G0080 1/21/69
R. A. FREIBURGHOUSE

SIGNIFICANT FEATURES OF MULTICS PL/I

The Multics PL/I implementation embodies a number of interesting features including new implementation strategies and new language constructs. This paper is a brief description of some of the more significant and unusual features. More complete documentation of the implementation and a detailed specification of the language will be available at a later date. The reader is urged to also read the companion document titled COMPATIBILITY CONSIDERATIONS OF THE PL/I IMPLEMENTATION.

The PL/I Language

The language of the Multics PL/I implementation is defined by IBM publication Y33-6003-0. A number of features (listed in Appendix 1) are not implemented in the initial compiler. Certain more exotic features (tasking, sterling data, etc.) will never be implemented.

The PL/I language implemented for Multics is a new language having fewer restrictions than EPL and containing many features not found in EPL. Some of the more interesting features are described in the following sections.

Data Initialization

The PL/I compiler implements the full form of the initial attribute for all storage classes. This permits the use of based initialized structures and all forms of array initialization. Initialization of internal static is done entirely at compile time making it a useful method of writing table driven routines.

Data Packing

The attributes "aligned" and "unaligned" may be applied to strings to control their packing within aggregates. Unaligned strings and structures may be passed as arguments and may serve as the argument to the "addr" function. In such cases, the image (the based declaration or parameter declaration) which is used to reference the unaligned item must also be declared unaligned. These two attributes allow the programmer to completely control the packing of data.

Self-defined Based Structures

The PL/I language contains a feature known as the "refer option" which is useful for declaring self-defining structures.

Example:

```
dcl  1  self based,
     2  size fixed,
     2  s char (n refer(size));
```

At the time of allocation the length of the string "s" will be computed using the value of "n". All subsequent references to the string will compute the length using the value of "size". The assignment of "n" to "size" occurs automatically at the time of allocation. This feature eliminates the existing EPL ackwardness of allocating and referencing self-defining structures.

Multiple Entries and Returns

The PL/I compiler correctly implements the multiple entry multiple return features of the PL/I language. The number of parameters may differ for each entry, the position of any particular parameter may be different in each entry, and the return value of each entry may have different attributes. A return (exp) statement will cause the expression to be converted to the correct type determined dynamically by the entry used to enter the procedure. This feature has always existed in PL/I but was not correctly implemented in EPL.

Reference Qualification

References to elements of structures do not need to be fully qualified, they need only be sufficiently qualified to make them unique according to the rules of PL/I.

References to based items do not have to be explicitly pointer qualified if the based attribute used to declare the item provides a pointer, i.e., based(p). Note also that it is no longer necessary to provide the pointer in the based attribute, i.e., based is legal.

Scope Rules

The PL/I compiler obeys the rules of the PL/I language with regard to the scope of declarations. Declarations do not have to precede their use as is the case in EPL. The compiler also obeys all PL/I rules related to the establishment of default declarations and attributes.

The compiler produces a storage map which shows all declarations of identifiers and lists the attributes either declared or assumed for each declaration. Declarations established contextually are listed separately to provide the programmer a quick method of checking for missing declare statements or mis-spelled identifiers.

IMPLEMENTATION STRATEGIES

Argument Passing

All arguments passed by PL/I calls are passed directly. This means that the pointers which constitute the argument list of the Multics call always point directly to the data. Specifiers and dope are never used. This direct passing of arguments means that all types of arguments are passed as efficiently as arithmetic scalars are passed by EPL.

If a parameter is declared to have * bounds or lengths, then the calling sequence must include argument descriptors which supply the missing size data. PL/I argument descriptors are an extension of the Multics standard argument descriptors described in BD.7.02. The PL/I compiler allows the programmer to control the creation of argument descriptors through the use of the entry attribute. If the PL/I programmer always completely declares every entry he calls, then argument descriptors will only be created when they are actually needed. This means that PL/I calls are as simple and efficient as they could be in any language using the Multics standard call.

Accessing of Data

PL/I object code addresses all data, including members of adjustable aggregates, directly through the use of efficient in-line code. If the

address of the data is constant, it is computed at compile time. If it is a mixture of constant and variable terms, the constant terms are combined at compile time. Dope and specifiers are never used to address or allocate data. They do not exist in PL/I object programs.

String Operations

All string operations are done by in-line code or by "tsb" type subroutinized code. No descriptors or calls are produced for string operations. The substr builtin function is implemented as a part of the normal addressing code and is therefore extremely efficient.

String Temporaries

String temporaries or dummies are designed in such a way that they appear to be both a varying and a non-varying string. This means that the programmer does not need to be concerned with whether a string expression is varying or non-varying when he uses such an expression as an argument.

All string temporaries are stored in the stack. The stack is extended during the execution of each statement by the amount necessary to hold the temporaries of that statement. The allocate and free machinery is never employed for string processing.

Varying Strings

The PL/I implementation of varying strings uses a new data format which consists of an integer followed by a non-varying string whose length is the declared maximum of the varying string.

The integer is used to hold the current size of the string in bits or characters.

Using this new data format, operations on varying strings are just as efficient as operations on non-varying strings. No epilogue is needed to free automatic varying strings. PL/I will never create an epilogue for any reason.

On Conditions

On conditions, including the condition prefix, are implemented by PL/I in a very efficient manner which uses the stack rather than the Multics condition machinery. This use of the stack means that as long as the programmer wishes the scope of an enabled condition to be within his own ring or set of programs he need not use the Multics machinery. If he wishes the effects of the Multics machinery, he must call it directly.

Label Variables

The PL/I compiler implements and believes the label (x,y..) form of the label attribute. If all of the identifiers in the label list appear as labels in the same block as the declaration, then the label variable will be assumed to have only values from that block.

This means that all transfers to this label variable which occur in this block will transfer to the text segment location without trying to pop the stack via the unwinder.

The label ( ) attribute should be used to replace the initialize label array of EPL. It has the advantage of being a general purpose variable which followes all the rules of the language.

Label variables may be initialized by an initial attribute or by the use of the following form:

```
dcl  lab(3)  label;

     lab(1):  ...
     lab(2):  ...
     lab(3):  ...
```

General Features

The PL/I compiler obeys the rules of the PL/I language as defined by Y33-6003-0. It behaves in a consistent and defined manner for all legal programs. The use of features which are not yet implemented or which are temporarily restricted will result in a meaningful diagnostic. The compiler can produce approximately 350 unique diagnostics. It will also optionally produce a complete list of all symbols used in the program with all of their declared and derived attributes. Items which have not been declared by a declare statement or by use as labels will be listed separately.

Object Code Efficiency

The PL/I compiler performs a great deal of analysis at compile time in an attempt to produce efficient code for all language features. The notion of an "efficient subset" which has developed out of experience with the EPL implementation is not valid for the PL/I implementation. Certain constructs are more efficient than others but there exists no sharp division between efficient and inefficient features. This phenomenon of EPL is due primarily to the use of dope, specifiers, allocate, free, epilogue, etc. for all but a limited set of cases. The PL/I compiler

does a good job with all language features. The user is urged to expand his coding style to take full advantage of the language. A programmer's guide to efficient PL/I object code will be published at a later date.

The Object Code Design

The compiler develops a complete expansion of the program in terms of its internal representation. In this process all accessing operations are brought out to the same level as user written explicit computations. A future version of the compiler will perform general optimization over this expanded code eliminating common subexpressions and removing code from loops. The internal representation is translated by the code generator into a set of macros which are expanded into 645 instructions in such a way as to make the text segment as short as possible. Extensive use is made of out of line sequences transferred to via "tsb" type instructions. This strategy results in the subroutinization of invariant code and the reduction of page faults, while retaining most advantages of in line code.

The code generator produces text, link and symbol segments directly. An assembly-like listing is also optionally produced.

SUMMARY OF PL/I FEATURES

1. Epilogues are never created.
2. Varying strings are implemented just as efficiently as non-varying strings.
3. Dope and specifiers are never created.
4. All arguments are passed as efficiently as EPL passes arithmetic scalars.
5. String temporaries are designed to appear as both varying and non-varying strings.
6. No calls are used to implement string operators. The substr builtin function is expanded into normal accessing code and is therefore very efficient.
7. The compiler obeys the rules of the language as defined by Y33-6003-0. It issues diagnostics and provides other useful information about the program being compiled.
8. The object code is very efficient and makes heavy use of out of line "tsb" subroutines located in an operator segment. This strategy reduces the number of page faults but does not increase the execution time by any significant amount.

? First version ...

APPENDIX 1

LANGUAGE FEATURES NOT IMPLEMENTED IN THE FIRST VERSION OF THE PL/I COMPILER

1. All input/output features including all related statements, declarations, on conditions and builtin functions.
2. Sterling data and pictured data.
3. Tasking - all related options, declarations, on conditions and builtin functions.
4. Scaled fixed point arithmetic.
5. Complex arithmetic.
6. Precision controlled arithmetic.
7. Decimal arithmetic is implemented as binary with the appropriate conversion of the declared precision.
8. Controlled storage class.
9. The attributes: defined, position, like, cell, generic.
10. Conversion between character string and arithmetic and between arithmetic and character string.
11. Aggregate expressions and array cross sections.
12. Check and size condition prefix.
13. Some builtin functions are omitted but the EPL subset is available.
14. Division of fixed point values must be done using the divide function.
15. Prologue dependencies are not resolved. Values available upon entry to a block do not include values declared as automatic in that block.