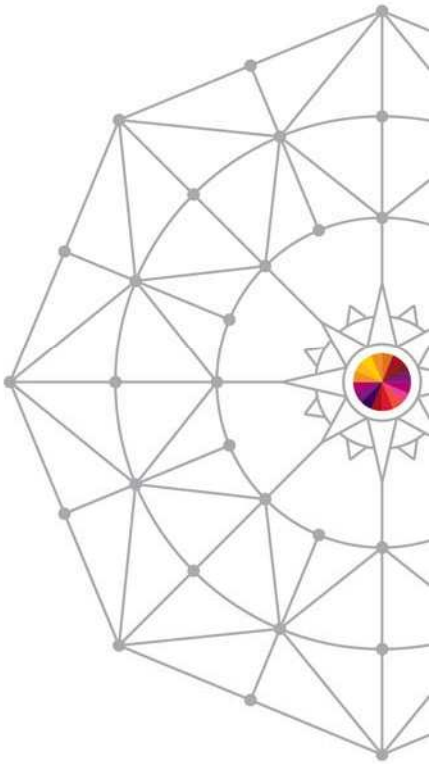
S16152 - Coding in COBOL for optimum performance



Tom Ross IBM

August 7, 2014

#SHAREorg

SHARE is an independent volunteer-run information technology association that provides **education**, professional **networking** and industry **influence**.

SHARE in Pittsburgh 2014

Title: Coding in COBOL for optimum performance

- Compiler options
- Dealing with data types
- Dealing with data items
- COBOL statements
- Sign processing

Finding inefficient COBOL coding

- Future: COBOL V5 may add flagging via RULES option
  - (Similar to the PL/I RULES compiler option)

- Inefficient compiler options
- Inefficient use of data types in calculations
- Inefficient use of data types in specific statements
- Inefficient use of data items

- You can find these manually today

Inefficient Compiler options

- NOBLOCK0
  - Use BLOCK0!
- NOFASTSRT
  - Use FASTSRT!
- SSRANGE
  - Use NOSSRANGE
  - If range checking desired you might use loop control tests to minimize performance impact
    - SSRANGE is much easier to turn on and off

Inefficient Compiler options

- TRUNC(STD)
  - Should never be used! Use TRUNC(OPT)
- TRUNC(BIN)
  - Recommend TRUNC(OPT) and COMP-5 for special case data items
- Performance considerations using TRUNC:
  - On the average, TRUNC(OPT) was 10% faster than TRUNC(BIN), with a range of 80% faster to equivalent.
  - On the average, TRUNC(STD) was 5% faster than TRUNC(BIN), with a range of 75% faster to 60% slower.
  - On the average, TRUNC(OPT) was 4% faster than TRUNC(STD), with a range of 64% faster to equivalent.

Inefficient Compiler options

- NUMPROC(NOPFD)
  - NUMPROC(PFD) is faster
- Performance considerations using NUMPROC:
  - On the average, NUMPROC(PFD) was 1% faster than NUMPROC(NOPFD), with a range of 21% faster to equivalent.
- Investigate your signed data in External Decimal and Packed-decimal
  - How can you do that? It is not easy, but if you really want to…
  - If NUMERIC with NUMPROC(PFD) will tell you if you need NOPFD
    1. Create a sniffer program from existing programs to access all of the data
    2. Use IF NUMERIC (CLASS TEST) for every data item in files and DBs
    3. If 100% NUMERIC, change to NUMPROC(PFD)!

Investigate whether you can use NUMPROC(PFD)


```
*> Compile ‘sniffer’ with NUMPROC(PFD)
EXEC SQL SELECT Ext-Dec Packed-Dec
           INTO …  :X,  :Y            END-EXEC

If X NUMERIC and Y NUMERIC Then
  Display ‘Use NUMPROC(PFD)!’
  Move 2 To Return-Code
Else
  Display ‘Sorry, use NUMPROC(NOPFD)!’
  Move 16 To Return-Code    *> Or call
  CEE3ABD
  Stop Run
End-If
```

Dealing with data types

- Calculations using numeric USAGE DISPLAY data items
- Perform VARYING identifier-2 data items defined with USAGE DISPLAY
- Perform VARYING operands with different data types
- Accessing a table with USAGE DISPLAY subscripts
- MOVEs and COMPUTEs that convert data types within loops

Dealing with data types

- Calculations using numeric USAGE DISPLAY data items
  Examples:

```
Set index_name_I Up By Usage_display_x
If Usage_display_y > Usage_display_z *
                              Usage_display_w
Compute  x = y ** z
```

  ADD, SUBTRACT, MULTIPLY,DIVIDE

- Use BINARY or PACKED-DECIMAL

Dealing with data types

- Perform VARYING identifier-2 data items defined with USAGE DISPLAY

```
PERFORM VARYING Usage_display_x
        FROM something BY something
        UNTIL something_else
END-PERFORM

PERFORM my_section VARYING
Usage_display_y
        FROM something BY something
        UNTIL something_else
END-PERFORM
```

PERFORM VARYING with different data types


```
PERFV1.
    PERFORM OTHER-PARA  VARYING EXT-DEC
        FROM PACKED BY BIN3
        UNTIL EXT-DEC   > FLOAT
    END-PERFORM

PERFV1.
    PERFORM OTHER-PARA  VARYING EXT-DEC
        FROM EXT-DEC2 BY EXT-DEC3
        UNTIL EXT-DEC   > EXT-DEC4
    END-PERFORM

PERFV3.
    PERFORM OTHER-PARA VARYING Bin
        FROM Bin2 BY Bin3
        UNTIL Bin   > Bin4
    END-PERFORM
```

PERFORM VARYING with different data types

- Measurements using COBOL V4.2 and V5.1.1
  - W/loop control set to 1000
  - PERFORM VARYING executed 100,000 times
- PERFV1: All operands different types
  - **V4.2 CPU: 0 HR 00 MIN 02.88 SEC**
  - **V5.1 CPU: 0 HR 00 MIN 02.23 SEC**
- PERFV2: All operands external decimal
  - **V4.2 CPU: 0 HR 00 MIN 01.59 SEC**
  - **V5.1 CPU: 0 HR 00 MIN 01.17 SEC**
- PERFV3: All operands BINARY
  - **V4.2 CPU: 0 HR 00 MIN 00.99 SEC**
  - **V5.1 CPU: 0 HR 00 MIN 00.30 SEC**

Dealing with data types

- Accessing a table with USAGE DISPLAY data items

```
PERFORM 1000 TIMES
    Add 1 to U_disp_x
    Move stuff To Table_element (
U_disp_x)
END-PERFORM
```

- Use BINARY or INDEX-NAMEs:

```
02 Table_element OCCURS 1000 Times
                      Indexed By
Index_Name_1.
```

Dealing with data types

- MOVEs and COMPUTEs that convert data types within loops

```
        PERFORM blah VARYING blah_blah

blah.
     Move Binary_b To Usage_display_x
     Compute Binary_b =
             Binary_b * Packed_C +
Float_f
```

- Avoid conversions if possible
- Use EXTERNAL DECIMAL for output only

Dealing with data types

- If IBM provided a DFP (Decimal Floating Point) data type, would you use it?
  - DFP is much faster than other data types
  - Is it possible to change a data type for stored data? DB2, IMS?
- COBOL V5 already uses DFP instructions
  - For converting External Decimal before calculations
  - For doing calculations with large Packed-Decimal data items

Dealing with data items

- Alphanumeric data item inadvertent padding
- Numeric data item truncation
- Numeric data item overflow
- Initialization of data items

Dealing with data items

- Alphanumeric data item inadvertent padding

```
Move Cust_Name to Cust_record     <* These MOVEs both put
Move Cust_Name to Cust_rec_name   <* the name in bytes 1-40.
```

- Looks harmless, right?

```
77 Cust_Name                  Pic x(40).
01 Cust_record.
   05 Cust_rec_Name            Pic x(40).
   05 Cust_rec_Account         Pic 9(30).
   05 Cust_rec_Address         Pic x(50).
   05 Cust_rec_Policy          Pic 9(15).
   05 Cust_rec_email           Pic x(25).
   05 Cust_rec_other.
      10  Cust_other1          Pic x(140).
      10  Cust_other1          Pic x(200).
      10  Cust_other1          Pic x(500).
```

Dealing with data items

- These moves are quite different!

```
Move Cust_Name to Cust_rec_name   <* Moves 40 bytes
Move Cust_Name to Cust_record     <* Moves 1000 bytes!
```

- The extra bytes moved cost CPU cycles

```
77 Cust_Name  Pic x(40).
01 Cust_record.
   05 Cust_rec_Name        Pic x(40).
   05 Cust_rec_Account     Pic 9(30).
   05 Cust_rec_Address     Pic x(50).
   05 Cust_rec_Policy      Pic 9(15).
   05 Cust_rec_email       Pic x(25).
   05 Cust_rec_other.
      10  Cust_other1      Pic x(140).
      10  Cust_other1      Pic x(200).
      10  Cust_other1      Pic x(500).
```

Dealing with data items

- Numeric data item truncation
  - DIAGTRUNC compiler option
  - Can help find coding ‘errors’

```
77 Binary_b PIC S9(9) BINARY.
77 Binary_c PIC S9(4) BINARY.

77 Packed_p PIC S9(7)V9(2) COMP-3.
77 Packed_q PIC S9(5)V9(2) COMP-3.

    Move Binary_b to Binary_c
    Move Packed_p to Packed_q
```

Dealing with data items

- Numeric data item overflow
- COBOL normally either ignores decimal overflow conditions or handles them by checking the condition code after the decimal instruction.
- ILC (Inter Language Communication) triggers switch to a language-neutral or ILC program mask
  - This ILC program mask enables decimal overflow (COBOL-only program mask ignores overflow)
  - COBOL code also tests condition after decimal instructions
  - Overflows cause program calls to condition handling
  - Overflows can be very common in COBOL
- Result: COBOL math can get bogged down

Dealing with data items

- Numeric data item overflow
- Performance considerations for a mixed COBOL with C or PL/I application with COBOL using PACKED-DECIMAL data types in 100,000 arithmetic statements that cause a decimal overflow condition (100,000 overflows):
  - Without C or PL/I: 0.040 seconds of CPU time
  - With C or PL/I: 1.636 seconds of CPU time

Dealing with data items

- Using XML or calling C now common, forcing ILC
- What to do? Make receiving data items larger … or if you can't change your data definitions …
- ON OVERFLOW for performance!

```
Compute  x = y ** z
    On Overflow CALL 'CEE3ABND'
End-Compute

Add 1 to U_disp_x
    On Overflow Write Error-record-info
End-Add
```

Dealing with data items

- ON OVERFLOW for performance?
- With ON OVERFLOW phrase, compiler generates code to check for the condition. If the condition happens, thousands of instructions and LE condition management overhead are avoided
- This should be especially considered for programs that use
  - ILC with C or PL/I or
  - XML PARSE or XML GENERATE or
  - Enterprise COBOL V5!
- All of these cases involve ILC
  - Enterprise COBOL V5 always uses C

Dealing with data items

- Best performance and usability would be achieved with larger data items to avoid overflow condition
- But ON OVERFLOW can be an alternative if you can only change the program you are working on or if data areas are not under your control

Dealing with data items

- Initialization of data items
  - Runtime option STORAGE(00) could be wasting lots of instructions
    - STORAGE(00) is almost a standard!
    - STGOPT (or older OPTIMIZE(FULL) could help
  - Initialize only those variables that need to be set
    - Use XREF compiler option and listings to see which ones need it
- INITIALIZE statement
  - Group MOVE faster than INITIALIZE for tables ?
  - Consider INITIALIZE for 1st element of table and then propagate that value to other elements of the table ?

INITIALIZE

```
01 WS-GROUP.
  02 WS-GROUP-TABLE OCCURS 1000 TIMES INDEXED BY T-IDX.
    05 WS1-COMP3 COMP-3 PIC S9(13)V9(2).
    05 WS2-COMP  COMP   PIC S9(9)V9(2).
    05 WS3-COMP5 COMP-5 PIC S9(5)V9(2).
    05 WS4-COMP1 COMP-1.
    05 WS5-ALPHANUM  PIC X(11).
    05 WS6-DISPLAY   PIC 9(13) DISPLAY.
    05 WS7-COMP2 COMP-2.

  . . .

  INITIALIZE WS-GROUP
```

INITIALIZE + MOVE

```
01 WS-GROUP.
  02 WS-GROUP-TABLE OCCURS 1000 TIMES INDEXED BY T-IDX.
    05 WS1-COMP3 COMP-3 PIC S9(13)V9(2).
    05 WS2-COMP  COMP   PIC S9(9)V9(2).
    05 WS3-COMP5 COMP-5 PIC S9(5)V9(2).
    05 WS4-COMP1 COMP-1.
    05 WS5-ALPHANUM  PIC X(11).
    05 WS6-DISPLAY   PIC 9(13) DISPLAY.
    05 WS7-COMP2 COMP-2.

  SET T-IDX TO 1
  INITIALIZE WS-GROUP-TABLE(T-IDX)
  PERFORM 999 TIMES
    SET T-IDX UP BY 1
    MOVE WS-GROUP-TABLE(1) TO WS-GROUP-TABLE(T-IDX)
  END-PERFORM
```

Group MOVE

```
01 WS-GROUP.
  02 WS-GROUP-TABLE OCCURS 1000 TIMES INDEXED BY T-IDX.
    05 WS1-COMP3 COMP-3 PIC S9(13)V9(2).
    05 WS2-COMP  COMP   PIC S9(9)V9(2).
    05 WS3-COMP5 COMP-5 PIC S9(5)V9(2).
    05 WS4-COMP1 COMP-1.
    05 WS5-ALPHANUM  PIC X(11).
    05 WS6-DISPLAY   PIC 9(13) DISPLAY.
    05 WS7-COMP2 COMP-2.


   Move X'00' To WS-GROUP

Ooops, what did I do wrong?
```

Group MOVE

```
01 WS-GROUP.
  02 WS-GROUP-TABLE OCCURS 1000 TIMES INDEXED BY T-IDX.
    05 WS1-COMP3 COMP-3 PIC S9(13)V9(2).
    05 WS2-COMP  COMP   PIC S9(9)V9(2).
    05 WS3-COMP5 COMP-5 PIC S9(5)V9(2).
    05 WS4-COMP1 COMP-1.
    05 WS5-ALPHANUM  PIC X(11).
    05 WS6-DISPLAY   PIC 9(13) DISPLAY.
    05 WS7-COMP2 COMP-2.
```

```
Move ALL X'00' To WS-GROUP
```

INITIALIZE + MOVE

- Well, I tried it with V4.2!
  - Each test PERFORMed 1,000,000 times
- INITIALIZE by itself:
  - CPU: 0 HR 00 MIN 02.37 SEC
- INITIALIZE + MOVE
  - CPU: 0 HR 00 MIN 04.13 SEC
- Group MOVE
  - CPU: 0 HR 00 MIN 05.18 SEC

- It turns out the V4.2 compiler generates INITIALIZE + MOVE already!

INITIALIZE + MOVE

- Then I tried it with V5.1.1!
  - Each test PERFORMed 1,000,000 times
- INITIALIZE by itself:
  - CPU: 0 HR 00 MIN 04.31 SEC
- INITIALIZE + MOVE
  - CPU: 0 HR 00 MIN 06.78 SEC
- Group MOVE
  - CPU: 0 HR 00 MIN 05.15 SEC

- The V5.1 compiler generates INITIALIZE + MOVE already, but slower than V4.2 … I will look into that!

INITIALIZE + MOVE

- I always thought INITIALIZE was slow
- Customers told me so and so did the COBOL Performance Tuning Paper:
- Performance considerations for INITIALIZE on a program that has 5 OCCURS clauses in the group:
  - When each OCCURS clause in the group contained 100 elements, a MOVE to the group was 8% faster than an INITIALIZE of the group.
  - When each OCCURS clause in the group contained 1000 elements, a MOVE to the group was 23% faster than an INITIALIZE of the group.

- I found differently!

COBOL Statements

- Move calculations outside of loops whenever possible
- SEARCH ALL
- Examples from clients

Move calculations outside of loops

```
      PERFORM blah VARYING blah_blah
  . . .
 blah.
* If tran processed after close of business
    If Function CURRENT-DATE (19:6)
         > 180000 Then
       Add 1 to effective-date
    End-If
```

Move calculations outside of loops


```
 77 tofday      PIC 9(8).
  . . .
   Move Function CURRENT-DATE (19:6)
                To tofday
   PERFORM blah VARYING blah_blah
  . . .
 blah.
* If tran processed after close of business
   If tofday > 180000 Then
      Add 1 to effective-date
   End-If
```

Move calculations outside of loops AND use more efficient data type!

```
 77 tofday      PIC 9(8) BINARY.
  . . .
   Move Function CURRENT-DATE (19:6)
                To tofday
   PERFORM blah VARYING blah_blah
  . . .
 blah.
* If tran processed after close of business
   If tofday > 180000 Then
      Add 1 to effective-date
   End-If
```

SEARCH ALL vs SEARCH

- SEARCH - binary versus serial
  - We got the question: Is there a point (a small enough number of items searched) where a serial search is faster than a binary SEARCH?
- Answer: it depends on your data! I tried a set of tests…
  - Using a binary search (SEARCH ALL) to search a 50-element table was 343% slower than using a sequential search (SEARCH)
    - BSRCHXS: CPU: 0 HR 00 MIN 01.41 SEC
    - SRCHXS: CPU: 0 HR 00 MIN 00.41 SEC
  - Using a binary search (SEARCH ALL) to search a 100-element table was 100% slower than using a sequential search (SEARCH)
    - BSRCHSM: CPU: 0 HR 00 MIN 01.47 SEC
    - SRCHSM: CPU: 0 HR 00 MIN 00.73 SEC
  - Using a binary search (SEARCH ALL) to search a 1000-element table was 70% faster than using a sequential search (SEARCH)
    - BSRCHBIG: CPU: 0 HR 00 MIN 02.21 SEC
    - SRCHBIG: CPU: 0 HR 00 MIN 06.52 SEC

Coding tips from customer situations

- One customer found that COBOL performance was better than PL/I and wanted to start using only COBOL for new applications (they are 50/50 COBOL and PL/I)
- The customer wanted to have replacements for commonly used PL/I functions:
  - **`VERIFY`**
  - **`TRIM`**
  - **`INDEX`**
- When they tried to code these in COBOL they found they were too slow
- They asked me to try to do better…

Coding tips from customer situations


```
*   VERIFY PL/I function in COBOL using INSPECT: slow

    MOVE '02.04.2010' TO TEXT1

    MOVE TEXT1 TO TEXT2
    INSPECT TEXT2 REPLACING ALL '.' BY '0'

    IF TEXT2 IS NOT NUMERIC
        MOVE 'NOT DATE' TO TEXT1
    END-IF
```

Coding tips from customer situations


```
*   VERIFY PL/I function in COBOL using CLASS test:
*   40% faster

SPECIAL-NAMES.
          CLASS VDATE IS '0' thru '9' '.'.
.   .   .

  MOVE '02.04.2010' TO TEXT1

  IF TEXT1 IS Not VDATE Then
       MOVE 'NOT DATE' TO TEXT1
  END-IF
```

Coding tips from customer situations


```
*   TRIM PL/I function written in COBOL using INSPECT and
*   FUNCTION REVERSE: slow

   MOVE '   This is string 1   ' TO TEXT1
   COMPUTE POS1 POS2 = 0

   INSPECT TEXT1
        TALLYING POS1
         FOR LEADING SPACES
   INSPECT FUNCTION REVERSE(TEXT1)
        TALLYING POS2
          FOR LEADING SPACES
   MOVE TEXT1(POS1:LENGTH OF TEXT1 - POS2 - POS1) TO TEXT2
```

Coding tips from customer situations

```
*  TRIM PL/I function written in COBOL: 31% faster

  MOVE '   This is string 1   ' TO TEXT1
  PERFORM VARYING POS1 FROM 1 BY 1
        UNTIL TEXT1(POS1:1) NOT = SPACE
  END-PERFORM

  PERFORM VARYING POS2 FROM LENGTH OF TEXT1
          BY -1 UNTIL TEXT1(POS2:1) NOT = SPACE
  END-PERFORM

  COMPUTE LEN = POS2 - POS1 + 1
  MOVE TEXT1(POS1 : LEN) TO TEXT2 (1 : LEN)
```

Coding tips from customer situations

```
*   INDEX PL/I function written in COBOL: slow

    MOVE 'TestString1 TestString2' TO BUFFER

    COMPUTE POS = 0

    INSPECT BUFFER
            TALLYING POS
            FOR CHARACTERS
            BEFORE INITIAL 'TestString2'
```

Coding tips from customer situations

```
*   INDEX PL/I function written in COBOL: 83% faster

    MOVE 'TestString1 TestString2' TO BUFFER

    PERFORM VARYING POS FROM 1 BY 1
        UNTIL BUFFER(POS:11) = 'TestString2'
    END-PERFORM
```