps A6 and A7, requires that the end of the list beginning at the MARK field be found in order to correctly restore the address within the previous block (the value of Q). This is necessary because the address that is saved in the BREF field in step A4 is the address of the block that caused the *first* reference to the nested block. Thus the corresponding offset in that block is found at the *end* of the list. The MARK field of every block must initially be zero. This is done when a block is allocated, but can be done by a linear pass through the FLID region prior to marking. If this condition is maintained, each list of addresses pointing to an accessible block is terminated with a zero.

The state of the FLID after Phase I is shown in Figure 3(b). The MARK field of the accessible blocks, 2, 4 and 5, each head a list of addresses pointing to that block. Note that block 3 still contains a pointer to block 5, but that the MARK field is zero.

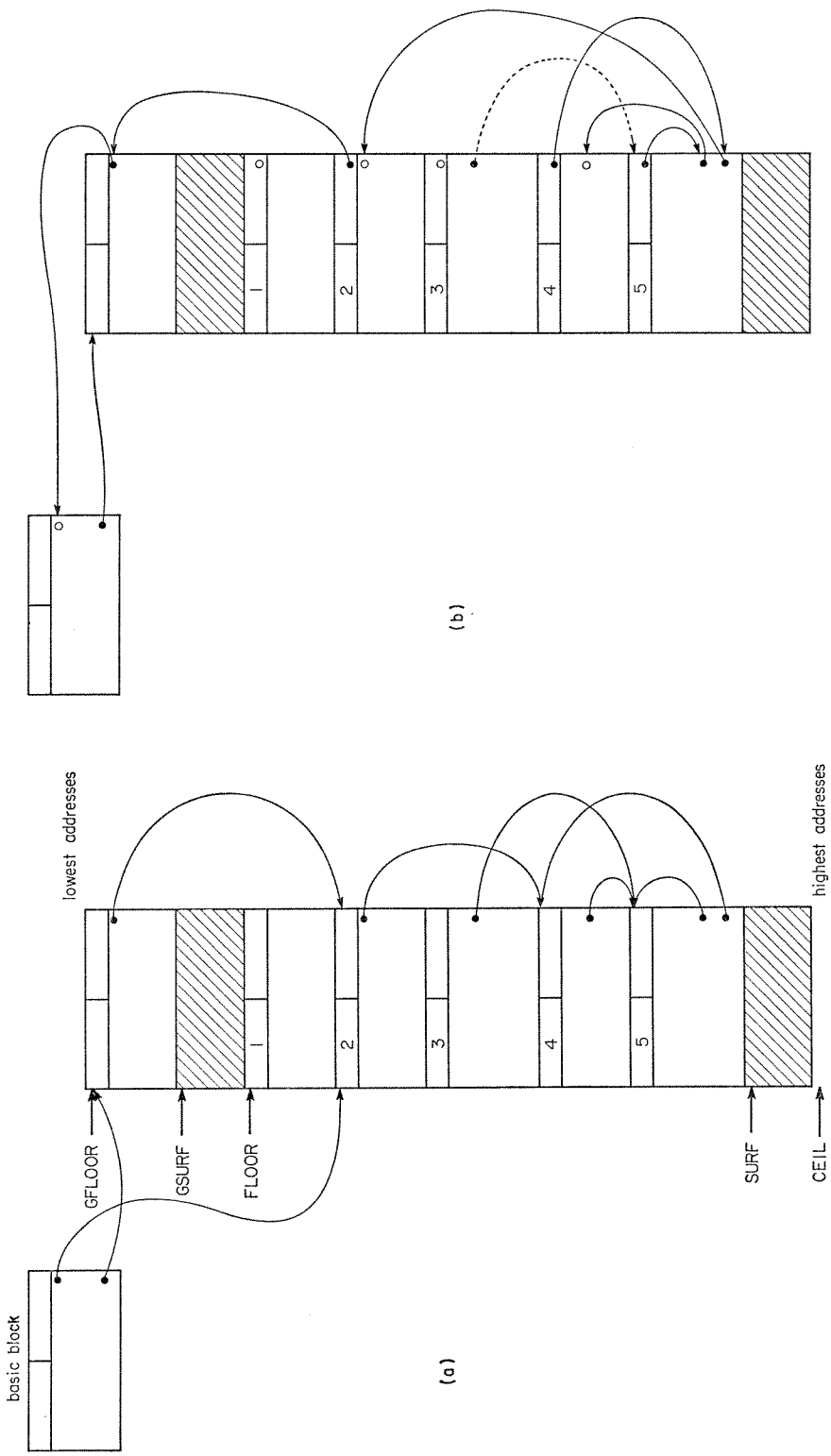

Figure 3. An example of Phases I–III. (a) Before reclamation, (b) after Phase I

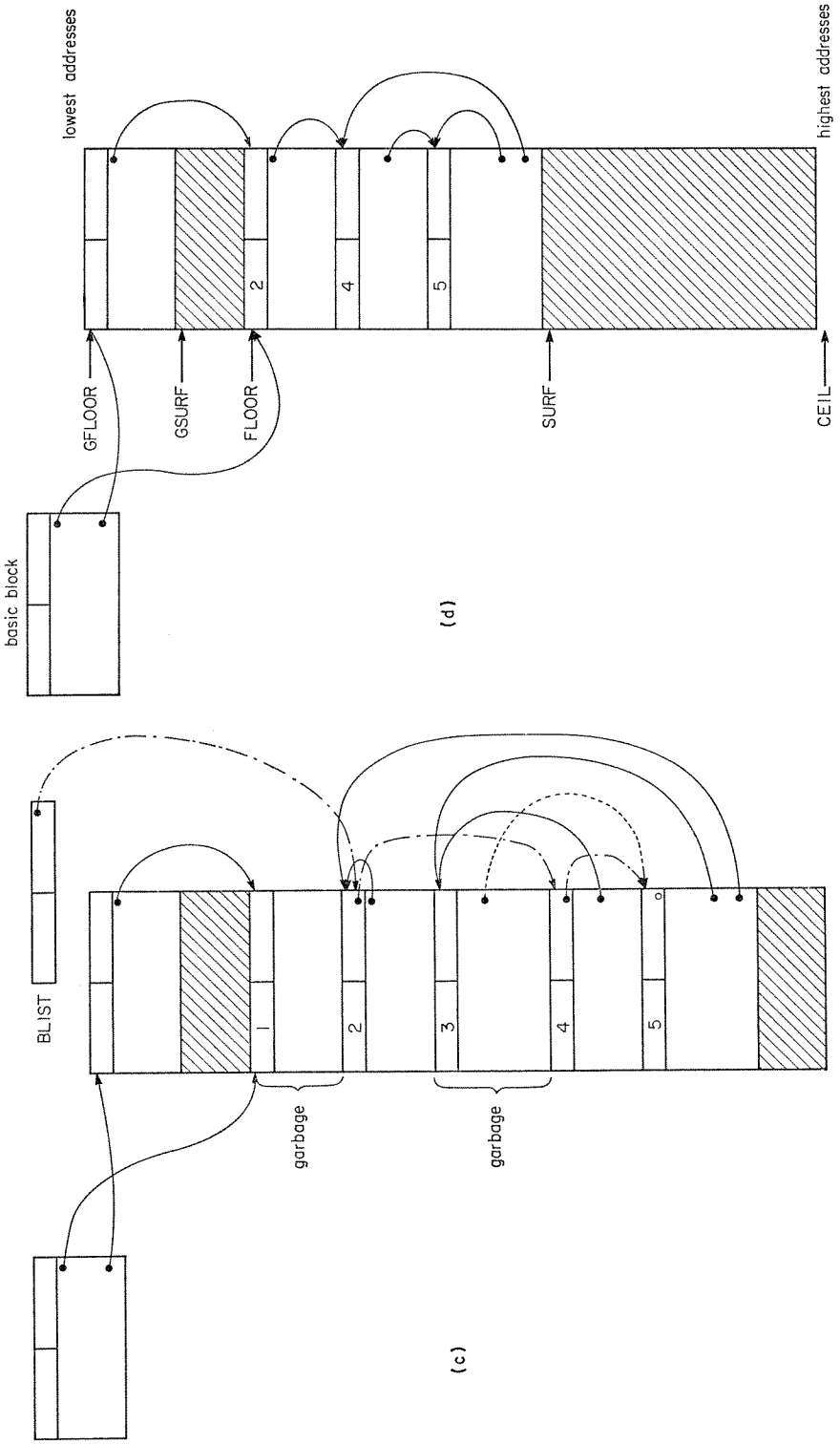

Figure 3 (cont'd). (c) After Phase II, (d) after Phase III

Phase II. Address adjustment

In Phase II, the lists of addresses pointing to each accessible block are traversed (and destroyed) in order to insert the new address of the block in the ADDR field of each descriptor pointing to it. This is accomplished by a linear sweep through the floating region checking for non-zero MARK fields.

In addition, a list of accessible blocks is constructed using the MARK field for the link. This list is constructed in ascending order of block addresses to facilitate processing in Phase III. The MARK field of a dummy block at location BLIST serves as the list head.

The contents of NFLOOR is the new value of the FLOOR for the floating region, and is established when the reclamation process is called. If the procedure is called in an attempt to obtain more FLID storage, NFLOOR is equal to FLOOR. If the GRID is to be expanded, NFLOOR reflects the desired amount of expansion. In either case, addresses are adjusted to their final value.

B1. [Initialize.] P points to the block whose MARK field is the next receptacle for the construction of the list of accessible blocks. Q is used to traverse the list of addresses pointing to a block. Set P to the address of the block at BLIST, NEW ← NFLOOR and OLD ← FLOOR. NEW points to the new location of a block.

B2. [Finished ?] If OLD ⩾ SURF then set SURF ← NEW, MARK(P) ← 0 and terminate Phase II.

B3. [Is the current block garbage ?] If MARK(OLD) = 0, this block is garbage; set OLD ← OLD + SIZE(OLD) and go to B2. Otherwise, add this block to the list of accessible blocks by setting MARK(P) ← OLD and P ← OLD.

B4. [Traverse list of addresses.] Set Q ← MARK(OLD).

B5. [Adjust an address.] Set T1 ← ADDR(Q) and ADDR(Q) ← NEW. Then set Q ← T1. If Q ≠ 0, repeat this step.

B6. [Increment NEW and OLD.] Set NEW ← NEW + SIZE(OLD) and OLD ← OLD + SIZE(OLD). Go to B2.

At the completion of Algorithm B, all floating addresses have been adjusted to reflect the new location of the accessible blocks. The list beginning at MARK(BLIST) contains all the accessible blocks.

Figure 3(c) shows the state of the FLID after Phase II. Figure 3(c) has a curious appearance because the floating addresses have been adjusted to point to the new locations of blocks 2, 4 and 5 *before* compaction. The list of accessible blocks is indicated by the dot-dashed lines. Figure 3(c) clearly shows that blocks 1 and 3 are garbage.

Phase III. Data movement

Algorithm C performs the necessary list traversal and data movement to compact the FLID region. Note that the BREF and MARK fields of the accessible blocks are cleared during this phase. This must be done to ensure the proper operation of the algorithms during the next reclamation.

C1. [Initialize.] In this phase, NEW points to the new location of a block and P traverses the list of accessible blocks. Set NEW ← FLOOR and let P point to the first block on the list.

C2. [Prepare to move a block.] If P = 0 terminate Phase III. Otherwise set T1 ← MARK(P) and clear the BREF and MARK fields, i.e. set BREF(P) ← 0 and MARK(P) ← 0. If P = NEW, data movement is unnecessary; go to C4.

C3. [Move a block.] Move SIZE(P) words from consecutive locations beginning at P to those beginning at NEW.

C4. [Update NEW and point to next block.] Set NEW ← NEW + SIZE(P). Advance to the next block by setting P ← T1 and go to C2.

The final state of the floating region is illustrated in Figure 3(d). The surface has been lowered because the space previously occupied by blocks 1 and 3 has been reclaimed.

Phase IV. GRID expansion

If more storage is needed for the GRID region, the FLID is moved upwards. This step is performed by Algorithm D.

D1. [Should the FLID be moved?] If NFLOOR = FLOOR, Phase IV is unnecessary; reclamation is finished.

D2. [Relocate the FLID.] Compute the current location of the end of the FLID, i.e. set T1 ← SURF − (NFLOOR − FLOOR). Move SURF − NFLOOR words from consecutive locations starting at T1 to locations beginning at SURF. Note that this movement is highest address first.

D3. [Reset FLOOR.] Set FLOOR ← NFLOOR; reclamation is finished.

IMPLEMENTATION

In the marking phase described above, it was assumed that blocks are homogeneous. The reclamation algorithms can be written to handle blocks that have an arbitrary, predefined structure. In particular, the block layout shown in Figure 2 is a simplification of the actual layout used in SITBOL.

The specific layout of a block is determined by the contents of the TYPE field. There are several types of blocks that have a layout as shown in Figure 2, but some block types contain no floating addresses. For example, a character string is stored in a block of this type. Some blocks have floating addresses in non-standard positions. Algorithm A in SITBOL is implemented as a group of table-driven procedures that operate in essentially a coroutine fashion. In step A4, a block is marked by calling the appropriate procedure given by an entry in a dispatch table indexed by TYPE codes. The previous marking procedure is temporarily suspended. This suspended procedure is reactivated in step A7 via a table dispatch on TYPE(P). This method of structuring resulted in a storage management subsystem that is modular and easily extended to include additional block formats.

HEURISTICS

The sediment

In actual experience using SITBOL, it was discovered that a number of blocks are allocated in the FLID region and remain active throughout program execution. These blocks tend to settle to the bottom of the floating region forming what might be called a *sediment* near the FLOOR. A significant fraction of the processing time consumed by the reclamation process is wasted processing blocks in the sediment. One possible solution to this problem is to provide language constructs that allow the programmer to explicitly designate in which region to allocate storage for certain objects. Another solution, adopted in SITBOL, is to

keep track of the sedimentary level and not attempt to reclaim the space occupied by blocks within the sediment unless absolutely necessary. A pointer, SEDIM, indicates the current level of the sediment. SEDIM points to the first word above the sediment region. Initially, SEDIM is equal to FLOOR. The sedimentary level rises during the course of program execution. A new sedimentary level is defined after the completion of Phase II by the address of the first unmarked block in the FLID region. This scheme requires some modifications to the algorithms given above.

In Phase I, the sediment region is treated as a part of the basic block. A check is performed in step A3 to determine if ADDR(Q) points into the sediment. If it does, the MARK field of the block pointed to is set to -1. Since the sediment region is treated as a part of the basic block, no list of addresses is formed. The active blocks are marked by simply setting the MARK fields to -1.

In Phase II, given by Algorithm B, OLD is initialized to SEDIM rather than FLOOR in step B1. At the end of Phase II, a linear scan is made to determine the new value of SEDIM. MARK fields with a value of -1 are also cleared since those blocks do not participate in Phase III. Blocks in the sediment are not added to the list of accessible blocks.

In the third phase, NEW is initialized to SEDIM instead of FLOOR in step C1. If the reclamation is performed to obtain additional growing storage, SEDIM is set to FLOOR before and after the reclamation process.

The BREF Field

The BREF field is needed only if a block contains more than one floating address. In some instances, it can be used for other purposes or simply deleted. For example, in the design of a SNOBOL4 system for the PDP-11,[9] the BREF and MARK fields each require a 16-bit word. To save space, blocks that do not contain floating addresses do not have a BREF field. For example, integers, reals and strings are stored in blocks without a BREF field.

Strings

A string in SITBOL is represented by a descriptor whose ADDR field points to a block containing the actual characters comprising the string. This type of block is called a *string block*. The INFO field of a string descriptor contains the length of the string and the character offset from the beginning of string block to the first character of the string. Any number of string descriptors can point to the same string block and, in many cases, strings effectively 'share' characters housed in the string block. An advantage of this scheme is that operations such as substring formation are trivial; only the contents of the INFO field must be modified. A disadvantage is that, during program execution, a string block may contain a substantial number of characters that are not part of any of the strings described by accessible string descriptors pointing to that string block.

This problem is partially alleviated during Phase II of the reclamation process. At the completion of Phase I, the MARK field of each accessible block heads a list of addresses that point to that block. For a string block, this list contains only string descriptors. By inspecting each descriptor, the maximum penetration into the string block is determined. The unreferenced portion of the string block is discarded by simply changing the SIZE field to reflect the maximum penetration.

This heuristic is introduced in Algorithm B between steps B3 and B4. Step B6 is modified so that NEW is incremented by the new size of the string block and OLD is incremented by

the original size. To save time in Phase II, this heuristic is performed only on every eighth reclamation in the current implementation of SITBOL.

STORAGE OVERHEAD

The techniques described above for dynamic allocation and reclamation require that the storage reserved for the reclamation process be spatially distributed throughout the allocatable region. Each block includes space reserved for the storage management scheme that is not related to the object contained in the block. Thus, the storage available to house data is reduced due to the overhead required by this method of storage management.

Consider an allocatable region of M words. The available storage, A, that can be used for data is

$$A = M(1-f) \tag{1}$$

where f is the fraction devoted to the storage management scheme. Assume that the average block size is $\bar{n}$, including the BREF and MARK fields, and that t is the number of words required for these fields. The number of blocks in the allocatable region is approximately equal to $M/\bar{n}$. Each block contains $\bar{n}-t$ words available for data. Therefore, the fraction devoted to storage management is

$$f = t/\bar{n} \tag{2}$$

Notice that for a given average block size, the fraction of storage required for storage management is a constant independent of M.

An alternative scheme is to separate the storage required by the storage management scheme from the allocated region. In this method, a stack and bit map are required to perform the marking and address adjustment phases. This technique is described in Reference 15.

The latter method, referred to as Method 2, has the advantage that a floating address may point into the middle of a block. The method used in SITBOL, referred to as Method 1, requires that all floating addresses point to the head of a block. This restriction, however, limits the number of valid floating addresses to the number of blocks in the FLID, whereas using Method 2, the number of valid floating addresses is equal to the size of the allocated region. Thus, in debugging SITBOL, it was easy to identify an invalid floating address, and the apparent restriction was an important aid in the construction of a reliable system. In addition, this restriction is not imposed on pointers into the GRID region since that region is not subject to reclamation.

The principal disadvantage of Method 2 is that the stack must be pre-allocated before program execution. It is possible that the marking phase will cause stack overflow if there exists a chain of pointers longer than the size of the stack. This cannot happen using Method 1 since the size of the 'stack' is increased by one whenever a block is allocated. Method 1 is used in SITBOL because it has the important advantage that the marking phase always succeeds. Furthermore, Method 2 requires extensive bit manipulation in order to maintain the bit map during the marking and address adjustment phases. This is usually more time consuming than simple address manipulations on most machines.

The storage overhead for Method 2 consists of the space required for the stack and the bit map. Assume that the additional stack allocated for this method occupies s words and that the word size of the machine is b bits per word. The size of the bit map needed for an

allocatable region of A words is approximately A/b. Thus,

$$M = A + (A/b) + s$$

and the fraction of storage devoted to storage management in this case is

$$f = \frac{1 + (sb/M)}{1 + b} \tag{3}$$

In situations in which more than one bit is required to represent each word of available storage in the bit map, b may be replaced by b/k in equation (3) where k is the number of bits required.

Equations (2) and (3) can be used to determine at what average block size Method 1 is preferable to Method 2 or *vice versa*. Given s, b and M the value of $\bar{n}$ that results in equation (2) being equal to equation (3) is

$$\bar{n}_e = \frac{t(1+b)}{1 + (sb/M)} \tag{4}$$

For example, if $M = 4{,}000$, $s = 100$, $b = 36$ and $t = 1$, $\bar{n}_e$ is approximately 19. Therefore, in this example, Method 1 would result in more available storage if the average block size is greater than 19 words.

In Method 1, f is a constant, while in Method 2, f decreases as the size of the allocatable region (M) increases. Equation (4) indicates that as M increases, $\bar{n}_e$ increases so that for large regions the average block size must be large in order to make Method 1 more economical than Method 2. These results suggest that Method 2 is preferable to Method 1 in cases where M is large. Empirical measurement of the average block size in the FLID region for a number of actual SITBOL programs tends to substantiate this suggestion. A summary of the measurements is given in Table I. The computations for the values of f and $\bar{n}_e$ have been made using $s = 100$, $b = 36$ and $t = 1$. The values of f in Table I are given as a percentage.

Table I. Measurements of SNOBOL4 programs

M	$\bar{n}$	$\bar{n}_e$	$f_1(\%)$	$f_2(\%)$
22,225	14	32	7·1	3·4
19,162	16	31	6·3	3·5
16,641	20	30	5·0	3·6
15,828	21	30	4·8	3·6
14,294	24	30	4·2	3·1
7,647	16	25	6·3	4·3
6,994	19	24	5·3	4·4
6,559	23	24	4·4	4·5
5,659	40	23	2·5	4·7
4,149	30	20	3·3	5·4
3,093	46	17	2·2	6·2
3,018	34	17	2·3	6·2
1,364	10	10	10·0	10·1
1,005	12	8	8·3	12·7
973	14	8	7·1	13·0

For large values of M, the value of f for Method 2 is less than that for Method 1. The value of f for Method 1, however, decreases for smaller values of M and becomes less than f for Method 2.

The set of programs that provided the data given in Table I consisted of compilers, preprocessors and document preparation systems. All of the programs were written by advanced SNOBOL4 programmers and most made extensive use of many SNOBOL4 language features such as programmer-defined datatypes and pattern-matching. The programs in Table I that require a large region are generally the compilers and one rather large document preparation program. Simple text processing and the preprocessor programs required less storage.

CONCLUSIONS

The storage management scheme described here is simple yet provides enough generality to be used for a system such as SNOBOL4. The algorithms can be implemented in a small, fixed amount of storage. For example, the entire storage allocation and reclamation module in SITBOL, programmed in assembly language, occupies only 450 decimal words—slightly less than 5 per cent of the entire system. The allocation mechanism is particularly simple and fast. Essentially, no time overhead is incurred in allocation until reclamation is necessary. Furthermore, the exact amount of storage requested is allocated so that there is no internal fragmentation.

The partitioning of the available storage into the growing and floating regions reduces the overhead due to reclamation and facilitates inclusion of language features that require the allocation of storage for immovable objects. For example, in SITBOL, this type of allocation is necessary for the dynamic loading of object code for external functions[17] and for the allocation of input and output buffers.

One of the most important concepts in the design and implementation of a dynamic storage management system is the notion of the tended region that provides the information needed to locate all pointers into the floating region. It is absolutely necessary that firm programming conventions be established dictating exactly how floating addresses are to be handled. The set of active pointers into the floating region comprises the data that drives the reclamation process. The use of a tended region aids in the concise definition of this set and ensures that all of the elements are processed during reclamation.

Although the techniques described in this paper have been given in terms of their application to SITBOL, they are by no means limited to this single application. Provided that suitable conventions can be established concerning the use of floating addresses, these techniques can be used in other applications requiring a general dynamic storage allocation and reclamation subsystem.

ACKNOWLEDGEMENTS

To the best of my knowledge, the variation of the marking algorithm described in this paper was devised by Robert B. K. Dewar and Kenneth Belcher at the Illinois Institute of Technology for SPITBOL. The heuristic used to discard the unreferenced portion of a string block is included in both SPITBOL and SITBOL. The sediment region originated with SITBOL. SITBOL was designed, and its implementation directed, by James F. Gimpel of Bell Laboratories. The two-level storage scheme, the sediment region and some of the terms used in this paper are due to Dr. Gimpel. In addition, the version of the storage management module in SITBOL from which a part of this paper is drawn was written by him. I would like to thank James Gimpel, Ralph E. Griswold and Robert Dewar for many stimulating discussions concerning storage management techniques and programming language implementation.

REFERENCES

1. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, 2nd ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1971.
2. R. E. Griswold, *The Macro Implementation of SNOBOL4, A Case Study of Machine-Independent Software Development*, W. H. Freeman, San Francisco, 1972.
3. J. F. Gimpel, *A Design for SNOBOL4 for the PDP-10*, SNOBOL4 Project Document S4D29b, Bell Laboratories, Holmdel, New Jersey, 1973.
4. J. F. Gimpel, *SITBOL Version 3.0*, SNOBOL4 Project Document S4D30b, Bell Laboratories, Holmdel, New Jersey, 1973.
5. J. F. Gimpel, 'Some highlights of the SITBOL language extensions to SNOBOL4', *SIGPLAN Notices*, **9**, 11–20 (1974).
6. R. B. K. Dewar, *SPITBOL Version 2.0*, SNOBOL4 Project Document S4D23, Illinois Institute of Technology, Chicago, 1971.
7. P. J. Santos, Jr., *FASBOL, A SNOBOL4 Compiler*, Electronics Research Laboratory Memorandum No. ERL–M134, The University of California, Berkeley, 1971.
8. W. R. Sears, III, *The Design of SIXBOL, A Fast Implementation of SNOBOL4 for the CDC 6000 Series Computers*, SNOBOL4 Project Document S4D45, The University of Arizona, Tucson, 1974.
9. J. F. Gimpel and D. R. Hanson, *The Design of ELFBOL—A Full SNOBOL4 for the PDP–11*, SNOBOL4 Project Document S4D34, Bell Laboratories, Holmdel, New Jersey, 1973.
10. D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, Massachusetts, 1973.
11. B. K. Haddon and W. M. Waite, 'A compaction procedure for variable-length storage elements', *Comput. J.* **10**, 162–165 (1967).
12. T. W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.
13. H. Schorr and W. M. Waite, 'An efficient machine-independent procedure for garbage collection in various list structures', *Comm. ACM*, **10**, 501–506 (1967).
14. W. M. Waite, *Implementing Software for Non-Numeric Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.
15. B. Wegbreit, 'A generalized compactifying garbage collector', *Comput. J.* **15**, 204–208 (1972).
16. J. F. Gimpel, *A Hierarchical Approach to the Design of Linkage Conventions*, SNOBOL4 Project Document S4D41, Bell Laboratories, Holmdel, New Jersey, 1974.
17. D. R. Brandt, 'An External Function Facility for SITBOL,' *SNOBOL4 Project Document S4D53*, The University of Arizona, Tucson, 1976.