COBOL performance: Myths and Realities

Speaker Name: Tom Ross
Speaker Company: IBM

Date of Presentation: August 10, 2011
Session Number: 9655

Agenda

- Performance of COBOL compilers - myths and realities
- Performance improvements over the years
- Highlights of updates to Performance Tuning Paper
- Coding tips

Myths and Realities

- Performance of COBOL compilers - myths and realities
  - IBM marketing materials imply performance improvements
    - Improved generated code is available in PL/I and C/C++
  - Wishful thinking adds to the misconception
  - IBM COBOL compilers are extremely efficient!
  - Dev process includes regular performance scrutiny
  - COBOL does run faster on newer processors

IBM Compilers exploit System z for Maximum Performance

- Compilers exploit new hardware instructions introduced by System z **(z9? z10?)**
- Code generated by the compilers is highly tuned for System z
- Boost in performance of applications running on System z

z/OS XL C/C++

- standards compliant C/C++ compilers to support porting code
- METAL C compiler option to support low-level programming

Enterprise COBOL for z/OS

- support for modernization of applications (XML support and Java support)
- integration with middleware such as CICS, DB2, and IMS

Enterprise PL/I for z/OS

- facilitates repurposing of existing business processes into new business models
- Integration with IBM middleware (CICS, DB2, and IMS)

- **135 new / changed Instructions (z196)**

Myths/facts about z196 and COBOL

- The 135 new and changed instructions were added or changed for many reasons, not just for performance
  - Examples:
    - Cryptographic facility instructions
    - Binary Floating Point (BFP) instructions
    - Decimal Floating Point (DFP) instructions
- The z196 processor processes all instructions faster than the z10 does, even old COBOL!
- IBM COBOL development is working on a new compiler design to make it easier to exploit new hardware instructions as they are introduced

Summary of new z196 instructions

- The IBM zEnterprise 196 provides a broad range of new facilities to improve performance and function:
  - High-word facility (30 instructions)
  - Interlocked-access facility (12 instructions)
  - Load/store-on-condition facility (6 instructions)
  - Distinct-operands facility (22 instructions)
  - Population-count facility (1 instruction)
  - Enhanced-floating-point facility (25 new, 30 changed instructions)
  - MSA-X4 facility (4 new, 3 changed instructions, new functions)
  - Etc.
- Potential for:
  - Significant performance improvement
  - Enhanced capabilities
  - Simpler code

Performance improvements over the years for COBOL compilers

- VS COBOL II
  - Many performance improvements over 6 releases result in very fast code produced by IBM COBOL compilers
- COBOL V2R2
  - Significant performance improvement in processing binary data with the TRUNC(BIN) compiler option
- COBOL V4R1
  - Performance of COBOL application programs has been enhanced by exploitation of new z/Architecture® instructions. The performance of COBOL Unicode support (USAGE NATIONAL data) has been significantly improved.

Performance tuning paper updated

- As the result of a SHARE requirement, we were able to apply resources to get the COBOL Performance Tuning Paper updated for COBOL V4R2
- The last time it was updated was for COBOL V3R1, 2001

- Online at:

http://www-01.ibm.com/software/awdtools/cobol/zos/library/

- New info since V3R1 version:
  - BLOCK0, XMLPARSE, INTERRUPT
- Updated section
  - CICS communication

Performance tuning paper updated

- BLOCK0 compiler option
  - New V4R2 option to change default behavior for QSAM output files
  - For 40 years, no BLOCK CONTAINS clause meant:
    - **BLOCK CONTAINS 1 RECORD**
    - The slowest possible!
    - Counterpoint: the file is always current
- BLOCK0 changes the compiler default for QSAM files from unblocked to blocked (as if **BLOCK CONTAINS 0** were specified) and thus gain the benefit of system-determined blocking for output files.

Performance tuning paper updated

- Specifying BLOCK0 activates an implicit BLOCK CONTAINS 0 clause for each file in the program that meets the following three criteria:
  - The FILE-CONTROL paragraph either specifies ORGANIZATION SEQUENTIAL or omits the ORGANIZATION clause.
  - The FD entry does not specify RECORDING MODE U.
  - The FD entry does not specify a BLOCK CONTAINS clause.

Performance tuning paper updated

- BLOCK 0 compiler option…results?
- Performance considerations using BLOCK0 on a program with a file that meets the criteria:
  - One program using BLOCK0 was 88% faster than using NOBLOCK0 and used 98% fewer EXCPs.

Performance tuning paper updated

- XMLPARSE compiler option
- There are 3 parsers in COBOL today
  - COBOL V3 parser, available in V4 as XMLPARSE(COMPAT)
    - Selected by compiler option
  - XMLSS non-validating parser (COBOL V4R1)
    - Selected by compiler option
  - XMLSS validating parser (COBOL V4R2)
    - Selected by compiler option + VALIDATING WITH clause
  - Do not change to XMLSS from V3 (COMPAT) parser unless you need the extra functionality!
    - Customer feedback and testing show it is a lot slower

Performance tuning paper updated

- XMLPARSE compiler option…Results?
- Performance considerations for XML PARSE example:
  - Five programs using XML PARSE were from 20% to 108% slower when using XMLPARSE(XMLSS) compared to using XMLPARSE(COMPAT).

Performance tuning paper updated

- INTERRUPT run-time option
- The 3R1 version of performance tuning paper did not cover this option
- The INTERRUPT option causes attention interrupts to be recognized by Language Environment. When you cause an interrupt, Language Environment can give control to your application or to Debug Tool.
- Performance considerations using INTERRUPT:
  - On the average, INTERRUPT(ON) was 1% slower than INTERRUPT(OFF), with a range of equivalent to 18% slower

Performance tuning paper updated

- SIMVRD run-time option…removed support!
- The SIMVRD option specifies whether COBOL programs use a VSAM KSDS to simulate variable-length relative organization data set. This support is only available with VS COBOL II through Enterprise COBOL Version 3 programs. Starting with Enterprise COBOL Version 4 programs, this support is no longer available.
- Performance considerations using SIMVRD:
  - One VSAM test case compiled with Enterprise COBOL 3.4 was 5% slower when using SIMVRD compared to NOSIMVRD.
- Those concerned with performance will not miss SIMVRD!

Performance tuning paper updated

- Program communication under CICS
- Choices: static CALL, dynamic CALL or EXEC CICS LINK
- In many cases EXEC CICS LINK can be replaced with COBOL dynamic CALL (similar separate load module characteristic)
  - DYNAM compiler option is not allowed for programs with EXEC CICS statements in CICS, so you must use CALL identifier to do dynamic CALL in these cases
- In some cases dynamic CALL cannot replace CICS LINK:
  - Cross systems EXEC CICS LINK
  - If subprograms ABEND or STOP RUN, they will stop the caller unless EXEC CICS LINK is used

Performance tuning paper updated

- Program communication under CICS
- Performance considerations using CICS (measuring call overhead only):
  - One test case was 446% slower using EXEC CICS LINK compared to using COBOL dynamic CALL with CBLPSHPOP(ON)
  - The same test case was 7817% slower using EXEC CICS LINK compared to using COBOL dynamic CALL with CBLPSHPOP(OFF)
  - The same test case was 1350% slower using COBOL dynamic CALL with CBLPSHPOP(ON) compared to using COBOL dynamic CALL with CBLPSHPOP(OFF)

Performance tuning paper updated

- To show the magnitude of the difference in CPU times between the above methods, here are the CPU times that were obtained from running each of these tests on our system and may not be representative of the results on your system.

'call' type	CPU Time (seconds)
EXEC CICS LINK	0.475
COBOL dynamic CALL CBLPSHPOP(ON)	0.087
COBOL dynamic CALL CBLPSHPOP(OFF)	0.006

Performance tuning paper updated

- COBOL normally either ignores decimal overflow conditions or handles them by checking the condition code after the decimal instruction. ILC triggers a 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 to use condition handling
  - Overflows can be very common in COBOL
  - Result: COBOL math can get bogged down

Performance tuning paper updated

- 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: .040 seconds of CPU time
  - With C or PL/I: 1.636 seconds of CPU time

Performance tuning paper updated

- XML GENERATE and XML PARSE result in bringing a C signature into your module - ILC!
- Solutions?
  - Ensure that your COBOL code does not encounter decimal overflow conditions
    - Larger data items
  - If XML processing is a special case, move XML processing into a different application
  - Process XML in separate enclaves or processes if possible
    - Examples: EXEC CICS LINK, SVC LINK

Performance tuning paper updated

- 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! (or maybe NO…)
- Performance considerations for search example:
  - Using a binary search (SEARCH ALL) to search a 100-element table was 15% faster than using a sequential search (SEARCH)
  - Using a binary search (SEARCH ALL) to search a 1000-element table was 500% faster than using a sequential search (SEARCH)

Performance tuning paper updated

- UPPER and LOWER case conversion
- When converting data to upper or lower case, it is generally more efficient to use INSPECT CONVERTING than the intrinsic functions FUNCTION UPPER-CASE or FUNCTION LOWER-CASE.
- Performance considerations for character conversions:
  - One test case that does 1,000 uppercase conversions was 35% faster when using INSPECT CONVERTING compared to using FUNCTION UPPER-CASE or FUNCTION LOWER-CASE
  - For this same test case, these intrinsic functions used 70% more storage than INSPECT CONVERTING

Performance tuning paper updated

- **Initializing Data**
- The INITIALIZE statement sets selected categories of data fields to predetermined values.
  - However, it is inefficient to initialize an entire group unless you really need all the items in the group to be initialized to different value.
- If you have a group that contains OCCURS data items and you want to set all items in the group to the same character (for example, space or x'00'), it is generally more efficient to use a MOVE statement instead of the INITIALIZE statement.

Performance tuning paper updated

- **Initializing Data**
- 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.

Coding tips from customer situations

- Avoid INITIALIZE unless the functionality is really needed
  - Much faster to MOVE SPACES or x'00' to the group
  - If individual fields need to be set to spaces or different types of zero (external decimal, packed-decimal, numeric-edited) then by all means use INITIALIZE
  - Rule: Don't use INITIALIZE just because it is there!

Coding tips from customer situations

* A customer had a suggestion for an improvement in our implementation of INITIALIZE, which sounded like a good idea

* When a table structure needs to be initialized, and you want both performance and the flexibility to change the structure without having to remember to change the code that initializes it…

* Combine INITIALIZE with group moves!

INITIALIZE tip…before

```
1 Grp.
    2 Struct OCCURS …
       3 Item1 PIC S9(9) BINARY.
       3 Item2 PIC S9(5) PACKED-DECIMAL.
        etc
```

INITIALIZE Grp

INITIALIZE tip…after

```
1 Grp.
    2 Struct OCCURS …
       3 Item1 PIC S9(9) BINARY.
       3 Item2 PIC S9(5) PACKED-DECIMAL.
        etc
```

```
INITIALIZE Struct(1)

COMPUTE j = LENGTH OF GRP / LENGTH OF STRUCT

PERFORM j TIMES

   MOVE Struct(1) To Struct(i)

END-PERFORM
```

INITIALIZE tip…wait a minute!

- I tested this out, and the compiler already generates a loop, and a better one than the user suggestion!
- In my simple case, with this structure:

```
1 Grp.
   2 Struct OCCURS 500 Times.
     3 Item1 PIC S9(9) BINARY.
     3 Item2 PIC S9(5) PACKED-DECIMAL.
     3 Item3 PIC +BB9(5).99 DISPLAY.
     3 Item4           COMP-2.
```

- INITIALIZE on GRP took:

  .07 CPU Seconds

- INITIALIZE on Struct(1) with MOVE in PERFORM loop took:

  .23 CPU Seconds

Coding tips from customer situations

- One customer got recommendation from consultant to code in Java instead of COBOL
- Customer would have preferred to code in COBOL
- Customer complained of continued issues with slow performance and missing Service Level Agreements(SLAs) due to poor Java performance
- Solution: re-code in COBOL?

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 written in COBOL: 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 written in COBOL: 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: 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
```

Questions about variables in dumps

- One program with a large data division (about 1 million items) using TEST(NOHOOK) took 330 times more CPU time to produce a CEEDUMP with COBOL's formatted variables compared to using NOTEST to produce a CEEDUMP without COBOL's formatted variables.
- Do you use formatted dumps with COBOL variables?
  - IE: Compile with TEST(NOHOOK) or TEST(NONE) for production programs
- Do you care about DUMP performance?
  - Usually not done in online environments