Measuring the Performance and Behavior of Icon Programs

CARY A. COUTANT, RALPH E. GRISWOLD, AND DAVID R. HANSON, MEMBER, IEEE

Abstract—The importance of the ability to measure the performance of programs written in high-level languages is well known. Performance measurement enables users to locate and correct program inefficiencies where automatic optimizations fail and provides a tool for understanding program behavior. This paper describes performance measurement facilities for the Icon programming language, and shows not only how these facilities provided insight into program behavior, but also how they were used to improve the implementation.

Index Terms—Icon, program measurement, storage management.

I. Introduction

THE importance of the ability to measure the performance of programs written in high-level languages is well known [12]. The most obvious advantage of measurement facilities is the possibility of locating and correcting inefficiencies. While compilers can perform many optimizations automatically, there are also many situations in which user optimizations, based on measurement data, can improve program performance where automatic techniques cannot. Measurement also provides an experimental tool for analyzing and understanding program behavior. In the case of complex algorithms whose performance is sensitive to data load, analytic techniques may be impractical and measurement may be the only practical solution.

High-level languages, by their nature, present problems in understanding performance and behavior that lower level languages do not. A Fortran programmer, for example, can usually relate Fortran language constructs to the machine code produced by the Fortran compiler, and hence to the basic operations of the computer. Most conventional computers after all are "Fortran machines." A SNOBOL4 or SETL programmer, on the other hand, has no such ready mapping between the source language and the operations of the computer. This is, of course, the intent of such high-level languages: to provide language constructs suitable for formulating solutions to complex problems that are unrelated to conventional computer architecture.

Despite the evident importance of being able to measure the performance of programs written in high-level languages, existing facilities typically are difficult to use and the results are frequently misleading. Major shortcomings include inadequate or incomplete resolution, inability to relate measurement data to the source program, and excessive overhead.

Earlier work with the SNOBOL4 programming language [17] showed that measurement tools, as expected, can aid the programmer of a high-level programming language in improving

Manuscript received September 10, 1980; revised August 13, 1982. This work was supported by the National Science Foundation under Grants MCS-7521757, MCS-7903890, and MCS-8101916.

C. A. Coutant was with the Department of Computer Science, University of Arizona, Tucson, AZ 85721. He is now with the Information Systems Laboratory, Hewlett-Packard, Cupertino, CA 95014.

R. E. Griswold and D. R. Hanson are with the Department of Computer Science, University of Arizona, Tucson, AZ 85721.

and understanding programs. They also can give insight into the behavior of complex implementations, and, in fact, can lead to improvements in implementations that cannot be achieved by other means.

The work with SNOBOL4, however, required adding instrumentation to a long finished implementation. One of the conclusions of that work was that much better measurement tools would have been possible if their design and implementation were done concomitant with the implementation of the language itself, as opposed to being ad hoc appendages (as is typically the case).

This paper describes measurement tools developed for Icon, a recently developed high-level programming language [5], [7]. Since the measurement tools in Icon were designed and implemented in conjunction with the implementation of the language, they provided an opportunity to test the conclusions drawn from the earlier work with SNOBOL4. In addition, Icon has a number of unusual features for which the measurement of program performance and behavior is, in itself, of interest.

Some of the instrumentation of Icon is conventional and is not described here. An example is sampling in order to observe the behavior of the runtime routines. Such information is easily obtained and is typical of measurements made by implementors at the stage where implementation improvements are being considered. This paper, however, is concerned with less traditional approaches and techniques, particularly measurement tools that give the user information on program behavior and performance directly related to the program test, and the instrumentation of the storage management system to aid in its development and improvement.

The next section gives an overview of Icon and its implementation. Section III reviews the major issues in the design and implementation of performance measurement tools for programming languages in general. In Section IV, the instrumentation of Icon is described and examples of the more important measurement facilities are given. Some typical experiences in the use of the measurement facilities are given in Section V. Analysis of the storage management system in Icon and its use to improve the implementation are the subjects of Section VI.

II. The Icon Programming Language and its Implementation

The characteristics of the programming language being measured strongly affect the nature of the measurement facilities. This is both inevitable and desirable—the user needs measurement results that are easily related to the programming language being used.

The organization of the Icon system and the way that it is implemented affect program performance (but, hopefully, not behavior). They also affect how instrumentation may be done and, to some extent, what is measured. The relevant aspects of Icon and its implementation are described in the following sections.

A. Characteristics of the Icon Programming Language

Syntactically, Icon is an expression-based language in the style of Algol 68 and Bliss [22], and has most traditional control structures. The syntax of a language influences both the selection of data to be measured and the methods of presenting it. There is a substantial difference in the approaches that can be taken between a language whose syntax is hierarchical, such as SNOBOL4 [16], and a language with a recursive syntax such as Icon. This matter is discussed in more detail in the next section.

The major semantic characteristics of Icon, as they affect program measurement are as follows:

1) automatic, dynamic storage management,

2) a variety of data types, including some unusual ones such as character sets and SNOBOL4-style tables,

3) lack of type declarations, with automatic type checking and coercion in context,

4) goal-directed expression evaluation.

The first three characteristics of Icon are shared by other, longer established languages like SNOBOL4. Goal-directed evaluation is more novel, especially since it is a general feature of Icon, instead of being limited to a particular part of the language, as it is in SNOBOL4 pattern matching [6].

Associated with goal-directed evaluation in Icon are conditional expressions that may succeed or fail (as in SNOBOL4) and *generators*, which are expressions that may produce more than one value, if this is required to achieve success (the "goal") in a larger surrounding expression. Comparison operations are typical of conditional expressions. For example,

$x > y$

succeeds if x is greater than y and fails otherwise. In Icon, success and failure drive control structures, as in

if $x > y$ then $z := x$ else $z := y$.

The concepts of success and failure, as opposed to Boolean values, are more general in that they are transmitted to enclosing expressions, so that any operation in Icon may fail (if, for example, one of its operands fails).

One of the basic generators is alternation, $e1 | e2$, which produces first one value and then another if the first value does not lead to the success of the expression in which the alternation is contained. An example the use of alternation is

if $(x | y) > u$ then $z := x$ else $x := u$

which assigns the value of x to z if either x or y is greater than u; otherwise is assigns the value of u to x.

Another generator is

i to j by k

which generates successive integer values from i to j, inclusive, using k as an increment. For example,

if x = (0 to 10 by 2) then z := 0

assigns zero to z if the value of x is an even number between zero and ten, inclusive.

There are many other generators in Icon, a number of which are associated with string processing. For example,

find ($s1$, $s2$)

returns the position in $s2$ at which $s1$ occurs as a substring of *s*2 (failing if there is no such position). Furthermore, it generates additional positions, as needed, from left to right, in case *s*1 occurs as a substring of *s*2 in more than one place. Thus,

if *i* := find (*s*1, *s*2) > 10 then write (*i*)

writes the first position at which *s*1 occurs as a substring of *s*2 at a position greater than ten, if there is such a position.

One important control structure in Icon causes generators to produce all their values in sequence

every *e*1 do *e*2.

For every value produced by *e*1, *e*2 is evaluated. For example,

every *i* := find (*s*1, *s*2) do write (*i*)

writes all the positions at which *s*1 occurs as a substring of *s*2.

A knowledge of all features of Icon is not necessary to understand the examples given in this paper—most of the features used in the examples are obvious, at least in their general nature. For a more complete description of Icon, see [5]-[7].

B. The Implementation of Icon

The portable Icon system is written in Ratfor [11]. It has been implemented on a number of computers, including the CDC Cyber, Cray-1, DG MV8000, DEC-10, Honeywell 66/60, IBM 370, PRIME 400, and VAX-11/780. Its instrumentation, as part of the portable system is, also essentially portable. The results described in this paper were obtained from the DEC-10 implementation.

The Icon system consists of two parts: a translator and a runtime system. The translator converts an Icon program into an executable code. This translated program, in turn, consists primarily of calls to the runtime routines that carry out the language operations. Because of language features like runtime type checking and coercion, very little actual computation is performed by the code produced by the translator. This implementation strategy, which is used in a number of implementations of SNOBOL4 [2], is not actually interpretive, since the translated program *is* executable. However, the translated program is primarily a driver and most program execution occurs in the runtime system.

A major component of the runtime system consists of storage management routines [8]. Because of the importance of storage management in Icon and its effect on the overall performance of the language, storage management is an important issue in itself. In fact, Icon programmers, like SNOBOL4 programmers, are often forced to consider the utilization of storage, even though it is not part of the semantics of the language *per se*.

The Icon storage management system consists of allocation routines and regeneration (garbage collection) routines. Allocation is a simple and fast process. Space is allocated contiguously within a region. A pointer to the beginning of free space is incremented to provide the space specified in an allocation request. Storage regeneration, which occurs when there is not enough space remaining to satisfy an allocation request, is a complicated and relatively slow process that involves distinguishing data that must be saved from data that may be discarded, relocation of saved data, and so on.

There are four regions in which data are allocated, corresponding to the nature of the data: integer, string, qualifier, and heap. Integers are allocated because the Fortran virtual machine model does not allow any spare bits to differentiate between words that contain integers and those that contain pointers and such. A level of indirectness is therefore needed to differentiate types. The string region contains character data, while the qualifier region contains pointers into the string regions that identify specific strings. Miscellaneous objects, such as lists, records, and character sets are allocated in the heap region.

Each region has its own allocation routine and regeneration routine. There are provisions for enlarging regions and relocating adjacent regions if necessary.

III. Technical Problems in the Design of Measurement Facilities

There are a number of problems related to the design and implementation of programming language measurement facilities. General discussions of the problems and various approaches to solving them are given in [12], [18], and [20]. Examples of measurement facilities for specific languages are given by [12], [21], [14] for Fortran, [18] for Algol-W, [15] for Pascal, and [1] and [3] for Algol 68-R. This section provides a brief survey of the following major issues:

1) methods of measurement,
2) selection of activities to be measured,
3) charge back of activity to program function,
4) instrumentation,
5) space and time artifact,
6) presentation of the results.

A. Measurement Methods

There are several methods of measurement that are commonly used. One is periodic sampling, in which the measurement facilities are activated by external interrupts from a system clock. When an interrupt occurs, measurement routines gain control and generate measurement data such as the location at which the interrupt occurred. If enough samples are taken, the measurement data resemble randomly selected data and, on the average, give an approximation to the actual behavior of the program. For example, average times for the execution of an operation can be approximated from the percentage of samples in which that operation is active. This technique is the standard way of generating program histograms and similar data [9], [10].

There are several problems with periodic sampling. The clocks available to the user typically lack resolution—a 60 Hz frequency is common. With such a low sampling rate, a program must run for a long period of time in order to produce enough samples to give a meaningful picture of program activity. For fast CPU's such long periods of program execution may be inordinately expensive or may require artificial techniques (such as multiple runs) to obtain enough samples to measure real programs.

In some cases, other uses of the clock may bias or invalidate the results of sampling. For example, if the operating system and a measurement routine use the same clock for different purposes, there may be effects that violate the underlying assumptions on which the interpretation of measurement data are based. For example, on the DEC-10, which uses the clock for scheduling, programs running on a heavily loaded system may be timed as taking as much as 20 percent longer than when run under light load conditions. These kinds of problems, which depend strongly on hardware and operating system characteristics, reduce the accuracy of and confidence in sampling measurements and make comparisons between runs virtually useless.

The other commonly used method of measurement is event monitoring. In this technique, selected program activities trigger measurement routines. For example, a garbage collection may cause a measurement routine to be called. Thus data can be gathered on specific events or on classes of activities. This method is strongly influenced by the properties of the language being measured and by the characteristics of its implementation.

B. Selection of Activities for Measurement

The selection of activities to be measured and the actual data to be produced is an interesting and complex problem and is strongly influenced by the information that the user seeks. At one extreme, specific aspects of program activity, such as storage allocation, can be selected for measurement. At the other extreme, a measurement facility may attempt to provide data on all aspects of program behavior. While there are clearly limitations to this approach, an approximation to it is an appealing starting point, especially for high-level languages, since *a priori* assumptions about the most important aspects of program behavior are suspect.

One of the first problems to resolve is the viewpoint from which measurements are to be taken. Measurement may be at the source-language level, in terms of the syntax and semantics of a program: elementary operations, statements, blocks, procedures, and so forth. Measurement may also be in terms of categories of program behavior, such as input/output, storage allocation, structure referencing, and so forth. Progressing farther from the source language, measurement may be in terms of the specific implementation characteristics, such as interpretive overhead and garbage collection for languages like Lisp and SNOBOL4. For the implementor, information on activity of the translated program or on the utilization of run-time routines may be of interest.

The data that are to be produced when program activity is sampled or when an event occurs must be determined. For the most complete measurement, a record of relevant data, perhaps including the time, may be produced for each sample or event. For any significant amount of measurement, such records cannot be kept in memory and must be written to external storage. Other kinds of data, such as the total counts for particular activities, usually can be kept in memory.

C. Charge Back

Charge back is concerned with the attribution of measurement data to components of a program. The fact that a particular routine is called frequently may not be as interesting as are the program activities that the routine causes. For example, an operation that allocates storage may cause a garbage collection. As with the selection of activities to be measured, charge back may be related to the hierarchical structure of the program itself, to categories of behavior, or to implementation specifics.

D. Instrumentation

Instrumentation to obtain measurement data is an implementation matter and of less direct interest here than problems related to the selection and interpretation of the data. Furthermore, instrumentation varies widely with the details of the implementation being measured, as well as with hardware and operating system architecture.

There are, however, two significantly different types of instrumentation: external and internal. External instrumentation does not require modification of the system being instrumented. As such, it is generally easier to implement, but often cannot provide the desired data. Internal instrumentation, consisting of modifications to the implementation itself, requires considerable knowledge of the system being instrumented. Internal instrumentation is, of course, much easier to include as part of the original language implementation than after the implementation is complete. In fact, consideration of the desired characteristics of internal instrumentation may allow instrumentation to be accommodated easily in the early stages of language implementation, while such instrumentation may be impractical if provisions are not made for it in advance.

The type of instrumentation depends on the type of measurement. Periodic sampling can generally be done primarily by external instrumentation, while event monitoring usually requires internal instrumentation. Most instrumentations have both internal and external components, although the former may be minor. For example, periodic sampling may, in itself, only require an external interrupt routine. Interpretation of the data, especially charge back, may require some internal modification. For example, relating the value of the program counter to the appropriate program activity may require insertion of entry points not present in the original implementation.

E. Measurement Artifact

The artifact of measurement—additional computational resources that are needed for the measurement process—is important. For measurement to be useful, its artifact must be tolerable. Execution-time artifact can generally be kept within acceptable ranges. Some forms of measurement can be accomplished with less than a 10 percent increase in program running time, although 30 percent is more typical. Memory space artifact similarly is usually in manageable ranges, unless the system being measured is already using most of the available memory. Artifacts of 5-20 percent are typical, depending on the instrumentation.

External storage requirements often pose the most serious problem, especially when "historical" measurement data are needed. If the measurement facility tends toward the "complete," it may be necessary to write dozens or even hundreds of words for each sample or event measured, which can quickly get out of hand.

Behavioral artifact also deserves consideration. In some systems, notably those with dynamic storage management systems, performance may be sensitive to the environment. Measurement systems may, unintentionally, affect the system they measure. For example, a measurement tool that uses significant amounts of memory may produce misleading results for a system that relies on dynamic memory allocation. This problem may be subtle and it usually requires analytical treatment and a clear understanding of the implementation and its sensitivity to, for example, the amount of memory available to it.

F. Presentation of Measurement Results

Perhaps the most challenging aspect of the design and implementation of measurement facilities is the presentation of the results in a meaningful and useful manner. Common measurement tools simply present a histogram of program location counters, displayed against a program load map [12]. While such displays may be of some use in lower level languages, such as Fortran, they are essentially useless in a higher level language such as Icon.

Selection is a basic difficulty in presentation. Most measurement facilities are capable of generating enormous amounts of data. Those that generate historical data are prone to this problem. Programmers, however, need measurement data that are related to the program in a meaningful way. Moreover, in order to gain insight and isolate problems, the relevant aspects of the data must be sharply focused.

It is typical to use postprocessing programs to display measurement data. This technique provides flexibility—a particularly important commodity—that is not feasible to incorporate in the instrumentation of a processor itself. Postprocessing artifact cannot be ignored. A large amount of measurement data combined with complex postprocessing can easily take longer to process than it takes to collect the data in the first place. It is not only the cost of this processing that is important—the user of the program may be discouraged from using a measurement tool that significantly adds to the time and cost of program development.

IV. The Instrumentation and Measurement of Icon

The design of measurement tools for Icon presented an interesting problem. Some unusual aspects of the language, especially generators and goal-directed evaluation, raised questions about the type of measurement data that would be useful. Furthermore, there was little experience with some of the techniques used in the implementation. As a result, several measurement tools were developed. The more traditional tools, such as sampling, are not described here.

A. Choice of Measurement Tools

One problem is relating measurement data to the syntax of Icon. In SNOBOL4, statements provide clearly delimited syntactic units that are also natural semantic units. In Icon, expressions may be nested to an arbitrary depth; unlike SNOBOL4, there is no fixed hierarchy of program structure except the procedure. Therefore, it was decided to associate measurement data with elementary "tokens"—literals, identifiers, function calls, operators, structure references, and so on. An example is given by the following program segment, in which the beginning of each token is identified by an arrow:

```
sum := sum + 1
↑   ↑  ↑   ↑↑
line := process (read (f))
↑    ↑  ↑      ↑↑     ↑
count[n] = 0
↑    ↑↑ ↑↑
```

A procedure call, such as process (x), involves both runtime access of the procedure name and its invocation. Hence, there is a token both for the procedure name and for the left parenthesis. For the call of a built-in function, like read (f), there is only a single token.

The usefulness of tallying program activities has long been recognized [12]. This method, which has a low artifact, also provides detail—an important issue with respect to the behavior of generators and goal-directed evaluation. The two most important kinds of data that are obtained for the tokens in a program are

1) activity—tallying each activation of a token,

2) allocation—keeping track of the amount of storage allocated by each token (not all types of tokens cause allocation).

The results of such measurements are simply totals: how many times each token is activated or how many words of storage are allocated by each token.

The major advantages of this scheme are that measurement data can be kept in memory during execution and written out at program termination, rather than continually writing during execution. Furthermore, the amount of data is small compared to that required for historical records. The artifact is thus reduced in all ways: running time, disk storage, and postprocessing. The penalty, of course, is that less information is obtained by tallying than for historical records. The data are continually integrated, and the instantaneous details of program behavior are lost. The use of tallying constituted, in part, an experiment to determine if that technique was sufficiently useful to compensate for the loss of information. Conclusions concerning the usefulness of tallying are given in Section VII.

B. Instrumentation Techniques

To support the various kinds of tallying, the Icon translator was modified to generate, conditionally, extra code to post token numbers and save and restore them as necessary during operations that change program context. As a byproduct of translation, a file relating token numbers to their position in the source program is produced. This file is used by postprocessing programs that produce displays of measurement data. Counts are kept in internal arrays during execution—one each for token activity and one for allocation. The size of these arrays is proportional to the number of tokens in the program. At the end of program execution, these arrays are written to files that are used by postprocessing programs.

C. Displays of Measurement Data

There are two basic forms of displays: tallies and averages. Tallies simply give the total count for the program tokens—activity or allocation. Averages are more useful for per-token allocation.

Fig. 1 shows a portion of a typical display of tallying token activity. The leftmost digit of each value is aligned under the leftmost character of the token. Values are written on successive lines where there is inadequate space between tokens to place the values on the same line. Counts show the number of times each token is activated. For example, the every loop in Fig. 1 was executed 100 times. A total of 6750 positions of the string "ab" were found in line. The additional 100 activations of find occurred for each of the 100 times "ab" was not found. Note that in goal-directed evaluation, the generator is repeatedly activated without reevaluating the arguments [7].

Program activity gives insight into program behavior such as the number of times a loop is entered. Token activity may also show interesting characteristics of familiar computations. Fig. 2 shows the activity resulting from the computation of Ackermann's function. The call that produced these results was acker (3, 5).

An example of the number of words allocated per token is shown in Fig. 3. This procedure constructs "meandering strings," strings that contain all substrings of a given length from a specified alphabet of characters [4].

```
every i := find("ab", line) do write(i)
100   100 6850 100    100  6750     6750
        6750                   6750
```

Fig. 1. An example of tallying token counts.

```
procedure acker(n,m)
  if n = 0 then return m + 1
  42438 42438 21096 21096
    42438            21096
      42438            21096
  if m = 0 then return acker(n - 1,1)
  21342 21342 247     247 247 247
    21342                247  247
      21342                247
  return acker(n - 1,acker(n,m - 1))
  21095 21095 21095 21095 21095 21095
              21095       21095
                21095        21095
                  21095        21095
end
```

Fig. 2. Computation of Ackermann's function.

```
procedure meander(alpha,n)
  local s, t, i, c, k
  i := k := size(alpha)
  t := 1 - n
       1.00
  s := repl(alpha[1],n - 1)
       2.75          2.00
  while c := alpha[i] do {
                     1.99
    if find(section(s,t) || c, s)
       0.28              2.70
           2.00
    then i := i - 1 else {s := s || c; i := k}
                                 34.8
    }
  return s
end
```

Fig. 3. An example of average allocation.

V. Experience with Use of the Measurement Facilities

The use of periodic sampling to locate "hot spots" is well established [12]. In high-level programming languages, such measurements are less meaningful than they are in lower level languages, since complex processes may be associated with apparently simple language constructions and it may be incorrect to assume that the performance of a program in a high-level language can be improved by concentration of the areas of the program where most of the time is spent. It is often difficult to determine if such "hot spots" are due to inefficient coding, unusual amounts of storage allocation (possibly indicating inappropriate data representation), poor algorithms, or a combination of causes. The following examples, taken from real experiences using the various performance measurement tools described in this paper, illustrate the range of possibilities.

A. Automatic Type Coercion

A potential inefficiency in Icon relates to automatic type coercion. For the Icon programmer, not having to worry about whether a value is a string or a character set, for example, is a convenience—but a potentially expensive one. If a value happens to be a character set that is used in an operation requiring a string, the coercion is performed automatically and the program works the same way it would if the value were a string. The cost may be high, indeed, if the coercion is performed in an inner loop. For each iteration of the loop, there is both the cost of the coercion itself and the space allocated for the string. Periodic sampling may show the loop to be time-consuming, but not show the cause (if the operation is a high-level one, the programmer may assume that the operation itself is consuming the time). Measurement of average storage allocation, however, makes the problem clear.

```
while line := read() do
                 9.71
  write(lfile,map(line,ucase,lcase))
              9.75     8.50  8.50
```

Fig. 4. Average storage allocation showing type conversion.

Consider Fig. 4, in which the line of code in the inner loop copies an input file to an output file, converting uppercase letters to lowercase letters. Here *lcase* and *ucase* are character sets containing the upper and lowercase letters, respectively. The function *map*, however, requires its arguments to be strings. The allocation measurement shows a clear problem, since there is allocation associated with the arguments as a result of conversion of characters sets to strings. This section of code runs about one-third faster if *lcase* and *ucase* are coerced to strings outside the loop, eliminating the allocation resulting from coercion within the loop.

Examples such as this have led to the development of coding caveats for Icon programmers. Such examples also suggest possible heuristics for the implementation and even potential changes in the design of the language.

B. Ordering Program Components

In some programs, the order in which tests are made or in which processing is done is optional, but the choice may affect program performance. The best order may be impossible to determine analytically if it depends on data. Here token counting proves useful.

An example occurred in a typesetting program in which formatting codes in the input document select processing functions through a very large case expression. In Icon, case selectors are examined linearly. Originally, the case selectors were arranged alphabetically (a logical choice and one that is useful for program development). Unfortunately, alphabetical order was far from optimal in terms of case selection and the problem was obvious, even if the optimal order was not.

One method of obtaining a more nearly optimal order is to analyze existing documents. This requires writing an auxiliary program to do the analysis, and in some situations may be impractical since representative documents may not be accessible to the author of the typsetting program. Tallying, however, provides the data as the program is actually used and shows unexpectedly frequent use of some formatting codes. A reordering of the case selectors to reflect this empirical data resulted in an 8 percent improvement in overall program performance, and considerably more in some situations.

C. Algorithm Design

Whether an algorithm is "good" or "bad" often depends on the data it processes. This was shown dramatically in a program for determining transitive closure of a graph. In this program, graph nodes are represented by single characters and arcs by character pairs. For example, *AB* represents an arc from node *A* to node *B*. A graph is then represented by a list of two items, one consisting of the nodes and the other consisting of a string of its arcs. One frequently used procedure, successors(n, g) determines the set of successors of a set of nodes *n* in graph *g*. There are basically two alternative approaches to computing the required set:

1) examine every odd position to determine if it is a member of the set *n*,

2) examine every position in which a member of *n* occurs to determine if it is odd.

Method 1 was the one used when the program was initially written. On examination of token activity, it was immediately obvious that this was the wrong choice. When method 2 was selected, the entire program ran two to five times faster, depending on the graph! Figs. 5 and 6 clearly show why.

It might be argued that method 2 should have been used in the first place. While that may be true, it is nonetheless a fact that the program was written using method 1 and the source of inefficiency was discovered by examining token activity. Furthermore, information from token sampling, as opposed to token activity, is open to different interpretations (such as possible disparities in timings for different operations). It is also interesting to note that the most efficient choice actually depends on the data: for graphs that are very dense, method 1 is more efficient; for sparse graphs, method 2 is more efficient.

```
procedure successors(n,g)
  local i, arcs, t
  arcs := g[2]
  67     67 67
            67
             67
  n := cset(n)
  67   67    67
    67
  every i := 1 to size(arcs) by 2 do
  67       67  67   67    67            67
             11792                    11792
                 11859
    if upto(n,arcs[i]) then t := t || arcs[i+1]
    11792     11792  11792      171  171 171  171
      11792     11792     171     171  171      171
                      11792                     171
                                                  171
  return cset(t)
  67      67   67
end
```

Fig. 5. Token counts for method 1.

```
procedure successors(n,g)
  local i, arcs, t
  arcs := g[2]
  67     67 67
            67
             67
  n := cset(n)
  67   67    67
    67
  every i := upto(n,arcs) do
  67       67  1190  67      1123
            1123       67
    if mod(i,2) = 1 then t := t || arcs[i+1]
    1123      1123 1123      171  171  171  171
      1123     1123  1123     171  171      171
                   171                      171
                                              171
  return cset(t)
  67      67   67
end
```

Fig. 6. Token counts for method 2.

D. Behavior of Generators

As indicated earlier, generators and goal-directed evaluation are important components of Icon. In particular, they allow concise and natural formulations for many kinds of computations. For example,

find$(s1, s2) >$ find$(s3, s4)$

succeeds provided that *s*1 occurs as a substring of *s*2 at a position that is greater than an occurrence of *s*3 as a substring of *s*4. With generators and goal-directed evaluation, this computation occurs automatically; without them, the programmer must provide loops and auxiliary identifiers.

In such expressions, the goal-directed evaluation mechanism of Icon "searches" for a solution and the programmer need not be concerned with the details. Such searches, however, may involve significant computation and the combinatorial possibilities in complex expressions are a potential cause for concern. The same situation exists in SNOBOL4, where a backtracking pattern-matching routine may perform time-consuming searches in apparently innocuous situations [6]. The greatest sources of inefficiency occur when a search is conducted in an inappropriate way or unnecessarily.

The activity of tokens shows such situations clearly, since each time a generator is activated to produce a value, that activation is tallied. An example is given in Fig. 7, which shows the results of measuring a loop that counts all the occurrences of the character *t* between positions 10 and 20 in a text file. This measurement shows an excessively large number of activations of the

10 to 20

expression. The problem with this naive use of generators is that the position of every instance of *t* is compared with each of the values from 10 to 20, instead of using a range check. Furthermore, the comparison continues even after the positions become greater than 20.

```
count := 0
1       1  1
while line := read() do
1       301  300        300
             301
  every find("t",line) = (10 to 20) do
  300     722  300 300   4642   5064    132
                            422    422
     count := count + 1
     132     132       132
                132       132
```

Fig. 7. A use of generators.

```
count := 0
1       1  1
while line := read() do
1       301  300        300
             301
  every i := find("t",line) do {
  300     300  529  300  300   349
          349
    if i < 10 then next
    349 349         97
          349
    if i > 20 then break
    252 252         120
          252
    count := count + 1
    132     132       132
               132       132
```

Fig. 8. An alternative approach.

The results of a more reasonable coding technique are shown in Fig. 8. This approach is about twice as fast as the former one for typical data files.

Thus, tallies of token activity may show not only "hot spots" due to the inappropriate use of generators, but also indicate the causes of the unnecessary computation.

VI. Instrumentation of Storage Management in Icon

The management of storage in high-level programming languages presents many problems. It is particularly important to be able to measure particular storage management techniques in the environment of the running program [13]. Earlier work on the measurement of the storage management system of an implementation of SNOBOL4 [17] indicated that performance analysis could give new insights and suggest improvements, even to well-established systems. In particular, it was discovered that some heuristics, which appeared to be sound in the abstract, either did little to improve performance or actually degraded it. Similarly, measurement suggested new heuristics that produced significant performance improvements. That work produced the following recommendations.

1) A basically simple strategy for storage management should be chosen for the initial implementation.

2) A measurement facility 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.

A. Measurement of Storage Management

Icon provided an ideal opportunity to test these recommendations. The Icon storage management system must support the allocation of many kinds of objects for a language with which experience was lacking. The storage requirements of Icon were sufficiently different from those of SNOBOL4 that details of earlier work were not directly applicable. Finally, the implementation of storage management in Icon could be modified easily if the results of measurement suggested changes.

There were two specific *a priori* concerns about storage management in Icon. One was the issue of allocating integers. Integers are also allocated in the MACRO SPITBOL implementation of SNOBOL4 [2]. While the allocation of integers appears to have no significant impact on the overall performance of MACRO SPITBOL, there is no quantitative data to verify this [19].

Another issue was "thrashing," which may occur when the available space in a storage region is small compared to the amount needed. In this situation, an allocation request may result in a regeneration of storage with very little excess space being recovered beyond the amount that was requested. As a result, storage regenerations may occur very frequently.

Following the recommendations given above, the storage management system made no *a priori* provision for handling these two issues. Rather, instrumentation was added and measurements were performed.

This instrumentation simply accumulates, in memory, the following information for each storage region:

1) the number of allocation requests,

2) the number of elements allocated (the number of words per element is machine dependent and varies from region to region),

3) the number of storage regenerations,

4) the time, in milliseconds, required for storage regeneration,

5) the number of times a region must be expanded,

6) the time, in milliseconds, required for expansion,

7) the final size of each region.

The accumulated information is printed when program execution is completed. Fig. 9 shows a typical summary of storage management activity. The CPU time is the total time required for program execution, which may, for example, be compared with the time required for storage regenerations. Fewer integers are allocated than requested, since some frequently used integers are preallocated (see Section VI-B).

The time required for allocation is not given, since it is so small that the measurement artifact would be unacceptably large. Allocation time, however, can be computed from analysis of the code in the allocation routines and the number and amount of allocation shown in the summary. Regeneration time, however, depends very much on the history of program execution and the configuration of memory when regeneration occurs and is not amenable to analytic approaches.

The summary in Fig. 9 is for a program that does a great deal of string processing, but in which most of the data is of a transient nature. As indicated, storage regeneration reclaims space for continued processing without the need for expanding the storage regions. To illustrate how much storage management may vary from program to program, storage activity for the computation of Ackermann's function is shown in Fig. 10.

B. Results

Measurement confirmed that while integer allocation is not a major source of inefficiency in most programs, significant

```
CPU time:  396600 ms

                    String     Qual.     Int.      Heap      Total
Allocations         60008      60011     10002     21
Elements alloc.     2160222    60011     9901      653
Regenerations       3076       3076      102       0
Elements recov.     2159352    59982     9894      0
Regen. time         16559      11637     1126      0         29322
Expansions          0          0         0         0
Expan. time         0          0         0         0         0
Final size          999        200       200       653
```

Fig. 9. Summary of storage management activity.

```
CPU time: 42906 ms

                    String     Qual.     Int.      Heap      Total
Allocations         5          8         63537     16
Elements alloc.     139        8         11954     609
Regenerations       0          0         73        0
Elements recov.     0          0         11649     0
Regen. time         0          0         2041      0         2041
Expansions          0          0         1         0
Expan. time         0          0         56        0         56
Final size          999        200       416       609
```

Fig. 10. Summary of storage management for Ackermann's function.

improvement could be made for some programs by special casing commonly occurring integers and preallocating them. The effects of preallocation are reflected in Figs. 9 and 10. This heuristic helps to reduce the number of integer allocations for loop indexes, for example. The improvement ranged from 5 percent for most programs to 30 percent for programs that allocated many transient integers, such as the program in Fig. 10. Measurement also guided the implementation of this heuristic; beyond a certain value, permanent allocation of integers did not result in significant improvement. At present, the integers 0-100 are preallocated.

The most dramatic improvement was obtained by adding a dynamic "breathing room" heuristic [8]. This heuristic allows storage areas to adapt their expansion requirements to the demands that are experienced by the running program. By performing measurements and experimenting with heuristics, an average improvement of over 50 percent in the *overall* running speed of Icon programs was obtained, and some programs run five times faster than before. (Figs. 9 and 10 show times after the heuristic was added.) While some improvement was expected, the magnitude of the effect was a surprise.

VII. Conclusions

Few of the instrumentation techniques used in Icon are novel. It is unlikely, however, that the instrumentation for tallying could have been incorporated if it had not been anticipated in the early stages of the implementation. This illustrates the value of incorporating measurement facilities as part of the implementation design. A measurement system added onto an existing implementation of Icon would probably have had quite different characteristics, dictated by the problems of modifying a completed implementation.

A. Usefulness of Measurement Tools to Programmers

Experience has shown that performance measurement facilities for high-level programming languages can be useful in aiding the programmer to improve the efficiency of programs, to locate errors, and to understand program behavior. Information gained from studying measurement data may lead to better programming techniques in general, especially in the use of language features that have no correspondence in conventional machine architecture. Algorithmic inefficiencies, especially in cases where performance depends on data, can also be detected by use of performance measurement. Similarly, data representation can be improved by experiments in situations where analysis is intractable.

Tallying proved to be remarkably useful. The low artifact makes the tools easy and economical to use and the information obtained is adequate for most purposes. While historical records have the inherent appeal of providing detail and also for showing the way performance and behavior may change during the time a program executes, experience with the use of the tools described here, compared with those developed earlier for SNOBOL4 [16], [17], suggests that simplicity and economy are often more desirable than completeness and detail.

Measurement of program activity at the token level proved unexpectedly useful, both in locating performance problems and in understanding program behavior. Much of the usefulness of activity measurement stems from its exact nature. While timings may have numerous interpretations, tallies of activations do not. Furthermore, program activity, reflected in such tallying, whether at the token level or some other, is relevant in almost all programming languages and should be given greater attention.

A relatively unexplored area in Icon is the measurement of token activity to illuminate the processes that go on during goal-directed evaluation. The combinatorial aspects of generators deserve study, especially as they relate to the relative efficiency or inefficiency of searches. Icon allows concise programming techniques using goal-directed evaluation without exposing potential combinatorial problems that would be self-evident in more traditional loop-oriented paradigms. On the other hand, goal-directed evaluation often allows more efficient computation by internalizing loops. A simple illustration is given by the activity shown in Fig. 1.

Despite the success of the measurement tools, there are limitations to the usefulness of performance measurement in Icon as well as in other high-level programming languages. The most fundamental problem is the inherent conflict between measurement and the motivation for high-level languages. In a lower level language such as Fortran, a programmer can readily relate measurement data to the program and see direct ways of making improvements. One of the motivations for high-level languages, however, is to get closer to the problem domain and farther from the constraints of conventional computer architecture. Program constructs are phrased in terms that the programmer can relate to the problem to be solved and not in terms of machine instructions. As a result, measurement data related to the machine on which a high-level program is run may be essentially meaningless to the programmer. On the other hand, if the measurement data are related to the high-level constructs, it is hard for the programmer to detect inefficiencies or to see how to correct them.

In fact, the user of a high-level language may need to have an expert understanding of its implementation in order to use measurement data to its best advantage. This, however, is in conflict with the motivation for high-level languages—that the programmer not have to know about what is going on, but rather may concentrate on the problem domain and the concepts appropriate to it. This conflict appears fundamental and unreconcilable.

B. Usefulness of Measurement Tools to Implementors

The implementors of high-level programming languages may be able to make better use of such facilities than the programmers that use the languages. In a number of instances program measurement has highlighted an implementation problem—either a bug or an inefficiency. Whether or not it should be the case, it is clear that implementors of high-level languages rely on conventional wisdom, experience (perhaps imperfectly verified), and on intuition, especially in the design of systems to support high-level processes such as automatic storage management. Instrumentation and measurement reveals the actual situation and replaces conjecture by fact. There appears to be no better method to dispel myths in this complex and difficult area.

Experience with Icon highlights the importance of incorporating measurement facilities in the initial design of an implementation rather than waiting until the implementation is complete. It is usually very difficult to add measurement facilities to a completed implementation. Even if they can be added, it may be necessary to make compromises that would not have been necessary if they had been considered in the design. Since a major benefit of performance measurement of high-level languages appears to be in improving the quality of implementations, measurement tools and their instrumentation should be an integral part of the initial design and should be used while there is still time to modify the implementation.

Acknowledgment

J. Davidson, T. Korb, and S. Wampler all made contributions to the work described in this paper. The suggestions of the referees were also very helpful.

References

[1] D. F. Brailsford, E. Foxley, K. C. Marder, and D. J. Morgan, "Runtime profiling of Algol 68-R programs using DIDYMUS and SCAMP," *SIGPLAN Notices*, vol. 12, pp. 27-33, June 1977.
[2] R. B. K. Dewar and A. P. McCann, "MACRO SPITBOL—A SNOBOL4 compiler," *Software–Practice and Experience*, vol. 7, pp. 95-113, Jan. 1977.
[3] E. Foxley and D. J. Morgan, "Monitoring the runtime activity of Algol 68-R programs," *Software–Practice and Experience*, vol. 8, pp. 29-34, Jan. 1978.
[4] J. F. Gimpel and W. Keister, "Minimal meandering strings," Bell Labs., Holmdel, NJ, Tech. Rep., July 1970.
[5] R. E. Griswold, D. R. Hanson, and J. T. Korb, "The Icon programming language: An overview," *SIGPLAN Notices*, vol. 14, pp. 18-31, Apr. 1979.
[6] R. E. Griswold and D. R. Hanson, "An alternative to the use of patterns in string processing," *ACM Trans. Programming Languages and Systems*, vol. 2, pp. 153-172, Apr. 1980.
[7] R. E. Griswold, D. R. Hanson, and J. T. Korb, "Generators in Icon," *ACM Trans. Programming Languages and Systems*, vol. 3, pp. 144-161, Apr. 1981.
[8] D. R. Hanson, "A portable storage management system for the Icon programming language," *Software–Practice and Experience*, vol. 10, pp. 489-500, June 1980.
[9] D. Ingalls, "The execution time profile as a measurement tool," in *Design and Optimization of Compilers*, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 107-128.
[10] S. Jasik, "Monitoring program execution on the CDC 6000 series machines," in *Design and Optimization of Compilers*, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 129-136.
[11] B. W. Kernighan and P. L. Plauger, *Software Tools*. Reading, MA: Addison-Wesley, 1976.
[12] D. E. Knuth, "An empirical study of Fortran programs," *Software–Practice and Experience*, vol. 1, pp. 105-133, Apr. 1971.
[13] B. W. Leverett and P. G. Hibbard, "An adaptive system for dynamic storage allocation," *Software–Practice and Experience*, vol. 12, pp. 543-556, June 1982.
[14] G. Lyon and R. B. Stillman, "Simple transforms for instrumenting Fortran programs," *Software–Practice and Experience*, vol. 5, pp. 347-358, Oct. 1975.
[15] S. Matwin and M. Missala, "A simple, machine independent tool for obtaining rough measures of Pascal programs," *SIGPLAN Notices*, vol. 11, pp. 42-45, Aug. 1976.
[16] G. D. Ripley, "Program perspectives: A relational representation of measurement data," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 296-300, July 1977.
[17] G. D. Ripley, R. E. Griswold, and D. R. Hanson, "Performance measurement of storage management in an implementation of SNOBOL4," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 130-137, Mar. 1978.
[18] E. Satterthwaite, "Debugging tools for high level languages," *Software–Practice and Experience*, vol. 2, pp. 197-217, July 1972.
[19] D. Shields, private communication, 1979.
[20] R. L. Sites, "Programming tools: Statement counts and procedure timings," *SIGPLAN Notices*, vol. 13, pp. 98-101, Dec. 1978.
[21] W. M. Waite, "A sampling monitor for applications programs," *Software–Practice and Experience*, vol. 3, pp. 75-79, Jan. 1973.
[22] W. A. Wulf, D. B. Russell, and A. N. Habermann, "BLISS: A language for systems programming," *Comm. ACM*, vol. 14, 780-790, Dec. 1971.

Cary A. Coutant was born in Chicago, IL, in 1956. He received the B.S. degree in physics in 1977 from Furman University, Greenville, SC, and the M.S. degree in computer science in 1979 from the University of Arizona, Tucson.

He is presently a Development Engineer in the Information Systems Laboratory, Hewlett-Packard, Cupertino, CA. His areas of interest are programming languages, operating systems, and document preparation systems.

Mr. Coutant is a member of the Association for Computing Machinery.

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 in the Department of Computer Science, University of Arizona, Tucson. His research interests include programming language design and implementation, nonnumeric computing, programming methodology, and software engineering.

Dr. Griswold is a member of the IEEE 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 from Oregon State University, Corvallis, and the M.S. degree in optical sciences in 1972, and the Ph.D. 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 the 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 Associate Professor of Computer Science and Head of the Department of Computer Science at University of Arizona, Tucson. He is also an Editor of *Software–Practice and Experience* and a consultant for several industrial laboratories. His areas of interest include programming language design and implementation, software engineering, document preparation systems, and operating systems.

Dr. Hanson is a member of the Association for Computing Machinery and the IEEE Computer Society.