Measuring the Performance and Behavior of the Icon Programming Language*

Cary A. Coutant, Ralph E. Griswold, and David R. Hanson

TR 80-20

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.

August 1980

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants MCS75-21757 and MCS79-03890.

Measuring the Performance and Behavior of the Icon Programming Language

1. Introduction

The importance of the ability to measure the performance of programs written in high-level languages is well known [knu71]. 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 [rip77]. 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, lack of appropriate charge-back information and excessive overhead.

Earlier work with the SNOBOL4 programming language [rip78] showed that measurement tools as expected can aid the programmer of a high level programming language in improving program performance. They also can give insight into the behavior of complex language 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 [gri79, gri80a]. Since the measurement tools in Icon were designed and implemented in conjunction with the implementation of the language, they provided and 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 of a conventional nature and is not described here. An example is sampling with charge back to runtime routines. Such information is fairly easy to obtain 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 (1) measurement tools that give the user information on program behavior and performance directly related to the program text, and (2) instrumentation of the storage management system to aid in the development and improvement of its implementation.

The next section gives an overview of Icon and its implementation. Section 3 reviews the major issues in the design and implementation of performance measurement tools for programming languages in general. In Section 4, 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 5. Analysis of the storage management system in Icon and its use to improve the implementation are the subjects of Section 6 In Section 7 results are summarized and conclusions drawn

2. 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 aspects of the Icon language and its implementation that are relevant to these concerns are described in the following sections

2.1 Characteristics of the Icon Programming Language

Syntactically, Icon is an expression-based language in the style of Algol 68 and Bliss [wul71] and has most of the traditional control structures The syntax of a programming language has important effects both in the selection of data to be measured and in methods of presenting this data to the user There is a substantial difference in the approaches that can be taken between a language whose syntax is hierarchical such as SNOBOL4 [rip77b], 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

(1) automatic, dynamic storage management,

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

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

(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 [gri80c]

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 yield 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 most basic generators in Icon is alternation, 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 Alternation is a binary operation, e1 | e2 An example of 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 y, otherwise it assigns the value of u to x

Another Icon 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 if 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 s2 (failing if there is no such position). Furthermore, it generates additional positions, as needed, from left to right, in case s1 occurs as a substring of s2 in more than one place. Thus

```
if find(s1,s2) > 10 then z := 10
```

assigns ten to z if s1 occurs as a substring of s2 at a position greater that ten.

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

```
every e1 do e2
```

For each value produced by e1, e2 is evaluated. For example

```
every i := find(s1,s2) do write(i)
```

prints all the positions at which s1 occurs as a substring of s2.

A knowledge of all the 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 [gri79, gri80a, gri80c].

2.2 The Implementation of Icon

The Icon system is written in Ratfor [ker76, gri80b] and is designed to be portable. It has been implemented on a number of computers including the DEC-10, Cyber 175, IBM 370, and VAX-11/780. Its instrumentation, as part of the portable system is, itself, 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 [han80a]. The translator converts an Icon program into 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 [dew77], 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. In fact, measurements of Icon programs indicate that typically less than 5% of program execution time is spent in the translated program itself. (This figure, obtained by traditional measurement methods, indicates that little improvement in program performance can be expected from optimization of the translator output.)

A major component of the runtime system consists of storage management routines [han80b]. Because of the importance of storage management in Icon and its affect 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 is allocated, corresponding to the nature of the data: integer, string qualifier, and heap. Integers are allocated because the Fortran virtual machine model used by Icon 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.

3. Technical Problems in the Design of Measurement Facilities

In general, there are a number of problems related to the design and implementation of programming language measurement facilities:

(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, and

(6) presentation of the results.

3.1 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 where the interrupt occurred. If enough samples are taken, the measurement data resembles randomly selected data and, on the average, gives an approximation to the actual behavior of the program. For example, average times for the execution of an operation may 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 [ing72, jas72].

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 get enough samples to give a meaningful picture of program activity. Furthermore for fast CPUs such long periods of program execution may be inordinately expensive or may require artificial techniques (such as multiple runs) to get 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, there may be effects that violate the underlying assumptions on which the interpretation of measurement data are based. For example, on the DEC-10 running under TOPS-10, the clock generally available for measurement is also used by the system for scheduling. Under heavy load conditions, the two uses tend to interfere, resulting in "drift". For example a program running under heavy load conditions may be timed as taking as much as 20% longer than when run under light load conditions. Such problems not only reduce the accuracy of and confidence in sampling measurements but may make comparisons between runs virtually useless. This problem is strongly dependent on hardware and operating system characteristics.

The other commonly used method of measurement is event monitoring. In this technique, selected program activities trigger measurement routines. For example, a garbage collection might 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.

3.2 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. At one extreme, specific aspects of program activity can be selected for measurement. For example storage allocation might be measured as an item of particular interest. At the other extreme a measurement facility might attempt to provide data on all aspects of program behavior. While there are clearly limitations to this approach, an approximation to it is appealing as a starting point, especially with the measurement of high-level languages, since *a priori* assumptions about the most important aspects of program behavior are suspect, given the present state of knowledge

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 On the other hand, measurement may 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 utilization of runtime routines may be of interest

Determination must be made as to what data are to be produced when program activity is sampled or when an event occurs 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, typically a disk file At the other extreme there are summaries, such as the total counts for particular activities Such data usually can be kept in memory

3.3 Charge Back

Charge back is concerned with the attribution of measurement data to components of a program For example the fact that a particular routine is called frequently may not as interesting as are the program activities that cause this routine to be called 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

3.4 Instrumentation

Instrumentation, the means by which measurement data is obtained, 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 is that which does not require modification of the system being instrumented As such it is usually the most tractable External instrumentation, while generally easier to implement often cannot provide the desired data Internal instrumentation, consisting of modifications to the implementation itself usually 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 almost always 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 Similarly, if measurement data is kept in memory rather than being written to a file, some modifications to permit communication of locations may be desirable if not actually essential

3.5 Measurement Artifact

The artifact of measurement—additional computation time, memory space, and external storage required by the measurement process—is an important matter For performance measurement to be useful its artifact must be tolerable

Computational artifact can generally be kept within acceptable ranges Some forms of measurement can be accomplished with less than a 10% increase in program running time, although 30% is more typical

Memory space artifact similarly is usually in manageable ranges, unless the processor is already using most of the memory available Artifacts of 5% to 20% are typical, depending on the nature of the instrumentation

External storage requirements often pose the most serious problem, especially when "historical" measurement data is 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 Such measurement data 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 A measurement system may unintentionally, affect the system it is measuring For example, a measurement tool that uses significant amounts of memory may produce very misleading results for implementations that use dynamic memory allocation within a fixed region This problem may be subtle and it usually requires analytical treatment and clear understanding of the implementation and its sensitivity to, for example, the amount of memory available to it

3.6 Presentation of Measurement Results

Perhaps the most challenging aspect of the design and implementation of performance measurement facilities is the presentation of the results in a manner that is meaningful and useful to the user The commonest measurement tools simply present a histogram of program location counters, displayed against a program load map [knu71] 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

One of the most basic difficulties in presentation is selection Most measurement facilities have the capability of generating enormous amounts of data Systems that generate historical records are the most prone to this problem A programmer, however, needs to relate measurement data to the corresponding program in a meaningful way Moreover, in order to gain insight and isolate problem areas, the relevant aspects of the measurement data must be sharply focused and properly related to the source program

It is typical for the display of measurement data to be produced by post-processing programs This provides for flexibility—a particularly important commodity—that is not feasible to incorporate in the instrumentation of a processor itself

Post-processing artifact cannot be ignored If the amount of measurement data is large and post processing is complex the processing time easily may exceed the time required to obtain 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 program development effort

4. The Instrumentation and Measurement of Icon

The novel aspects of Icon and lack of experience with some techniques used in its implementation motivated extensive instrumentation to provide a variety of measurement tools

4.1 Choice of Measurement Tools

One problem immediately encountered was relating measurement data to the syntax of Icon In SNOBOL4, statements provide clearly delimited syntactic units that are also a natural semantic units In Icon expressions may be nested to arbitrary depths and, unlike SNOBOL4, there is no fixed hierarchy of program structure except the procedure

The decision was made, therefore, to associate measurement with elementary "tokens"—literals identifiers function calls, operators, structure references, and so on [cou79a, cou79b] For example the following expressions consist of tokens beginning at the places marked by arrows

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

The call of a programmer-defined procedure, 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 built-in functions, like read(f), there is only a single token

In order to reduce the measurement artifact, most of measurement tools for Icon were designed around tallying rather than the production of historical records Three kinds of data are obtained for the tokens in a program

(1) Activity—counting each time a token is evaluated,

(2) Sampling—noting at periodic intervals which token is currently being evaluated, and

(3) 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 evaluated, how often each token is sampled (hence an approximation to the time spent evaluating each token), or how many words of storage are allocated by each token

The major advantages of this scheme are that (1) measurement data can be kept in memory during execution and written out at program termination, rather than continually writing during program execution and (2) 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 post processing The penalty, of course, is that less information is obtained by counting than with historical records The data is continually integrated, and the details of program behavior over the course of program execution are lost The choice of counting constituted an experiment to determine if that technique would prove sufficiently useful in practice to compensate for the loss of information Conclusions concerning the usefulness of tallying are given in Section 7

4.2 Instrumentation Techniques

To support the various kinds of counting, 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 [cou79a] As a byproduct of translation, a file relating token numbers to their position in the source program is produced This file is used by post-processing programs that produce displays of measurement data Counts are kept in internal arrays during execution—one each for token activity, token sampling and allocation The sizes of these arrays are 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 post processing programs

Storage management has a significant effect on sampling Most of the storage management time is usually associated with reclamation (garbage collection), not allocation On the other hand, reclamation may be caused by any allocation request, regardless of the amount of storage required This tends to distort time distributions, since a token that triggers reclamation may be charged for the time needed to reclaim the space allocated by many other tokens To compensate for this effect, samples that occur during reclamation are not charged directly to any token, but rather are distributed to all tokens that cause allocation in proportion to the amount of storage they allocate This technique gives only a first approximation to accurate charge back, since storage management is a complex process It is, however, generally within the accuracy that is obtainable with low-frequency sampling

4.3 Displays of Measurement Data

Measurement data is formatted by post processing programs to put it in a form that can be easily analyzed by the user There are two basic forms of displays counts and averages

Counts simply give the total count for the program tokens—activity, sampling, or allocation Averages are more useful for samples and the number of words allocated per token

Figure 1 shows a portion of a typical output from counting the activity of tokens Note that 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

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

Figure 1 An Example of Token Counting

Counts show the number of times each token is activated For example, the every loop in the example above 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 no substring was found Note that in goal-directed evaluation, the generator is repeatedly activated without re-evaluating the arguments

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 Figure 2 shows the activity resulting from the computation of Ackerman's function The call that produced these results was acker(3,5)

```
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
```

Figure 2 Computation of Ackerman's Function

An example of the number of words allocated per token is shown in Figure 3 This procedure constructs "meandering strings", strings that contain all substrings of a given length from a specified alphabet of characters [cou80, gim70]

For sampling, the average values are adjusted to correspond to milliseconds of residency An example using the substring location code given above, is shown in Figure 4

5. Experience with Use of the Measurement Facilities

The use of periodic sampling to locate "hot spots" is well established [knu71] 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 is often misleading to assume that the performance of a program in a high-level language may 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

```
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- else {s  = s || c, i  = k}
                        34 8
  }
  return s
end
```

Figure 3   An Example of Average Allocation

```
every i  = find("ab",line) do write(i)
16 5   0 03               0 12     0 06
         0 76                0 20
```

Figure 4   An Example of Average Time

5.1 Automatic Type Coercion

One 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, but it is used in an operation that requires 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 itself 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.

Consider the two lines of code in an inner loop shown in Figure 5.

```
write(map(line,lcons,ucons))
      18 5     7 25    7 25
write(map(line,ucons,lcons))
      18 5     7 25    7 25
```

Figure 5   Average Storage Allocation Showing Coercion

Here lcons and ucons are character sets containing the upper- and lower-case consonants, respectively. The function map, however, requires its arguments to be strings. The allocation measurement shows the problem clearly, since there is allocation associated with arguments. This section of code runs nearly four times faster if lcons and ucons are converted to strings outside the loop.

Examples such as this have led to the development of coding caveats [cou80] for Icon programmers. Such examples also suggest possible heuristics for the implementation and even potential changes in the design of the language.

5.2 Ordering Program Components

In some programs, the order in which tests are made or in which processing is done is optional, but it 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 document being processed 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 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 would be to analyze existing documents. This would require writing an auxiliary program to do the analysis, and in some situations might have been unsatisfactory, since the documents might not be accessible to the author of the typesetting program. Token counting, however, provided the data as the program was actually used—and showed unexpectedly frequent use of some formatting codes. A reordering of the case selectors to reflect this empirical data resulted in an 8% improvement in overall program performance.

5.3 Coding Errors in "Correct" Programs

Some kinds of errors in coding may not affect program output, but they may, nonetheless, be sources of inefficiency. An example occurred in the division of the large case expression described in the example above into a number of smaller case expressions to accommodate the limitations of a particular Icon translator. After the modification, the program worked properly. An analysis of token counts, however, immediately pinpointed an error in the way the division into smaller case expressions was done. Instead of exiting a case expression once a clause in it was selected, the program continued on through all the subsequent case expressions. Of course, no clause was selected in these subsequent case expressions and the program was "correct". However, considerable time was spent needlessly searching for subsequent matches.

Correcting the coding mistake was worthwhile—because of the very large number of case selectors, overall program performance improved nearly 10% when the mistake was rectified.

It is worth noting that, since the program produced correct output, the coding error would never have been located due to program malfunction. Neither would periodic sampling have shown the problem. The problem was easily identified because of an *exact* match in token counts at the head of each of the case expressions.

5.4 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 approaches to computing the required set:

(1) Examine every position in which a member of n occurs to determine if it is odd, or

(2) Examine every odd position to determine if it is a member of the set n.

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! Figures 6 and 7 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, while 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
```

Figure 6 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
```

Figure 7 Token Counts for Method 2

6. Instrumentation of Storage Management in Icon

Earlier work on the measurement of the storage management system of an implementation of SNOBOL4 [rip78] 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, and

(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.

6.1 Measurement of Storage Management

Icon provided an ideal opportunity to test these recommendations. The Icon storage management system had to support allocations 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 [dew77]. 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 [shi79].

Another issue was "thrashing", which may occur when the available space in a 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, and

(7) the final size of each region.

The accumulated information is printed when program execution is completed. Figure 8 shows a typical summary of storage management activity.

```
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
```

Figure 8. 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

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 Figure 8 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 Ackerman's function is shown in Figure 9

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	

Figure 9 Summary of Storage Management for Ackerman's Function

6.2 Results

Measurement confirmed that integer allocation is not a major source of inefficiency in most programs (although it does degrade performance in some kinds of programs) Measurement indicated some improvement could be made by special casing commonly occurring integers and effectively pre-allocating them permanently This heuristic helps to reduce the number of integer allocations for loop indices, for example The improvement ranged from 5% for most programs to 30% for programs that allocated many transient integers The implementation of this heuristic was directed by experiment and measurement It was found that beyond a certain point, little performance improvement was obtained and the permanent allocation of additional integers was not justified At the present time, the integers 0 through 100 are pre-allocated

The most dramatic improvement was obtained by adding a dynamic "breathing room" heuristic [han80b] 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% in the *overall* running speed of Icon programs was obtained as a result and some programs run five times faster than before While some improvement was expected, the magnitude of the effect was a surprise

7. Conclusions

Few of the instrumentation techniques used in Icon are novel It is unlikely, however, that the instrumentation for counting could have been incorporated if it had not been anticipated in the early stages of the implementation of Icon This is an example of the value of incorporating measurement facilities as part of the implementation design Phrased another way, 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

7.1 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 has proved remarkably useful The low artifact makes the tools easy and economical to use and the information obtained is adequate for most purposes Historical records have an inherent appeal because they have the potential for providing great detail and also for showing the way performance and behavior may change during the time a program executes However, experience with the use of the tools described here compared with those developed earlier for SNOBOL4 [rip77b, rip78] suggests that simplicity and economy are more desirable, in actual practice, 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 often admit of numerous interpretations, counts of activations do not Furthermore, program activity, reflected in such counting, 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 contains the potential hazard of allowing concise expression through 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 Figure 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 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 user 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

7.2 Usefulness of Measurement Tools to Implementors

The implementors of high-level programming languages may be able to make more direct 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 brings the real situation to light and reduces conjecture to fact There appears to be no better method to dispel myths in this complex and difficult area

Experience with Icon has highlighted the importance of incorporating measurement facilities in the initial design of an implementation, rather than waiting until the implementation is complete. In the first place, 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 design and implementation and should be used while there is still time to modify the implementation.

Acknowledgements

Jack Davidson, Tim Korb, and Steve Wampler all made contributions to the work described in this paper.

References

[cou79a]
C. A. Coutant and R. E. Griswold, *Instrumenting Icon for Performance Measurement*, Tech. Rep. 79-9, Univ. Arizona, Tucson, AZ, May 1979.

[cou79b]
C. A. Coutant and R. E. Griswold, *Tools for the Measurement of Icon Programs*, Tech. Rep. 79-10, Univ. Arizona, Tucson, AZ, May 1979.

[cou80]
C. A. Coutant, R. E. Griswold, and S. B. Wampler, *Reference Manual for the Icon Programming Language, Version 3*, Tech. Rep. 80-2, Univ. Arizona, Tucson, AZ, May 1980.

[dew77]
R. B. K. Dewar and A. P. McCann, "MACRO SPITBOL—a SNOBOL4 Compiler," *Software—Practice and Experience*, vol. 7, pp. 95-113, Jan.-Feb. 1977.

[knu71]
D. E. Knuth, "An empirical study of Fortran programs," *Software—Practice and Experience*, vol. 1, pp. 105-133, Apr.-June 1971.

[gim70]
J. F. Gimpel and W. Keister, *Minimal Meandering Strings*, Tech. Rep., Bell Labs., Holmdel, NJ, July 1970.

[gri79]
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.

[gri80a]
R. E. Griswold and D. R. Hanson, *Reference Manual for the Icon Programming Language*, Tech. Rep. 79-1a, Univ. Arizona, Tucson, AZ, Feb. 1980.

[gri80b]
R. E. Griswold, D. R. Hanson, and S. B. Wampler, *Transporting the Icon Programming Language*, Tech. Rep. 79-2b, Univ. Arizona, Tucson, AZ, Feb. 1980.

[gri80c]
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.

[han80a]
D. R. Hanson, *Icon Implementation Notes*, Tech. Rep. 79-12b, Univ. Arizona, Tucson, AZ, Feb. 1980.

[han80b]
D. R. Hanson, "A portable storage management system for the Icon programming language," *Software—Practice and Experience*, vol. 10, pp. 489-500, June 1980.

[ing72]
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

[jas72]
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

[ker76]
B W Kernighan and P L Plauger, *Software Tools*, Reading, MA Addison-Wesley, 1976

[rip77a]
G D Ripley and G R Owens, *Code Segment Optimization of Fortran Programs*, Tech Rep , Univ Arizona, Tucson, AZ, 1977

[rip77b]
G D Ripley, "Program perspectives a relational representation of measurement data," *IEEE Trans Software Eng* , vol SE-3, pp 296–300, July 1977

[rip78]
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

[shi79]
D Shields, private communication, 1979

[wul71]
W A Wulf, D B Russell, and A N Habermann, "BLISS a language for systems programming" *Comm ACM*, vol 14, 780 790, Dec 1971