# LE Performance Tips and Techniques - COBOL and PL/I issues

Tom Ross

SHARE Session: 8213

March, 2002

IBM Software

# *Performance Tips for COBOL and PL/I: topics*

- **Run-time tips are good for COBOL and for PL/I**
- **Coding specifics are for COBOL**
- **This presentation assumes good application design**
  - Your biggest improvements are made in application design
  - Example: You should eliminate I/O requests through design changes before trying to make each request faster
- **This session focuses on cases where IBM gets in the way of fast code, or provides opportunities for improvement after the design is scrubbed and optimal**

# *Performance Tips for COBOL and LE: topics*

- Run-time library access for performance
- Run-time options for speed and efficiency
- IMS performance considerations
- CICS performance considerations
- COBOL Compiler options for speed and profit
- COBOL and LE features affecting run-time performance
- COBOL coding tips

# *Run-time library access for performance*

- **SCEERUN in LPALST provides best performance**
  - Member LPALSTxx in SYS1.PARMLIB
  - Reentrant modules are loaded into LPA at system IPL
  - VM: Place SCEERUN in Shared Segments (DCSS)
- **STEPLIB is next fastest**
  - Optimized if SCEERUN placed in LLA with FREEZE option
- **LNKLST is no faster than STEPLIB!**
  - Tests at STL show: LNKLST 2.6% slower than STEPLIB during heavy loads
  - Tests at STL show: LNKLST .01% faster than STEPLIB during off peak time
  - Modules from LNKLST are loaded into the users private region
  - Optimized if SCEERUN placed in LLA with FREEZE option
- **There are 2 factors to consider:**
  - Name look up: DASD search or cross memory search?
    LNKLST/LPALST use cross memory search
  - Module load from DASD or already loaded?
    LPA or LLA with FREEZE means modules are already loaded

# *Run-time options for speed and efficiency*

- **AIXBLD**
  - For VSAM files with alternate indexes
  - One VSAM program using AIXBLD is 8% slower than with NOAIXBLD
- **ALL31**
  - ALL31(ON) is 1% faster than ALL31(OFF), range 4% faster to equal
  - One program with many library routine calls was 10% faster w/ALL31(ON)
- **CBLPSHPOP**
  - Changes behavior of CICS condition handling
  - CBLPSHPOP(OFF) much faster for 2nd through nth CALL
- **CHECK**
  - CHECK(ON) with SSRANGE is 2% slower than with CHECK(OFF) (range equal to 20% slower)

# *Run-time options for speed and efficiency*

- **RPTOPTS**
  - **RPTOPTS(ON) is equivalent to RPTOPTS(OFF), range equal to 2% slower**
- **RPTSTG**
  - **RPTSTG(ON) is 5% slower than RPTSTG(OFF) (range equal to 37% slower)**
  - **RPTSTG(ON) can degrade CALL intensive applications 160% or more!**
- **STORAGE**
  - **STORAGE(00,00,00) is 7% slower than STORAGE(NONE,NONE,NONE) (Range equal to 57% slower) (CALL intensive applications can be 160% slower)**
  - **We have customer reports of STORAGE(NONE,NONE,00) causing major performance problems! It should be avoided at ANY cost. This includes recoding to intialize PL/I or C/C++ variables.**

    Only affects LOCAL-STORAGE variables in COBOL
- **RTEREUS**
  - **For special cases only! Bad side-effects**
  - **Overhead of assembler MAIN calling COBOL sub is 99% less**

# *Run-time options for speed and efficiency*

- **TRAP**
  - TRAP(ON) is equal to TRAP(OFF)
- **STORAGE tuning**
  - Use the RPTSTG(ON) option to get storage reports
  - Use the values returned by the RPTSTG(ON) option as the size of the initial blocks of storage for the HEAP, ANYHEAP, BELOWHEAP, STACK, and LIBSTACK run-time options
- **IBM defaults for storage options(non-CICS)**
  - ANYHEAP(16K,8K,ANYWHERE,FREE)
  - BELOWHEAP(8K,4K,FREE)
  - HEAP(32K,32K,ANYWHERE,KEEP,8K,4K)
  - LIBSTACK(8K,4K,FREE)
  - STACK(128K,128K,BELOW,KEEP)
- **For COBOL-only applications:**
  - STACK(64K,64K,BELOW,KEEP)
- **For AMODE(31) applications:**
  - STACK(,,ANYWHERE,)

# *Run-time options for speed and efficiency*

- **Recommendations for speed and usability**
  - **NOAIXBLD**
  - **ALL31(ON) if CICS or if all programs are AMODE=31**
  - **CHECK(OFF)**
  - **NODEBUG**
  - **RPTOPTS(OFF)**
  - **RPTSTG(OFF)**
  - **NORTEREUS**
  - **STORAGE(00,,,) if coming from COBOL WSCLEAR environment**
  - **STORAGE(NONE,,,) if coming from NOWSCLEAR environment**

    Never use STORAGE with 3rd parm other than NONE!
  - **TRAP(ON)**

# *IMS performance considerations*

- **Use Library Routine Retention (LRR)**
  - Similar to VS COBOL II LIBKEEP option
  - Keeps LE environment initialized and retains loaded LE routines and LE storage for library routines
  - See LE Customization manual, SC28-1941-xx
- **Preload Library routines**
  - CEEBINIT, IGZCPAC, IGZCPCO, CEEV005, CEEPLPKA, IGZETRM, IGZEINI
  - IGZCFCC (for COBOL for MVS & VM R2 and later)
  - IGZCLNK (for COBOL/370 R1 programs)
  - IGZCLNK, IGZCTCO, IGZEPLF, IGZEPCL (for VS COBOL II programs)
  - IGZCLNC, IGZCTCO, IGZEPLF, IGZEPCL and any ILBO routines that were preloaded before LE (for OS/VS COBOL programs)

# *IMS performance considerations*

- **Preload application programs**
  - **Heavily used application programs can be compiled with RENT and preloaded to reduce the LOAD overhead for them**
  - **Especially helpful for shared dynamically called subprograms**
- **For OS/VS COBOL programs**
  - **STACK(32K,32K,BELOW,KEEP)**

    Default is STACK(128K,128K,BELOW,KEEP)
  - **Maybe use CEEUOPT**

# *CICS performance considerations - bad news*

- **EXEC CICS LINK slower under LE than under VS COBOL II**
  - **Each LINK or XCTL creates a new run-unit (enclave)**
  - **VS COBOL II transactions with lots of CICS LINKs can be up to 50% slower**
- **LE uses more storage under CICS than VS COBOL II**
  - **May need to increase CICS extended user DSA**
  - **Will need to increase CICS user DSA if using ALL31(OFF)**
  - **Why more storage?**

    LE load modules are bigger than VS COBOL II

    LE has bigger control blocks than VS COBOL II

    LE has more pools of storage than VS COBOL II

# *CICS performance considerations - CALL*

- **EXEC CICS LINK much slower than COBOL CALL**
  - TestcaseA: COB1--LINK-->COB2--LINK-->COB3
    COB1 EXEC CICS LINK to COB2 1000 times
    return via EXEC CICS RETURN
  - TestcaseB: COB1--DYNCALL-->COB2--DYNCALL->-COB3
    COB1 CALL identifier to COB2 1000 times
    return via GOBACK
  - TestcaseA CPU time: 0.85 SEC
  - TestcaseB CPU time: 0.17 SEC
  - EXEC CICS LINK has about 5 TIMES the overhead of
    COBOL dynamic CALL
- **PL/I: Convert CICS LINK/XCTL to CALL or FETCH**
- **COBOL: Convert CICS LINK/XCTL to CALL**
  - Get even more benefit by going to CBLPSHPOP(OFF)!

# *CICS performance considerations - CALL*

## CBLPSHPOP run-time option

- Performance testing shows that overhead of CBLPSHPOP(ON) is significant!

  (No affect on EXEC CICS LINK/XCTL)

- For 2nd through nth CALL to a program, overhead is 1500%!

- Each CALL takes 15 TIMES the overhead of CBLPSHPOP(OFF)

## When can I use CBLPSHPOP(OFF)?

- If no CICS HANDLE ABEND, HANDLE AID, HANDLE CONDITION, or IGNORE CONDITION statements

- Any programs that do their own EXEC PUSH/POP HANDLE

- Any programs that don't use CALL statements will not be affected

# *CICS performance considerations - good news!*

- **LE APAR PQ14883 and COBOL/LE APAR PQ16794**
  - **Reduce the overhead of a CICS LINK - available August 6, 1998**
  - **For an ALL31(ON) application by approximately 10%.**
  - **For ALL31(OFF), we've achieved nearly a 30% CPU savings per CICS LINK.**
- **LE APAR PQ22514 and CICS APAR PQ19878**

  (CICS APAR for V4.1, CICS TS 1 APAR PQ19370)
  - **Approximately 15-20% CPU savings per LINK for ALL31(ON)**
  - **Approximately 5-10% per LINK for ALL31(OFF).**

# *CICS performance considerations - storage*

- **What to do?**
  - **Increase ERDSASZE/EDSALIM**
  - **Put LE modules in ELPA**
  - **Use storage tuning to reduce GETMAIN/FREEMAIN**
    - remember the 16 byte storage buffer when setting values:
      specified-value = desired-value - 16
    - Example: If you want 4K, specify 4080
  - **Do not change RESERVE stack size to anything other than 0K.**
    - Some customers said they changed theirs to 8K because they heard it has to be this for C++. Under CICS, this is not correct.
    - LE Prog Ref example for WSCLEAR shows: STORAGE=(0,NONE,NONE,8K).
    - This almost implies that the RESERVE stack should be 8K in order for WSCLEAR to work.
    - The RESERVE stack is not needed under CICS. If an amount is coded, we end up spending an extra getmain BELOW for that size for every rununit.

# CICS performance considerations - storage

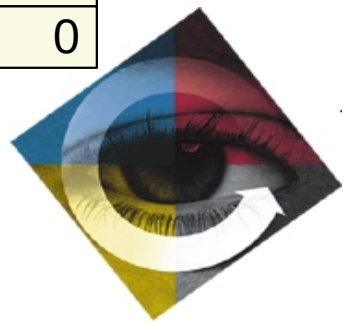
- **What to do?**
  - **Use ALL31(ON)**
  - **Here are tables of storage usage for COBOL with LE under CICS:**

With ALL31(OFF) and STACK(4K,4K,BELOW,KEEP)

| ALL31(OFF) | TRAN(*) | TRAN below | Enclave above | Enclave below |
|---|---|---|---|---|
| LE 1.7 | 13952 | 1848 | 12128 | 21160 |
| LE 2.7 | 14052 | 1896 | 12208 | 17872 |

With ALL31(ON) and STACK(4K,4K,BELOW,KEEP)

| ALL31(ON) | TRAN(*) | TRAN below | Enclave above | Enclave below |
|---|---|---|---|---|
| LE 1.7 | 13952 | 0 | 29192 | 0 |
| LE 2.7 | 14052 | 0 | 25984 | 0 |



# *COBOL/LE features affecting performance*

## Assembler driver calling COBOL programs repeatedly

- Use a REUSABLE run-time environment

- **ILBOSTP0, IGZERRE**
  - ILBOSTP0 will set up both OS/VS COBOL and COBOL portion of LE
  - IGZERRE sets up only COBOL portion
  - ILBOSTP0 should be converted to IGZERRE

    ILBOSTP0 does not have method for enclave termination, storage may not be freed
  - Assembler main calling IGZERRE before calling COBOL is 99% faster than not calling IGZERRE first
- **CEEENTRY and CEETERM**
  - LE-conforming assembler MAIN calling COBOL is 99% faster than non LE-conforming
- **CEEPIPI**
  - Similar to IGZERRE, but works for all languages.
  - Can invoke MAIN or SUB programs
- **Add a COBOL stub in front of assembler driver**
- **LRR - see IMS section**

# *COBOL/LE features affecting performance*

## Mixing older COBOL programs with newer ones

- **Programs linked with older LE releases**
  - If a COBOL/370 program was link-edited with LE R2, R3, or R4, re-link with newer LE, and REPLACE IGZCBSN
  - If a VS COBOL II NORES program was link-edited with LE R2, R3, or R4, re-link with newer LE, and REPLACE IGZENRI
- **OS/VS COBOL mixed with COBOL/370 or later**
  - Both OS/VS COBOL part of LE and new COBOL part of LE must be used
  - Costs extra at INIT and TERM
  - Converting to COBOL for MVS & VM or COBOL for OS/390 & VM will avoid OS/VS COBOL run-time init and term

# *COBOL/LE features affecting performance*

- **Which compiler to use? Should you recompile all of your VS COBOL II programs with COBOL for OS/390 & VM?**
  - Tests with mixed COBOL and assembler show significant improvement to be gained with newer compilers
- **COBOLA -> AssemblerB -> COBOLC**
  - COBOLA is VS COBOL II calling assembler with dynamic CALL
  - AssemblerB LOADs and BALRs to COBOLC 100,000 times
  - COBOLC is VS COBOL II
  - If using VS COBOL II run-time library: CPU = 4.703 SECS
  - If using LE 2.7 run-time library: CPU = 2.962 SECS
  - If COBOLC compiled w/COBOL for OS/390 & VM: CPU = 0.525 SECS
- **Assembler to COBOL overhead reduced 89%**
  - Just by recompiling! No code changes necessary.

# *COBOL coding tips - table element references*

- **Subscripting**
  - External decimal (USAGE DISPLAY)
  - Binary
- **Indexing**
  - INDEXED BY phrase of OCCURS
  - USAGE IS INDEX
- **Which is fastest, and which slowest?**
  - External Decimal = 4 instructions per reference
  - Binary with TRUNC(OPT or STD) = 2 instructions
  - Binary with TRUNC(BIN) = 4 instructions including CVD
  - Indexes = 1 instruction
- **Slowest is binary with TRUNC(BIN)**
- **For programs with lots of table processing, use INDEXES to really speed things up!**

# *COBOL coding tips - table element references*

- **Given these data descriptions:**

```
77 SUB1  PIC 9(4) USAGE BINARY.
01  GRP1.
    05  TAB1 OCCURS 1000 INDEXED BY TABINDX.
        10 SALES       PIC 9(7) PACKED-DECIMAL.
        10 EXPENSES  PIC 9(7) PACKED-DECIMAL.
        10 INVENTORY PIC 9(7) PACKED-DECIMAL.
```

- **Slow code:**

```
PERFORM   VARYING SUB1
                  FROM 1 BY 1
                  UNTIL SUB1 > 1000
   COMPUTE SALES-TOTAL = SALES-TOTAL + SALES(SUB1)
   COMPUTE EXPENSE-TOTAL = EXPENSE-TOTAL + EXPENSES(SUB1)
   COMPUTE INVENTORY-TOTAL = INVENTORY-TOTAL + INVENTORY(SUB1)
END-PERFORM
```

- **Fast code:**

```
PERFORM   VARYING TABINDX
                  FROM 1 BY 1
                  UNTIL TABINDX > 1000
   COMPUTE SALES-TOTAL = SALES-TOTAL + SALES(TABINDX)
   COMPUTE EXPENSE-TOTAL = EXPENSE-TOTAL + EXPENSES(TABINDX)
   COMPUTE INVENTORY-TOTAL = INVENTORY-TOTAL + INVENTORY(TABINDX)
END-PERFORM
```

# *COBOL coding tips - I/O performance*

## QSAM files/datasets

- **Data Striping: Extended format datasets**
  - Large data sets with high I/O activity are best candidates
  - Your installation's ACS routines must request a data class which specifies data set name type of EXTENDED
  - Storage must be SMS-managed
  - You should use more buffers to fully exploit striping
    BUFNO subparameter of the DCB parameter of JCL or
    RESERVE clause of the SELECT statement in the FILE-CONTROL paragraph
  - Your buffers should be above the 16MB line
- **Block size for performance**
  - Use large blocksizes or
  - Use BLOCK CONTAINS 0 and omit the BLKSIZE parameter from JCL
- **QSAM buffers above the 16 MB line**
  - Programs compiled with VS COBOL II Release 3 or later
  - Running under Language Environment R3 or later
  - All programs compiled with RENT and DATA(31) or RMODE(ANY) and NORENT
  - EXTERNAL files: Use ALL31(ON)

# *COBOL coding tips - I/O performance*

## VSAM files/datasets

- **Increase number of buffers**
  - Data buffers (BUFND) for sequential access
  - Index buffers (BUFNI) for random access
- **Control interval size (CISZ)**
  - A smaller CISZ results in faster retrieval for random processing at the expense of inserts
  - A larger CISZ is more efficient for sequential processing
  - In general: large CISZ and buffer space may improve performance
- **Fastest access mode? In general:**
  - SEQUENTIAL is fastest
  - DYNAMIC is next
  - RANDOM access is the least efficient
- **VSAM buffers above the 16 MB line**
  - Programs compiled with VS COBOL II Release 3 or later
  - Running under Language Environment R3 or later

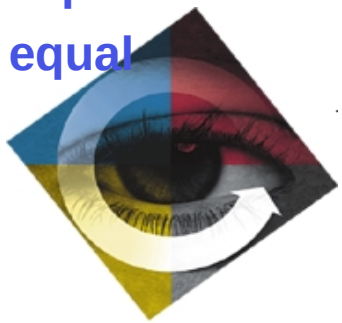
# *COBOL Compiler options for speed and profit*

- **AWO**
  - For QSAM variable length record blocked files
  - Results: One program using var-len files and AWO was 10% faster than NOAWO
- **NUMPROC**
  - PFD is fastest, but MIG is faster than NOPFD (11% faster to equal)
- **OPTIMIZE**
  - OPTIMIZE(STD) is 4% faster than NOOPT, with a range of 17% faster to equal
  - OPTIMIZE(FULL) can be much faster for programs with VALUE clauses that are called in initial state (CALL and CANCEL, PROGRAM IS INITIAL)

    One program that had 500 unreferenced data items was 80% faster with OPT(FULL) over STD
- **RENT**
  - and NORENT are equivalent

# *COBOL Compiler options for speed and profit*

- **SSRANGE**
  - Compiled w/SSRANGE, run w/CHECK(ON) is 4.3% slower than NOSSRANGE
  - Compiled w/SSRANGE, run w/CHECK(OFF) is 2.1% slower than NOSSRANGE
  - Compiled w/SSRANGE, run w/CHECK(ON) is 1.4% slower than w/CHECK(OFF)
- **TEST**
  - TEST(ALL,SYM) is 14% slower than NOTEST, range equal to 59% slower
  - TEST(NONE,SYM) is equal to NOTEST
- **TRUNC**
  - TRUNC(OPT) is 27% faster than TRUNC(BIN) range 92% faster to equal
  - TRUNC(STD) is 26% faster than TRUNC(BIN) range 88% faster to equal
  - TRUNC(OPT) is 5% faster than TRUNC(STD) range 49% faster to equal



# *COBOL for OS/390 & VM V2R2 - improved TRUNC(BIN)!*

- **Improvement by number of digits for both signed and unsigned binary data:**
  - **BEFORE: on the average, 2.1.0 compiler with TRUNC(BIN) was 4.8 times slower than TRUNC(OPT), with a max of 30 times slower**
  - **AFTER: on the average, 2.2.0 compiler with TRUNC(BIN) was 1.8 times slower than TRUNC(OPT), with a max of 13 times slower**
  - **2.2.0 compiler with TRUNC(OPT) compared to 2.1.0 compiler with TRUNC(OPT)**

| | | |
|---|---|---|
| signed | 1-18 digits | equivalent |
| unsigned | 1-9 digits | equivalent |
| | 10-17 digits | 70% faster |
| | 18 digits | 8% faster |

  - **2.2.0 compiler with TRUNC(BIN) compared to 2.1.0 compiler with TRUNC(BIN)**

| | | |
|---|---|---|
| signed | 1-9 digits | 85% faster |
| | 10-18 digits | equivalent |
| unsigned | 1-9 digits | equivalent |
| | 10-17 digits | 70% faster |
| | 18 digits | equivalent |

# *COBOL Compiler options for speed and profit*

- **Recommendations for speed and usability**
  - **AWO**
  - **NOCMPR2**
  - **FASTSRT**
  - **NUMPROC(MIG)**
  - **OPTIMIZE(STD)**
  - **RENT**
  - **NOSSRANGE**
  - **NOTEST or TEST(NONE,SYM) or TEST(NONE,SYM,SEPARATE)**
  - **TRUNC(OPT)**