Instrumenting Icon for Performance Measurement*

Cary A. Coutant
Ralph E. Griswold

TR 79-9

May 1979

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS75-21757.

Instrumenting Icon for Performance Measurement

1. Introduction

Several features have been added to the DEC-10 implementation of the Icon programming language [1] to allow performance measurement of various aspects of program execution. During execution of an Icon program, several forms of data may be gathered and written to files. Performance is monitored only when it has been requested. Usually, the request is made as an option when the program is translated. Some forms of measurement, however, can be requested at execution time. For details on collecting and interpreting the data, see Reference 2.

Performance measurement data may be collected in two ways: in response to events that occur in the program itself, or as a result of some periodic, external sampling. Events that may cause accumulation of data are evaluation of a portion of a source-language expression, allocation, expansion, or regeneration of any of the four storage regions [3]. Sampling is performed by a special subroutine that receives control from the operating system periodically. The sampling routine detects where the program is currently executing in terms of both source program position and runtime system.

2. Tokens

Execution in terms of the source program is monitored in terms of tokens. A token is the basic syntactic unit of an Icon program: identifiers, operators, and literals are examples of tokens. The Icon translator generates a list of "executable" tokens, tokens that have semantic significance. For example, the assignment operator (":=") causes a value to be stored in a variable; a left bracket ("[") might cause subscripting to occur. These tokens are considered "executing" when the assignment or subscripting operation is being evaluated. An identifier is considered executing while its value is being obtained. A right bracket ("]"), however, has only syntactic significance; no code is executed as a result of this token.

In the example expressions below, carets appear underneath the beginning of executable tokens.

```
count := count + 1
while line := read(f) do
   process(line)
```

Note that the left parenthesis is executed for calls on source language procedures, but not for built-in procedures.

Every executable token is assigned a number, beginning with 1. The token list generated by the translator indicates the line number and column where each numbered token begins in the source program text. This list is written to a file, which is used by several programs that postprocess the measurement data.

3. Event Monitoring

The Icon translator and runtime system have facilities for monitoring two types of events: token execution, and calls on the storage management system.

When token monitoring is requested, the translator produces code that maintains an internal array of counters. Each element of this array corresponds to one of the numbered tokens in the source program. When a token is executed, the counter corresponding to that token is incremented by one. When the program finishes, this array is written to a file for postprocessing.

Storage management events are tallied on request into another internal array. Three different events are monitored: allocations, expansions, and regenerations of each of the four data regions used by the Icon system (string space, string qualifiers, integers, and the heap). In addition to tallying these events, the system also totals, for each region, the number of elements allocated, the number of elements recovered by regeneration, and the time spent during expansion and regeneration. This data is printed in tabular form after the program has finished.

Allocation requests made of the storage management system are also monitored on a per-token basis. An internal array, one element per token, counts the amount of storage allocated on behalf of each token. This array is also written to a file at program termination.

4. Sampling

When token monitoring is requested, the translator generates code that posts the number of the currently executing

token in a known global location. This token number is tallied at each sampling period in an internal array of token samples.

At each invocation of the sampling routine, the current program counter (pc) is also available. The pc contains an address that is either within the generated code or within one of the runtime system modules. Typically, at least 95% of the samples taken are within the runtime system. When pc samples are correlated with a symbol table of the runtime system, each sample indicates which module in the runtime system was active when the sample was taken.

Since periodic sampling is largely random with respect to program execution, a large number of samples produces a reasonable time profile of program execution, both at the source program level (by token samples) and at the implementation level (by pc samples). When the two sampling methods are combined, samples of source-level operations are "charged back" to the runtime system modules that implement the operation. If both token sampling and pc sampling have been requested, the sampling routine writes both the token number and the program counter to the pc sampling file at each sampling period, so that a postprocessor can analyze the charge back.

Samples frequently occur during a regeneration of one of the data regions. Regenerations generally result from the combined effect of many allocation requests by many different tokens, but the time spent in regeneration would normally be charged to the few tokens whose requests trigger the regeneration. Ideally, the regeneration time should be distributed among all tokens which request allocations, according to the total amount of space requesed by each token. As mentioned in Section 3, the total amount of space requested per token is monitored. The storage management system sets a global flag, GCFLAG, during a regeneration, so that samples made during regeneration are charged not to the currently executing token, but to a special token which absorbs all regeneration time. The regeneration time can then be distributed among the proper tokens at termination of the program.

5. Implementation

Icon is implemented as two independent systems, a translator and a runtime library. An Icon source program is translated into a Fortran subroutine consisting largely of calls to runtime library routines. The Fortran subroutine is then compiled and linked with a main program and the runtime library, producing an executable program. The main program performs a system-dependent initialization sequence, calls the generated subroutine, then performs a system-

dependent termination sequence. The main program and the header of the generated Fortran subroutine are shown in Appendix A.

Storage management events are accumulated in two ways. Totals for the number of allocation requests, the number of elements allocated, the number of regenerations, the number of expansions, the number of elements recovered by regeneration, the time (in milliseconds) spent during regeneration, and the time spent during expansion are kept on a per-region basis in individual internal arrays. The size of an element is peculiar to both the region and the implementation. An element refers to the basic unit of allocation for each region: a character (four per word on the DEC-10) in the string region, a qualifier (two words each on the DEC-10) in the string qualifier region, or a Fortran integer (one word each on the DEC-10) in the integer and heap regions.

Allocation requests per token are collected in terms of machine bits in an integer array ALC. For convenience, ALC is treated as if subscripting were zero-based. A total of all allocation requests is kept in ALC(0). At program termination, each element is divided by the number of bits per word (36 on the DEC-10), and the array is written to a file.

Program counter sampling is performed by the module CLOCK in the runtime library. If pc sampling is requested, the initialization sequence calls a routine in the CLOCK module to start sampling. This routine creates a file for the sampling data, and enables the DEC-10 software interrupt system. Once enabled, the operating system interrupts the running program at fixed intervals (1/60th of a second), and calls the sampling routine in the CLOCK module. The value that was in the program counter just before the interrupt occurred is stored by the software in an interrupt block. The sampling routine writes this value to the sampling data file in the right half of one 36-bit word, then returns from the interrupt. The termination sequence calls a routine in the CLOCK module to stop sampling, the interrupt system is disabled, and the sampling data file is closed. A listing of the CLOCK module is given in Appendix B.

If any form of token monitoring is desired, the translator must generate extra Fortran code to support it. For token counting, an integer array T is created local to the generated subroutine, with one element per source program token. For each token, there is a sequence of Fortran statements that executes that token. In front of each such sequence, the translator places the statement

```
T(i) = T(i) + 1
```

where i is the number of the token about to be executed. Thus, each time a token is executed, a counter in the array

T is incremented by one. Prior to returning to the main program, the runtime system routine ZDUMP is called to write the array T to a file.

For token sampling, an integer array TSAMP is allocated in the common block CTOKEN, with one element for absorbing regeneration samples, and one element for each source program token. For convenience, this array is considered to have zero-based subscripting. In the same common block is an integer TN, which always contains the number of the currently executing token. In front of each sequence of code that executes a token, and at each point where that sequence of code might be re-entered, the translator places the statement

```
TN = i
```

where i is the token number. TN must always contain the proper token number, since the time of invocation of the sampling routine is unpredictable and is independent of the flow of execution. The sampling routine used for pc sampling is also used for token sampling: at each interrupt, the equivalent of the Fortran statement

```
TSAMP(TN) = TSAMP(TN) + 1
```

is executed. The value of TN is also written to the pc sampling file in the left half of the output word, to allow token charge back. If GCFLAG is set, however, the equivalent of the Fortran statement

```
TSAMP(0) = TSAMP(0) + 1
```

is executed instead, and zero is written to the pc sampling file in the left half of the output word. Prior to returning to the main program, the runtime system routine ZDUMP is called to distribute the regeneration samples and to write the array TSAMP to a file. The regeneration samples are charged to each token which caused allocation, proportionately to the amount of allocation, by the formula

```
TSAMP(i) = TSAMP(i) + ALC(i) * TSAMP(0) / ALC(0)
```

where i is the token number, TSAMP(0) contains the total number of regeneration samples, and ALC(0) contains the total amount of allocation.

A section of Fortran code from an example program is given in Appendix C.

The token counting, token sampling, and allocation request files are ASCII files consisting of one line per token. Each line contains an integer (string of digits) corresponding to the number of counts, samples, or words

allocated for the token represented by that line.

6. Artifact

Measuring the performance of a program imposes an additional cost on the user over that of running the program with no measurement. This cost is manifested in four ways: additional execution time, larger object program, file space for the measurement data, and cost of postprocessing the data.

Sampling is the major cause for an increase in execution time. A typical program might take up to a 50% longer to execute due to token and program counter sampling. Token counting without sampling causes an increase of about 20%.

Token monitoring code lengthens the resulting Fortran program by about 50% in terms of number of lines, and the corresponding relocatable object file by about 40%.

There are five files that are directly related to performance measurement. The token list, token counting file, token sampling file, and allocation request file each require one line per source program token, and their sizes are independent of the running time of the program. The pc sampling file requires one word for each sample, about 60 words per CPU second that the program runs.

Most of the postprocessors have little to do but format the data. This usually involves using the token list to correlate data files with the original source text. These operations are relatively inexpensive, since the data files are fixed in length. Charge back of source-program tokens to implementation routines, however, is a fairly expensive process. The pc sampling file is generally rather large, since long runs are needed to accurately profile the program.

7. Acknowledgements

Tim Korb first implemented token counting, and Walt Hansen adapted it to the present system. Dave Hanson developed the storage management system, and most of the related monitoring features.

References

1. Griswold, R. E., and D. R. Hanson. Reference Manual for the Icon Programming Language, Technical Report TR 79-1, Department of Computer Science, The University of Arizona, Tucson, January 1979.

2. Griswold, R. E., and C. A. Coutant. Tools for the Measurement of Icon Programs, Technical Report TR 79-10, Department of Computer Science, The University of Arizona, Tucson, April 1979.

3. Hanson, D. R. A Portable Storage Management System for the Icon Programming Language, Technical Report TR 78-16a, Department of Computer Science, The University of Arizona, Tucson, February 1979.

Appendix A -- Main Program and ICON Header

Subroutine IMAIN, written in Ratfor, appears below. IMAIN is effectively the Icon main program, since the actual main program is implementation-dependent, and does nothing except call IMAIN. The version of IMAIN shown below is also implementation-dependent; it has been modified locally to provide for the measurement options.

```
include idef
include adef
define(PCOPT,1)
define(TOPT,2)
define(MOPT,4)
define(GTPRG,3)          # gettab table number for job name
define(THISJOB,-1)       # index for current job in gettab

## l0imain -- Icon main program.
#
subroutine imain
   character arg(60)
   integer i, j, n, junk
   common /ctoken/ tn, tsamp(1)
   integer tn, tsamp
   common /calc/ alc(1)
   integer alc
   integer tock, getarg, ctoi, loc, gettab, sixtoc
   include csizes
   include cparm

   alcoff(1) = loc(alc(1)) - loc(alcoff(1)) + 1
   tnoff(1) = loc(tn) - loc(tnoff(1)) + 1
   jbver(1) = 0137 - loc(jbver(1)) + 1
   gcflag = 0
   if (getarg(1, arg, 60) ~= EOF & arg(1) == MINUS & arg(2) == LPAREN) {
      call delarg(1)
      for (i = 3; arg(i) ~= EOS & arg(i) ~= RPAREN; i = j) {
         j = i + 1
         n = ctoi(arg, j)
         if (arg(i) == LETM)
            jbver(jbver(1)) = jbver(jbver(1)) | MOPT
         else if (arg(i) == LETT)
            jbver(jbver(1)) = jbver(jbver(1)) | TOPT
         else if (arg(i) == LETP)
            jbver(jbver(1)) = jbver(jbver(1)) | PCOPT
         else if (arg(i) == LETS & n > 0)
            strsiz = n
         else if (arg(i) == LETQ & n > 0)
            sqlsiz = n
         else if (arg(i) == LETI & n > 0)
            intsiz = n
         else if (arg(i) == LETH & n > 0)
            hepsiz = n
         else if (arg(i) == LETK & n > 0)
```

```
            stksiz = n
         else if (arg(i) == LETL)
            intlb = n
         else if (arg(i) == LETU)
            intub = n
         }
      }
   call apr
   tn = 1
   tsamp(1) = 0
   junk = sixtoc(gettab(GTPRG, THISJOB), prgnam, NAMSIZ)
   call tick(tn)
   call icon
   call tock
   if (jbver(jbver(1)) >= 4)
      call zpstat
   return
end
```

Subroutine ICON is the generated code from the Icon translator. The parts of the subroutine which contain code generated for all Icon programs are shown below. Code corresponding to each source-program procedure follows the RETURN statement. The statement labelled 2 acts as a distribution point for all source-level transfers of control.

```
      SUBROUTINE ICON
      COMMON/CMAIN/SIGNAL,LABEL,FLABEL
      INTEGER SIGNAL,LABEL,FLABEL
      INTEGER XCMP,XCOMP,XLCMP,XNCMP
      COMMON/CTOKEN/TN,TSAMP(11)
      INTEGER TN,TSAMP
      COMMON/CALC/ALC(11)
      INTEGER ALC
      INTEGER TCOUNT(11),T(10)
      EQUIVALENCE(TCOUNT(2),T(1))
      INTEGER S(22),P(12),G(2),I(1),L(1)
      REAL R(1)
      DATA S/21,109,97,105,110,10002,108,105,110,101,10002,114,101,97,10
     *0,10002,119,114,105,116,101,10002/
      DATA G/1,1/
      DATA P/11,1,0,4,357,13,0,0,1,6,0,0/
      DATA I/0/
      DATA L/0/
      DATA R/0.0/
      DATA TCOUNT,TSAMP,ALC,TN/33*0,1/
      CALL SINIT(S,G,P,I,R,L)
      CALL XGLOBL(1)
      CALL XDEREF
      CALL XCPROC
      TSAMP(2)=0
```

```
      ALC(2)=0
      CALL XINVOK(3,0)
      GOTO 2
3     CALL ZDUMP(TCOUNT,TSAMP,ALC,11)
      RETURN

      generated code for each Icon procedure appears here

1     LABEL=FLABEL
2     GOTO (1,2,3,4,5),LABEL
      CALL SYSERR(29HICON: ILLEGAL INTERNAL LABEL.)
      END
```

Appendix B -- ZCLOCK

The module ZCLOCK contains the subroutines TICK and TOCK, which enable and disable the clock interrupt, respectively. The actual sampling routine begins at the label TKR, which receives control at each clock interrupt when enabled. MAKNAM, CREATE, CLOSE, and WRITEF are subroutines in the Ratfor I/O system.

```
        title   clock interrupt routines

        search  uuosym
        search  ioparm
        sall
        purge   close, open
        twoseg
        reloc   400000

wordmode==10*^d36
pcopt==1
tkopt==2
EOS==^d10002

prgnam==cparm##
gcflag==cparm##+^d10

define  .stop(msg) <
        jrst    [outstr [asciz/msg
/]
                 exit]
        >

; tick(ctoken) - test right half of .jbver (edit number, set by link-10).
;   Bit 34 is one if token sampling is selected, bit 35 is one if
;   pc sampling is selected.  Creates name.mon for pc sampling, turns
;   on clock for either.  The argument is the address of the token
;   counting array.

tick::  movei   t1,@(a)
        movem   t1,ctoken
        hrrzs   t1,.jbver
        trnn    t1,pcopt
         jrst   tick1
        movem   t1,pcflag
        movei   a,[exp <-4,,0>,prefix,prgnam,suffix,fname]+1
        pushj   p,maknam##
        movei   a,[exp <-2,,0>,fname,[write+wordmode]]+1
        pushj   p,create##
        cain    r,err
         .stop  <can't create monitor file>
        movem   r,monfile
        hrrz    t1,.jbver
tick1:  trne    t1,tkopt
         movem  t1,tkflag
```

```
        trnn    t1,pcopt+tkopt
         popj   p,
        movei   t1,intvec
        piini.  t1,
         .stop  <can't init priority interrupt>
        movsi   t1,(ps.fon)
        pisys.  t1,
         .stop  <can't turn on priority interrupt>
        move    t1,[ps.fac+intarg]
        pisys.  t1,
         .stop  <can't start the clock>
        popj    p,

; tock() - turns off clock if it was on, and closes icon.mon if
;   pc sampling was being done.

tock::  hrrz    r,.jbver
        trnn    r,pcopt+tkopt
         popj   p,
        move    t1,[ps.frc+intarg]
        pisys.  t1,
         jfcl
        skipn   pcflag
         popj   p,
        movei   a,[exp <-1,,0>,monfile]+1
        pushj   p,close##
        popj    p,

; tkr - processes clock interrupts.  For pc sampling, writes out
;   current pc; for token sampling, increments token count array.

tkr:    movem   z,save
        move    z,[1,,save+1]
        blt     z,save+17
        skipn   pcflag
         jrst   tkr1
        hrrz    t1,oldpc
        cail    t1,tick
         jrst   tkrend
        skipe   gcflag
         hrl    t1,@ctoken
        movem   t1,pcout
        movei   a,[exp <-3,,0>,pcout,[1],monfile]+1
        pushj   p,writef##
tkr1:   skipn   tkflag
         jrst   tkrend
        move    t1,ctoken
        skipn   gcflag
         add    t1,(t1)
        aos     1(t1)
tkrend: movsi   17,save
        blt     17,17
        debrk.
         .stop  <debreak failed 1>
```

```
          .stop  <debreak failed 2>

         reloc

intvec: tkr
 oldpc: block   3

intarg: .pcapc
         block   2

prefix: exp     "d", "s", "k", ":", EOS
suffix: exp     ".", "m", "o", "n", EOS
fname:  block   30
monfile:block   1
ctoken: block   1
pcout:  block   1
pcflag: 0
tkflag: 0

save:   block   20

         end
```

Appendix C -- Sample Fortran Code

The Icon program below is shown with each executable token marked. Below that is the Fortran code generated for the program, excluding the parts shown in Appendix A. Note that no token monitoring code is generated for tokens 5 or 9 (the left parentheses following read and write), since these correspond to built-in system functions.

```
procedure main
   while line := read(&input) do
   ^     ^     ^  ^    ^ ^       ^

      write(line)
      ^     ^^

end
```

```
4     CONTINUE
      CALL XLINE(2)
      T(1)=T(1)+1
      TN=1
      CALL XLPBEG
23002 CALL XMARK(5)
      CALL XLINE(2)
      T(2)=T(2)+1
      TN=2
      CALL XLOCAL(1)
      T(6)=T(6)+1
      TN=6
      CALL XKEYWD(605)
      T(4)=T(4)+1
      TN=4
      CALL XREAD
      IF(SIGNAL.EQ.0)GOTO 1
      T(3)=T(3)+1
      TN=3
      CALL XASG
      IF(SIGNAL.EQ.0)GOTO 1
5     CALL XDRIVE
      IF(LABEL.NE.0)GOTO 2
      TN=1
      CALL XPOP
      IF(SIGNAL.EQ.0)GOTO 23001
      CALL XLINE(3)
      T(10)=T(10)+1
      TN=10
      CALL XLOCAL(1)
      CALL XDEREF
      T(8)=T(8)+1
      TN=8
      CALL XWRITE(1)
      T(7)=T(7)+1
      TN=7
```

```
      CALL XPOP
      GOTO 23002
23001 TN=1
      CALL XLPEND
      CALL XPOP
      CALL XPNULL
      SIGNAL=1
      CALL XLINE(4)
      CALL XRETRN
      GOTO 2
```