To: Distribution

From: Richard Barnes

Subject: A Guide to Efficient PL/I Constructs in the Multics Environment

Date: October 23, 1974

1. Introduction

This document is an informal guide to efficient use of the Multics PL/I compiler. It provides advice on how to take advantage of the good features of the compiler while avoiding its weaknesses. Emphasis is placed on discussing which constructs produce more efficient code than others. The document assumes that the reader is familiar with PL/I.

For a semi-formal definition of the language supported by the Multics PL/I compiler, see the "PL/I Language Manual".

2. The Alignment Attributes

The use of the aligned attribute and the unaligned attribute can have a great effect on the speed of a program and the size of its database. Whereas unaligned items can start on a bit boundary (character boundary for character strings, pictures, and decimal variables), aligned items must start on at least a fullword boundary and occupy an integral number of fullwords. If a value requires 72 bits or less of storage to represent it, access of the value will be faster if its generation of storage is aligned because it can be directly loaded into the aq registers.

2.1 Use of the Alignment Attributes with Arithmetic and Pointer Variables

Access of aligned binary and pointer variables is usually much faster than that of unaligned variables. The only exception to the above is that unaligned pointers that happen to be fullword aligned are accessed at speeds comparable to that of aligned pointers, but the former cannot be indirected through. In general one should use aligned binary and pointer

----

variables for local scalar variables, and only use unaligned binary and pointer variables in large data structures where size is important, but speed of access is not.

The alignment attribute has no effect on the access time of decimal variables.

2.2 Use of the Alignment Attributes with Short Strings

A short string is defined to be a nonvarying string with constant extents whose length is less than or equal to 72 bits (eight characters). Access of aligned short strings is usually much faster than that of unaligned short strings. Thus, it is recommended that one use aligned short strings for local scalar variables, and restrict the use of unaligned short strings to large data structures where space is important.

2.3 Use of the Alignment Attribute with Long Strings

All nonvarying strings that are not short are considered to be long. Because, in general, these strings are too long to fit into the aq registers, the use of the aligned attribute does not speed up their access. Therefore, it is recommended that one use the default alignment attribute--unaligned.

All varying strings are considered to be aligned whether declared aligned or unaligned.

2.4 Use of Unaligned Short Variables in Arrays and Structures

For the purposes of this discussion, short variables are those variables which occupy no more than 72 bits (eight characters) of storage and are declared with constant extents.

When accessing an element of an array of short unaligned variables, the access code is quicker if a constant subscript is used, because the compiler uses an EIS (Extended Instruction Set) instruction, when the subscript is not constant, in accessing the variable. If an unaligned short variable is contained in an array of structures, and the variable is accessed with a nonconstant subscript, access code is faster if the array is declared aligned, because the use of an EIS instruction is avoided.

3. Use of the Precision Attribute in Offset and Length Expressions

Because 6180 index registers can only hold 18 bits of information, while up to 24 bits may be needed to express the offset or length of a string for use in an EIS instruction, the compiler must make use of the precision attribute in deciding which register to use. If a subscript expression, the second or third argument of the substr builtin, or the declared length of a string has a precision of 18 or less, it can be kept in an index register, whereas if the precision is more than 18,it must be kept in the a or q register. This means, for example, that if a user knows that he wants a substring that may be more than 262,143 items long, then the precision of the third argument of substr should reflect that fact (otherwise the high-order bits of the length may be lost). Conversely, if the user knows that a string is less than 262,144 items long, he should reflect that knowledge in the precision used for subscripts and arguments to substr. (Besides looking at the precision of the length and offset expressions, the compiler also makes use of the declared string size in cases of constant extents to determine where the offset or length may be kept.)

4. The Use of Internal Static to Simulate Named Constants

If a variable is declared to be internal static with an initial attribute and it is never set within a program, the compiler will treat it as if it were a constant. (A variable is considered set if it appears on the left side of an assignment statement, is the first argument of a pseudovariable, appears in the list of a get statement, appears as the target of a read statement, appears in a set option, is passed as an argument, is an argument to the addr builtin, or is the base reference of a defined attribute.) Converting an internal static variable to a constant means that more efficient code will often be generated to use the variable, sometimes avoiding storage references, and that the variable will not have to be copied into the combined linkage section upon initiation of the segment. Since passing a variable as an argument is equivalent to setting it, one must enclose the variable in parentheses if it is to appear in an argument list. This will make the variable be passed by value and force a copy to be made at call time. Making sure that such an internal static variable, which the user intends to use as a constant, is considered by the compiler to be a constant is worthwhile if the variable is not a long string which is only used in a few calls. This feature of the compiler is a good substitute for named constants which the PL/I language generally does not provide.

5. Use of the Initial Attribute

The compiler's implementation of the initial attribute for automatic, based, and controlled arrays is inefficient compared with the code the user can get from explicit assignment statements. Therefore, use of the initial attribute in the above cases is discouraged. Since the use of the initial attribute does not generate code for static variables, the above statement does not apply in that case. Users are warned, however, that use of the initial attribute can make a program more difficult to read in some cases, and that initialization of large external static arrays this way can cause creation of a larger object segment than intended.

6. The Assignment Operation

6.1 The Multiple Assignment Statement

In deciding whether or not to use a multiple assignment statement rather than separate assignment statements, it is useful to know under which circumstances multiple assignment statements produce inefficient code. A multiple assignment statement of the form

```
T1, T2, ---, Tn = E;
```

where E is not a constant, is semantically equivalent to the separate statements

```
 V = E;

T1 = V;

T2 = V;

    .

    .

    .

Tn = V;
```

If the temporary represented by V can be kept in a machine register throughout the assignment, then the multiple assignment statement is efficient. Clearly, this implies that if E is longer than two words, the multiple assignment statement will <u>not</u> be efficient, since E cannot fit in a register. Thus, multiple assignment statements are not efficient when the right hand side is a long string, a varying string, an entry variable, a label value, a file value, a format value, an area, a decimal value, a complex value, or an aggregate.

6.2 Conversions

All of the PL/I conversions are efficient, many of them producing inline code, while the others produce calls to any_to_any_. Inline code is produced for all cases where neither the source nor target are complex, decimal, character string, or picture (See 6.3). Of the other cases, the following produces inline code:

complex_float binary ($\leq$27)	=	real binary;
real binary	=	complex_float binary($\leq$27);
real decimal	=	real decimal;
complex decimal	=	complex decimal;
real binary integer	=	real decimal;
real decimal	=	real binary integer;
character	=	real fixed decimal;
character	=	real binary integer;

All other cases produce calls to any_to_any_.

The convert builtin function can be used to effect conversion between character and binary and avoid intermediate conversions that other builtins might cause.

6.3 Pictures

The use of pictures provides a convenient way to get efficient controlled conversion between arithmetic and character. When using pictures, the user can avoid PL/I's inconvenient conversion rules by specifying the format he/she wishes.

While picture unpacking (going from character to arithmetic form) is done by pl1_operators_ call, the most common cases of picture editing (going from arithemtic to character form) are done inline. At present, inline code is generated for the majority of cases of editing into real fixed pictures. The cases of editing into real fixed pictures that produce pl1_operators_ calls are any of the following:

- o the absolute value of the number's scale is greater than 31
- o a "y" picture character appears in a drifting field picture (e.g., $$$y99)
- o a zero suppression character or drifting character appears to the right of the "v" picture character
- o the inline sequence requires more than 63 micro-ops for the MVNE instruction

7. Arithmetic Operations

Most arithmetic operations are implemented with fast inline code. The one general exception is the power operator - ** which is sometimes implemented with pl1_operators_ calls or subroutine calls. USERS ARE CAUTIONED AGAINST USING THE "/" OPERATOR WITH FIXED POINT OPERANDS AS THE PL/I PRECISION RULES MAY CAUSE UNEXPECTED RESULTS.

7.1 Binary Operations

Most binary arithmetic operations produce inline code. Multiplication of fixed binary (≥36) numbers produces pl1_operators_ calls, all division invoked by the "/" operator cause calls to slow pl1_operators_ routines.

The "**" operation normally generates pl1_operators_ calls for real operands and full subroutine calls for complex operands. If the operands are both real, and the second operand is a positive integer constant that could be represented as a fixed bin(35) value, inline code will be generated to do the power operation as repeated multiplications.

7.2 Decimal Operations

Most decimal arithmetic operations cause efficient inline code to be generated. The major exception is the case of one or both of the operands having a scale greater than 32 or less than -31. This case will often cause additional assignments or multiplications to be generated since the 6180 hardware only handles scales within the range -31 to 32.

If the power operator has decimal operands, a conversion to and from binary and/or a subroutine call will be generated.

8. String Operations

All string operations (as opposed to builtins) cause inline code to be generated. In addition, some special cases cause better than usual code to be generated.

8.1 Special Case of Concatenation

Concatentation is often used in constructing varying strings. A normal concatenation of the form

```
a = b || c;
```

results in three (3) moves -- b and c are moved into a temporary, and the result is moved into a. However, a concatenation of the form

```
vs = vs || c;
```

where vs is a varying string, results in just one move -- c is moved to the end of vs. The latter special case can be used to great advantage in building varying strings. Consider the following example:

```
vs = a || b || c;
```

results in four moves and perhaps some instructions to allocate temporaries, while

```
vs = a;

vs = vs || b;

vs = vs || c;
```

results in three moves with no temporaries allocated.

8.2 Operations on Long Strings

Most statements of the form

```
a = b <bool_op> c;

a = translate (b,...);

a = bool (b,c,<bolr>);
```

where a, b, and c are long nonvarying strings, cause code to be generated that performs the operation in a temporary and then moves the result into a. However, if a is the same length as the temporary would be, and if the compiler believes that a could not possibly overlap with b or c then the operation will be performed directly in a and no temporary will be allocated. (Note, that due to a problem with the current implementation, this optimization only occurs if a is unaligned for boolean operations, and only if a is aligned for builtins.)

In a statement of the form

```
if a <op> b
```

or

```
if bool (a, b, <bolr>)
```

where a and b are long strings, the compiler will attempt to do the operation, without allocating a temporary, by using an SZTL instruction if the value is not needed elsewhere.

8.3 Aggregate Operations

Most aggregate operations, other than simple assignment and the use of the string and unspec builtins and pseudovariables, are relatively inefficient in the present Multics PL/I implementation and should be avoided. By simple, assignment, we mean assignment statements of the form.

```
p -> aggregate = q -> aggregate;
```

9. Use of the Builtin Functions

Most of the standard PL/I builtin functions and pseudovariables are implemented efficiently in the Multics compiler. There are certain exceptions and special cases that should be mentioned explicitly.

9.1 Arithmetic Builtins

With the exception of the divide builtin, all the arithmetic builtins cause efficient code to be generated. The divide builtin is inefficient only for some cases in which a fixed binary result is produced. If a fixed binary result is produced, a call to a very slow pl1_operators_ divide routine is generated unless the result and both operands are unscaled with a precision less than or equal to 35.

9.2 String Builtins

Efficient inline or out-of-line code is generated for all but three string builtins and pseudovariables. The builtins that are handled inefficiently are before, after, and decat. Execution of these three builtins is about 50 times slower than might be expected.

There are special cases of some of the other string builtins that cause more efficient code to be generated than is normally generated for the general case. These are:

```
index (<char_str>, <char1>)

index (<char_str), <char2>)

index (reverse(<char_str>), <char1>)

index (reverse(<char_str>), <char2>)

search (<char1>, <char_str>)

verify (<char2>, <char_str>)

search (<char_str>, <constant>)

verify (<char_str>, <constant>)

search (reverse(<char_str>), <constant>)

verify (reverse(<char_str>), <constant>)

translate (<char_str>, <constant> [,<constant>])
```

Note that the search, verify, and translate builtin functions expect that the characters in their input are all legal ASCII characters. These builtins may <u>not</u> by used to process strings with non-ASCII characters.

9.3 Mathematical Builtins

References to the mathematical builtin functions are compiled either into fast calls to pl1_operators_ or into slower normal subroutine calls. The following math builtins are compiled into pl1_operators_ calls if they have real arguments:

atan	exp	sin	tand
atand	log	sind	
cos	log10	sqrt	
cosd	log2	tan	

All other cases produce subroutine calls.

10. The Call Statement and Function References

When a call statement or function reference is executed,in the general case, an argument list must be constructed which takes 3 + 2*number_of_arguments instructions to do. When the new procedure block is entered, a new stack frame is established by a pl1_operators_ routine that takes around 30 instructions. This is a high overhead to have when using an important feature of PL/I that is necessary for good programming practice. The Multics PL/I compiler has two optimizations which can greatly reduce this overhead. First, it can decide that an internal procedure or begin block may share the stack frame of another block rather than obtaining its own. A block that does not obtain its own stack frame is called a "quick" block or procedure. Second, the compiler can build argument lists to quick procedures at compile time, if the arguments have constant addresses known at compile time. These two optimizations greatly reduce the cost of call statements and function references.

10.1 Determining the "Quickness" of a Block

The Multics PL/I compiler goes through a two stage process to determine which (procedure or begin) blocks can be quick, that is which ones need not obtain stack frames. The first stage excludes blocks from being quick because of their properties. The following properties can make a block non-quick.

o it is the external procedure block

o it is an ON-unit

o it has I/O statements

o it has format statements

o it has ON, revert, or signal statements

o it has automatic variables with expression extents

- it has an entry that is assigned to an entry variable or passed as an argument

- it has an entry with a star-extent return value

- it has an entry with a star-extent parameter that is called with the corresponding argument being an expression whose length is non-constant

- it has an entry that is referenced in the argument list of such a call after the aforementioned argument

In the second stage, the compiler uses a graph of the calls between blocks, to determine which of the remaining eligible blocks can be quick. The algorithm used in this stage is an iterative one based on the constraint that a quick block may use the stack frame of one and only one non-quick block and thus may effectively be invoked from only one non-quick block. In fact, the algorithm states that a quick block may be invoked from only one stack frame, and an invocation from a quick block is considered an invocation from its owner's stack frame.

A user can determine which block has been made quick by examining the symbols listings produced by the compiler. In the section marked, "STORAGE REQUIREMENTS FOR THIS PROGRAM" is a list of all the blocks in the program. If the line for a particular block contains the words, "shares stack frame of", that block is quick.

10.2 Determining which Calls and Function References Use Constant Argument Lists

In generating a quick procedure call, the Multics PL/I compiler can often generate a constant argument list if the addresses of the arguments are known at compile time. This saves the cost of executing instructions to set up the argument list at runtime. At this time the following constraints must be satisfied for the compiler to generate a constant argument list:

- the quick procedure must contain no non-quick blocks

- the stack frame of the caller must be smaller than 16,384 words

- the arguments must be constants, expressions with operators, builtin references, function references, or automatic variables

- o all automatic arguments must be allocated in the stack frame of the caller

- o all automatic arguments must have constant extents

- o all subscripted arguments must have constant subscripts

11. Using If Statements

In handling if statements containing logical operators, such as

```
          if x = 0 | p ^=null | x + 3 < 2
             then call a;

          if z > 3 & q = null & loaded
             then call b;
```

the present Multics PL/I compiler generates code that evaluates the entire logical expression before making the jumps, even though evaluation of one item of a logical operator might suffice to decide the result. In addition, converting the result of a relational operator to bit (1) requires a call to a very short sequence in pl1_operators_. (This sequence averages about three instructions.) This causes a conflict between considerations of speed and style. While the above examples may be easier to read and debug, the following examples would be faster:

```
          if x = 0 then go to lab;
          if p ^= null then go to lab;
          if x + 3 < 2
             then
     lab:            call a;

          or

          if z > 3
          then if q = null
          then if loaded
              then call b;
```

Resolving such a conflict is an individual decision. In any particular case, one has to judge whether the extra speed of the latter approach is worth the lack of clarity and structure it entails. Obviously the answer will differ from case to case and programmer to programmer.

There are plans to improve the PL/I compiler's handling of such if statements in a future implementation so that the structured approach will get more efficient code.

12. Other Constructs That Are Costly or Dangerous

- default statements

- multidimensional arrays with star bounds

- arrays of elements of star extents

- programs requiring a stack frame of more than 16,384 words