Published: 12/07/67

Identification

An example of EPL string manipulation
D. B. Wagner

Epigraph

There is a lot to do in Liddypool, but not all convenience.

- John Lennon.

Purpose

This section is a companion piece to Section BN.10.00. We take a simple sample program and investigate some of the ways in which the efficiency of the compiled code is affected by source-level decisions. We then present a rather terse annotation which should help a system programmer to read the object code given in BN.10.01A for the various versions of the program.

The investigation of the source program assumes a thorough knowledge of the language and some feeling for EPL data formats. The object code annotations assume a thorough knowledge of EPL data formats, 645 machine language, and EPLBSA conventions. Every system programmer ought to know these things, and those who do not may find this a useful firehose-style learning aid.

This Section is long on discussion and short on conclusions. This is due in part to the author's tragic flaw and in part to the difficulty of finding any universally valid principles in the irrational world of EPL. Our basic purpose here must be to provide the beginning of the kind of understanding which can be used as a basis for action in particular situations.

The Program

The program we are interested in is a trivial version of R. R. Fenichel's wordflipper. When called, it reads a line, reverses each word in the line, prints the result, and returns. Thus if it read "Where did you get those great big beautiful eyes" it would print "erehW did uoy teg esoht taerg gib lufituaeb seye".

The version we start with is on page 1 of BN.10.01A. Look at this program. The call to read_in reads a line from the console typewriter into the varying string line and sets l to the number of characters read. The use of index and the if's following it put into word the next word (sequence of characters delimited by blanks) in the line. The do loop with concatenations reverses the word and the statement "revword = ..." puts the reversed word into the proper place in the result. The call to write_out prints the result on the console.

With this introduction it should not be hard to follow the various versions and the history of versions presented below.

Efficiency Measurements

Just to what the reader's appetite, the size and timing of the various versions are given in the table below.

		size of text and link in words	time in ms. first call	time in ms. second call
1.	Varying strings and concatenation	603	481	46
2.	Non-varying strings and substr's	356	224	105
3.	Using mismatched declarations in place of the substr's	253	85	27
4.	Making the mismatched array "synchronous"	243	76	19
5.	Making the mismatched array have one-word elements.	465	75	17

The timing tests were made using the 645 simulator in 6.36.

There are good reasons for ignoring the measurement of the speed of the first call. In the first place this includes all the link-popping and (perhaps) some dynamic loading, and these are the specific system functions which will be done very differently in Multics. Second, the

first call happens only once, so that for example the first-call speed increase by almost a factor of 3 from version 2 to version 3 is really just a saving of 139 ms. A reasonable price policy might make this a saving of one cent per process using the program. Using the program in a thousand different processes would just begin to pay for one compilation of the improved version. The spectacular increase in after-first-call speed is what we should be concentrating on.

The Versions

Initial experimenting with modifications of this program was rather discouraging. We changed all the varying strings to static; changed line and word to non-varying; used a mismatched declaration instead of the substr in the inner loop; and we were most annoyed to find that the program size changed by perhaps 10 percent and the time changed by about 1 percent. Apparently the varying-string concatenation in the inner loop just swamps all other timing effects. Investigation of the run-time library shows that each concatenation with a varying result involves:

one call from the program to stgop_$ctcs_

one call from ctcs_ to alloc_

two calls from ctcs_ to movstr_

one call from ctcs_ to freen_

So the only hope was to change the algorithm to avoid varying strings and concatenations. The resulting program is on page 17 of BN.10.01A. The inner loop is tricky: examine it warily.

As the timings given above indicate, this attempt at improving the program is a thorough bust. The reason is that we still have a vast number of calls to run-time routines in the main loop. (See BN.10.01A pages 21-25, lines 184-324).

But note that all the substr's are on non-varying strings and have a constant "length" argument. Those in the know, know that EPL does a much better job on subscripting than it does on substringing. So if we treat the character-string line as an array of single characters, the code should improve.

We will treat line as an array by using a mismatched declaration. Mismatched declarations are, as a matter of policy and common sense, frowned upon. They create the possibility of incredible bugs, they cause us to lose some measure of machine-independence, and they are hard to find later unless the programmer has taken special care to make them obvious. Nevertheless, since some of the most effective strategies for improving program efficiency demand mismatched declarations, we must grumble and go ahead. Note in what follows that whenever a mismatched declaration is used it is very carefully annotated for the benefit of posterity.

See the program on page 29 of BN.10.01A. The program corresponds statement for statement with the previous one, so it should not be hard to follow. Note that the array declaration takes the form of an array inside a structure. This is because, if we had simply declared,

```
dcl dummy_line (100) char (1) based (lp);
```

EPL would have considered this array aligned, i.e. with each character at a word boundary. BN.9.01A gives the rules which determine whether an EPL aggregate is aligned or packed. It suffices here to say that the declaration,

```
dcl 1 dummy based (lp),
      2 line (100) char (1);
```

yields a packed array.

A glance at the timing table given earlier shows that we have done rather well this time: a fourfold speed increase and about 30 percent space improvement. But it still is rather galling to think that it doesn't require a very smart compiler to recognize this special case and give it to us free, without mismatched declarations, extra debugging time, and a messy program.

But this program can be milked even further: notice in the EPLBSA listings for this program the frequent comment "NOT SYNCHRONOUS". They indicate that our array was planned inefficiently. The concept of synchrony of aggregates is defined in BN.9.01 and is rather complicated. In this case it turned out that the first structure declared below is not synchronous, while the second is:

```
dcl 1 dummy based (lp);
      2 line (100) char (1);
```

```
dcl 1 dummy based (lp),
      2 line (0:99) char (1);
```

The program on page 38 of BN.10.01A makes suitable adjustments and uses the second declaration above instead of the first. We again find an improvement, although of course we couldn't really expect it to be as dramatic as the first: roughly 30 percent timing improvement.

Just to show the lengths to which one can go, we have done great violence to our program and produced the version on page 47 of BN.10.01A. Here the array declaration is,

```
dcl 1 dummy (25),
      2 (c0,c1,c2,c3) char (1);
```

So the array has one-word elements, and whenever an access is made to a character in the array its position with respect to word boundaries is known so shifts do not have to be calculated at execution time.

The speedup in the actual access must be at least tenfold, but most of this improvement is masked by the large amount of expensive arithmetic we have placed in the inner loop. Thus the speedup is only about 10 percent. There is, of course, the possibility that a cleverer algorithm might obviate the need for this arithmetic. But the whole point of using a higher language is to avoid the necessity of worrying about word boundaries and such.

Annotations of the EPLBSA object programs

Below is a set of notes which should help in reading the EPLBSA code in BN.10.01A.

The EPLBSA listings given for these programs are those produced on-line on the 645 by the EPLBSA assembler. These listings are printed by the GECOS program SYSOUT and so use the GE Hollerith character set instead of Ascii; therefore all letters are printed as upper-case and certain unavailable special characters are printed as random other characters. What GE character replaces a given Ascii character depends upon which of the two printers happens to be configured into the system. The two important transformations are:

Ascii character	One printer	Other printer
underscore	~	\
vertical bar	^	^

Look quickly at one of the EPLBSA listings and notice the following ways in which EPL helps someone reading the code:

1. The code produced for a given source statement is marked off by blank comment lines, that is lines which consist of nothing but a double-quote character.

2. Scattered throughout will be found curious little comments in the "comments field" of various lines. These comments are explained in the notes below.

In addition, note the "line numbers" in the EPLBSA listing. (Third column of numbers.) These line numbers will be used throughout this Section in referring to the EPLBSA code.

Page 2, lines 2-4.

These comments give the version for each of the three passes of the compiler.

Page 2, lines 5-7.

These statements appear at the beginning of every object program. The segref for datmk_ may or may not be necessary, but is always there. The symbol .ds marks the beginning of the "display" in each stack frame: the display is used by internal blocks to keep track of the stack levels of all containing blocks. The symbol .u0 marks the beginning of a 4-word "utility" block of storage in the stack frame. The important thing about .u0 is that this storage is available at each block level possible in the program.

Page 2, lines 8-17.

This kind of code is compiled for each external entry. The link on line 8 is compiled in case the program later refers to itself. Use is the pseudo-op which specifies a location counter to be used (see BN.8.01). Note the names of the two most important location counters, prolc1 for the prologue code sequence and mainc1 for the main code sequence.

Lines 14-15 are the call to .sv, the standard save sequence. The symbol .as1 is the size of the stack frame needed by this program, and is defined at the end, on page 11, line 446.

Pages 2-3, lines 19-54

This code is included in any program which mentions varying strings. It is usually merely useless, but can occasionally be dangerous. This code tries to set up and initialize the free storage area at free_$free_. But we normally don't need this initialization since the library free_ segment is present. Furthermore this code may never be invoked, as in this case it is not.

This code is that which would be compiled for a declaration like

```
dcl x area((1024)) external static;
```

Lines 29-37 are the code to grow 1024 words at the first reference to the segref free_. Lines 38-43 are the dope vector for the area. Lines 45-53 are the code in the "internal static specifiers" code sequence to set up the specifier for the area.

The references to location 39 of the stack frame on lines 36, 46 and 51 have to do with making sure we don't attempt to initialize until the specifier exists. I have never been able to understand this code, but it doesn't work anyway so don't worry about it. It will be changed to something else eventually.

Page 3, lines 54-69.

This is the code compiled for the declaration of the varying character string line.

The equ on line 55 gives the location in the stack set aside for the string's eight words of specifier and data. The comment on this line indicates the source program name. xx0026 is the alias used by EPL in referring to this variable.

Lines 58-65 are in the prologue code sequence: they manufacture a specifier for the varying string and then call the subroutine .v1 to initialize the string. Lines 66-69 are in the epilogue code sequence: they free the storage occupied by the value of the varying string.

Page 4, lines 127-131.

These equ's specify the locations in the stack frame which will hold the fixed variables i, j, k, and l. Again note the aliases and the comments giving source program names.

Page 4, lines 133-149.

This is the code compiled for the call to read_in. Note that the statements

```
link      xx0038, <read_in>|[read_in]

...

call      lp|xx0038,*(...)
```

have precisely the same effect as,

```
call      <read_in>|[read_in] (...)
```

The former code is more difficult to read in the listing, however, and is therefore preferred by EPL.

Line 134 is the code for the constant 100, and lines 136-140 copy this constant into a temporary for the call. PL/I does specify this copying of all constants passed as arguments, but it seems to be rather a waste.

The symbol .a1 is the location in the stack frame of a block of storage big enough to hold the largest argument list used in the program. The code in lines 141-148 set up the argument list for the call. The fld instruction on line 147 is very clever: it sets up the a - and q - registers for a standard argument list header, and does it in one instruction without needing any literals. Check the 645 manual for how it works.

Page 5, lines 151-156

This is the character-string constant " ". Note that it has a specifier which can live in a pure procedure. There seems to be no reason for the even pseudo-op on line 152.

Page 5, lines 157-165

This is the call to stgop_$cscs_ which performs the assignment

```
result = " ";
```

See BN.7.09 for stgop_$cscs_.

Page 5, lines 167-171.

This is the assignment

```
i = 1;
```

as can be seen by looking back in the program to find out what xx0034 is the alias of.

Page 5, line 173.

This line is for the statement label word_loop. xx0043 is the alias for word_loop.

Page 5, lines 174-199.

This code evaluates the subexpression

```
substr (line, i, 1-i+1)
```

using the run-time routine stgop_$sscs_. This routine just goes directly to the routine substr_$sscs_. The latter is described in BN.7.05, and the reader is referred to that Section for what is going on here.

Page 6, line 219-223.

This is the substatement

```
if j=0 then
```

Note that this code is about the best that could be expected from anything but a supercompiler. It seems to be generally true that EPL's arithmetic is excellent and everything else is mediocre-to-disastrous.

The if-action and the else-action are lines 225-255 and 259-290 respectively.

Pages 8-9, lines 301-322.

This is the code for the do statement. Lines 360-361 are for the corresponding end statement.

Lines 304-309 evaluate the subexpression

```
length (word)
```

Note that the four instructions on lines 304-307 could be replaced by

```
ldq  sp|xx0028+6
```

Page 10, lines 363-399.

This code is for the statement

```
result = result||" "||revword;
```

Lines 367-378 set up a varying string temporary with alias xx0051. Then the code for the statement is the same as that for

```
t = result||" ";

revword = t|| revword;
```

where t is the temporary.

Page 11, line 421.

This tra is the code for the return statement.

Page 11, lines 423-442.

This kind of code appears at the end of any block which has an epilogue. See BP.3.00 for an overview of what is happening.

Lines 433-442 are the prologue code to establish the epilogue for the benefit of the unwinder. The symbol end.1 is where control goes at a normal return, and epi.1 is the point at which the unwinder can call the epilogue.

The code on lines 427-430 is essentially the standard beginning for an EPL internal procedure. The job of .cp (lines 495-502, q.v.) is to get the linkage base pair lb←lp set properly and to put a pointer to the stack frame of the main procedure into .ds, the "display", in the epilogue's own stack frame.

The code on lines 423-424 reverts the epilogue so that the unwinder no longer knows about it, and sets .ds to point directly to the current stack frame.

Looking quickly back to line 67, note how the epilogue does its job. By always referring to things through .ds it does not need to know whether it is operating in a stack frame of its own or in the same stack frame of the main procedure.

Page 11, lines 444-448.

Equ's like this appear at the end of every block. They establish several numbers important to the block. The symbols are:

.a1	argument list storage
.u1	utility storage
.as1	size of stack frame
.w1, .m1	unclear. They have something to do with adjustable data, of which there is none in this program.

Pages 11-12, lines 449-459.

This code is to set up the internal static storage block. It only got into this program by accident (because of the free_$free_ nonsense of lines 19-54) and will not be discussed here. See the notes in BN.10.02 for a full discussion of this code.

Page 18, lines 19-26.

This is the code for the declaration of the non-varying string line. The prologue code sets up an ordinary 4-word specifier. No epilogue is needed.

Page 19, lines 68-69.

This is the code for the statement

```
do j = i by 1 to 1;
```

The corresponding end statement compiles into the tra on line 146.

This code may be thought of as something like the following:

```
dcl xx0040 fixed;
xx0040 = i;
dcl xx0041 fixed;
xx0041 = 1;
j = xx0040;
go to xx0042;
xx0038:  j = j+1;
xx0042:  if j > xx0041 then go to xx0039;
         ...
         go to xx0038;
xx0039:
```

This code is four or five instructions more expensive than it needs to be, but it really isn't bad compared to the rest of EPL. (And it is nearly an order of magnitude better than EPL produced for this statement in the bad old days).

Page 20, line 100

This equ is the result of Pass 1.5's pooling of temporaries. The comment "equ temp" tells the story.

Page 21, line 139.

Comments like "A reference to a temporary" are added to much of the code produced by the part of the compiler written by J.F. Gimpel. The main thrust of Gimpel's work on EPL was optimizing short string references, so that these comments generally point out pieces of very good compiled code. This particular instruction is picking up the value of the bit-string temporary xx0047 without going through the specifier. In the bad old days this reference would have taken perhaps four or five instructions.

Page 21, lines 150-187.

This is the code for the second do statement in the program. The amazing array of floating-point operations such as fad, lde, etc., etc., indicates to the weary reader that something happened which he didn't bargain for.

The problem is that EPL's language specifications demand that the result of a division be a double-precision floating-point number. So the compiled code dutifully converts everything in sight to floating-point. Every time around the loop, k is converted to double-floating and compared with the double-floating result of the expression (j-i)/2.

It would have been slightly more efficient to write the statement,

```
do r = 0 by 1 to fixed ((j-i)/2,17);
```

but not by very much. Hard to get worked up about.

Page 25, line 351.

This is what appears at the end of a block that doesn't need an epilogue.

Page 30, line 34.

Here is the code for the statement

```
dcl c char (1);
```

Compare with the code for the same statement, page 18, lines 34-40. Pass 1.5 has determined that in the present case no specifier is needed, because c is never passed as an argument.

Page 32, lines 98-99.

This is the constant " " again, but in this version of the program Pass 1.5 has determined that the constant does not need a specifier, since it is never passed out as an argument. Compare page 6, lines 201-205.

Page 32, lines 98-115

This is the code for the substatement,

```
if lp->dummy.line(j) = " " then
```

The hassle this code goes through is better than once-upon-a-time, but still rather peculiar. What .mx0 and .mx1 do is not terribly clear, but the result is that index register 6 contains a word offset and index register 5 contains a shift.

The call to .mx1 is made necessary by the fact that the array is "not synchronous". See BN.9.01 for a rather opaque definition of this term.

Once the character has been accessed out of the array, the comparison (lines 112-115) goes along rather well, although it certainly could be pared by at least 2 instructions.

Page 41, lines 97-116.

This is the substatement,

```
if lp->dummy.line(j-1) = " " then
```

where the declaration of dummy has been changed so that it is "synchronous". Compare with page 32, lines 98-115. Now we see that the call to .mx1 has disappeared, so that the statement executes much faster.

Page 51, lines 46-60.

This is the code for the declaration,

```
dcl blc(0:3) label init (blc_0,blc_1,blc_2,blc_3);
```

As can be seen, this does not look much like a label array. The actual "data" of the array is the set of four tra's, lines 46-49. So the data is not ordinary label data. Furthermore it cannot be assigned to, since it lives in the procedure segment. The wonderful thing about this label array, however, is that using it is incredibly fast and efficient.

Lines 56-60 are dope and specifier for the label array. They are never referred to, but are there only because a bug in Pass 1.5 at this writing forces them to be made.

Pages 53-54, lines 197-198.

This is the code for,

```
go to blc(jc);
```

Remarkably good code.

Page 54, lines 205-213.

This is the code for the substatement,

```
if lp->dummy (jw).c0 = " " then
```

The circumstances are ideal for a short-string reference now: the array has one-word elements, and the position within the array of the substring is known. The code turns out to be not bad: only three instructions less, but now the call to .mx0 is not necessary so each instruction carries only its own weight.

Pages 55-56.

Note the vast array of floating-point operations, a consequence of using divisions. With this much code in the inner loop doing arithmetic we had no reason to expect this program to be faster than its predecessor, but in fact tests show that this program is faster. The general rule seems to be: any amount of arithmetic is worthwhile if it in some way allows better string manipulation.

Page 57-59.

The high density of "A reference to ...." comments is a sign that we must be doing something right, since as we mentioned before this comment usually indicates the neighborhood of a better-than-average piece of compiled code.