Published: 10/18/68

Identification

Interpretation of entry counts in stgop_ transfer vector; coding styles which produce calls to stgop_.
C. Garman

Introduction

Various questions have recently been raised about the amount of time spent during Multics and 6.36 runs executing procedures of the EPL run-time support package, especially the procedures for manipulating long bit and character strings.

This document describes how to find out how many times EPL-compiled code called for various functions, and includes a section on coding styles which produce such calls, as well as how better to code them if possible.

The interface procedure stgop_(STrinG OPerations) serves as a transfer vector to shield the code from changes in the actual procedures which are eventually invoked.

A recent change in compiler-produced code uses a modified calling sequence which reduces the number of links generated per program by putting the transfer through the real link into one program only, and using an indexed TRA instruction to dispatch to the ultimate handler.

The calling sequence is

```
eapap     arglist          standard multics arguments
eax0      subno            "number" of procedure
tsbbp     <lib_>|[lib_]
```

(In 6.36, procedure lib_ is merely a transfer to <stgop_>|[lib_], in Multics, the segment named stgop_ also has the name lib_).

The stgop_ dispatcher stores the contents of bp->bb in sp|20 so that the called procedures may effect a standard return, (since no save is required the return is directed to the caller of stgop_), adds 1 to the contents of the location in a vector corresponding to subno and dispatches into a vector of transfer instructions which in turn transfer to the appropriate procedures. (The dispatch vector is shared with the entries used for individual calls to stgop_, such as EPLBSA procedures which do similar manipulation.)

Locating and Interpreting the Vector

The vector containing the indicated "fan-out" counts is located at 10(8) in the linkage section for stgop_; it is currently 42(8) words long, of which for the time being only 0-27(8) should ever be non-zero. (In Multics, stgop_ is part of bound_lib_1_wired, and its linkage block begins at 740(8) (currently). The linkage for the bound segment is itself copied into the combined linkage of each ring: in ring 0, it is in "wired_sup_linkage", with its base at 2356(8); and in the ring 1 combined linkage it generally starts at 3646(8). Thus to find the location of the values in a given ring's combined linkage, the three numbers must be added: the base of bound_lib_1_wired's linkage, the offset of stgop_'s linkage in bound_lib_1, and the offset of the vector in stgop_).

The order of the values is:

#8	name	function
0	bsbs_	move bit strings
1	cscs_	move char strings
2	ctbs_	concatenate bit strings
3	ctcs_	concatenate char strings
4	ixbs_	index function--bit strings
5	ixcs_	index function--char strings
6	ntbs_	not function--bit strings
7	ndbs_	and function--bit strings
10	orbs_	or function--bit strings
11	eqbs_	comparisons--bit string =
12	eqcs_	comparisons--char strings =
13	nebs_	comparisons--bit strings ¬=
14	necs_	comparisons--char strings ¬=
15	lebs_	comparisons--bit strings <=
16	lecs_	comparisons--char strings <=

#8	name	function
17	gebs_	comparisons--bit strings >=
20	gecs_	comparisons--char strings >=
21	ltbs_	comparisons--bit strings <
22	ltcs_	comparisons--char strings <
23	gtbs_	comparisons--bit strings >
24	gtcs_	comparisons--char strings >
25	ssbs_	compute subscript--bit string
26	sscs_	compute subscript--char string
27	bsfx_	convert bit string to double precision fixed

Also located in the linkage for stgop_, at 57(8) and 65(8), are the entry counts for the entry points [lib_] and [stgop_] respectively (the latter is for a projected compiler change that will further reduce the overhead of the dispatch function); the sum of these two values should equal the sum of the values of the elements of the fan-out vector.

These values are also included in the entry sequences of the individually called routines; however, certain routines, such as movstr_, are themselves called by other procedures in the string package, e.g., catstr_, and thus their indicated entry counts may be higher than those in the fan-out vector.

Also in the linkage for stgop_ are the entry sequences for the originally compiled calls: the counts for these are conveniently located at intervals of 10(8) beginning at 73(8), in the same sequence as indicated above for the dispatch vector.

Coding styles which produce string-package calls

A. General considerations

Although it is discussed in some detail in the BN sections on code generation, a brief review of the conditions under which EPL will produce calls to the runtime string package is appropriate here:

1. When the length of one or more of the arguments is

   a. greater than 36 bits (or 4 characters)

   b. an expression

   c. unknown

   under (c) is included string parameters with a '*' as the length, virtual strings resulting from a call to cv_string, and references to the substr function and pseudo-variable.

2. Whenever any variable is a varying string, regardless of its declared maximum length. Note that the length function is executed as in-line code; however, when the length desired is that of a parameter, care must be exercised that both the declared formal parameter and the actual parameter are of the same type, that is, both varying or both non-varying (otherwise incorrect execution will likely result). When a parameter may be either varying or non-varying the lg functions (BY.10.02) should be invoked instead.

3. Varying string temporaries are created whenever there is more than one expression in an assignment statement involving an unknown-length string, or a function call is made with a string expression involving one or more varying or unknown-length strings. Thus,

```
fixed_result=var||fixed;
```

   will not produce a temporary varying string, however

```
fixed_result=fixed||adj||fixed;
```

   will create varying-string temporaries.

   (Note that

```
fixed_result=fixed||fixed||adj;
```

   will also produce varying temporaries where

```
fixed_result=(fixed||fixed)||adj;
```

   will not).

   Likewise

```
call zorch (fixed||adj);
```

   produces varying string temporaries.

B. Specific Examples.

1. Testing for a null character string. The sequence

```
if string = "" then ...
```

is better stated

```
if length(string) = 0 then ...
```

2. Moving the first part of a string into another string:

```
string_1 = substr(string, 1, 1);
```

where string_1 is of length 1, results in two calls to the string package: one to compute the substring, and one to perform the actual movement (No, Virginia, substr-s are never done in-line).

An entirely equivalent result is obtained by:

```
string_1 = string;
```

which results in at most 1 call to the string package. (The assignment is done in-line if both strings are known by the compiler to be 36 bits or less in length).

3. Testing bits in a short bit string: When a procedure wishes to inquire if 1 or more bits in a short string (<36 bits) are zero or non-zero, a common coding practice is

```
if substr(bits, 4, 6) then ...
```

or

```
of substr(bits, 18, 9) ¬= "0" then ...
```

THIS GENERATES INCREDIBLE CODE: in the first case the code is precisely equivalent to

```
if index(substr(bits, 4, 6), "1"b) ¬= 0 then ...
```

with the attendant calls to first compute the substring and then search the resultant string for non-zero bits, or in the second case to compute the substring and then compare it with the constant string.

Much cheaper, in both generated code and execution time, would be

```
if bits & "000111111"b" then ...
```

or

```
if (bits & "000111111"b) ¬= "0"b then ...
```

4. Examining short bit string a few bits at a time.

Where a program must examine bits sequentially a common sequence might be

```
do i = 1 to 9;
     if substr(bits, i, 1) then ...
end;
```

There is slightly more justification for this code than that inveighed against above, but a much shorter, cleaner (albeit less transparent) solution appears below:

```
dcl  b9 bit (9),
     f10 fixed bin(10);
...
b9 = "1"b;
do i = 1 to 9;
     if bits & b9 then ...
     ...
     f10 = b9;
     b9 = f10;
end;
```

The statements f10=b9; b9=f10; effect the shifting of the mask (b9) 1 bit to the right per iteration. Extensions for larger field sizes should be relatively obvious, as well as algorithmns for reversing the direction in which the mask is shifted; the proof of the method is derivable from a close reading of the PL/1 manual.

5. Scanning character strings one character at a time.

If a program must examine a character string 1 character at a time, and then perhaps determine whether the character is a member of a fairly limited subset of the possible characters (and especially if the probability of its being such a member is relatively low), the example which follows will generally result in excessive running time for the program:

```
dcl  c1  char(1);

... (indexing)

c1 = substr(string, i, 1);

if index(subset, c1) ¬= 0 then ...
```

If a large number of characters are not going to be in subset, and if they are easily grouped in contiguous ranges (such as the alphabetic and numeric characters in ASCII), the if-statements shown below will be shorter in execution if such eliminations are made:

```
if c1 >= "a"
then if c1 <= "z"
     then go to alphabetic;

if c1 >= "A"
then if c1 <= "Z"
     then go to upper_case;

if c1 >= "0"
then if c1 <= "9"
     then go to numeric;

if index(subset, c1) ¬= 0
then ...
```

If the subset is fairly small the rest of the tests should probably be performed in line:

```
if c1=s1
then go to member;

if c1=s2
then go to member;

...
```

(The ultimate in speed of subdivision is, of course, a dispatch table that accounts for all the ASCII characters; however, since it has 128 entries, there would seem to be a little overhead unless a fine partitioning were really needed.)

6. Operations with bit-strings of lengths between 36 and 72.

As was mentioned under General Considerations earlier, the EPL compiler does not do in-line operations with bit-string fields larger than 36 bits; however, for frequently referenced items in packed structures, significant reductions in code and access time may be accomplished at the expense of padding the structures they lie in, and then referring to the data with a different structure containing double-precision integer identifiers (or alternatively packing the other items into a sub-structure at level 3). The following example shows a declaration as it might be written, and then as re-arranged and a "reference" structure declared for access to the longer items:

```
dcl  1  slow  based(p),
          2 uid bit(70),
          2 (time1, time2) bit (52),
          2 item1 bit (6),
          2 item2 bit(36),
          2 item3 bit(18),
          2 item4 bit(12),
          2 item5 bit(6);
```

As re-written:

```
dcl  1 padded  based(p)
         2 (pad1 bit(2),                  /*words 0-1*/
            uid bit(70)),

         2 (pad2 bit(20),                 /*words 2-3*/
            time1 bit(52)),

         2 (pad3 bit(20),                 /*words 4-5*/
            time2 bit(52)),

         2 (item2 bit(36),                /*words 6-8*/
            item1 bit(6),
            item3 bit(18),
            item4 bit(12),
            item5 bit(6),
            pad4 bit(30));
```

```
dcl  1 access based(p),
       2 uid fixed bin(70),                    /*0-1*/
       2 time1 fixed bin(52),                  /*2-3*/
       2 time2 fixed bin(52),                  /*4-5*/
       2 packed,                               /*6-8*/
         3 (item2 bit(36),
            item1 bit(6),
            item3 bit(18),
            item4 bit(12),
            item5 bit(6),
            pad4 bit(30));
```

Notes:

a. Item2 was moved around so that as a bit string it would not overlap word boundaries. (Otherwise an EPL "idiotic" reference would occur; see BN.9 for further information about structure packing.)

b. The structure as declared occupied 9 words; however, if the structure were to be instead an array of structures (adding dimensioning parentheses after the level-1 identifier), 36 bits more of padding would be required to make the double precision items always lie on even word addresses in all replications (the compiler would automatically make this adjustment on references to the access structure, but not on addressing calculations involving the padded structure). In this regard, the pointer returned in an EPL allocate statement will always be pointing at an even-address word. If references are made to the double-precision items with an odd-address, the 645 will automatically force it to the lower even address, with generally deleterious results.

7. Testing fixed-point values.

While this topic is not directly related to strings and string manipulation, it is relevant to the production of unnecessary code: when a program wishes to test a variable for zero/non-zero values, an all-too-common sequence is as follows (assume code is fixed bin(17)):

```
if code then go to error;
```

where a cleaner way would require a very simple change:

```
if code ¬= 0 then go to error;
```

The code expansions appear below

case 1	case 2
lda code	lda code
als 19	cmpa =0
anaq = v17/-1, 55/0	tze *+2
tze *+2	tra error
tra error	

The als-anaq instructions are solely for the purpose of converting the fixed point number (right-justified) into a bit string of the same length as the precision of the number (left-justified). Note that the literal reference in the first case is two words, which is a slower memory access than is required by the 1-word literal in the 2nd case. (Unfortunately EPL is not yet clever enough to realize that the 2nd sequence could be reduced to

```
lda code
tnz error
```

Note especially

```
if ¬ code then

error: ...
```

is not what it looks like; the code at error will be executed only if the rightmost 17 bits of code were all 1-s (see PL/1 manual for further information.)

8. Multi-conjunction if-statements.

While this topic, like topic 7, does not directly relate to calls to stgop_, it is relevant to the production and execution of rather gruesome instruction sequences: specifically statements of the form

```
if (a rel b) & (c rel d) & ... then ...
```

where rel stands for any of the 8 comparison operators (=, ¬=, <, <=, >=, ¬>, ¬<), and where the "&" may also be replaced by "|". For the case of and conjunctions ("&") much more efficient code results by the replacement of the given statement by

```
if a rel b
then if c rel d
     then if e rel f
          ...
```

The reason is that the compiler must create 1-bit bit-strings from the results of the relational expressions, or or and them together, and then perform the then-clause only if the result of all the operations leaves non-zero bit(s) in the temporary test string. Thus, all expressions are evaluated, even if the value of any one being 0 or 1, (for and and or respectively) ensures that no further tests need by made. On the other hand, given an ordering of the expressions such that the proper branch is taken as early as possible, significant speed-up in execution time may be obtained as well.

For expressions joined by "|" in the if-statement, the equivalent restatement is

```
     if a rel b
     then
doit:    ...
     else if c rel d
          then go to doit;
          else if e rel f
               then go to doit:
                    ...
```

Alternatively, by use of DeMorgan's theorem, one may reverse the sense of each rel and write the statement in the expanded form for and conjunctions, changing the last then to be a transfer around the code which is really to be executed. (local go to-s are cheap.)

Epitaph

The night has been long,

ditto, ditto my song,

and thank goodness they're both

of them over.

Gilbert and Sullivan -
"Iolanthe"