Published: 5/5/67

Identification

The efficient accessing of data.

James F. Gimpel

Purpose

The purpose of this section is to indicate how source language decisions can influence the size and complexity of machine code needed to access data.

Abstract

Three major themes are sounded. First, data should be made adjustable only with good reason and with care. Second, the sheer number of instructions needed to access certain kinds of packed bit strings indicates that special attention be paid to the design of packed aggregates. Third, the inherent inefficiency of accessing certain kinds of parameters implies that the selection of arguments to be passed across call boundaries should not ignore efficiency considerations.

Introduction

We regard as data, for the purpose of this section, strings, arithmetics, and pointers. We are ignoring label and area data, as their effect on efficiency is relatively small.

Let alpha, beta, gamma and delta be arbitrarily qualified names restricted only in that they do not have pointer qualification. Then a PL/I statement of the form:

```
alpha = beta + gamma -> delta;
```

is translated into assembly language as four separate data accesses. One to load the value of beta into an active register (the "a" or "aq" register), one to load a base register (bp) with the value of gamma, another to add the value of delta to the active register (using, of course, the previously loaded base register) and finally one to store the contents of this active register into alpha.

A data access can be more or less complex depending on the extent of qualification and, as importantly, the degree of adjustability of the data item with respect to the first physical location of its containing aggregate.* For example, in the structure,

```
dcl 1 alpha,
      2 beta fixed,
      2 gamma (-3:n) fixed,
      2 delta fixed;
```

---

* In most cases, the compiler has little difficulty in obtaining the first physical location of an aggregate; but, if any of its parts is adjustable, the compiler has no idea where the last location of the aggregate lies. This asymmetric with respect to storage allocation weaves an asymmetric thread through much of this section.

the offset of element alpha.beta from the first physical location of the aggregate alpha is known to the compiler (viz. 0); the offset of alpha.gamma (i) is known given the value of i (viz. 4+i); the offset of alpha.delta is not known and dope must be interrogated to determine this value. Those elements of an aggregate for which this offset can be computed without referencing dope are called directly-addressable (a formal procedure to determine whether a data item is directly-addressable is given in the appendix). As another example, the elements of the array

```
declare   alpha (n:25);
```

are not directly addressable because the lower bound in addition to the subscript must be known in order to access a scalar element of alpha.

Code needed to access directly-addressable data is substantially more compact than code needed to access data which is not directly addressable. There are a few simple rules which, if observed at the source level, serve to increase the number of directly-addressable data items.

Definition: An adjustable data item is an array with an adjustable bound or a string with an adjustable length, or an aggregate containing one of these. Note that varying strings are not adjustable; they occupy a fixed number (viz. two) of words in a structure.

Rule 1: If a structure has adjustable data items, put the adjustable items last.

Example:

```
Faster          declare  1 alpha,
                            2 beta  bit (12),
                            2 gamma bit (n);

Slower          declare  1 alpha,
                            2 gamma bit (n),
                            2 beta  bit (12);
```

Rule 2: If an array has an adjustable bound, make it the first upper bound (if possible).

Example:

```
Faster          declare  alpha (0:n) fixed;
Slower          declare  alpha (-n:0) fixed;
```

Example:

```
Faster          declare  alpha (n,15) fixed;
Slower          declare  alpha (15,n) fixed;
```

Rule 3: Do not use an asterisk as a bounds indicator if only the first upper bound is adjustable.

Example:

```
Faster          a:  proc (p,n);
                    declare  alpha (n) based (p);
                    etc.
Slower          a:  proc (alpha)
                    declare  alpha (*); etc.
```

It is a weakness of the language (PL/I) that a bound of the form (1:*) cannot be given.

The Four Part Translation Sequence

A data access to a name of the form

$$A_1.A_2.\ \cdots.A_n$$

where each A_i is a possibly subscripted variable is decomposed into four distinct phases.

1. Subscript computation
2. Subscript multiplying
3. Address-preparation
4. Operation

These phases are not necessarily contiguous. Subscript computation for all data accessing is done immediately. Parts 2, 3, 4 are contiguous for directly-addressable data items. The first two are null if no subscripts are present. The third is null in certain cases and may be elaborate in others. The fourth is always present. It may be one instruction like the load of a fixed point number or it may be several instructions like the store of a packed bit string. For example

```
declare  1 alpha (20) static,
           2 beta fixed,
           2 gamma fixed;
declare (n,m) fixed;
        alpha (n+m).gamma = 13;
```

This last statement is an assignment statement consisting of two data accesses, a load and a store. The first access consists of simply loading the a-register with 13. Thus parts 1, 2, 3 for this data access are null; the accessing associated with storing illustrates the 4-part translation sequence indicated above.

1. Add n and m and store into a temporary (done before the load of 13).
2. Multiply the contents of this temporary by 2 and load into XR6 (done by a subroutine).
3. Set base pair bp pointing to the first location of static storage for this segment (done by eapbp lp|.is,*)
4. sta bp|alpha.alias+4-2+1,6

All except perhaps the last is self-evident. Alpha.alias is a symbol of the form xx____ which is set to equal the offset from the static storage region associated with this procedure to the location reserved for the aggregate alpha. The +4 represents the size of the specifier which is allocated into the first four locations of this region. The quantity -2 represents the offset to the virtual origin of the array (the hypothetical element with 0 subscripts [see the appendix]). The +1 represents the offset of element gamma from the start of an alpha-type structure. The ,6 represents the effect of the subscript. Note that structure

accessing for directly-addressable data items becomes cheap. The assembler does the work and not the compiled code.

Data Accessing Without Subscripts

If a qualified name contains no subscripts then parts 1 and 2 of the data access are null. The whole data access consists of an address preparation and an operation.

Address-Preparation

Address-preparation code is used to place the machine in a state such that the data item of interest can be referenced in the next instruction.

In general, the address-preparation code depends only on storage class; the number of instructions for each class is given in Table 1.

As indicated above, the accessing for controlled storage does not include loading a base pair with the expressed pointer. The number of instructions to do this is, in turn, subject to the same consideration as other data accesses.

Summarizing the results of Table 1, all storage classes lead to a small number of address-preparation instructions except parameters with specifiers (i.e., strings, structures and arrays).

Operation

Loads and stores and many operations with aligned scalars take one instruction. Access to a pointer used to

TABLE 1

Number of instructions in the address-preparation sequence (for directly accessible data items).*

Storage Class of Outer Aggregate	Number of Instructions
A. Automatic	
(1) not adjustable	0
(2) adjustable	1
B. Static	
(a) internal	1
(b) external	0
C. Based	0
D. Constants	0
E. Parameters	
(a) unpacked aggregates	5
(b) strings	5 in line plus 10 in sub-routine
(c) parameters w/o specifiers	2

* There is an extra instruction for automatic and parameter data if the access is made in a block other than where the declaration is made.

qualify a based variable takes one instruction. The trouble srots are with strings. Strings of unknown length or strings whose length is greater than 36 bits or varying strings are handled by runtime subroutines. Packed strings, depending on their position within a word, are more difficult to access than aligned strings. Table 2 gives the number of instructions needed to load and store packed strings, aligned strings and parameter strings. Incidentally, the algorithm which the EPL compiler uses to pack and align strings is not commonly known and is given in the appendix.

As Table 2 indicates, loading and storing of parameter strings is even more awkward than with packed strings and no improvement in the situation is anticipated. The unholy solution is to assign the value of otherwise frequently used string parameters to strings in automatic storage.

Although a programmer cannot explicitly declare a structure to be packed, he may, by insertion of a foreign element (e.g., an arithmetic variable) into an otherwise packable structure, cause the structure to become unpacked. By this and other techniques the programmer has considerable control over packing and unpacking. This raises the following question: should he pack? For example:

TABLE 2

Minimum number of instructions needed to carry out primitive operations on nondimensioned strings whose length is known and less than 36 bits.

Type of String	Instructions to Load	Instructions to Store
A. aligned	2	1
B. within a packed aggregate and		
(1) left adjusted in a machine word	2	3
(2) wholly contained within a word but not (1)	3	4
(3) straddling a word boundary	4	6
C. parameter		
(1) 1 bit long	3	4
(2) more than 1 bit long	6	12

```
declare  1 alpha,
           2 beta_1  bit (18),
           2 beta_2  bit (18),
           2 beta_3  bit (18),
           2 beta_4  bit (18);
```

is packed and we save two words of storage. Since the elements were declared, presumably they are somewhere, in somebody's program, each accessed. Assuming each is loaded just once and stored into just once (surely modest

figures) we lose, by Table 2, a total of 16 instruction locations, not to mention increased accessing time. More frequent accessing more grossly outbalances the situation; as would bit strings more poorly coordinated with word boundaries.

This is not to imply categorically that packing always leads to inefficiencies. It could be that there are many repetitions of the packed data, such as in arrays* or, for example, because all entries in a file directory have the same format. In such cases it may be that there are far more instances of the data being accessed in core at any given moment than there are instructions to access the data.

Another good reason to pack is to faithfully represent data already formatted by the hardware such as the DCWs in the I/O system.

It is therefore difficult to give general rules to cover all situations. It is almost never a good idea to have packed data in automatic storage but beyond that maxim all that can be stated is that each situation should be evaluated in its own merits. Table 2 should be helpful in this regard.

---

* Arrays of packed data deserve special caution. They are treated in the next section.

Assuming we are packing, what are good rules to follow:

Rule 1: Try not to straddle word boundaries.

Example:

```
Faster          declare  1 alpha,
                         2 beta  bit (36),
                         2 gamma bit (26),

Slower          declare  1 alpha,
                         2 gamma bit (26),
                         2 beta  bit (36),
```

Rule 2: The most frequently accessed strings should be left adjusted in a word.

Example: Suppose beta is used far more frequently than gamma.

```
Faster          declare  1 alpha,
                         2 beta  bit (18),
                         2 gamma bit (18),

Slower          declare  1 alpha,
                         2 gamma bit (18),
                         2 beta  bit (18);
```

Data Accessing With Subscripts

When subscripts appear in a name we enjoy the fullblown 4-part data accessing sequence mentioned earlier and consisting of

(1) Subscript computation

(2) Subscript multiplication

(3) Address-preparation

(4) Operation

If the array is directly addressable, the address preparation is identical to the nonsubscripted case. The operation for aligned data is identical except for modifying the address with XR6. Arrays of packed strings as well as (1) and (2) will be treated in detail here.

Subscript Computation

Subscripts, generally speaking, are computed and stored in a temporary. In the current compilers this is done in all cases except when the subscript is an integer variable in automatic storage used at the block level in which declared.

Example:

Faster

```
a: proc;
   declare  alpha (100) fixed;
   .
   .
   .
 b: begin;
    declare  i fixed;
    .
    .
    .
  c: do i = 1 to 100;
     .
     .
     .
     alpha (i)
     .
     .
     .
   end c;
  .
  .
  .
end b;
.
.
.
end a;
```

Slower is to leave out the declaration of i.

Explanation: If i is not specifically declared in the inner block, b, i is by default declared in the outer block. Then not only does it require an extra instruction to access i within block b but when used as a subscript i is loaded from the previous stack frame, stored into a temporary in the current stack frame and is reloaded at subscript multiplication time.

Subscript Multiplication

For arrays whose element size is one word, the "subscript multiplication" code consists a load of an index register (very cheap).

If, on the other hand, the element size is not one word, some nontrivial subscript multiplication must be performed. This is done by subroutine but the whole thing takes three instructions of in-line code not to mention the execution time.

Rule 1: All things being equal, make array sizes one word.

Example:

```
Faster      declare  1 alpha (100),
                       2 beta  bit (18),
                       2 gamma bit (18);

Slower      declare  1 alpha,
                       2 beta (100) bit (18),
                       2 gamma (100) bit (18);
```

Example:

```
Faster      declare  1 alpha,
                       2 beta  (100) fixed,
                       2 gamma (100) fixed;

Slower      declare  1 alpha (100),
                       2 beta  fixed,
                       2 gamma fixed;
```

The next most preferred size of an array element is a multiple of words. This is always the case unless the array is packed.

Rule 2: All things being equal, and failing the previous rule, make the array size a multiple of words.

Example:

```
Faster      declare  1 alpha (100),
                       2 beta  bit (18),
                       2 gamma bit (54);

Slower      declare  1 alpha,
                       2 beta  (100) bit (18),
                       2 gamma (100) bit (54);
```

Rule 3: Failing the two previous rules coordinate array elements with word boundaries in as orderly a manner as possible.

Example:

```
Faster      declare  1 alpha,
                       2 beta  (0:5) bit (12),
                       2 gamma bit (1);
```

```
Slower          declare  1 alpha,
                         2 gamma bit (1),
                         2 beta (0:5) bit (12);
```

Rule 3 also makes good sense from the standpoint of program hygiene. For example, dumps are easier to look at. The following rule is considerably less obvious; statements supporting the conclusion follow the rule.

Rule 4: In packed arrays attempt to align the virtual origin of the array with a word boundary.

Example:

```
Faster          declare  1 alpha,
                         2 beta bit (12),
                         2 gamma (5) bit (12);
```

Slower declare 1 alpha,
2 gamma (5) bit (12),
2 beta bit (12);

Explanation: In the first case the virtual origin* of gamma (i.e., alpha.gamma (0)) is aligned with a word boundary, viz. the first physical location of alpha. In the second case, it falls as offset 24 from a word boundary.

Operation (Packed Arrays)

If the array is packed and the multipliers are multiples of 36 bits then the operation code required to load and store strings is the same as indicated in Table 2. That is, the operation code is exactly as if the string or structure were not dimensioned. If, on the other hand, one or more of the multipliers is not a multiple of 36 bits, complexities are introduced.

Definitions: Let $m_1, m_2, \ldots, m_n$ be the multipliers† used in accessing a packed string. Then the precessing resolution R is defined as the greatest common divisor of $m_1, m_2, \ldots, m_n$ and 36. That is

---

* See appendix for explanation of virtual origin.

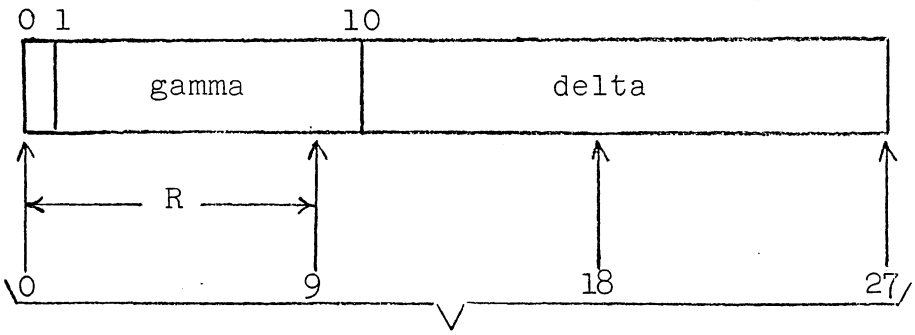
† Multipliers for arrays are discussed in the appendix. For packed arrays they are measured in bits.

$$R = \gcd(m_1, m_2, \ldots, m_n, 36)$$

Example:

```
declare 1 alpha (100),
          2 beta bit (1)
          2 gamma bit (9),
          2 delta bit (17);
```

Then the multiplier for this array is 27 bits. The precessing resolution R = gcd (27,36) = 9. This means that word boundaries can potentially crop up every nine bits. This is indicated schematically in Figure 1.



Virtual Word Boundaries

Figure 1

Note that the structure layout was not startlingly good. Since gamma straddles a virtual word boundary, it can sometimes straddle a real word boundary (for some subscripts). Such strings are especially difficult to access and we have a special name for them.

Definition: An array of strings is said to be contiguous if for all subscripts (not necessarily within the bounds of the array) no string scalar straddles a word boundary; it is called discontiguous if every such scalar straddles a word boundary; it is called idiotic if some of the scalars do and some of the scalars do not.

For every arrayed bit string there is a compile-time offset of the 0^{th} element from the base of the aggregate.* Call this offset C; it is measured in bits. In the above example the offset C for beta is -27, for gamma it is -26, and for delta the offset is -17. Word offsets do not concern the compiler but bit offsets do. Define

$$c = \text{mod}\ (C,36)$$

Then in the previous example

for beta $c = \text{mod}\ (-27,36) = 9$

for gamma $c = \text{mod}\ (-26,36) = 10$

for delta $c = \text{mod}\ (-17,36) = 19$

Let ℓ be the length of a scalar member of a string array.

Remark: Let $K = \ell + \text{mod}\ (c,R)$. Then a string is

(a) contiguous if $K \leq R$

(b) discontiguous if $K > 36$

(c) idiotic if $R < K \leq 36$.

Hence large precessing resolutions tend to squeeze out idiotic strings. In the limiting case when

---

* When this compile-time offset is added to the base of the aggregate, the virtual origin is obtained (see Section BN.9.01a).

the precessing resolution equals 36, we have no precessing and hence no idiotic strings.

In the previous example for beta,

$$K = \ell + \text{mod}\ (c,R) = 1 + \text{mod}\ (9,9) = 1$$

and hence beta is contiguous (as are all strings of length 1). For gamma,

$$K = \ell + \text{mod}\ (c,R) = 9 + \text{mod}\ (10,9) = 10.$$

K is greater than R but less than 36 and hence gamma is idiotic. For delta,

$$K = \ell + \text{mod}\ (c,R) = 17 + \text{mod}\ (19,9) = 27$$

and so delta is also idiotic.

Definition: A packed array of strings is synchronous if $c < R$.

The semantics of this definition derive from the following: Let E be the result in bits of multiplying the subscripts by their respective multipliers. Let V be the word containing the virtual origin of the array. Then, if the packed array is synchronous, the first bit of the string scalar will always fall in word $V + [E/36]$, and thus the array is spacially synchronized with the subscript. The compiler can compress code if an array is synchronous independently of whether it is contiguous, discontiguous or idiotic.

TABLE 3

Minimum* instructions required for primitive operations on elements of packed arrays. The strings are assumed to be nonadjustable, nonvarying and no longer than 36 bits.

Type of String	Operation Load	Store
1. Synchronous		
(a) contiguous	3	6
(b) discontiguous	4	8
(c) idiotic	6	12
2. Not Synchronous		
(a) contiguous	5	8
(b) discontiguous	6	10
(c) idiotic	8	14

* These numbers seem to be at or near the theoretical lower limits given the machine and the instruction set.

Table 3 indicates the number of instructions required to perform primitive operations on these strings given the category in which they fall.

Returning to our previous example we find that for beta we need five and eight instructions to load and store respectively. For both gamma and delta we need eight and fourteen to load and store. If we were to rearrange the order of beta and gamma, then both gamma and beta would be contiguous. If, moreover, we changed the bounds of the array to

```
declare  1 alpha (0:99),
           2 gamma bit (9),
           2 beta bit (1),
           2 delta bit (17);
```

we will have made gamma synchronous. We now require only three and six instructions to load and store gamma compared to the previous eight and fourteen. The number of instructions required to access beta and delta remain unchanged, however.

This is not very much of an improvement. The situation alters dramatically, however, if a dummy bit string long enough to fill out a word is inserted into the structure. For example:

```
declare  1 alpha (100),
           2 beta  bit (1),
           2 gamma bit (9),
           2 delta bit (17),
           2 filler bit (9);
```

The array size is now one word so that we save about 10 instructions in the subscript multiplication phase of each data access. In addition, by referring to Table 2, beta can be loaded and stored in two and three instructions respectively; whereas gamma and delta will both require three and four. This represents a dramatic saving in each data access offset only by the fact that 25 locations of storage have been lost through a partial unpacking of the structure.