Performance of Storage Management in an Implementation of SNOBOL4

G. DAVID RIPLEY, MEMBER, IEEE, RALPH E. GRISWOLD, AND DAVID R. HANSON, MEMBER, IEEE

Abstract—Results of measuring the performance of the storage management subsystem in an implementation of SNOBOL4 are described. By instrumenting the storage management system, data concerning the size, lifetime, and use of storage blocks were collected. These data, like those obtained from conventional time measurement techniques, were used to locate program inefficiencies. In addition, these measurements uncovered some deficiencies in the storage management system, and provided the basis upon which to judge the heuristics used in the garbage collector.

Index Terms—Program measurement, SNOBOL4, storage management.

Manuscript received May 11, 1977; revised August 17, 1977. This work was supported by the National Science Foundation under Grant MCS75-21757.

The authors are with the Department of Computer Science, The University of Arizona, Tucson, AZ 85721.

I. INTRODUCTION

CONVENTIONAL techniques of program measurement concentrate on execution time [1]. Storage, however, is an important and expensive resource. This is particularly true of high-level languages such as SNOBOL4, Lisp, and APL, which rely on dynamic storage management to support many language features. Measuring the performance of storage management in these kinds of languages may lead to better implementation techniques, and to a better understanding of the ways in which the dynamic features of such languages are actually used [2].

In SNOBOL4, as in languages such as Lisp and APL, creation of data objects is a run-time activity. Construction of strings, patterns, and arrays all require the allocation of storage during program execution. Other activities, such as pattern matching,

0098-5589/78/0300-0130$00.75

may or may not require allocation. Allocation is an implicit process—there are no declarations or allocation statements in SNOBOL4. Although some operations, such as array creation, obviously require allocation, other operations that cause allocation are not so evident to the programmer, and may even depend on the history of program execution.

SNOBOL4 also has no explicit deallocation operations. Storage reclamation (garbage collection) is implicit and occurs automatically when it is required. Data objects that must be preserved over garbage collection are determined by their accessibility, rather than by any particular program constructs.

It is typical for a SNOBOL4 program to allocate storage continually during execution and for garbage collection to occur at irregular intervals in order to reclaim inaccessible storage for subsequent allocation. While allocation can be charged back to operations that request the storage, garbage collection occurs only when it is required, and hence cannot be reasonably charged back to the operation whose allocation request is unfortunate enough to trigger the collection process.

Since storage management is implicit in SNOBOL4, the SNOBOL4 programmer typically has little conception of program storage requirements unless the total amount of memory required causes problems in running the program. Generally, the time required for allocation and deallocation, as well as the space needed to support transient storage utilization, is largely unknown to the programmer.

A complicating factor is the richness of the SNOBOL4 language, which typically offers many alternative methods of accomplishing the same task. The same data structure may be represented using an array, a string, or a linked list of data objects. Computation may be performed in conventional ways or by using the search and backtrack algorithms of pattern matching. The choice of method has a significant impact on storage requirements, but in the absence of knowledge of these requirements, choices usually are made on *ad hoc* bases.

This paper describes the results of measuring the performance of storage management in SITBOL [3], a widely used implementation of SNOBOL4 for the DEC-10. At the beginning of this investigation, it was uncertain what results might be expected. Even implementers of SNOBOL4 processors have little objective knowledge about the performance of these systems. The basic hypothesis motivating this work is that storage was an important resource whose measurement could provide insight into implementation techniques and program behavior, and possibly provide tools for reducing costs of running SNOBOL4 programs. In addition, the results should have general applicability to other programming languages and implementations that employ dynamic storage management.

The remainder of this paper describes storage management in SITBOL, the instrumentation used to measure the performance of storage management, and the tools developed for processing the resulting data. Some results obtained from storage measurement are described and some of the uses of this kind of measurement are discussed.

II. Storage Management in SITBOL

SITBOL is an interpreter-based implementation of SNOBOL4 designed specifically for the DEC-10. In spite of its interpretive nature, it is sufficiently efficient for production work. One of the reasons for using SITBOL in this investigation was that the results may be of practical value because of the widespread use of SITBOL. In addition, the implementation is well structured and extensively documented [4], [5], facilitating evaluation and the modifications needed for storage measurement. Finally, the storage management facilities of SITBOL are sophisticated and include many of the features included in other recent language implementations such as Algol 68 [6] and SL5 [7].

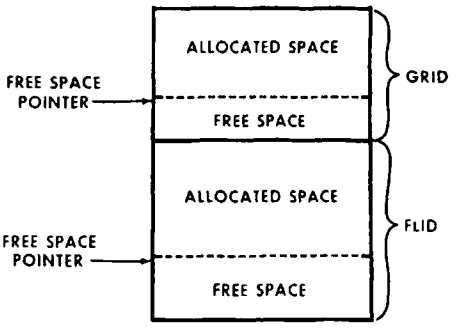


Fig. 1. Allocated storage regions in SITBOL.

SITBOL uses heap storage management techniques with a variety of elaborations and heuristics that are designed to improve performance. There are two main areas from which storage is allocated: the GRID and the FLID. The GRID (*Gr*owing *I*mpure *D*ata) contains dynamically allocated objects that are not subject to reclamation. Examples are I/O buffers and blocks for variables. The FLID (*Fl*oating *I*mpure *D*ata) contains dynamically allocated objects that are subject to reclamation. Examples are strings, patterns, and arrays. In each region, space is allocated upon request, linearly from the beginning of the region. Thus each region is subdivided into an allocated region and a free region. The two regions are contiguous, as shown in Fig. 1.

Allocation is both simple and fast: a free space pointer is incremented by the amount of storage that is requested. Each allocated object constitutes a block of storage and has heading information containing its length and type. The type may correspond to a source-language data type (such as ARRAY), or identify a particular kind of internal object (such as blocks used temporarily during pattern matching).

When an allocation request cannot be satisfied because the remaining free space is inadequate, a garbage collection occurs. A marking phase, tracing possible access paths, is carried out to identify those blocks that must be preserved. Although access paths exist through the GRID, no attempt is made to reclaim space in this region. When the accessible objects in the FLID have been determined, storage in the FLID is compacted, eliminating the inaccessible blocks, and pointers are adjusted accordingly. Following a garbage collection, all the allocated space in the FLID is at the beginning, so that allocation may proceed as before. Details of this process are described in [4].

If the GRID becomes full, a garbage collection is first performed to compact the FLID. The FLID is then relocated to make more free space available in the GRID.

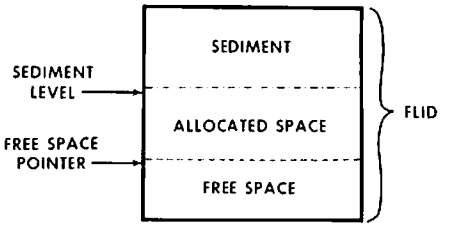


Fig. 2. "Sediment" in the FLID.

One problem with this kind of storage management is the potential for thrashing that results from garbage collections that produce only a small amount of free space. An attempt is made to mitigate this problem by requiring that garbage collection produce an excess of at least 1000 words over the amount needed to satisfy the allocation request. This assures some "breathing room" for subsequent allocation. If a garbage collection does not provide the required breathing room, more memory is requested from the operating system.

Garbage collection also may return more free space than is justified by expected future allocation requests, thus increasing memory charges unnecessarily. If too much space is returned (more than 3000 words), the excess is returned to the operating system. The default values of 1000 and 3000 for these free space limits can be changed by the SNOBOL4 programmer using the explicit COLLECT function with the desired values as arguments.

Another potential cause of thrashing occurs in the expansion of the GRID. When the GRID is expanded, a margin of 500 words is provided. There is no provision for the programmer to change this value.

Several other heuristics are used in an attempt to improve performance [4]. One of these heuristics concerns blocks that accumulate at the bottom of the FLID. This "sediment" is due to objects with long retention periods (such as compiled code and patterns and arrays that are set up during program initialization and which remain accessible throughout program execution). As garbage collections occur, such objects are forced to the bottom of the FLID by compaction, as illustrated in Fig. 2.

The sediment level is determined for the succeeding garbage collection by observing the first unmarked block in the FLID. In hopes of reducing processing time during garbage collection, no attempt is made to collect objects in the sediment unless garbage collection fails to produce the required margin. The sediment is broken up when the GRID is expanded and when one garbage collection results in a sediment level below the previous one. In the latter case, the next garbage collection reclaims inaccessible blocks in the sediment.

Another heuristic has to do with the handling of strings. Strings are stored in blocks of contiguous characters and referenced by pointers indicating the length and offset of referenced substrings. As program execution continues, situations may arise in which there are pointers into a string block, but significant portions of the block may be no longer accessible. In such a case, the unused trailing portion of the string block is discarded during garbage collection. To balance the savings in space produced by this heuristic against the time required for the processing, string block compression is attempted only on every eighth garbage collection.

III. Collection and Analysis of Storage Management Data

Information about individual blocks provides most of the data needed to measure the performance of storage management. Relevant facts include the type of block, its size, its time of creation, its "lifetime," and the source-language operation responsible for its creation. The first two items are contained in the standard header that is part of all blocks in the FLID, the region of greatest interest. An additional word was added to this header and the allocation routines were modified to store the creation time and the number of the statement which caused the allocation (providing an approximation to the specific source-language operation). The information contained in this enlarged header is called the *creation history*.

Since SNOBOL4 has no specific deallocation primitives, the lifetime of a block is determined not by its actual period of accessibility from the source-language program, but rather by its accessibility when garbage collection occurs. That is, a block may be created and subsequently become inaccessible, but still occupy storage until the next garbage collection occurs. Since this block does occupy storage until garbage collection under such a system, it is reasonable to consider block lifetimes as ending only at garbage collections. Consequently, the state of storage when garbage collection occurs provides a realistic picture of storage utilization. Furthermore, since garbage collections tend to occur fairly frequently, adequate resolution is obtained by this approach.

To provide the data needed to measure the performance of storage management, the state of the FLID including all block creation histories is output before and after each garbage collection and at program termination. These accumulated data provide the raw material for the analysis of various aspects of storage management.

Using the method just described, a substantial amount of data are gathered during the execution of a typical SNOBOL4 program. As is usually the case in such investigations, the major problem is not getting the data, but rather in making sense out of it—suppressing detail and integrating useful information into a concise and meaningful form. To accomplish this, programs were written to postprocess the raw storage management data.

Block type	Stmt	Blocks Allocated	Blocks Collected	Words Allocated		Words Collected		Average Block Lifetime	Collected / Allocated
datatype field	700	2	0	12	0.5%	0	0.0%	0.013	0.0%
defined datatype	700	1	0	7	0.3	0	0.0	0.026	0.0
string		6	3	32	1.2	12	0.6	0.031	37.5
	100	3	1	13	0.5	4	0.2	0.054	30.8
	800	3	2	19	0.7	8	0.4	0.008	42.1
table		2	1	2506	97.7	2003	99.1	0.037	79.9
	200	1	1	2003	78.1	2003	99.1	0.039	100.0
	600	1	0	503	19.6	0	0.0	0.036	0.0
table element	300	1	1	7	0.3	7	0.3	0.039	100.0

Fig. 3. Summary of storage management data.

```
Garbage Collection 3, at statement 700, after executing 7 statements:

   Location          Before    After

Beginning of FLID      3977     4480
Sediment level         3977     4480
Free pointer           6610     5101
End of FLID            7103     7103

                      Blocks     Words

Storage recovered        2      2010 79%
Storage preserved        4       519 21%

    Action              Block Type          Stmt   Size   Lifetime

Storage recovered     table                   200   2003     0.039
                      table element           300      7     0.039

Storage preserved     string                  100      4     0.056
                      string                  100      5     0.056
                      table                   600    503     0.010
                      defined datatype        700      7     0.000
```

Fig. 4. Detailed storage management data.

One of these programs condenses the raw data and provides a summary "profile" of various storage parameters and the blocks in the FLID. Options permit the presentation of data by source-program statement number, block type, or both. Fig. 3 illustrates a typical profile produced by this program.

Such summaries often raise questions that can be answered only by examination of more detailed data. To provide a tool for gleaning this information, another program was written to provide a "motion picture" of FLID usage. An example of the results produced by this program is shown in Fig. 4, which depicts data from a single garbage collection.

Most of the performance analysis was obtained by using these two postprocessing programs, applied to a variety of SNOBOL4 programs. The programs chosen represent a variety of programming styles and levels of quality, but were mostly selected from programs in production use or appearing in the literature. Some of the more significant results of these analyses are given in the following sections.

IV. Locating Program Inefficiencies

It is well known that one of the most efficacious uses of performance measurement is to locate high spots that contribute disproportionately to the cost of program execution [1]. In many programs, rewriting of only a few high spots can lead to substantial performance improvements.

One interesting question is whether storage management data can be used in this fashion. Storage management is relatively uncorrelated with specific language constructs. Therefore it is possible that storage measurement cannot be used to significantly improve program performance because of this lack of correlation. On the other hand, storage high spots may identify problems that would not be evident in conventional time measurement. In SNOBOL4, garbage collection occurs when it is needed. In time sampling measurements, the time spent in garbage collection is charged to the statements in which garbage collection happens to occur, even if those statements are not responsible for significant allocation. Thus a statement with heavy allocation demands actually may not be charged for the subsequent processing costs that are incurred. A number of tests were made in an attempt to settle this issue.

In a program called SLICER, which produces summaries of SNOBOL4 program execution time activity from measurement samples [8], two heavily exercised statements were of

the form

```
LE(A,B) IDENT(C,D)                    :S(L)
```

The programmer intended a multiple predicate: if A is less than or equal to B and C is identical to D, then proceed to the statement labeled L. While having the same control effect as a multiple predicate, this statement actually produces a needless pattern match of "pattern" IDENT(C,D) on subject LE(A,B), resulting in the allocation of 17 words of temporary storage per statement execution. During one execution of SLICER, storage measurements indicated that these statements accounted for about 4000 words of storage allocated and two garbage collections. The solution, of course, is to enclose the two predicates in parentheses, forming a compound subject. This results in roughly a 10 percent decrease in program execution time. Although time measurement with intrastatement resolution would have identified pattern-matching activity and hence raised suspicions, a statement-level time measurement would not have indicated this problem.

The second example, from a formatting program [9], illustrates how deceptive storage management in SITBOL can be. Storage measurement identified a single pattern-matching statement in which an inordinately large amount of allocation occurred. This statement was particularly simple:

```
GET    CONTROL                    :S($CSTRING)F($MODE)
```

The purpose of this statement is to test input lines in order to distinguish control lines from text lines. The sources of allocation in this statement are the pattern-matching process itself and an unevaluated component in the pattern CONTROL, which is defined as

```
CONTROL = POS(0) *CW (BREAK(" ") . CSTRING REM . VALUE
+                  | REM . CSTRING NULL . VALUE)
```

The identifying control character is the value of CW, which is left unevaluated so that it can be changed by the user during formatting if desired.

Since CW is not evaluated until pattern matching occurs, a pattern component for it is built dynamically each time the statement is executed. This is done once for each line of input. The resulting allocation of storage is significant and shows up as a high spot. The unevaluated component in CONTROL can be removed without affecting the program behavior by simply changing the mechanism by which the user redefines the control character so that the entire pattern CONTROL is rebuilt. Since changing the control character is done very infrequently, the additional processing required to rebuild CONTROL is insignificant in practice.

The results of this trivial change—modifying only two lines of the program—were dramatic. In formatting a typical document, the overall amount of storage allocated dropped by 15 percent and the overall time and cost of running the program dropped by 8 percent.

Interestingly, time sampling does not show this high spot so clearly. The reason is that 25 percent of program execution time saved by this optimization is due to reduction of garbage collection time, which in general would not be attributed to these statements by time sampling. Even if this high spot had been identified by conventional time-sampling techniques, the source of the problem would not have been as evident and would likely have been attributed to the complexity of the pattern and the frequency with which the pattern-matching statement was executed. Thus storage measurement not only located the high spot but also identified its specific cause, making an improvement possible.

There are cases, however, in which storage measurement provides no additional information concerning program inefficiencies. For instance, experience from these tests indicates that in some cases either storage requirements are fairly obvious, time sampling identifies the same high spots as storage management, or little improvement in program performance can be obtained by attempting to rewrite program sections identified as allocation high spots.

An example is the line justification process in ACOLYTE [10], a rather large document preparation program. Line justification typically accounts for about 30 percent of the storage allocated and 10 percent of ACOLYTE's execution time. Examination of the code, which inserts extra blanks in lines to produce even margins, suggests several possible optimization techniques. The overall saving in storage resulting from one of these optimizations was only 3 percent, however. Furthermore, it was not at all clear that additional optimization would reduce storage usage significantly, or that reductions would result in significantly lower execution costs.

ACOLYTE also uses a dictionary file for hyphenation. In order to hyphenate a word, ACOLYTE searches the file sequentially. Search of the dictionary requires the words to be read into memory, which causes the allocation of a large number of string blocks. Measurement of a typical case showed that the few statements involved in searching the hyphenation file consumed 37 percent of the storage used by ACOLYTE and 27 percent of its execution time. While the hyphenation facility certainly could be redesigned (such as using a hyphenation algorithm and a user-supplied dictionary of exceptions), such optimizations also would have been suggested by a high point in time sampling, giving storage measurement no significant advantage over time measurement.

Another example occurs in SLICER. Dynamic measurement of SLICER's storage usage proved interesting and useful in some ways as described above. However, its storage can largely be estimated statically given a few facts such as the number of words required in SITBOL to store numeric data and pointers, which items are pointed to and which are self-contained, and table element size. This static information, used in conjunction with time measurements, led to many of the reductions made in storage usage of SLICER.

V. Storage Usage and Language Implementation

The analysis of actual storage utilization can be used to improve the design and implementation of a storage management system [2]. This section describes some of the investi-

gations and results that relate to the implementation of storage management in SITBOL.

Heuristics

The various heuristics used in SITBOL storage management operate in an environment of such complexity that analytic techniques for their evaluation are impractical. Heuristics frequently have intuitive appeal, but their actual usefulness is hard to demonstrate. To get a better assessment of the heuristics, the storage management algorithm was modified and the resulting changes in performance were measured.

Storage management was first simplified by making the sediment and string compression heuristics optional. The range of performance differences observed for a number of SNOBOL4 programs are as follows. (Here "time-space" refers to kilocore-seconds.)

	String Compression	Sediment
Time improvement:	-1 percent to 1 percent	0 percent to 5 percent
Time-space improvement:	-1 percent to 1 percent	1 percent to 4 percent

Although these figures are in the "noise" range with respect to the reliability of the timing, it is clear that the value of string compression is very questionable and the potential improvement of performance from sedimentation is minor. For string compression, it seems that the time spent in doing the compression may be greater than the savings effected.

In another area, a new feature, "dynamic breathing room," was added after measurement data showed that for certain programs only a small fraction of the FLID was collectable. Due to the static size of breathing room in the FLID, this can result in many time-consuming garbage collections (garbage collection time in SITBOL is roughly proportional to the amount of preserved storage). The dynamic breathing room heuristic modifies these minimum and maximum values based on storage usage patterns. Both ends of the range are incremented each time memory expands and decremented each time memory contracts. This usually results in fewer regenerations, somewhat more memory used, shorter execution time, and an overall execution cost savings. The range of improvement was

	Breathing Room
Time improvement:	-1 percent to 17 percent
Time-space improvement:	-1 percent to 9 percent

System Deficiencies

In addition to evaluating the basic properties of the storage management system, a number of logical flaws or oversights were discovered through the use of storage measurement tools. Some of these problems are rather minor. Other deficiencies vary in importance, depending on the characteristics of the program being executed. What follows is a list of some of these, not as a criticism of SITBOL but rather to illustrate the value of storage measurement in uncovering deficiencies that are bound to exist in large, complex systems, and are difficult to uncover in any other way.

Unnecessary Allocation: One problem was exposed by a summary by source statement and block type of a program that used SNOBOL4 tables. The profile indicated that seven words of storage were consumed for each table element accessed, regardless of whether the element existed prior to accessing. Thus unnecessary allocation continually occurred when existing table elements were accessed. In one program this accounted for 4000 words of allocated storage and several garbage collections. Investigation showed that the table element access mechanism always allocates space for a new table element, on the possibility that the element had not yet been defined. Although no attempt was made to remedy most problems uncovered by storage measurement, a minor change was made to fix this one, with the following improvement.

Time improvement:	2 percent to 9 percent
Time-space improvement:	1 percent to 9 percent

Another example of unnecessary allocation occurs in the CODE function that compiles additional source code at runtime. SITBOL allocates sequences of 400-word blocks for the compiled code, the unused portion of which becomes garbage if no additional space is allocated by CODE. Storage measurement showed that frequently only a few words of the block were used, producing extraneous garbage on each call of CODE. Since code created during execution tends to be short, a smaller block size for this purpose is desirable.

Matching Strings: Storage measurements pointed out that simple string patterns are handled in the same way as arbitrarily complex patterns, invoking the general pattern-constructing mechanism and unnecessarily consuming storage. While some extra code is required to handle this case specially, it does seem to occur frequently and in fact most other SNOBOL4 implementations do consider it worthwhile to treat this case separately. In addition, by treating this case separately, the allocation resulting from the multiple predicate

```
LE(A,B) IDENT(C,D)
```

discussed earlier could be avoided without requiring the programmer to change the source statement.

Transient Storage: Heavy storage consumption in a number of programs measured was due to "transient" strings, i.e., strings created and discarded within the execution of one statement. An example is

```
OUTPUT = "The temperature," TEMP ", and humidity,"
+        HUMID ", result in a chill factor of" CF
```

Here five concatenations are performed per statement execution, four of which are transient. Furthermore, when OUTPUT is assigned a value, its previous value is of course discarded. A statement of this type in SLICER accounted for 8 percent of execution time and 16 percent of storage usage. Hyphenation in ACOLYTE, described earlier, involves input to a variable that was examined. If no match was found, the statement was reexecuted, resulting in the previous value of the variable being discarded. The point is that transient storage can sometimes be immediately deallocated, and hence never contribute to accumulating garbage. Other proposed SNOBOL4 implementations, such as SIXBOL [11], use a separate storage region

for strings in order to treat this important type of storage specially.

Use of Character-Set Blocks: The example concerning the pattern assigned to CONTROL, given in Section IV, illustrates another use of storage measurement to suggest a refinement in an implementation. A profile of the block types allocated by the pattern-matching statement prior to modification shows approximately one-fourth of the storage is allocated for character-set blocks [12]. Ordinarily character sets are used by lexical pattern-matching primitives such as BREAK and SPAN. Each construction of a pattern component for *CW also consumes a portion of a character-set block, which is not subsequently reused, accounting for the large amount of allocation of this block type. An examination of the pattern at first reveals no explanation of this unusual situation—there is nothing inherent in the pattern that requires the use of a new character-set block for each pattern match.

The cause of this problem actually lies in an unnecessary application of a heuristic that SITBOL uses to quickly position the cursor to the first character of interest in a string. This heuristic only applies when the first component of a pattern is a character string and matching is unanchored. In the situation here, the dynamic construction of a pattern component that results from *CW treats this as a separate pattern without recognizing that it is imbedded in a context that makes this heuristic meaningless. The entire pattern still works properly because of the presence of POS(0), which causes the heuristic to be bypassed. The extra storage is consumed needlessly, however. It is doubtful that this "bug" would ever have been discovered without the attention provided by storage measurement.

VI. Conclusions

The techniques used to measure the performance of storage management in SITBOL have a brute-force character. The major reason for this approach was the lack of prior work in this field and hence the absence of *a priori* information on the areas that were important to study. Adding facilities to an existing system rather than designing them in conjunction with the design of the system contributed to the nature of the result.

The main consequence of the brute-force approach is the massive amount of data that accumulates from a typical measurement. As a result, the artifact is large, the measurement tool is relatively cumbersome to use, and it is difficult to extract meaningful results from the measurement data. Postprocessing programs reduce the last problem to a manageable level, but there still is much information inherent in the measurement data that is difficult to extract. Examples of possible further analysis are the determination of average block lifetimes, average lifetimes of specific block types, "spectral analysis" by block type, and statistical summaries of more sophisticated kinds. The problem is a typical one: the amount of work needed is large, even prohibitive. Therefore selections must be made. There is no guarantee that the selections made in this paper are the most meaningful ones, although that seems likely. Conversely, if it could be determined that a more restricted set of data was adequate to provide the information needed for performance measurement, the instrumentation might be considerably simplified and the artifact and related problems correspondingly reduced.

Applicability of the Results

As indicated in the preceding sections, one of the areas that may benefit from storage management performance measurement is the optimization of programs written in SNOBOL4 and other languages in which dynamic storage management plays a major role. Unfortunately, the usefulness of storage management measurement as an optimization tool is questionable. Conventional time-measurement facilities often indicate storage utilization inefficiencies, thereby reducing the usefulness of a separate tool. In addition, even in languages with many dynamic features, it appears that static analysis, which is simpler and cheaper, often proves as useful as dynamic measurement and furthermore often provides more directly useful information on methods of improving program performance.

In many cases, the tradeoff between space and time that takes place internally balances out the total cost. For example, a greater allocation of storage may offset inefficient access mechanisms. Perhaps more importantly, alternative forms of storage usage present complexities that the average source-language programmer may not wish or be able to cope with. Therefore, storage management measurement may be difficult for the source-language programmer to interpret and use constructively. In SNOBOL4, at least, implicit management of storage is a strong component of the language design. For the SNOBOL4 programmer to have to become sufficiently expert in the language internals and to become preoccupied with "what is going on behind the scenes" runs contrary to the philosophy of the language and neutralizes many of the benefits the language otherwise offers.

Storage management measurements appear to be most useful for improving the algorithms and strategies used in the implementation of dynamic storage management subsystems.

It is generally known that almost all implementations of complex systems have large *ad hoc* components and that accurate, quantitative measures of their performance are unavailable. Quality of performance is therefore largely a matter of conjecture, inference, and comparison with other, similar systems. In the abstract, algorithms can be analyzed, compared, and used as the basis for design. In the extremely complex environment of the running program, measurements of the type used in this study may be the only practical method of validating performance and locating unsuspected problems.

The discoveries made in measuring the performance of storage management in SITBOL are not important in themselves, but they provide a case in point, and are all the more significant because of the basically high quality of the SITBOL implementation.

Since SITBOL distills much of the conventional wisdom about storage management techniques suitable for SNOBOL4-like languages, the problems located in this study may be useful in the continuing attempt to produce viable implemen-

tations of these kinds of languages on smaller and smaller computers. In these efforts, there is a fundamental problem: nonnumeric data inherently occupy more space than numeric data. Strings simply require more memory than numbers. In limited-memory environments, the question is not so much one of performance but of feasibility. Perhaps the most significant problem here is that of transient storage, mentioned in Section V. Unfortunately, the measurement techniques used for SITBOL do not provide enough resolution to determine the extent of this problem, since retention periods in SITBOL are extended to times of garbage collections. Techniques to effectively deal with this problem are essential for implementations on small machines. One approach to this problem is described in [13].

It is difficult to draw general conclusions about the implementation of dynamic storage management systems from the specific experiences with SITBOL. These experiences do, however, lead to several recommendations concerning the design and implementation of dynamic storage management systems:

1) a basically simple strategy for storage management should be chosen for the initial implementation;

2) a measurement facility similar to that described in this paper should be incorporated in the design from the beginning;

3) using this measurement facility, sources of inefficiency should be sought and heuristics or more complex strategies should be added only as there is evidence of the need for them and their utility in practice.

Acknowledgment

The authors are indebted to C. C. Daugherty for implementing most of the instrumentation and for running test programs.

References

[1] D. E. Knuth, "An empirical study of Fortran programs," *Software–Practice and Experience*, vol. 1, pp. 105–133, Apr.–June 1971.

[2] D. W. Clark and C. C. Green, "An empirical study of list structure in Lisp," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 78–86, Feb. 1977.

[3] J. F. Gimpel, *SITBOL Version 3.0*, Tech. Rep. S4D30b, Bell Lab., Holmdel, NJ, Nov. 1972.

[4] D. R. Hanson, "Storage management for an implementation of SNOBOL4," *Software–Practice and Experience*, vol. 7, pp. 179–192, Mar.–Apr. 1977.

[5] J. F. Gimpel, *A Design for SNOBOL4 for the PDP-10*, Tech. Rep. S4D29b, Bell Lab., Holmdel, NJ, May 1973.

[6] U. Hill, "Special runtime organization techniques for Algol 68," in *Lecture Notes in Computer Science, Vol. 21: Compiler Construction*, G. Goos and J. Hartmanis, Ed. Berlin: Springer-Verlag, 1974, pp. 222–252.

[7] R. E. Griswold and D. R. Hanson, "An Overview of SL5," *SIGPLAN Notices*, vol. 12, pp. 40–50, Apr. 1977.

[8] G. D. Ripley, "Program perspectives: A relational representation of measurement data," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 296–300, July 1977.

[9] R. E. Griswold, *String and List Processing in SNOBOL4, Techniques and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1975, pp. 168–191.

[10] R. O. Anderson and R. E. Griswold, *ACOLYTE, A Document Formatting Program*, Tech. Rep. S4PD11b, Univ. Arizona, Tucson, AZ, Feb. 1976.

[11] W. R. Sears, *The Design of SIXBOL: A Fast Implementation of SNOBOL4 for the CDC6000 Series Computers*, Tech. Rep. S4D45, Univ. Arizona, Tucson, AZ, Nov. 1974.

[12] J. F. Gimpel, "The minimization of spatially-multiplexed character sets," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 315–318, June 1974.

[13] J. F. Gimpel and D. R. Hanson, *The Design of ELFBOL–A Full SNOBOL4 for the PDP-11*, Tech. Rep. S4D34, Bell Lab., Holmdel, NJ, Oct. 1973.

G. David Ripley (M'72) received the B.A. degree in mathematics from the California State University at Sacramento, the M.A. degree in mathematics from the University of California, Berkeley, and the Ph.D. degree in computer science from Iowa State University, Ames, in 1965, 1967, and 1970, respectively.

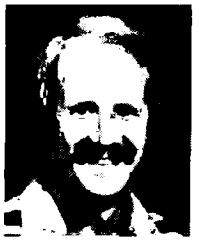
He was a member of the Programming Language Research Group at RCA Laboratories in Princeton from 1970 to 1972. He is presently an Assistant Professor of Computer Science, University of Arizona, Tucson, with interests in program performance, programming languages, and language translators.

Dr. Ripley is a member of the Association for Computing Machinery, Phi Kappa Phi, and Sigma Xi.

Ralph E. Griswold was born in Modesto, CA, on May 19, 1934. He received the B.S. degree in physics in 1956, and the M.S. and Ph.D. degrees in electrical engineering in 1960 and 1962, respectively, all from Stanford University, Stanford, CA.

From 1962 to 1971, he was a member of the Technical Staff of Bell Laboratories, where he was head of the Programming Research and Development Department. He is presently Professor and Head of the Department of Computer Science, University of Arizona, Tucson, AZ. His research interests include programming language design and implementation, nonnumeric computing, programming methodology, and software engineering.

Dr. Griswold is a member of the Computer Society, the Association for Computing Machinery, and the Association for Computational Linguistics.



David R. Hanson (M'72) was born in Oakland, CA, in 1948. He received the B.S. degree in physics in 1970 from Oregon State University, Corvallis, the M.S. degree in optical sciences in 1972, and the Ph.D. degree in computer science in 1976, both from the University of Arizona, Tucson.

From 1970 to 1973, he was a member of the Research Staff at Western Electric Engineering Research Center, Princeton, NJ, where he did applied research initially in laser physics and then in computer science. From 1976 to 1977, he was an Assistant Professor of Computer Science at Yale University, New Haven, CT. He is presently an Assistant Professor of Computer Science at the University of Arizona, Tucson. He is also a consultant for ITT Telecommunications Technology Center, Stamford, CT. His areas of interest include the design and implementation of programming languages, programming methodology, operating systems, and software engineering.

Dr. Hanson is a member of the Association for Computing Machinery and the American Physical Society.