IBM Enterprise COBOL Version 3 Release 1 Performance Tuning

January 16, 2002

R. J. Arellanes

IBM Corporation
Software Solutions
555 Bailey Avenue
San Jose, CA 95141

Disclaimer

The performance considerations contained in this paper were obtained by running sample programs in a particular hardware/software configuration using a selected set of tests and are presented as illustrations. Since performance varies with configuration, sample program characteristics, and other installation and environment factors, results obtained in other operating environments may vary. We recommend that you construct sample programs representative of your workload and run your own experiments with a configuration applicable to your environment.

IBM does not represent, warrant, or guarantee that a user will achieve the same or similar results in the user's environment as the experimental results reported in this paper.

Distribution Notice

| **Fifth Edition, January 2002**

| This edition applies to IBM Enterprise COBOL Version 3 Release 1 running with z/OS Language Environment Version 1 Release 2, and to all subsequent releases and modifications until otherwise indicated in new editions.

| This edition replaces all previous editions of this document. All changes made in this edition are marked with change bars as indicated to the left of this paragraph.

Contents

Introduction

This paper identifies some of the factors that can affect the performance of COBOL programs when using IBM Enterprise COBOL Version 3 Release 1 running with z/OS Language Environment Version 1 Release 2. It also contains a summary of the performance experiences that we have had with IBM Enterprise COBOL. The tuning methods discussed here are intended to assist users in selecting from the various options which are available for compiling COBOL programs with IBM Enterprise COBOL Version 3 Release 1 and executing them with z/OS Language Environment Version 1 Release 2. These methods provide the COBOL programmer with the opportunity to tune the LE run-time environment to potentially improve CPU time performance and the use of system resources.

First, we will look at some compiler and run-time options that affect the run-time performance of a COBOL application. Next, we will look at some efficient COBOL coding techniques that may improve the run-time performance. Finally, we will look at a check list of items to be aware of if a performance problem is encountered with IBM Enterprise COBOL.

Referenced IBM Publications

Throughout this paper, all references to OS/VS COBOL refer to OS/VS COBOL Release 2.4, all references to MVS refer to z/OS, unless otherwise indicated. Additionally, several items have page number references after them, enclosed in parentheses. The manual abbreviations are in **bold face** followed by the page numbers in that manual. The abbreviations and manuals referenced in this paper are listed in the table below:

Abbrev	IBM Publication and Number
COB PG	*Enterprise COBOL for z/OS and OS/390 Programming Guide Version 3 Release 1*, SC27-1412-00
COB LRM	*Enterprise COBOL for z/OS and OS/390 Version 3 Release 1 Language Reference*, GC27-1408-00
COB MIG	*Enterprise COBOL for z/OS and OS/390 Version 3 Release 1 Compiler and Run-Time Migration Guide*, GC27-1409-00
LE PG	*z/OS V1R2.0 Language Environment Programming Guide*, SA22-7561-01
LE REF	*z/OS V1R2.0 Language Environment Programming Reference*, SA22-7562-01
LE CUST	*z/OS V1R2.0 Language Environment Customization*, SA22-7564-01
LE MIG	*z/OS V1R2.0 Language Environment Run-Time Migration Guide*, GA22-7565-01

These manuals contain additional information regarding the topics that are discussed in this paper, and it is strongly recommended that they be used in conjunction with this paper to receive increased benefit from the information contained herein.

Background

In general, we do most of our measurements on MVS batch, but sometimes we also measure the language products on CICS. In measuring the performance of the language products on MVS batch, we use a variety of different tools that we have developed specifically for this purpose. Our tools can collect data such as CPU time used by a program (TCB and SRB Time), elapsed time, EXCP counts, and virtual storage usage. When measuring on CICS, we usually enlist the help of other departments that are more familiar with transaction environments in collecting and analyzing data from RMF and SMF.

In measuring COBOL, we have three different types of benchmarks: compile-only, compile and execute, and kernels. The compile-only set contains programs which are designed to be compiled but not executed. These

programs measure the compiler performance as a function of program size and range from 200 statements to more than 7,000 statements. The compile and execute set contains programs which are either subsets of "real" application programs or are entire application programs. These programs are somewhat representative of "real" application programs. Some are actual customer programs and the rest have been written by IBM. The kernel set contains programs which are a collection of specific language statements enclosed in a loop of 1,000 to 100,000 times. Each kernel program consists of several of these loops, with each loop having a different variation of that language statement. Each loop is surrounded by a call to a timing routine so that we can measure the performance of the individual statements.

| The performance considerations reported in this paper were made on a 2064 Model 116 with 2 sysplexes and
| 2 systems on each sysplex. The programs run for this paper were run under one of the systems on one of the
| sysplexes running z/OS Version 1 Release 2 PTF Level 0107. The programs used were batch-type (non-interactive) applications. **Unless otherwise indicated, all performance comparisons made in this paper are referencing CPU time performance and not elapsed time performance.**

Tuning the Run-Time Environment

This section focuses on some of the options that are available for tuning an application, as well as the overall LE run-time environment. This in itself may not produce high performing code since both the coding style and the data types can have a significant impact on the performance of the application. In fact, the coding style and data types usually have a far greater impact on the performance of an application than that of tuning the application via external means (e.g., compiler options, run-time options, space management tuning, and placing the library routines in shared storage). First, we will look at each of these external options in a little more detail.

Compiler Options that Affect Run-Time Performance

Compiler options, by far, have been the cause of a vast majority of the performance problems reported by customers. Many customers are not aware of the performance implications that many of the compiler options have at run time, especially the ARITH, AWO, DYNAM, FASTSRT, NUMPROC, OPTIMIZE, RENT, SSRANGE, TEST, THREAD, and TRUNC options. Let's look at each of these options to see how they might affect the performance of an application program.

(**COB PG:** p 279)

ARITH - EXTEND or COMPAT

The ARITH compiler option allows you to control the maximum number of digits allowed for decimal numbers (packed decimal, zoned decimal, and numeric-edited data items and numeric literals). With ARITH(EXTEND), the maximum number of digits is 31; with ARITH(COMPAT), the maximum number of digits is 18. However, ARITH(EXTEND) will cause some degradation in performance for all decimal data types due to larger intermediate results. The amount of degradation that you experience depends directly on the amount of decimal data that you use.

Performance considerations using ARITH:

- On the average, ARITH(EXTEND) was 1% slower than ARITH(COMPAT), with a range of equivalent to 38% slower.

(**COB PG:** pp 37, 41, 48-49, 95, 283-284, 557-566)

AWO or NOAWO

The AWO compiler option causes the APPLY WRITE-ONLY clause to be in effect for all physical sequential, variable-length, blocked files, even if the APPLY WRITE-ONLY clause is not specified in the program. With APPLY WRITE-ONLY in effect, the file buffer is written to the output device when there is not enough space in the buffer for the next record. Without APPLY WRITE-ONLY, the file buffer is written to the output device when there is not enough space in the buffer for the maximum size record. If the application has a large variation in the size of the records to be written, using APPLY WRITE-ONLY can result in a performance savings since this will generally result in fewer calls to Data Management Services to handle the I/Os.

Performance considerations using AWO:

- One program using variable-length files and AWO was 88% faster than NOAWO. This faster processing was the result of using 98% fewer EXCPs to process the writes.

(**COB PG:** pp 11-12, 284, 543)

DATA(24) or DATA(31)

Using DATA(31) with your RENT program will help to relieve some below the line virtual storage constraint problems. When you use DATA(31) with your RENT programs, most QSAM file buffers can be allocated above the 16MB line. When you use DATA(31) with the run-time option HEAP(,,ANYWHERE), all non-EXTERNAL WORKING-STORAGE and non-EXTERNAL FD record areas can be allocated above the 16MB line.

With DATA(24), the WORKING-STORAGE and FD record areas will be allocated below the 16 MB line.

Notes:

1. For NORENT programs, the RMODE option determines where non-EXTERNAL data is allocated.
2. See "QSAM Files" on page 27 for additional information on QSAM file buffers.
3. See "ALL31" on page 9 for information on where EXTERNAL data is allocated.

Performance data using DATA is not available at this time.

(**COB PG:** pp 34-35, 137-138, 288, 308, 368, 379, 408, 543)

DYNAM or NODYNAM

The DYNAM compiler option specifies that all subprograms invoked through the CALL literal statement will be loaded dynamically at run time. This allows you to share common subprograms among several different applications, allowing for easier maintenance of these subprograms since the application will not have to be re-link-edited if the subprogram is changed. DYNAM also allows you to control the use of virtual storage by giving you the ability to use a CANCEL statement to free the virtual storage used by a subprogram when the subprogram is no longer needed. However, when using the DYNAM option, you pay a performance penalty since the call must go through a library routine, whereas with the NODYNAM option, the call goes directly to the subprogram. Hence, the path length is longer with DYNAM than with NODYNAM.

Performance considerations using DYNAM with CALL literal (measuring CALL overhead only):

On the average, for a CALL intensive application, the overhead associated with the CALL using DYNAM ranged from 16% slower to 100% slower than NODYNAM.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms is not degraded as much.

(**COB PG:** pp 293, 394-399, 404-405, 544)

FASTSRT or NOFASTSRT

For eligible sorts, the FASTSRT compiler option specifies that the SORT product will handle all of the I/O and that COBOL does not need to do it. This eliminates all of the overhead of returning control to COBOL after each record is read in or after processing each record that COBOL returns to sort. The use of FASTSRT is recommended when direct access devices are used for the sort work files since the compiler will then determine which sorts are eligible for this option and generate the proper code. If the sort is not eligible for this option, the compiler will still generate the same code as if the NOFASTSRT option were in effect. A list of requirements for using the FASTSRT option is in the COBOL programming guide.

Performance considerations using FASTSRT:

One program that processed 100,000 records was 35% faster when using FASTSRT compared to using NOFASTSRT.

| (**COB PG:** pp 189-191, 194, 294, 544)

NUMPROC - NOPFD, MIG, or PFD

Using the NUMPROC(PFD) compiler option generates significantly more efficient code for numeric comparisons. It also avoids the generation of extra code that NUMPROC(NOPFD) or NUMPROC(MIG) generates for most references to COMP-3 and DISPLAY numeric data items to ensure a correct sign is being used. With NUMPROC(NOPFD), sign fix-up processing is done for all references to these numeric data items. With NUMPROC(MIG), sign fix-up processing is done only for receiving fields (and not for sending fields) of arithmetic and MOVE statements. With NUMPROC(PFD), the compiler assumes that the data has the correct sign and bypasses this sign fix-up processing. NUMPROC(MIG) generates code that is similar to that of OS/VS COBOL. Using NUMPROC(NOPFD) or NUMPROC(MIG) may also inhibit some other types of optimization. However, not all external data files contain the proper sign for COMP-3 or DISPLAY signed numeric data, and hence, using NUMPROC(PFD) may not be applicable for all application programs. For performance sensitive applications, NUMPROC(PFD) is recommended when possible.

Performance considerations using NUMPROC:

| On the average, NUMPROC(PFD) was 1% faster than NUMPROC(NOPFD), with a range of 20% faster to equivalent.

| On the average, NUMPROC(PFD) was 1% faster than NUMPROC(MIG), with a range of 9% faster to equivalent.

| On the average, NUMPROC(MIG) was equivalent to NUMPROC(NOPFD), with a range of 13% faster to equivalent.

| (**COB PG:** pp 45-46, 302-303, 544)

OPTIMIZE(STD), OPTIMIZE(FULL), or NOOPTIMIZE

To assist in the optimization of the code, you should use the OPTIMIZE compiler option. With the OPTIMIZE(STD) or OPTIMIZE(FULL) options in effect, you may receive optimizations that include:

- eliminating unnecessary branches
- simplifying inefficient branches
- simplifying the code for the out-of-line PERFORM statement, moving the performed paragraphs in-line, where possible
- simplifying the code for a CALL to a contained (nested) program, moving the called statements in-line, where possible
- eliminating duplicate computations
- eliminating constant computations
- aggregating moves of contiguous, equal-sized items into a single move
- deleting unreachable code

Additionally, with the OPTIMIZE(FULL) option in effect, you may also receive these optimizations:

- deleting unreferenced data items and the associated code to initialize their VALUE clauses

Many of these optimizations are not available with OS/VS COBOL, but are available with IBM Enterprise COBOL. NOOPTIMIZE is generally used while a program is being developed when frequent compiles are necessary. NOOPTIMIZE also makes it easier to debug a program since code is not moved; NOOPTIMIZE is required when using the TEST compiler option with a value other than TEST(NONE). OPTIMIZE requires more CPU time for compiles than NOOPTIMIZE, but generally produces more efficient run-time code. For production runs, OPTIMIZE is recommended.

WARNING: If your program relies upon unreferenced level 01 or level 77 data items, you should not use OPTIMIZE(FULL), since OPTIMIZE(FULL) will delete all unreferenced data items. On way to prevent the data item from being deleted by the OPTIMIZE(FULL) option is to refer to the data item in the PROCEDURE DIVISION (for example, initialize the data item with a PROCEDURE DIVISION statement instead of with VALUE clauses).

Performance considerations using OPTIMIZE:

- On the average, OPTIMIZE(STD) was 1% faster than NOOPTIMIZE, with a range of 12% faster to equivalent.
- On the average, OPTIMIZE(FULL) was equivalent to OPTIMIZE(STD).
- One RENT program calling a RENT subprogram with 500 unreferenced data items with VALUE clauses was 9% faster with OPTIMIZE(FULL) or OPT(STD) compared to NOOPT.
- The same RENT program calling a RENT subprogram with 500 unreferenced data items with VALUE clauses using the IS INITIAL clause on the PROGRAM-ID statement was 90% faster with OPTIMIZE(FULL) compared to OPT(STD).

Note: The two RENT program tests measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

(**COB PG:** pp 304-305, 533-535, 538-539, 540-542, 544)

RENT or NORENT

Using the RENT compiler option causes the compiler to generate some additional code to ensure that the program is reentrant. Reentrant programs can be placed in shared storage like the Link Pack Area (LPA) or the Extended Link Pack Area (ELPA). Also, the RENT option will allow the program to run above the 16 MB line. Producing reentrant code may increase the execution time path length slightly.

Note: The RMODE(ANY) option can be used to run NORENT programs above the 16 MB line.

Performance considerations using RENT:

- On the average, RENT was equivalent to NORENT.

(**COB PG:** pp 34-35, 137-138, 308-309, 379, 381, 408, 435, 438, 453, 544, 547)

RMODE - AUTO, 24, or ANY

The RMODE compiler option determines the RMODE setting for the COBOL program. When using RMODE(AUTO), the RMODE setting depends on the use of RENT or NORENT. For RENT, the program will have RMODE ANY. For NORENT, the program will have RMODE 24. When using RMODE(24), the program will always have RMODE 24. When using RMODE(ANY), the program will always have RMODE ANY.

Note: When using NORENT, the RMODE option controls where the WORKING-STORAGE will reside. With RMODE(24), the WORKING-STORAGE will be below the 16 MB line. With RMODE(ANY), the WORKING-STORAGE can be above the 16 MB line.

Performance data using RMODE is not available at this time.

(**COB PG:** pp 34-35, 138, 308-309, 379, 408, 544)

SSRANGE or NOSSRANGE

Using SSRANGE generates additional code to verify that all subscripts, indexes, and reference modification expressions do not refer to data beyond the bounds of the subject data item. This in-line code occurs at every reference to a subscripted or variable-length data item, as well as at every reference modification expression, and it can result in some degradation at run time. In general, if you need to verify the subscripts only a few times in the application instead of at every reference, coding your own checks may be faster than using the SSRANGE option. For performance sensitive applications, NOSSRANGE is recommended.

Performance considerations using SSRANGE:

- On the average, SSRANGE with the run-time option CHECK(ON) was 1% slower than NOSSRANGE, with a range of equivalent to 27% slower.
- On the average, SSRANGE with the run-time option CHECK(OFF) was 1% slower than NOSSRANGE, with a range of equivalent to 9% slower.
- On the average, SSRANGE with the run-time option CHECK(ON) was 1% slower than SSRANGE with the run-time option CHECK(OFF) with a range of equivalent to 16% slower.

(**COB PG:** pp 313, 333, 545)

TEST or NOTEST

The TEST compiler option produces object code that enables Debug Tool to perform batch and interactive debugging. The amount of debugging support available depends on which TEST suboptions you use. The TEST option also allows you to request that symbolic variables be included in the formatted dump produced by Language Environment.

When using the SYM suboption of the TEST option, you can control where the symbolic information will be kept. If you use TEST(,SYM,NOSEPARATE), the symbolic information will be part of the object module, which could result in a much larger object module. If you use TEST(,SYM,SEPARATE), the symbolic information will be placed in a separate file and will be loaded only as needed.

Note: If you used the FDUMP option with VS COBOL II, TEST(NONE,SYM) is the equivalent option with IBM Enterprise COBOL.

Using TEST with a value other than NONE can cause a significant performance degradation when used in a production environment since this additional code occurs at ***each*** COBOL statement. Hence, the TEST option with a value other than NONE should be used only when debugging an application. Additionally, when TEST is used with a value other than NONE, the OPTIMIZE option is disabled. For production runs, NOTEST or TEST(NONE) is recommended.

Note: With the latest levels of Debug Tool, you can step through your program even if there are no compiled-in hooks, by using the overlay hooks function of Debug Tool. However, you must compile with the NOOPTIMIZE and TEST(NONE) options to use this feature. You should also use the SYM and SEPARATE suboptions of TEST to get the symbolic debug information without substantially increasing the size of your load modules.

Additionally, when using TEST(NONE,SYM) with a large data division and an abend occurs producing a CEEDUMP, a significant amount of CPU time may be required to produce the CEEDUMP, depending on the size of the data division.

Performance considerations using TEST:

- On the average, TEST(ALL,SYM) was 20% slower than NOTEST, with a range of equivalent to 200% slower when not producing a CEEDUMP.
- On the average, TEST(NONE,SYM) was equivalent to NOTEST when not producing a CEEDUMP.

On the average, TEST(NONE,SYM,NOSEPARATE) resulted in a 236% increase in the object module size compared to using NOTEST or TEST(NONE,SYM,SEPARATE), with a range of 9% larger to 670% larger.

On the average, TEST(NONE,SYM,SEPARATE) resulted in an increase of approximately 200 bytes in the object module size compared to using NOTEST

One program with a large data division (about 1 million items) using TEST(NONE,SYM) took 200 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.

(**COB PG:** pp 219, 314-315, 361, 545)

THREAD or NOTHREAD

The THREAD compiler option enables the COBOL program for execution in a Language Environment enclave with multiple POSIX threads or PL/I tasks. A program compiled with the THREAD compiler option can also run in a non-threaded environment, however there will be some degradation in the initialization and termination of the COBOL program due to the overhead of serialization logic that is done for these programs. The THREAD compiler option also requires the use of the IS RECURSIVE clause on the PROGRAM-ID statement.

Performance considerations using THREAD (measuring CALL overhead only):

One testcase (Assembler calling COBOL) using THREAD was 35% slower than using NOTHREAD.

One testcase (COBOL statically calling COBOL) using THREAD was 30% slower than using NOTHREAD.

One testcase (COBOL dynamically calling COBOL) using THREAD was 30% slower than using NOTHREAD.

Notes:

1. The IS RECURSIVE clause was used in both the THREAD and NOTHREAD cases.
2. This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms is not degraded as much.

(**COB PG:** pp 17, 316-317, 391, 433-439)

TRUNC - BIN, STD, or OPT

When using the TRUNC(BIN) compiler option, all binary (COMP) sending fields are treated as either halfword, fullword, or doubleword values, depending on the PICTURE clause, and code is generated to truncate all binary receiving fields to the corresponding halfword, fullword, or doubleword boundary (base 2 truncation). The full content of the field is significant. This can add a significant amount of degradation since typically some data conversions must be done, which may require the use of some library subroutines. BIN is usually the slowest of the three sub options for TRUNC.

When using the TRUNC(STD) compiler option, the final intermediate result of an arithmetic expression, or the sending field in the MOVE statement, is truncated to the number of digits in the PICTURE clause of the binary (COMP) receiving field (base 10 truncation). This can add a significant amount of degradation since typically the number is divided by some power of ten (depending on the number of digits in the PICTURE clause) and the remainder is used; a divide instruction is one of the more expensive instructions. TRUNC(STD) behaves in a similar way as TRUNC in OS/VS COBOL.

However, with TRUNC(OPT), the compiler assumes that the data conforms to the PICTURE and USAGE specifications and manipulates the result based on the size of the field in storage (halfword, fullword or doubleword). Although TRUNC(OPT) most closely resembles the behavior of NOTRUNC in OS/VS

| COBOL and is recommended for compatibility with NOTRUNC, there are some cases where the result will be different. Please consult the COBOL Migration Guide and Programming Guide for additional details.

TRUNC(STD) conforms to the ANSI and SAA standards, whereas TRUNC(BIN) and TRUNC(OPT) do not. TRUNC(OPT) is provided as a performance tuning option and should be used only when the data in the application program conforms to the PICTURE and USAGE specifications. For performance sensitive applications, the use of TRUNC(OPT) is recommended when possible.

Performance considerations using TRUNC:

- On the average, TRUNC(OPT) was 24% faster than TRUNC(BIN), with a range of 88% faster to equivalent.
- On the average, TRUNC(STD) was 15% faster than TRUNC(BIN), with a range of 78% faster to equivalent.
- On the average, TRUNC(OPT) was 6% faster than TRUNC(STD), with a range of 65% faster to equivalent.

Note: See "Recent Performance Improvements" on page 36 for additional details of improvements to TRUNC(BIN) and TRUNC(OPT) processing.

(**COB PG:** pp 41-42, 317-319, 371-372, 376, 545; **COB MIG:** p 154)

Run-Time Options that Affect Run-Time Performance

Selecting the proper run-time options is another factor that affects the performance of a COBOL application. Therefore, it is important for the system programmer responsible for installing and setting up the LE environment to work with the application programmers so that the proper run-time options are set up correctly for your installation. Let's look at some of the options that can help to improve the performance of the individual application, as well as the overall LE run-time environment.

(**LE PG:** pp 113-124; **LE REF:** pp 9-102; **LE CUST:** pp 19-28, 67-158)

AIXBLD

The AIXBLD option allows alternate indexes to be built at run time. However, this may adversely affect the run-time performance of the application. It is much more efficient to use Access Method Services to build the alternate indexes before running the COBOL application than using the NOAIXBLD run-time option. Note that AIXBLD is not supported when VSAM datasets are accessed in RLS mode.

Performance considerations using AIXBLD:

One VSAM program was 8% slower when using AIXBLD compared to using NOAIXBLD.

(**LE REF:** p 12-13; **LE CUST:** pp 73-74; **COB PG:** p 164-165, 168, 547)

ALL31

The ALL31 option allows LE to take advantage of knowing that there are *no* AMODE(24) routines in the application. ALL31(ON) specifies that the entire application will run in AMODE(31). This can help to improve the performance for an all AMODE(31) application because LE can minimize the amount of mode switching across calls to common run-time library routines. Additionally, using ALL31(ON) will help to relieve some below the line virtual storage constraint problems, since less below the line storage is used.

When using ALL31(ON), all EXTERNAL WORKING-STORAGE and EXTERNAL FD records areas can be allocated above the 16MB line if you also use the HEAP(,,ANYWHERE) run-time option and

compile the program with either the DATA(31) and RENT compiler options or with the RMODE(ANY) and NORENT compiler options. ALL31(OFF) is required for all OS/VS COBOL programs that are not running under CICS, all VS COBOL II NORES programs, and all other AMODE(24) programs. Note that when using ALL31(OFF), you must also use STACK(,,BELOW).

Note: Beginning with LE for z/OS Release 1.2, the run-time defaults have changed to ALL31(ON),STACK(,,ANY). LE for OS/390 Release 2.10 and earlier run-time defaults were ALL31(OFF),STACK(,,BELOW).

Performance considerations using ALL31 (measuring CALL overhead only):

On the average, ALL31(ON) was equivalent to ALL31(OFF).

One program with many library routine calls was 10% faster when using ALL31(ON).

Note: This test measured only the overhead of the CALL for a RENT program (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms will have different results, depending on the number of calls that are made to LE common run-time routines.

(**LE REF:** pp 13-15; **LE CUST:** pp 74-76; **COB PG:** pp 33-35, 137-138, 396-397, 406; **COB MIG:** p 55-56)

CHECK

The CHECK option activates the additional code generated by the SSRANGE compiler option, which requires more CPU time resources for the verification of the subscripts, indexes, and reference modification expressions. Using the CHECK(OFF) run-time option deactivates this code but still requires some additional CPU time resources at every use of a subscript, index, or reference modification expression to determine that this check is not desired during the particular run of the program. This option has an effect only on a program that has been compiled with the SSRANGE compiler option.

Performance considerations using CHECK:

On the average, CHECK(ON) with SSRANGE was 1% slower than CHECK(OFF) with SSRANGE, with a range of equivalent to 16% slower.

(**LE REF:** pp 21-22; **LE CUST:** pp 82-83; **COB PG:** pp 313, 333, 545)

DEBUG

The DEBUG option activates the COBOL batch debugging features specified by the USE FOR DEBUGGING declarative. This may add some additional overhead to process the debugging statements. This option has an effect only on a program that has the USE FOR DEBUGGING declarative.

Performance considerations using DEBUG:

The eleven programs measured ranged from equivalent to 2080% slower when using DEBUG compared to using NODEBUG.

Note: The programs measured in this test were modified to use WITH DEBUGGING MODE on the SOURCE-COMPUTER paragraph and to contain a USE FOR DEBUGGING ON ALL PROCEDURES declarative that did a DISPLAY DEBUG-ITEM. Since the debugging code in these cases is generated only for paragraph and section labels, other programs may have significantly different results.

(**LE REF:** pp 23-24; **LE CUST:** pp 84-85; **COB PG:** p 330)

RPTOPTS

The RPTOPTS option allows you to get a report of the run-time options that were in use during the execution of an application. This report is produced after the application has terminated. Thus, if the application abends, the report may not be generated. Generating the report can result in some additional overhead. Specifying RPTOPTS(OFF) will eliminate this overhead.

Performance considerations using RPTOPTS:

| On the average, RPTOPTS(ON) was equivalent to RPTOPTS(OFF).

| **Note:** Although the average for a single batch program shows equivalent performance for RPTOPTS(ON), | you may experience some degradation in a transaction environment (for example, CICS) where main pro- | grams are repeatedly invoked.

| (**LE REF:** pp 59-61; **LE CUST:** pp 117-120; **COB MIG:** p 99)

RPTSTG

The RPTSTG option allows you to get a report on the storage that was used by an application. This report is produced after the application has terminated. Thus, if the application abends, the report may not be generated. The data from this report can help you fine tune the storage parameters for the application, reducing the number of times that the LE storage manager must make system requests to acquire or free storage. Collecting the data and generating the report can result in some additional overhead. Specifying RPTSTG(OFF) will eliminate this overhead.

Performance considerations using RPTSTG:

| On the average, RPTSTG(ON) was 5% slower than RPTSTG(OFF), with a range of equivalent to | 88% slower. Note that when using call intensive applications, the degradation can be 200% slower or | more.

| (**LE REF:** pp 61-69; **LE CUST:** pp 120-127; **COB MIG:** p 99)

RTEREUS

The RTEREUS option causes the LE run-time environment to be initialized for reusability when the first COBOL program is invoked. The LE run-time environment remains initialized (all COBOL programs and their work areas are kept in storage) in addition to keeping the library routines initialized and in storage. This means that, for subsequent invocations of COBOL programs, most of the run-time environment initialization will be bypassed. Most of the run-time termination will also be bypassed, unless a STOP RUN is executed or unless an explicit call to terminate the environment is made (**Note:** using STOP RUN results in control being returned to the caller of the routine that invoked the first COBOL program, terminating the reusable run-time environment).

Because of the effect that the STOP RUN statement has on the run-time environment, you should change all STOP RUN statements to GOBACK statements in order to get the benefit of RTEREUS. The most noticeable impact will be on the performance of a non-COBOL driver repeatedly calling a COBOL subprogram (for example, an assembler driver that repeatedly calls COBOL applications). The RTEREUS option helps in this case. However, using the RTEREUS option does affect the semantics of the COBOL application: each COBOL program will now be considered to be a subprogram and will be entered in its last-used state on subsequent invocations (if you want the program to be entered in its initial state, you can use the INITIAL clause on the PROGRAM-ID statement). **WARNING: This means that storage that is acquired during the execution of the application will not be freed.** Therefore, RTEREUS may not be applicable to all environments.

Performance considerations using RTEREUS (measuring CALL overhead only):

One testcase (Assembler calling COBOL) using RTEREUS was 99% faster than using NORTEREUS.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

| See “Modifying COBOL's Reusable Environment Behavior” on page 17 for additional performance considerations when running in a COBOL reusable environment.

(**LE REF:** pp 69-71; **LE CUST:** pp 127-129; **LE MIG:** p 17; **COB PG:** pp 381, 436; **COB MIG:** pp 54, 85-86, 109-110, 436)

STORAGE

The first parameter of this option initializes all heap allocations, including all external data records acquired by a program, to the specified value when the storage for the external data is allocated. This also includes the WORKING-STORAGE acquired by a RENT program (unless a VALUE clause is used on the data item) when the program is first called or, for dynamic calls, when the program is canceled and then called again. Storage is not initialized on subsequent calls to the program. This can result in some overhead at run time depending on the number of external data records in the program and the size of the WORKING-STORAGE section.

Note: If you used the WSCLEAR option with VS COBOL II, STORAGE(00,NONE,NONE) is the equivalent option with Language Environment.

The second parameter of this option initializes all heap storage when it is freed.

The third parameter of this option initializes all DSA (stack) storage when it is allocated. The amount of overhead depends on the number of routines called (subroutines and library routines) and the amount of LOCAL-STORAGE data items that are used. This can have a significant impact on the CPU time of an application that is call intensive.

Performance considerations using STORAGE:

On the average, STORAGE(00,00,00) was 17% slower than STORAGE(NONE,NONE,NONE), with a range of equivalent to 130% slower. One RENT program calling a RENT subprogram using IS INITIAL on the PROGRAM-ID statement with a 40 MB WORKING-STORAGE was 28% slower. Note that when using call intensive applications, the degradation can be 200% slower or more.

On the average, STORAGE(00,NONE,NONE) was equivalent to STORAGE(NONE,NONE,NONE). One RENT program calling a RENT subprogram using IS INITIAL on the PROGRAM-ID statement with a 40 MB WORKING-STORAGE was 4% slower.

On the average, STORAGE(NONE,00,NONE) was equivalent to STORAGE(NONE,NONE,NONE). One RENT program calling a RENT subprogram using IS INITIAL on the PROGRAM-ID statement with a 40 MB WORKING-STORAGE was 13% slower.

On the average, STORAGE(NONE,NONE,00) was 17% slower than STORAGE(NONE,NONE,NONE), with a range of equivalent to 130% slower. One RENT program calling a RENT subprogram using IS INITIAL on the PROGRAM-ID statement with a 40 MB WORKING-STORAGE was 6% slower. Note that when using call intensive applications, the degradation can be 200% slower or more.

Note: The two RENT program tests and the call intensive tests measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms is not degraded as much.

(**LE REF:** pp 76-78; **LE CUST:** pp 134-136; **COB MIG:** pp 55-57, 83)

TEST

The TEST option specifies the conditions under which Debug Tool assumes control when the user application is invoked. Since this may result in Debug Tool being initialized and invoked, there may be some additional overhead when using TEST. Specifying NOTEST will eliminate this overhead.

Performance data using TEST is not available at this time.

(**LE REF:** pp 85-87; **LE CUST:** pp 142-144)

TRAP

The TRAP option allows LE to intercept an abnormal termination (abend), provide the abend information, and then terminate the LE run-time environment. TRAP(ON) also assures that all files are closed when an abend is encountered and is required for proper handling of the ON SIZE ERROR clause of arithmetic statements for overflow conditions. TRAP(OFF) prevents LE from intercepting the abend. In general, there will not be any significant impact on the performance of a COBOL application when using TRAP(ON).

Performance considerations using TRAP:

On the average, TRAP(ON) was equivalent to TRAP(OFF).

(**LE PG:** pp 194, 437, 456-457, 537, 598; **LE REF:** pp 92-95; **LE CUST:** pp 151-153; **COB PG:** pp 130, 160, 175, 221; **COB MIG:** pp 55, 57)

VCTRSAVE

The VCTRSAVE option specifies whether any language in the application uses the vector facility when the user-provided condition handlers are called. When the condition handlers use the vector facility, the entire vector environment has to be saved on every condition and restored upon return to the application code. Unless you need the function provided by VCTRSAVE(ON), you should run with VCTRSAVE(OFF) to avoid this overhead.

Performance considerations using VCTRSAVE:

On the average, VCTRSAVE(ON) was equivalent to VCTRSAVE(OFF).

(**LE REF:** p 97; **LE CUST:** pp 155-156)

COBOL and LE Features that Affect Run-Time Performance

COBOL and Language Environment have several installation and environment tuning features that can enhance the performance of your application. We will now look at some additional factors that should be considered for the application.

Storage Management Tuning

Storage management tuning can reduce the overhead involved in getting and freeing storage for the application program. With proper tuning, several GETMAIN and FREEMAIN calls can be eliminated. To better understand the need for storage tuning, let's look at how storage management works.

First of all, storage management was designed to keep a block of storage only as long as necessary. This means that during the execution of a COBOL program, if any block of storage becomes empty, it will be freed. This can be beneficial in a transaction environment (or any environment) where you want storage to

be freed as soon as possible so that other transactions (or applications) can make efficient use of the storage. However, it can also be detrimental if the last block of storage does not contain enough free space to satisfy a storage request by a library routine. For example, suppose that a library routine needs 2K of storage but there is only 1K of storage available in the last block of storage. The library routine will call storage management to request 2K of storage. Storage management will determine that there is not enough storage in the last block and issue a GETMAIN to acquire this storage (this GETMAINed size can also be tuned). The library routine will use it and then, when it is done, call storage management to indicate that it no longer needs this 2K of storage. Storage management, seeing that this block of storage is now empty, will issue a FREEMAIN to release the storage back to the operating system. Now, if this library routine or any other library routine that needs more than 1K of storage is called often, a significant amount of CPU time degradation can result because of the amount of GETMAIN and FREEMAIN activity.

Fortunately, there is a way to compensate for this with LE; it is called storage management tuning. The RPTSTG(ON) run-time option can help you in determining the values to use for any specific application program. You use the value returned by the RPTSTG(ON) option as the size of the initial block of storage for the HEAP, ANYHEAP, BELOWHEAP, STACK, and LIBSTACK run-time options. This will prevent the above from happening in an all VS COBOL II, COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL application. However, if the application also contains OS/VS COBOL programs that are being called frequently, the RPTSTG(ON) option may not indicate a need for additional storage. Increasing these initial values can also eliminate some storage management activity in this mixed environment.

The IBM supplied default storage options for batch applications are listed below:

```
ANYHEAP(16K,8K,ANYWHERE,FREE)
BELOWHEAP(8K,4K,FREE)
HEAP(32K,32K,ANYWHERE,KEEP,8K,4K)
LIBSTACK(4K,4K,FREE)
STACK(128K,128K,ANYWHERE,KEEP,512K,128K)
THREADHEAP(4K,4K,ANYWHERE,KEEP)
THREADSTACK(OFF,4K,4K,ANYWHERE,KEEP,128K,128K)
```

If you are running only COBOL applications, you can do some further storage tuning as indicated below:

```
STACK(64K,64K,ANYWHERE,KEEP)
```

If all of your applications are AMODE(31), you can use ALL31(ON) and STACK(,,ANYWHERE). Otherwise, you must use ALL31(OFF) and STACK(,,BELOW).

Overall below the line storage requirements have been substantially reduced by reducing the default storage options and by moving some of the library routines above the line. Here is a comparison of storage usage for a minimal STOP RUN COBOL program, using the IBM supplied default run-time options:

	Below	Above	
LE/370 Release 1.3	1156K	160K	
LE/370 Release 1.4	552K	316K	
LE for MVS & VM Release 1.5	272K	976K	
LE for OS/390 Release 1.6	272K	992K	
LE for OS/390 Release 1.7	276K	1000K	
LE for OS/390 Release 1.8	276K	1144K	
LE for OS/390 Release 1.9	276K	1156K	
LE for OS/390 Release 2.7	272K	1164K	
LE for OS/390 Release 2.8	272K	1192K	
LE for OS/390 Release 2.9	280K	1196K	
LE for OS/390 Release 2.10	292K	2144K	
LE for z/OS Release 1.2	280K	2388K	(comparable options - see note below)
LE for z/OS Release 1.2	88K	2520K	(default options)

Note: Beginning with LE for z/OS Release 1.2, the run-time defaults have changed to ALL31(ON),STACK(,,ANY). LE for OS/390 Release 2.10 and earlier run-time defaults were ALL31(OFF),STACK(,,BELOW).

As you can see, by moving to the latest release of Language Environment, you can reduce the amount of below the line storage used by your applications.

Performance data using Storage Management Tuning is not available at this time.

(**LE PG:** pp 165-173; **LE REF:** pp 15-16, 18-19, 32-34, 72-76, 87-91, 517-523; **LE CUST:** pp 38, 53-55, 76-77, 78-79, 92-94, 101-103, 131-134, 145-149, 175-192; **COB MIG:** pp 24-26, 54, 97-98, 319)

Storage Tuning User Exit

In an environment where Language Environment is being initialized and terminated constantly, such as CICS, IMS, or other transaction processing type of environments, tuning the storage options can improve the overall performance of the application. This helps to reduce the GETMAIN and FREEMAIN activity. The Language Environment storage tuning user exit is one way that you can manage the task of selecting the best values for your environment. The storage tuning user exit allows you to set storage values for your main programs without having to linkedit the values into your load modules.

(**LE CUST:** pp 175-192)

Calling IGZERRE

One way that an assembler program can set up a reusable run-time environment for COBOL is by calling IGZERRE. This module can be invoked to explicitly initialize and terminate COBOL's portion of the LE run-time environment. It allows a non-COBOL, non-LE-conforming program to initialize the LE run-time environment, thereby effectively establishing itself as the main COBOL program. As a result, the use of STOP RUN will cause control to be returned to the caller of the routine that invoked the IGZERRE initialization. IGZERRE is an enhanced version of ILBOSTP0 (for OS/VS COBOL) and accomplishes the same results as ILBOSTP0. Although, IGZERRE has been designed for Enterprise COBOL, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II applications, it also supports OS/VS COBOL applications. Using IGZERRE has the added benefits of supporting applications running above the 16 MB line, allowing the application to terminate the COBOL portion of the LE run-time environment and improving the performance of the application Using ILBOSTP0 in an LE environment will set up both the OS/VS COBOL and the COBOL portion of the LE environments whereas using IGZERRE will set up only the COBOL portion of the LE environment (the OS/VS COBOL environment will be set up only as needed).

WARNING: It is strongly recommended that ILBOSTP0 be converted to IGZERRE INIT and IGZERRE TERM calls since ILBOSTP0 does not provide a way to terminate the reusable run-time environment. As a result, storage may not be freed and modules may not be deleted upon termination of the application. This can result in an eventual out-of-storage condition in some environments. Additionally, since ILBOSTP0 is AMODE 24, it will automatically set ALL31(OFF) and STACK(,,BELOW) when used under Language Environment.

The semantic changes and performance benefits of using this method are the same as when using the RTEREUS run-time option. See “Using IGZERRE” on page 45 for an example of using IGZERRE.

Performance considerations using IGZERRE (measuring CALL overhead only):

One testcase (Assembler calling COBOL) using IGZERRE was 99% faster than not using IGZERRE.

See “First Program Not COBOL” on page 21 for additional performance considerations comparing calling IGZERRE with other environment initialization techniques.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

| See “Modifying COBOL's Reusable Environment Behavior” on page 17 for additional performance considerations when running in a COBOL reusable environment.

| (**LE CUST:** pp 62-63; **COB PG:** pp 381, 436; **COB MIG:** pp 69-70, 84-86, 109-110, 273)

Using the CEEENTRY and CEETERM Macros

To improve the performance of Assembler calling COBOL, you can make the Assembler program LE-conforming. This can be done using the CEEENTRY and CEETERM macros provided with LE. See “Using CEEENTRY” on page 42 for an example of using the CEEENTRY and CEETERM macros.

Performance considerations using the CEEENTRY and CEETERM macros (measuring CALL overhead only):

One testcase (Assembler calling COBOL) using the CEEENTRY and CEETERM macros was 99% faster than not using them.

See “First Program Not COBOL” on page 21 for additional performance considerations comparing using CEEENTRY and CEETERM with other environment initialization techniques.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

| (**LE PG:** pp 471-472, 474, 477-484, 492-495)

Using Preinitialization Services (CEEPIPI)

LE preinitialization services (CEEPIPI) can also be used to improve the performance of Assembler calling COBOL. LE preinitialization services lets an application initialize the LE environment once, execute multiple LE-conforming programs, the then explicitly terminate the LE environment. This substantially reduces the use of system resources that would have been required to initialize and terminate the LE environment for each program of the application. See “Using CEEPIPI with Call_Sub” on page 47 for an example of using CEEPIPI to call a COBOL subprogram and “Using CEEPIPI with Call_Main” on page 49 for an example of using CEEPIPI to call a COBOL main program.

Performance considerations using CEEPIPI (measuring CALL overhead only):

One testcase (Assembler calling COBOL) using CEEPIPI to invoke the COBOL program as a subprogram was 99% faster than not using CEEPIPI.

The same program using CEEPIPI to invoke the COBOL program as a main program was 95% faster than not using CEEPIPI.

See “First Program Not COBOL” on page 21 for additional performance considerations comparing using CEEPIPI with other environment initialization techniques.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

| (**LE PG:** pp 501-532; **COB MIG:** p 97, 273)

Using Library Routine Retention (LRR)

LRR is a function that provides a performance improvement for those applications or subsystems running on MVS with the following attributes:

- the application or subsystem invokes programs that require LE
- the application or subsystem is not LE-conforming (i.e., LE is not already initialized when the application or subsystem invokes programs that require LE)
- the application or subsystem repeatedly invokes programs that require LE running under the same MVS task
- the application or subsystem is not using LE preinitialization services

LRR is useful for assembler drivers that repeatedly call LE-conforming languages and for IMS/TM regions. LRR is supported only under MVS and not under VM. Also, LRR is not supported under CICS. See "IMS" on page 23 for information on using LRR under IMS.

When LRR has been initialized, LE keeps a subset of its resources in memory after the environment terminates. As a result, subsequent invocations of programs in the same MVS task that caused LE to be initialized are faster because the resources can be reused without having to be reacquired and reinitialized. The resources that LE keeps in memory upon LE termination are:

- LE run-time load modules
- storage associated with these load modules
- storage for LE startup control blocks

When LRR is terminated, these resources are released from memory.

LE preinitialization services and LRR can be used simultaneously. However, there is no additional benefit by using LRR when LE preinitialization services are being used. Essentially, when LRR is active and a non-LE-conforming application uses preinitialization services, LE remains preinitialized between repeated invocations of LE-conforming programs and does not terminate. Upon return to the non-LE-conforming application, preinitialization services can be called to terminate the LE environment, in which case LRR will be back in effect. See "Using CEELRR" on page 43 for an example of using LRR.

Performance considerations using LRR:

One testcase (Assembler calling COBOL) using LRR was 96% faster than using not using LRR.

See "First Program Not COBOL" on page 21 for additional performance considerations comparing using LRR with other environment initialization techniques.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

(**LE PG:** pp 474-477; **LE CUST:** p 57; **COB MIG:** p 96-97)

Modifying COBOL's Reusable Environment Behavior

Using the IGZRREOP customization job can be used to improve the performance of running in a COBOL reusable environment. REUSENV=COMPAT provides behavior that is compatible with VS COBOL II's reusable environment. With this setting, when a program check occurs while the COBOL reusable environment is dormant (i.e., after returning from the topmost COBOL program back to the non-Language Environment conforming assembler caller), a S0Cx abend will occur, but it significantly affects the performance of a such an application running under Language Environment. This degradation is due to an ESPIE RESET being issued prior to the return from COBOL to the assembler driver and then an ESPIE SET upon each re-entry to the topmost COBOL program.

| REUSENV=OPT changes this behavior by allowing Language Environment to trap all program checks, including those that occur while the COBOL reusable environment is dormant. This option provides behavior that is not the same as VS COBOL II and will result in an abend 4036 if a program check occurs while the COBOL reusable environment is dormant. However, since an ESPIE RESET and ESPIE SET do not have to be issued between each invocation of the topmost COBOL program, performance will be improved over using REUSENV=COMPAT.

| Sample source code to make these changes is in members IGZERREO and IGZWARRE of the SCEESAMP dataset.

| Performance considerations using IGZRREOP:

| When using IGZRREOP with REUSENV=OPT, assembler programs calling COBOL programs repeatedly under COBOL's reusable run-time environment can be 60 to 90% faster than using IGZRREOP with REUSENV=COMPAT.

| (**LE CUST:** pp 62-63)

Library in the LPA/ELPA

Placing the COBOL and the LE library routines in the Link Pack Area (LPA) or Extended Link Pack Area (ELPA) can also help to improve total system performance. This will reduce the real storage requirements for the entire system for COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, Enterprise COBOL, VS COBOL II RES, or OS/VS COBOL RES applications since the library routines can be shared by all applications instead of each application having its own copy of the library routines. For a list of COBOL library routines that are eligible to be placed in the LPA/ELPA, see members CEEWLPA and IGZWMLP4 in the SCEESAMP dataset.

Placing the library routines in a shared area will also reduce the I/O activity since they are loaded only once when the system is started and not for each application program.

Performance data placing the library routines in the LPA/ELPA is not available at this time.

| (**LE CUST:** pp 7, 43-44, 229-250; **LE MIG:** pp 3-4; **COB MIG:** p 31)

Using CALLs

When using CALLs, be sure to consider using nested programs when possible. The performance of a CALL to a nested program is faster than an external static CALL; external dynamic calls are the slowest. CALL identifier is slower than dynamic CALL literal. Additionally, you should consider space management tuning (mentioned earlier in this paper) for all CALL intensive applications.

With static CALLs, all programs are link-edited together, and hence, are always in storage, even if you do not call them. However, there is only one copy of the bootstrapping library routines link-edited with the application.

With dynamic CALLs, each subprogram is link-edited separately from the others. They are brought into storage only if they are needed. However, each subprogram has its own copy of the bootstrapping library routines link-edited with it, bringing multiple copies of these routines in storage as the application is executing.

Performance considerations for using CALLs (measuring CALL overhead only):

CALL to nested programs was 50% to 60% faster than static CALL.

Static CALL literal was 45% to 55% faster than dynamic CALL literal.

Static CALL literal was 60% to 65% faster than dynamic CALL identifier.

Dynamic CALL literal was 15% to 25% faster than dynamic CALL identifier.

Note: These tests measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

(**COB PG:** pp 394-400)

Using IS INITIAL on the PROGRAM-ID Statement

The IS INITIAL clause on the PROGRAM-ID statement specifies that when a program is called, it and any programs that it contains will be entered in their initial or first-time called state.

Performance considerations for using IS INITIAL on the PROGRAM-ID statement (measuring CALL overhead only):

One RENT program calling a RENT subprogram 100 times using IS INITIAL on the PROGRAM-ID statement was 1000% to 1500% slower than not using IS INITIAL, depending on the size of WORKING-STORAGE.

Note: These tests measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprogram is not degraded as much.

(**COB PG:** pp 6, 14-15, 130, 161, 393, 395-396; **COB LRM:** pp 81, 84)

Using IS RECURSIVE on the PROGRAM-ID Statement

The IS RECURSIVE clause on the PROGRAM-ID statement specifies that the COBOL program can be recursively called while a previous invocation is still active. The IS RECURSIVE clause is required for all programs that are compiled with the THREAD compiler option.

Performance considerations for using IS RECURSIVE on the PROGRAM-ID statement (measuring CALL overhead only):

One testcase (Assembler repeatedly calling COBOL) using IS RECURSIVE was 4% slower than not using IS RECURSIVE.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms is not degraded as much.

(**COB PG:** pp 6, 17, 403; **COB LRM:** pp 81, 83)

Other Product Related Factors that Affect Run-Time Performance

It is important to understand COBOL's interaction with other products in order to enhance the performance of your application. We will now look at some product related factors that should be considered for the application.

Mixing Assembler with VS COBOL II

If the application contains assembler calling or being called by VS COBOL II using the LE library, a significant amount of CPU time can be saved by recompiling the COBOL program with COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL for z/OS & OS/390 in the following cases:

- Assembler statically calling COBOL repeatedly
- COBOL dynamically calling Assembler which then does a LOAD of a second COBOL program and calls it repeatedly

| Performance considerations for Assembler and COBOL (measuring CALL overhead only):

| Assembler statically calling Enterprise COBOL 100 times is up to 25% faster than calling VS COBOL II using the LE library.

| Enterprise COBOL calling Assembler which LOADs a second Enterprise COBOL program and calls it 100,000 times is up to 80% faster than using VS COBOL II with the LE library.

Mixing VS COBOL II or COBOL/370 Rel 1 with COBOL for MVS & VM Rel 2 or later

| If a COBOL for MVS & VM Release 2, COBOL for OS/390 & VM Version 2, or Enterprise COBOL application program statically calls a COBOL/370 Release 1 program or a VS COBOL II program, you must ensure that the proper bootstrap routines are linked with the application. If you are linking only object decks (output from the compiler), the normal link-edit process will do this. However, if you have previously link-edited load modules that you are including in the application, you must do the following:

- if a VS COBOL II RES program was link-edited with the VS COBOL II Release 4 run-time library without APAR PN74000, you must include a REPLACE IGZEBST control statement in the linkage editor input.
- if a VS COBOL II RES program was link-edited with the LE/370 Release 2 or 3 run-time library without APAR PN74011, you must include a REPLACE IGZEBST control statement in the linkage editor input.

Additionally, to have the best performance, you should also do the following:

- if a COBOL/370 Release 1 program was link-edited with the LE/370 Release 2, 3, or 4 run-time library without APAR PN74011, you should include a REPLACE IGZCBSN control statement in the linkage editor input.
- if a VS COBOL II NORES program was link-edited with the LE/370 Release 2, 3, or 4 run-time library without APAR PN74011, you should include a REPLACE IGZENRI control statement in the linkage editor input.
- if a VS COBOL II RES program was link-edited with the LE/370 Release 4 run-time library without APAR PN74011, you should include a REPLACE IGZEBST control statement in the linkage editor input.

| (**COB MIG:** pp 75-103, 219-227)

Mixing OS/VS COBOL with COBOL/370 Rel 1 or COBOL for MVS & VM Rel 2 or later

If the application program is an OS/VS COBOL program using the LE library or if the application program
| has a mixture of OS/VS COBOL and COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM,
| or Enterprise COBOL, there will be some degradation at run time since both the OS/VS COBOL environment and the LE environment must be initialized and cleaned up. Converting the entire application to
| COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL will eliminate the need for setting up both environments.

| Performance data using mixed OS/VS COBOL and COBOL/370, COBOL for MVS & VM, COBOL for
| OS/390 & VM, or Enterprise COBOL programs is not available at this time.

| (**COB MIG:** pp 63-74, 219-227)

First Program Not COBOL

If the first program in the application is not COBOL or it is not LE-conforming, there can be a significant degradation if COBOL is repeatedly called since the COBOL environment must be initialized and terminated each time a COBOL main program is invoked. This overhead can be reduced by doing one of the following (listed in order of most improvement to least improvement):

- Use the CEEENTRY and CEETERM macros in the first program of the application to make it an LE-conforming program
- Call the first program of the application from a COBOL stub program (a program that just has a call statement to the original first program)
- Call CEEPIPI from the first program of the application to initialize the LE environment, invoke the COBOL program, and then terminate the LE environment when the application is complete
- Use the run-time option RTEREUS to initialize the run-time environment for reusability, making all COBOL main programs become subprograms
- Call IGZERRE from the first program of the application to make it appear as the COBOL main program and to initialize the run-time environment for reusability
- Call ILBOSTP0 (when using OS/VS COBOL) from the first program of the application to make it appear as the COBOL main program. This is provided for compatibility with OS/VS COBOL; calling IGZERRE is preferred over calling ILBOSTP0.
- Use the Library Routine Retention (LRR) function (similar to the function provided by the LIBKEEP run-time option in VS COBOL II)
- Place the LE library routines in the LPA, ELPA, or NSS. The list of routines to put in the LPA, EPLA, or NSS is release dependent and is the same routines listed under the IMS preload list considerations on page 23. Additionally, if the application is a VS COBOL II application using the LE library, include the following library routines: IGZCTCO, IGZEINI, IGZEPLF, and IGZEPCL. An alternative method (when not using RTEREUS, IGZERRE, or CEEPIPI) is to load these routines using the LOAD macro prior to the first time COBOL is called.

Make sure that you fully understand the implications of each one before you use them.

The performance considerations for Assembler calling COBOL will be presented in two different ways. The first will compare each of the tuning possibilities with using a standard, non-tuned assembler program to repeatedly call COBOL (the slowest method). The second will show the same comparison using an LE-conforming assembler program using the CEEENTRY and CEETERM macros (the fastest method).

Performance considerations for Assembler calling COBOL 50,000 times (measuring CALL overhead only) compared to a non-tuned assembler program, in order of most improvement to least improvement:

| CALL overhead was 99% faster when using the CEEENTRY and CEETERM macros (LE-conforming assembler)

| CALL overhead was 99% faster when calling IGZERRE with REUSENV=OPT before calling COBOL.

| CALL overhead was 99% faster when calling the Assembler program from a COBOL stub.

| CALL overhead was 99% faster when calling CEEPIPI to invoke the COBOL program as a subprogram.

| CALL overhead was 99% faster when calling IGZERRE with REUSENV=COMPAT before calling COBOL.

| CALL overhead was 99% faster when calling ILBOSTP0 before calling COBOL.

| CALL overhead was 99% faster when using the RTEREUS run-time option.

| CALL overhead was 98% faster when using LRR.

CALL overhead was 98% faster when calling CEEPIPI to invoke the COBOL program as a main program.

CALL overhead was 96% faster when using the LOAD SVC to load the LE library routines before calling the COBOL program.

Performance considerations for placing the library routines in the LPA/ELPA are not available at this time.

Performance considerations for Assembler calling COBOL 50,000 times (measuring CALL overhead only) compared to an LE-conforming assembler program, in order of least degradation to most degradation:

CALL overhead was 8% slower when calling IGZERRE with REUSENV=OPT before calling COBOL.

CALL overhead was 22% slower when calling the Assembler program from a COBOL stub.

CALL overhead was 83% slower when calling CEEPIPI to invoke the COBOL program as a subprogram.

CALL overhead was 1,338% slower when calling IGZERRE with REUSENV=COMPAT before calling COBOL.

CALL overhead was 1,394% slower when calling ILBOSTP0 before calling COBOL.

CALL overhead was 1,429% slower when using the RTEREUS run-time option.

CALL overhead was 7,815% slower when using LRR.

CALL overhead was 9,474% slower when calling CEEPIPI to invoke the COBOL program as a main program.

CALL overhead was 19,476% slower when using the LOAD SVC to load the LE library routines before calling the COBOL program.

CALL overhead was 487,485% slower when not using any of the above methods (non-tuned assembler)

In order to show the magnitude of the difference in CPU times using each of 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.

	CPU Time (seconds)	EXCP Counts	Elapsed Time
Untuned assembler	346.186	6,106,608	13 hrs
SVC LOAD lib rtns	13.899	126	22 secs
Using CEEPIPI main	6.798	127	9 secs
Using LRR	5.620	131	9 secs
Using RTEREUS	1.086	126	2 secs
Using ILBOSTP0	1.061	134	3 secs
Using IGZERRE COMPAT	1.021	130	2 secs
Using CEEPIPI sub	0.130	125	1 sec
Using a COBOL stub	0.087	126	1 sec
Using IGZERRE OPT	0.077	130	1 sec
Using CEEENTRY	0.071	124	1 sec

Notes:

1. These tests measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.
2. See “Modifying COBOL's Reusable Environment Behavior” on page 17 for information on how to change the reusable environment behavior setting of REUSENV to COMPAT or OPT.

(**COB MIG:** pp 84-86, 96-97, 109-110, 273, 436)

IMS

If the application is running under IMS, preloading the application program and the library routines can help to reduce the load/search overhead, as well as reduce the I/O activity. This is especially true for the library routines since they are used by every COBOL program. When the application program is preloaded, subsequent requests for the program are handled faster because it does not have to be fetched from external storage. The RENT compiler option is required for preloaded applications.

| (**COB PG:** pp 379-380, 406, 546-547; **COB MIG:** pp 32, 84-85, 315-318)

Using the Library Routine Retention (LRR) function can significantly improve the performance of COBOL transactions running under IMS/TM. LRR provides function similar to that of the VS COBOL II LIBKEEP run-time option. It keeps the LE environment initialized and retains in memory any loaded LE library routines, storage associated with these library routines, and storage for LE startup control blocks. To use LRR in an IMS dependent region, you must do the following:

- In your startup JCL or procedure to bring up the IMS dependent region, specify the PREINIT=xx parameter (xx is the 2-character suffix of the DFSINTxx member in your IMS PROCLIB dataset)
- Include the name CEELRRIN in the DFSINTxx member of your IMS PROCLIB dataset
- Bring up your IMS dependent region

You can also create your own load module to initialize the LRR function by modifying the CEELRRIN sample source in the SCEESAMP dataset. If you do this, use your module name in place of CEELRRIN above.

| (**LE PG:** pp 474-475; **LE CUST:** p 57; **COB MIG:** pp 96-97)

WARNING: If the RTEREUS run-time option is used, the top level COBOL programs of all applications must be preloaded. Note that using RTEREUS will keep the LE environment up until the region goes down or until a STOP RUN is issued by a COBOL program. This means that every program and its working storage (from the time the first COBOL program was initialized) is kept in the region. Although this is very fast, you may find that the region may soon fill to overflowing, especially if there are many different COBOL programs that are invoked.

When not using RTEREUS or LRR, it is recommended that you preload the following library routines:

- For all COBOL applications:
  CEEBINIT, IGZCPAC, IGZCPCO, CEEEV005, CEEPLPKA, IGZETRM, IGZEINI, and IGZCLNK
- If the application contains VS COBOL II programs:
  IGZCTCO, IGZEPLF, and IGZEPCL
- If the application contains OS/VS COBOL programs:
  IGZCTCO, IGZEPLF, IGZCLNC, and IGZEPCL
  any ILBO library routines that you previously preloaded.
  If ILBOSTT0 was in the preload list with OS/VS COBOL, replace it with ILBOSTT2 and make sure that it is in there twice.
  You should, at a minimum, include all heavily used ILBO library routines in your preload list.

Preloading should reduce the amount of I/O activity associated with loading and deleting these routines for each transaction.

| Since the two COBPACKs, IGZCPAC and IGZCPCO, only contain those COBOL library routines that
| run above the line (AMODE 31, RMODE ANY), you should also preload any of the below the line rou-
| tines that you need. A list of the below the line routines can be found in the Language Environment
| Customization manual.

| (**LE CUST:** pp 231-232)

Additionally, heavily used application programs can be compiled with the RENT compiler option and pre-loaded to reduce the amount of I/O activity associated with loading them.

The TRUNC(OPT) compiler option can be used if:

- you are not using a database that was built by a non-COBOL program
- your usage of all binary data items conforms to the PICTURE and USAGE specifications for the data items (e.g., no pointer arithmetic using binary data types).

Otherwise, you should use the TRUNC(BIN) compiler option or COMP-5 data types.

Performance data using IBM Enterprise COBOL with IMS is not available at this time.

| (**COB PG:** pp 317-318)

CICS

Language Environment uses more transaction storage than VS COBOL II. This is especially noticeable when more than one run-unit (enclave) is used since storage is managed at the run-unit level with LE. This means that HEAP, STACK, ANYHEAP, etc. are allocated for each run-unit under LE. With VS COBOL II, stack (SRA) and heap storage are managed at the transaction level. Additionally, there are some LE control blocks that need to be allocated.

| (**COB MIG:** pp 25-26)

In order to minimize the amount of below the line storage used by LE under CICS, you should run with ALL31(ON) and STACK(,,ANYWHERE) as much as possible. In order to do this, you have to identify all
| of your AMODE(24) COBOL programs that are not OS/VS COBOL. Then you can either make the necessary coding changes to make them AMODE(31) or you can link-edit a CEEUOPT with ALL31(OFF) and STACK(,,BELOW) as necessary for those run units that need it. You can find out how much storage a particular transaction is using by looking at the auxiliary trace data for that transaction. You do not need to be concerned about OS/VS COBOL programs since the LE run-time options do not affect OS/VS COBOL programs running under CICS. Also, if the transaction is defined with TASKDATALOC(ANY) and ALL31(ON) is being used and the programs are compiled with DATA(31), then LE does not use any below the line storage for the transaction under CICS, resulting in some additional below the line storage savings.

| There are two CICS SIT options that can be used to reduce the amount of GETMAIN and FREEMAIN
| activity, which will help the response time. The first one is the RUWAPOOL SIT option. You can set
| RUWAPOOL to YES to reduce the GETMAIN and FREEMAIN activity. The second is the AUTODST
| SIT option. If you are using CICS Transaction Server Version 1 Release 3 or later, you can also set
| AUTODST to YES to cause Language Environment to automatically tune the storage for the CICS region.
| Doing this should result in fewer GETMAIN and FREEMAIN requests in the CICS region. Additionally,
| when using AUTODST=YES, you can also use the storage tuning user exit (see “Storage Tuning User Exit” on page 15) to modify the default behavior of this automatic storage tuning.

| (**LE CUST:** pp 53-55; **COB MIG:** p 99)

The RENT compiler option is required for an application running under CICS. Additionally, if the program is run through the CICS translator (i.e., it has EXEC CICS commands in it), it must also use the
| NODYNAM compiler option. CICS Transaction Server 1.3 or later is required for Enterprise COBOL.

| (**COB PG:** pp 308, 406; **COB MIG:** p 99)

| Enterprise COBOL, COBOL for OS/390 & VM, COBOL for MVS & VM, and COBOL/370 support static
| and dynamic calls to Enterprise COBOL, COBOL for OS/390 & VM, COBOL for MVS and VM, COBOL/370, and VS COBOL II subprograms containing CICS commands or dependencies. Static calls are done with the CALL literal statement and dynamic calls are done with the CALL identifier statement. Converting EXEC CICS LINKs to COBOL CALLs can improve transaction response time and reduce virtual

storage usage. Neither Enterprise COBOL, COBOL for OS/390 & VM, COBOL for MVS & VM, nor COBOL/370 supports calls to or from OS/VS COBOL programs in a CICS environment. In this case, EXEC CICS LINK must be used.

Note: : When using EXEC CICS LINK under Language Environment, a new run-unit (enclave) will be created for each EXEC CICS LINK. This means that new control blocks will be allocated and subsequently freed for each LINKed to program. This will result in an increase in the number of storage requests. When using ALL31(ON), there is at least one storage request for each enclave. When using ALL31(OFF), there are two to three storage requests for each enclave. If storage management tuning has not been done, you may experience more storage requests per enclave. As a result of a new enclave being created for each EXEC CICS LINK, the CPU time performance will also be degraded when compared to VS COBOL II. If your application uses many EXEC CICS LINKs, you can avoid this extra overhead by using COBOL CALLs whenever possible.

(**COB PG:** pp 546-547; **COB MIG:** pp 25-26)

If you are using the COBOL CALL statement to call a program that has been translated with the CICS translator, you must pass DFHEIBLK and DFHCOMMAREA as the first two parameters on the CALL statement. However, if you are calling a program that has not been translated, you should not pass DFHEIBLK and DFHCOMMAREA on the CALL statement. Additionally, if your called subprogram does not use any of the EXEC CICS condition handling commands, you can use the run-time option CBLPSHPOP(OFF) to eliminate the overhead of doing an EXEC CICS PUSH HANDLE and an EXEC CICS POP HANDLE that is done for each call by the LE run-time. The CBLPSHPOP setting can be changed dynamically by using the CLER transaction.

(**LE PG:** pp 114, 438; **LE REF:** p 20; **LE CUST:** p 81; **COB PG:** pp 367, 373-374; **COB MIG:** pp 57, 100-102, 221)

As long as your usage of all binary (COMP) data items in the application conforms to the PICTURE and USAGE specifications, you can use TRUNC(OPT) to improve transaction response time. This is recommended in performance sensitive CICS applications. If your usage of any binary data item does not conform to the PICTURE and USAGE specifications, you can either use a COMP-5 data type or increase the precision in the PICTURE clause instead of using the TRUNC(BIN) compiler option. Note that the CICS translator does not generate code that will cause truncation and the CICS co-processor uses COMP-5 data types which does not cause truncation. If you were using NOTRUNC with your OS/VS COBOL programs without problems, TRUNC(OPT) on IBM Enterprise COBOL behaves in a similar way. For additional information on the TRUNC option, please refer to the compiler options section of this paper.

Performance considerations using CICS (measuring call overhead only):

- One testcase was 445% slower using EXEC CICS LINK compared to using COBOL dynamic CALL with CBLPSHPOP(ON)
- The same testcase was 4547% slower using EXEC CICS LINK compared to using COBOL dynamic CALL with CBLPSHPOP(OFF)
- The same testcase was 753% slower using COBOL dynamic CALL with CBLPSHPOP(ON) compared to using COBOL dynamic CALL with CBLPSHPOP(OFF)

In order to show the magnitude of the difference in CPU times using each of 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.

	CPU Time (seconds)
EXEC CICS LINK	0.790
COBOL dynamic CALL CBLPSHPOP(ON)	0.145
COBOL dynamic CALL CBLPSHPOP(OFF)	0.017

Note: This test measured only the overhead of 20,000 CALLs/EXEC CICS LINKs (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

(**COB PG:** pp 317-318, 372; **COB MIG:** pp 154, 208)

DB2

As long as your usage of all binary (COMP) data items in the application conforms to the PICTURE and USAGE specifications and your binary data was created by COBOL programs, you can use TRUNC(OPT) to improve performance under DB2. This is recommended in performance sensitive DB2 applications. If your usage of any binary data item does not conform to the PICTURE and USAGE specifications, you should use COMP-5 data types or use the TRUNC(BIN) compiler option. If you were using NOTRUNC with your OS/VS COBOL programs without problems, TRUNC(OPT) on COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL behaves in a similar way. For additional information on the TRUNC option, please refer to the compiler options section of this paper.

The RENT compiler option must be used for COBOL programs used as DB2 stored procedures.

Performance data using IBM Enterprise COBOL with DB2 is not available at this time.

(**COB PG:** pp 308-309, 317-318, 376, 406)

DFSORT

Use the FASTSRT compiler option to improve the performance of most sort operations. With FASTSRT, the DFSORT product performs the I/O on input and/or output files named in either or both of the SORT ... USING or SORT ... GIVING statements. If you have input or output procedures for your sort files, you cannot use the FASTSRT option. The complete list of requirements is contained in the Programming Guides.

Performance considerations using DFSORT:

One program that processed 100,000 records was 45% faster when using FASTSRT compared to using NOFASTSRT and used about 50% fewer EXCPs.

(**COB PG:** pp 189-191, 194, 294, 544)

Efficient COBOL Coding Techniques

This section focuses on how the source code can be modified to tune a program for better performance. Coding style, as well as data types, can have a significant impact on the performance of an application. Producing higher performing code usually has a far greater impact than that of tuning the application via compiler and run-time options, but producing such code may not be a viable option for many existing applications since it does require modifying the source code and at times may even require an extensive knowledge of how the program works. However, these techniques should be considered for all new applications.

| (**COB PG:** pp 533-542)

Data Files

Planning how the files will be created and used is an important factor in determining efficient file characteristics for the application. Some of the characteristics that affect the performance of file processing are: file organization, access method, record format, and blocksize. Some of these are discussed in more detail below.

QSAM Files

When using QSAM files, use large block sizes whenever possible by using the BLOCK CONTAINS clause on your file definitions (the default with COBOL is to use unblocked files). If you are using DFP Version 3 Release 1 or later, you can have the system determine the optimal blocksize for you by specifying the BLOCK CONTAINS 0 clause for any new files that you are creating and omitting the BLKSIZE parameter in your JCL for these files. This should significantly improve the file processing time (both in CPU time and elapsed time).

Additionally, increasing the number of I/O buffers for heavy I/O jobs can improve both the CPU and elapsed time performance, at the expense of using more storage. This can be accomplished by using the BUFNO subparameter of the DCB parameter in the JCL or by using the RESERVE clause of the SELECT statement in the FILE-CONTROL paragraph. Note that if you do not use either the BUFNO subparameter or the RESERVE clause, the system default will be used.

QSAM buffers can be allocated above the 16 MB line if all of the following are true:

- the programs are compiled with VS COBOL II Release 3.0 or higher, COBOL/370 Release 1.0 or
| higher, IBM COBOL for MVS & VM Release 2.0 or higher, IBM COBOL for OS/390 & VM, or IBM
| Enterprise COBOL
- the programs are running with LE/370 Release 3.0 or higher, IBM Language Environment for MVS &
| VM Release 5.0 or higher, Language Environment for OS/390 & VM, or z/OS Language Environment
- | the programs are compiled with RENT and DATA(31) or compiled with NORENT and
| RMODE(ANY)
- | the program is executing in AMODE 31
- | the program is executing on MVS
- the ALL31(ON) run-time option is used (for EXTERNAL files)

| (**COB PG:** pp 35-36, 125-127, 138, 543)

Variable-Length Files

When writing to variable-length blocked sequential files, use the APPLY WRITE-ONLY clause for the file or use the AWO compiler option. This reduces the number of calls to Data Management Services to handle the I/Os.

For performance considerations using the APPLY-WRITE-ONLY clause or the AWO compiler option, see "AWO or NOAWO" on page 3.

| (**COB PG:** pp 11-12, 284)

VSAM Files

When using VSAM files, increase the number of data buffers (BUFND) for sequential access or index buffers (BUFNI) for random access. Also, select a control interval size (CISZ) that is appropriate for the application. A smaller CISZ results in faster retrieval for random processing at the expense of inserts, whereas a larger CISZ is more efficient for sequential processing. In general, using large CI and buffer space VSAM parameters may help to improve the performance of the application.

In general, sequential access is the most efficient, dynamic access the next, and random access is the least efficient. However, for relative record VSAM (ORGANIZATION IS RELATIVE), using ACCESS IS DYNAMIC when reading each record in a random order can be slower than using ACCESS IS RANDOM, since VSAM may prefetch multiple tracks of data when using ACCESS IS DYNAMIC. ACCESS IS DYNAMIC is optimal when reading one record in a random order and then reading several subsequent records sequentially.

Random access results in an increase in I/O activity because VSAM must access the index for each request.

If you use alternate indexes, it is more efficient to use the Access Method Services to build them than to use the AIXBLD run-time option. Avoid using multiple alternate indexes when possible since updates will have to be applied through the primary paths and reflected through the multiple alternate paths.

VSAM buffers can be allocated above the 16 MB line if all of the following are true:

- the programs are compiled with VS COBOL II Release 3.0 or higher, COBOL/370 Release 1.0 or
| higher, IBM COBOL for MVS & VM Release 2.0 or higher, IBM COBOL for OS/390 & VM, or IBM
| Enterprise COBOL
- the programs are running with LE/370 Release 3.0 or higher, IBM Language Environment for MVS &
| VM Release 5.0 or higher, Language Environment for OS/390 & VM, or z/OS Language Environment

| (**COB PG:** pp 164-165, 169-170, 546-547)

Data Types

Using the proper data types is also an important factor in determining the performance characteristics of an application. Some of these are discussed below.

| (**COB PG:** pp 37-44)

BINARY (COMP or COMP-4)

When using binary (COMP) data items, the use of the SYNCHRONIZED clause specifies that the binary data items will be properly aligned on halfword, fullword, or doubleword boundaries. This may enhance the performance of certain operations on some machines. Additionally, using signed data items with eight or fewer digits produces the best code for binary items. The following shows the general performance considerations (from most efficient to least efficient) for the number of digits of precision for signed binary data items (using PICTURE S9(n) COMP) using TRUNC(OPT):

- n is from 1 to 8

  for n from 1 to 4, arithmetic is done in halfword instructions where possible

  for n from 5 to 8, arithmetic is done in fullword instructions where possible

- n is from 10 to 17

  arithmetic is done in doubleword format

- n is 9

  fullword values are converted to doubleword format and then doubleword arithmetic is used (**this is SLOWER than any of the above**)

- n is 18

  doubleword values are converted to a higher precision format and then arithmetic is done using this higher precision (**this is the SLOWEST of all for binary data items**)

Notes:

1. Using 9 digits is slower than using 10 digits.
2. The TRUNC compiler option significantly affects the performance of all binary (BINARY, COMP, and COMP-4) data items.

Performance considerations for BINARY datatypes using TRUNC(STD):

using 1 to 8 digits is the fastest

using 9 digits is 20% slower than using 1 to 8 digits.

using 10 to 17 digits is 280% slower than using 1 to 8 digits.

using 18 digits is 510% slower than using 1 to 8 digits.

Performance considerations for BINARY datatypes using TRUNC(OPT):

using 1 to 8 digits is the fastest

using 9 digits is 75% slower than using 1 to 8 digits.

using 10 to 17 digits is 30% slower than using 1 to 8 digits.

using 18 digits is 1340% slower than using 1 to 8 digits.

Performance considerations for BINARY datatypes using TRUNC(BIN):

using 1 to 4 digits is the fastest

using 5 to 9 digits is 40% slower than using 1 to 4 digits.

using 10 to 18 digits is 2600% slower than using 1 to 4 digits.

(**COB PG:** pp 41, 317-319, 533-536)

COMP-5

Binary COMP-5 data items are similar to the above BINARY (COMP and COMP-4) data items with the exception that they always behave as if the TRUNC(BIN) compiler option were in effect for them. Hence, the performance recommendations for BINARY (COMP or COMP-4) data items also apply for COMP-5 data items.

Performance considerations for COMP-5:

- using 1 to 4 digits is the fastest
- using 5 to 9 digits is 35% slower than using 1 to 4 digits.
- using 10 to 18 digits is 2200% slower than using 1 to 4 digits.

(**COB PG:** pp 41-42, 317-318, 371, 376)

Data Conversions

Conversion to a common format is necessary for certain types of numeric operations when mixed data types are involved in the computation. This results in additional processing time and storage for these conversions. In order to minimize this overhead, it is recommended that the guidelines discussed below be followed.

(**COB PG:** pp 43-44, 536)

DISPLAY

Avoid using USAGE DISPLAY data items for computations (especially in areas that are heavily used for computations). When a USAGE DISPLAY data item is used, additional overhead is required to convert the data item to the proper type both before and after the computation. In some cases, this conversion is done by a call to a library routine, which can be expensive compared to using the proper data type that does not require any conversion.

Performance considerations for DISPLAY:

- using 1 to 15 digits (with an odd number of digits) is the fastest
- using 2 to 16 digits (with an even number of digits) is 16% slower than using 1 to 15 digits (with an odd number of digits)
- using 17 to 18 digits is 60-70% slower than using 1 to 15 digits (with an odd number of digits)

(**COB PG:** p 535)

PACKED-DECIMAL (COMP-3)

When using PACKED-DECIMAL (COMP-3) data items in computations, use 15 or fewer digits in the PICTURE specification to avoid the use of library routines for multiplication and division. A call to the library routine is very expensive when compared to doing the calculation in-line. Additionally, using a signed data item with an odd number of digits produces more efficient code since this uses an integral multiple of bytes in storage for the data item.

Performance considerations for PACKED-DECIMAL:

- using an odd number of digits is 5% to 20% faster than using an even number of digits
- using the 16 to 18 digits is up to 140% slower than using 1 to 15 digits

(**COB PG:** pp 535-536)

Comparing Data Types

When selecting your data types, it is important to understand the performance characteristics of them before you use them. Shown below are some performance considerations of doing several ADDs and SUBTRACTs on the various data types of the specified precision.

Performance considerations for comparing data types (using ARITH(COMPAT)):

- Packed decimal (COMP-3) compared to binary (COMP or COMP-4) with TRUNC(STD)
  - using 1 to 9 digits: packed decimal is 30% to 60% slower than binary
  - using 10 to 17 digits: packed decimal is 55% to 65% faster than binary
  - using 18 digits: packed decimal is 74% faster than binary
- Packed decimal (COMP-3) compared to binary (COMP or COMP-4) with TRUNC(OPT)
  - using 1 to 8 digits: packed decimal is 160% to 200% slower than binary
  - using 9 digits: packed decimal is 60% slower than binary
  - using 10 to 17 digits: packed decimal is 150% to 180% slower than binary
  - using 18 digits: packed decimal is 74% faster than binary
- Packed decimal (COMP-3) compared to binary (COMP or COMP-4) with TRUNC(BIN) or COMP-5
  - using 1 to 8 digits: packed decimal is 130% to 200% slower than binary
  - using 9 digits: packed decimal is 85% slower than binary
  - using 10 to 18 digits: packed decimal is 88% faster than binary
- DISPLAY compared to packed decimal (COMP-3)
  - using 1 to 6 digits: DISPLAY is 100% slower than packed decimal
  - using 7 to 16 digits: DISPLAY is 40% to 70% slower than packed decimal
  - using 17 to 18 digits: DISPLAY is 150% to 200% slower than packed decimal
- DISPLAY compared to binary (COMP or COMP-4) with TRUNC(STD)
  - using 1 to 8 digits: DISPLAY is 150% slower than binary
  - using 9 digits: DISPLAY is 125% slower than binary
  - using 10 to 16 digits: DISPLAY is 20% faster than binary
  - using 17 digits: DISPLAY is 8% slower than binary
  - using 18 digits: DISPLAY is 25% faster than binary
- DISPLAY compared to binary (COMP or COMP-4) with TRUNC(OPT)
  - using 1 to 8 digits: DISPLAY is 350% slower than binary
  - using 9 digits: DISPLAY is 225% slower than binary
  - using 10 to 16 digits: DISPLAY is 380% slower than binary
  - using 17 digits: DISPLAY is 580% slower than binary
  - using 18 digits: DISPLAY is 35% faster than binary
- DISPLAY compared to binary (COMP or COMP-4) with TRUNC(BIN) or COMP-5
  - using 1 to 4 digits: DISPLAY is 400% to 440% slower than binary
  - using 5 to 9 digits: DISPLAY is 240% to 280% slower than binary
  - using 10 to 18 digits: DISPLAY is 70% to 80% faster than binary

Fixed-Point vs Floating-Point

Plan the use of fixed-point and floating-point data types. You can enhance the performance of an application by carefully determining when to use fixed-point and floating-point data. When conversions are necessary, binary (COMP) and packed decimal (COMP-3) data with nine or fewer digits require the least amount of overhead when being converted to or from floating-point (COMP-1 or COMP-2) data. Also, when using fixed-point exponentiations with large exponents, the calculation can be done more efficiently by using operands that force the exponentiation to be evaluated in floating-point.

An example of forcing the exponentiation to be evaluated in floating-point is as follows:

```
01  A  PIC S9(6)V9(12) COMP-3 VALUE 0.
01  B  PIC S9V9(12)    COMP-3 VALUE 1.234567891.
01  C  PIC S9(10)      COMP-3 VALUE -99999.

COMPUTE A = (1 + B) ** C.          (original)
COMPUTE A = (1.0E0 + B) ** C.      (forced to floating-point)
```

The above example was forced to floating-point by changing the fixed-point constant value 1 to a floating-point constant value 1.0E0.

Performance considerations for fixed-point vs floating-point:

- forcing an exponentiation to be done in floating-point can be up to 98% faster than doing it in fixed-point

(**COB PG:** pp 53-54, 536-537)

Indexes vs Subscripts

Using indexes to address a table is more efficient than using subscripts since the index already contains the displacement from the start of the table and does not have to be calculated at run time. Subscripts, on the other hand, contain an occurrence number that must be converted to a displacement value at run time before it can be used. When using subscripts to address a table, use a binary (COMP) signed data item with eight or fewer digits (for example, using PICTURE S9(8) COMP for the data item). This will allow fullword arithmetic to be used during the calculations. Additionally, in some cases, using four or fewer digits for the data item may also offer some added reduction in CPU time since halfword arithmetic can be used.

Performance considerations for indexes vs subscripts (PIC S9(8)):

- using binary data items (COMP) to address a table is 30% slower than using indexes
- using packed decimal data items (COMP-3) to address a table is 300% slower than using indexes
- using DISPLAY data items to address a table is 450% slower than using indexes

(**COB PG:** pp 59-61, 537-538)

OCCURS DEPENDING ON

When using OCCURS DEPENDING ON (ODO) data items, ensure that the ODO objects are binary (COMP) to avoid unnecessary conversions each time the variable-length items are referenced. Some performance degradation is expected when using ODO data items since special code must be executed every time a variable-length data item is referenced. This code determines the current size of the item every time the item is referenced. It also determines the location of variably-located data items. Because this special code is out-of-line, it may inhibit some optimizations. Furthermore, code to manipulate variable-length data items is substantially less efficient than that for fixed-length data items. For example, the code to compare or move a variable-length data item may involve calling a library routine and is significantly slower than the equivalent

code for fixed-length data items. If you do use variable-length data items, copying them into fixed-length data items prior to a period of high-frequency use can reduce some of this overhead.

| Performance considerations for fixed-length vs variable-length tables:

| using variable-length tables is 5% slower then using a fixed-length table

| using a variable-length table that references the first complex ODO element is 7% slower then using a fixed-length table

| using a variable-length table that references a complex ODO element other than the first is 140% slower then using a fixed-length table

| (**COB PG:** pp 66-68, 537-540)

Program Design

Using the appropriate program design is another important factor in determining the performance characteristics of an application. If an inefficient design is used, then you are limited in the amount of performance tuning you can do to the application. The biggest performance improvement you can make is to use an efficient design. After the design is determined, then you can look at other things such as efficient algorithms, data structures and data types, and coding style. Some of these are discussed below.

Algorithms

Examine the underlying algorithms that have been selected before looking at the COBOL specifics. Improving the algorithms usually has a much greater impact on the performance than does improving the detailed implementation of the algorithm. As an example, consider two search algorithms: a sequential search and a binary search. Clearly, both of them will produce the same results and may, in fact, have almost the same performance for small tables. However, as the table size increases, the binary search will be much faster than the sequential search. As in this case of the two searches, you may have to do some additional coding to maintain a sorted table for the binary search, but the additional effort spent here is more than saved during the execution of the program.

| (**COB PG:** p 533)

Data Structures and Data Types

After deciding on the algorithm, look at the data structures and data types. Ensure that both are appropriate for the selected algorithm. The algorithm may in general be a fast one, but if the wrong data structures or types are used, the performance can degrade significantly. As an example, consider two PERFORM VARYING loops, one using a USAGE DISPLAY data item for the loop variable and the other using a COMPUTATIONAL data item. In the case of DISPLAY data item, data conversion must be done for each iteration of the loop, whereas in the COMPUTATIONAL data item, binary fullword arithmetic can be used. Once again, they will both produce the same results, but the loop using the COMPUTATIONAL data item will be much faster than the loop using the DISPLAY data item.

| Performance considerations for loop control variables (PIC S9(8)):

| using packed decimal (COMP-3) is 280% slower than using binary (COMP)

| using DISPLAY is 575% slower than using binary (COMP)

| (**COB PG:** pp 533, 535-537)

Coding Style

Examine the coding style. Ensure that the program is well structured, utilizing the structured coding constructs that are available with IBM Enterprise COBOL. Avoid using the GO TO statement (in particular, the altered GO TO statement) and avoid using PERFORMed procedures that involve irregular control flow (for example, a PERFORMed procedure that cannot reach the end of the procedure). The optimizer can optimize the code better and over larger blocks of code if the programs are well structured and don't have a "spaghetti-like" control flow. Additionally, the programs will be easier to maintain because of the structured logic flow.

| (**COB PG:** pp 533-534)

Factoring Expressions

Factor expressions where possible, especially in loops. The optimizer does not do the factoring for you. For evaluating arithmetic expressions, the compiler is bound by the left-to-right evaluation rules for COBOL. In order for the optimizer to recognize constant computations (that can be done at compile time) or duplicate computations (common subexpressions), move all constants and duplicate expressions to the left end of the expression or group them in parentheses.

| (**COB PG:** pp 534-535)

Symbolic Constants

If you want the optimizer to recognize a data item as a constant throughout the program, initialize it with a VALUE clause and don't modify it anywhere in the program (if a data item is passed BY REFERENCE to a subprogram, the optimizer considers it to be modified not only at this CALL statement, but also at all CALL statements).

| (**COB PG:** p 534)

Subscript Checking

When using tables, evaluate the need to verify subscripts. Using the SSRANGE option to catch the errors causes the compiler to generate special code at each subscript reference to determine if the subscript is out of bounds. However, if subscripts need to be checked in only a few places in the application to ensure that they are valid, then coding your own checks can improve the performance when compared to using the SSRANGE compiler option.

| (**COB PG:** pp 333, 537)

Subscript Usage

Additionally, try to use the tables so that the rightmost subscript varies the most often for references that occur close to each other in the program. The optimizer can then eliminate some of the subscript calculations because of common subexpression optimization.

| Performance considerations for table reference patterns (PIC S9(8)):

| when referencing tables sequentially, having the leftmost subscript vary the most often can be 50% slower than having the rightmost subscript vary the most often

| (**COB PG:** pp 537-540)

Searching

When using the SEARCH statement, place the most often used data near the beginning of the table for more efficient sequential searching. For better performance, especially when searching large tables, sort the data in the table and use the SEARCH ALL statement. This results in a binary search on the table.

| Performance considerations for search example:

| using a binary search (SEARCH ALL) to search a 100-element table was 6% faster than using a sequential search (SEARCH)

| using a binary search (SEARCH ALL) to search a 1000-element table was 83% faster than using a sequential search (SEARCH)

| **Note:** The size of the table directly affects the performance of the search. Larger tables benefit more from a binary search.

| (**COB PG:** pp 69-71, 538)

Recent Performance Improvements

COBOL has made performance improvements in some specific areas of the compiled code and library routines. Unless otherwise stated, all performance improvements apply to the indicated release and all subsequent releases of the product. Here is a brief summary of the latest improvements:

IBM COBOL for OS/390 & VM Version 2 Release 2 (2.2.0) compiler provides improved performance over IBM COBOL for OS/390 & VM Release 2 Version 1 (2.1.0) compiler in the following areas:

- TRUNC(BIN) usage

Performance considerations for TRUNC(BIN) improvement:

On the average, TRUNC(BIN) with COBOL for OS/390 & VM Version 2 Release 2 and later is 15% faster than TRUNC(BIN) with COBOL for OS/390 & VM Version 2 Release 1 and prior, with a range of equivalent to 95% faster.

Here is a breakdown of the 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   approx equivalent
                10-17 digits   70% faster
                18    digits   approx equivalent

BEFORE: 2.1.0 compiler with TRUNC(BIN) compared to 2.1.0 compiler w/ TRUNC(OPT)
      signed    1-4   digits    8 times slower
                5-8   digits   12 times slower
                9     digits    8 times slower
                10-17 digits   27 times slower
                18    digits    2 times slower
      unsigned  1-18  digits   equivalent

AFTER:  2.2.0 compiler with TRUNC(BIN) compared to 2.2.0 compiler w/ TRUNC(OPT)
      signed    1-4   digits   equivalent
                5-8   digits    1 1/2 to 2 times slower
                9     digits   equivalent
                10-17 digits   27 times slower
                18    digits    2 times slower
      unsigned  1-18  digits   equivalent
```

A Performance Checklist

The following list of questions will help to isolate the problem area if a performance-related problem arises with COBOL. Most of the performance-related problems that have been reported in the past have fallen under one or more of the categories below. Each category does not always apply to all operating environments. Most of these have been discussed earlier in this paper, so additional detail will not be given here. This will just serve as a checklist of things to consider when investigating a performance problem.

1. What are all of the compiler options that were used? Is OPTIMIZE being used? Is TRUNC(OPT) being used? Make sure you understand the performance implications of the options you are using.
2. What run-time options were used? Make sure you understand the performance implications of the options you are using.
3. Is the program compiled with OS/VS COBOL or VS COBOL II and run with the LE library?
4. Does the application have a mixture of OS/VS COBOL or VS COBOL II with COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL programs?
5. Is the application called by any other non-COBOL or non-LE-conforming programs? For a non-COBOL driver repeatedly calling COBOL, is RTEREUS or IGZERRE or ILBOSTP0 being used? Is LRR or CEEPIPI being used? Are the CEEENTRY and CEETERM macros being used?
6. Does the application have any calls to any other programs? If so, what languages are involved? What is the approximate number of calls and the depth of the calls? Are the calls static or dynamic? How many unique programs are called?
7. What Space Management Tuning is being used? The RPTSTG(ON) run-time option can help you to determine the correct values to use.
8. What are the JOBLIB and STEPLIB datasets and where is the LE library in the search order? Is the LE library in LNKLST or LPA/ELPA?
9. For IMS, is the application and/or library preloaded? Is LRR being used?
10. Do you have an execution profiler? If so, have you used it to try to identify the "hot spots" of where time is spent in the application? This information can be very useful in identifying and solving performance problems.
11. What are the release levels of all COBOL products being used? Has all current maintenance been applied? If the most current release of IBM Enterprise COBOL is not being used, you should try it before reporting the problem to IBM since the problem may have already been addressed.
12. What are the release levels of the operating and subsystems being used (IMS, CICS, OS/390, z/OS)? Has all current maintenance been applied?
13. If using SORT, what release of DFSORT is being used? Has all current maintenance been applied? Is the FASTSRT compiler option being used?
14. In case you need to seek assistance from IBM in solving the performance problem, what other information can you tell us to help us understand the overall program structure (e.g., heavy use of a particular COBOL verb, the application program alters the save area in a non-standard way, subscripts that are not binary, data types used (USAGE DISPLAY, INDEX, COMP-n), etc.)?

Summary

This paper has identified some of the factors for tuning the performance of a COBOL application through the use of compiler options, run-time options, and efficient program coding techniques. Additionally, it has identified some factors for tuning the overall LE run-time environment. A variety of different tuning tips was provided for each of the above. The primary focus was on tuning the application using the compiler and run-time options with a secondary focus, for more in depth fine tuning, on examining the program design, algorithms, and data structures.

For the type of tuning suggested here, the costs and the skill level are relatively low, because in many cases, the program itself is not changed (except for, perhaps, data types). Hence, introducing errors is a low risk. The performance gains from this type of tuning may be sufficient to delay or eliminate the need for algorithmic changes, program structure changes, or further data type considerations.

In summary, there are many opportunities for the COBOL programmer to tune the COBOL application program and run-time environment for better CPU time performance and better use of system resources. The COBOL programmer has many compiler options, run-time options, data types, and language features from which to select, and the proper choice may lead to significantly better performance. Conversely, making the wrong choice can lead to significantly degraded performance. The goal of this paper is to make you aware of the various options that are available so that you -- both the system programmer installing the product as well as the COBOL programmer responsible for the application -- can choose the right ones for your application program that will lead to the best performance for your environment.

Appendix A. Intrinsic Function Implementation Considerations

The COBOL intrinsic functions are implemented either by using LE callable services, library routines, in-line code, or a combination of these. The following table shows how each of the intrinsic functions are implemented:

Table 1 (Page 1 of 2). Intrinsic Function Implementation

Function Name	LE Service	Library Routine	In-line Code
ACOS	X		
ANNUITY			X
ASIN	X		
ATAN	X		
CHAR			X
COS	X		
CURRENT-DATE	X		
DATE-OF-INTEGER	X		
DAY-OF-INTEGER	X		X
FACTORIAL			X
INTEGER			X
INTEGER-OF-DATE	X		
INTEGER-OF-DAY	X		X
INTEGER-PART			X
LENGTH			X
LOG	X		
LOG10	X		
LOWER-CASE		X	
MAX			X
MEAN			X
MEDIAN		X	
MIDRANGE			X
MIN			X
MOD			X
NUMVAL		X	
NUMVAL-C		X	
ORD			X
ORD-MAX			X
ORD-MIN			X
PRESENT-VALUE		X	
RANDOM	X		X
RANGE			X

Table 1 (Page 2 of 2). Intrinsic Function Implementation			
Function Name	**LE Service**	**Library Routine**	**In-line Code**
REM (fixed-point)			X
REM (floating-point)	X		
REVERSE		X	
SIN	X		
SQRT	X		
STANDARD-DEVIATION		X	X
SUM			X
TAN	X		
UPPER-CASE		X	
VARIANCE		X	X
WHEN-COMPILED			X[1]

[1] WHEN-COMPILED is a literal that is used whenever it is needed.

Appendix B. History of Prior Performance Improvements

COBOL has made performance improvements in some specific areas of the compiled code and library routines. Here is a brief history of the improvements made for prior COBOL releases, version, or products:

IBM COBOL for MVS & VM Version 1 Release 2 running with LE for MVS & VM Release 5 provides improved performance over COBOL/370 Version 1 Release 1 running with LE/370 Release 3:

- Dynamic and Static CALLs
- Non-COBOL to COBOL CALLs
- Reduction in below the line storage (LE improvement)

COBOL/370 Version 1 Release 1 with all current maintenance (Release 1.1) running with LE/370 Release 3 provides improved performance over COBOL/370 Version 1 Release 1 running with LE/370 Release 2:

- Eliminating storage and initialization code for unreferenced data items with the OPTIMIZE(FULL) option
- RMODE option added to support NORENT programs above the 16MB line
- Static initialization of WORKING-STORAGE variables at compile time for NORENT programs
- Optimized parameter list generated code for the USING phrase of the CALL statement
- Variable-length MOVEs for LINKAGE SECTION data items
- Optimized code generated for the main entry point
- Dynamic CALL literal
- Library Routine Retention (LRR) (available with LE/370 Version 1 Release 3 and later)

COBOL/370 Version 1 Release 1 running with LE/370 Version 1 Release 1 provides improved performance over VS COBOL II Release 3.2 and 4.0 for:

- Inter-language call with Assembler and C
- Dynamic CALL identifier
- UNSTRING (some additional cases are done in-line)

VS COBOL II Release 3.2 provides improved performance over VS COBOL II Release 3.1 for:

- EVALUATE (when using EVALUATE TRUE or EVALUATE FALSE)
- Passing parameters of large data structures to subprograms

VS COBOL II Release 3.1 provides improved performance over VS COBOL II Release 3.0 for:

- INITIALIZE (for tables with many subordinate items)
- INSPECT, STRING, and UNSTRING (some additional cases are done in-line)

VS COBOL II Release 3.0 provides improved performance over VS COBOL II Release 2 for:

- SSRANGE processing (code is now done in-line instead of through a library routine call, significantly reducing the overhead of subscript range checking)
- Decimal divide
- INSPECT, STRING, and UNSTRING (some of the simple cases of these statements are now done in-line instead of through a library routine call)
- CALL (by reducing some of the overhead)
- INDEX (when comparing INDEXes with constants and when using the SEARCH statement)

Appendix C. Coding Examples

Using CEEENTRY

```
* ================================================================
*   Bring up the LE environment
* ================================================================
CEE2COB  CEEENTRY PPA=MAINPPA,AUTO=WORKSIZE
         USING WORKAREA,13
* ================================================================
*   Set up the parameter list for the COBOL program
* ================================================================
         LA    5,OP1              Get address of 1st parameter
         ST    5,PARM1              and store it in parm list
         LA    5,OP2              Get address of 2nd parameter
         ST    5,PARM2              and store it in parm list
         LA    1,PARMLIST         Load addr of parm list in Reg 1
* ================================================================
*   Call the COBOL program
* ================================================================
         L     15,COBPGM          Get the address of the COBOL program
         BASR  14,15                and branch to it
* ================================================================
*   Terminate the LE environment
* ================================================================
         CEETERM  RC=0            Terminate with return code zero
*
* ================================================================
*   Data Constants
* ================================================================
COBPGM   DC    V(COBSUB)          Address of COBOL program
OP1      DC    X'00100C'          1st parameter for COBOL program
OP2      DC    X'00200C'          2nd parameter for COBOL program
*
MAINPPA  CEEPPA  ,                Constants describing the code block
* ================================================================
*   Workarea
* ================================================================
WORKAREA DSECT
         ORG    *+CEEDSASZ        Leave space for the DSA fixed part
*
PARMLIST DS    0F                 Parameter list for COBOL program
PARM1    DS    A                  Address of 1st parameter
PARM2    DS    A                  Address of 2nd parameter
*
         DS     0D
WORKSIZE EQU    *-WORKAREA
         CEEDSA  ,                Mapping of the Dynamic Save Area
         CEECAA  ,                Mapping of the Common Anchor Area
*
         END   CEE2COB
```

Using CEELRR

```
LRR2COB  CSECT
LRR2COB  AMODE 31
LRR2COB  RMODE ANY
*
         EXTRN COBSUB
*
* ================================================================
*  Save callers regs and chain save areas
* ================================================================
*
         STM   14,12,12(13)       Store incoming registers
         LR    12,15              Base LRR2COB on Register 12
         USING LRR2COB,12
*
         LA    15,SAVEAREA        Get this program's save area
         ST    13,4(,15)          Save caller's save area pointer
         ST    15,8(,13)          Save this program's save area pointer
         LR    13,15              Load standard save area Register 13
*
* ================================================================
*  Initialize Library Routine Retention (LRR)
* ================================================================
*
         CEELRR ACTION=INIT
*
* ================================================================
*   Set up the parameter list for the COBOL program
* ================================================================
*
         LA    5,OP1              Get address of 1st parameter
         ST    5,PARM1              and store it in parm list
         LA    5,OP2              Get address of 2nd parameter
         ST    5,PARM2              and store it in parm list
         LA    1,PARMLIST         Load addr of parm list in Reg 1
*
* ================================================================
*   Call the COBOL program
* ================================================================
*
         L     15,COBPGM          Get the address of the COBOL program
         BASR  14,15                and branch to it
*
* ================================================================
*   Terminate Library Routine Retention (LRR)
* ================================================================
*
         CEELRR ACTION=TERM
*
* ================================================================
*   Return to our caller
* ================================================================
*
         L     13,4(,13)          Point to incoming registers
         LM    14,12,12(13)       Restore caller's registers
         SR    15,15              Return code 0
         BR    14                 Return to caller
*
```

```
* ================================================================
*   Data Constants and Parameter Lists
* ================================================================
COBPGM   DC    V(COBSUB)          Address of COBOL program
OP1      DC    X'00100C'          1st parameter for COBOL program
OP2      DC    X'00200C'          2nd parameter for COBOL program
*
PARMLIST DS    0F                 Parameter list for COBOL program
PARM1    DS    A                  Address of 1st parameter
PARM2    DS    A                  Address of 2nd parameter
*
SAVEAREA DS    18F                Standard Save Area
*
         END   LRR2COB
```

Using IGZERRE

```
*********************************************************************
*                                                                   *
*  The IGZERRE module can be invoked to set up the COBOL Run-time   *
*  Environment running under LE before the first COBOL program is   *
*  called.  Invoking IGZERRE will explicitly drive the COBOL        *
*  initialization and termination functions of LE.                  *
*                                                                   *
*  LOAD/DELETE of IGZERRE must be done by the user.  This load module *
*  must remain loaded until after the LE run-time environment has   *
*  been terminated (either by IGZERRE termination or a STOP RUN).   *
*                                                                   *
*  When a reusable run-time environment has been created via IGZERRE  *
*  initialization, subsequent use of STOP RUN will result in control  *
*  being returned to the caller of the routine that invoked IGZERRE   *
*  initialization.                                                  *
*                                                                   *
*********************************************************************
RRE2COB  CSECT
RRE2COB  AMODE 31                This routine is 31 bit addressable
RRE2COB  RMODE ANY               And can reside above or below the
*                                  line
* ================================================================
*  Save callers regs and chain save areas
* ================================================================
         STM   14,12,12(13)      Store incoming registers
         LR    12,15             Base RRE2COB on Register 12
         USING RRE2COB,12
*
         LA    15,SAVEAREA       Get this program's save area
         ST    13,4(,15)         Save caller's save area pointer
         ST    15,8(,13)         Save this program's save area pointer
         LR    13,15             Load standard save area Register 13
*
         LOAD  EP=IGZERRE        Issue LOAD for Reusable Run-time
*                                  Environment INIT/TERM Routine
         ST    0,IGZERREA        Save address for termination
*
*********************************************************************
*  IGZERRE is AMODE(31), RMODE(ANY). The routine that invokes IGZERRE *
*  must do so via BASSM 14,15, if running in 24-bit mode. IGZERRE    *
*  will return via BSM 0,14.                                        *
*********************************************************************
*
         LA    1,1               Function code for init is "1"
         LTR   15,0              Get address of IGZERRE
         BM    ALTBRCH1          High bit on, so above the line
         BASR  14,15             Go initialize the COBOL environment
         B     TOCHECK           Check return code
ALTBRCH1 BASSM 14,15             Go initialize the COBOL environment
*
*********************************************************************
*  At this point, the user may wish to check the return codes:      *
*    0 - Function completed correctly                               *
*    4 - LE already initialized (initialization only)               *
*    8 - Invalid function code (not 1 or 2)                         *
*   16 - LE not initialized (termination only)                      *
*********************************************************************
```

```
*
TOCHECK  LA    14,4              "4" or less is OK
         CR    14,15             Test the return Register 15
         BL    ULTIMATE          Leave if return higher than "4"
* ================================================================
*  Set up the parameter list for the COBOL program
* ================================================================
         LA    5,OP1             Get address of 1st parameter
         ST    5,PARM1             and store it in parm list
         LA    5,OP2             Get address of 2nd parameter
         ST    5,PARM2             and store it in parm list
         LA    1,PARMLIST        Load addr of parm list in Reg 1
* ================================================================
*  Call the COBOL program
* ================================================================
         L     15,COBPGM         Get the address of the COBOL program
         BASR  14,15               and branch to it
* ================================================================
*  Terminate the reusable environment
* ================================================================
         L     15,IGZERREA       Address of IGZERRE was saved here
         LA    1,2               Function code for term is "2"
         LTR   15,15             IGZERRE might be above the line
         BM    PENULTMT          If so, go issue BASSM, else
         BASR  14,15             Go terminate the COBOL environment
         B     ULTIMATE          COBOL environment cleaned up
*                                  (unless return code non-zero)
PENULTMT BASSM 14,15             Go terminate the COBOL environment
*
* ================================================================
*  Ready to return to our caller
* ================================================================
*
ULTIMATE DELETE EP=IGZERRE       Delete IGZERRE
         L     13,4(,13)         Point to incoming registers
         RETURN (14,12),RC=(15)  Return to caller
*
* ================================================================
*   Data Constants and Parameter Lists
* ================================================================
COBPGM   DC    V(COBSUB)         Address of COBOL program
OP1      DC    X'00100C'         1st parameter for COBOL program
OP2      DC    X'00200C'         2nd parameter for COBOL program
*
PARMLIST DS    0F                Parameter list for COBOL program
PARM1    DS    A                 Address of 1st parameter
PARM2    DS    A                 Address of 2nd parameter
*
SAVEAREA DS    18F               Standard Save Area
IGZERREA DC    A(0)              Address of IGZERRE
*
         END   RRE2COB
```

Using CEEPIPI with Call_Sub

```
PIPI2COB CSECT
*********************************************************************
* Since CEEPIPI must run as AMODE ANY / RMODE 24, this assembler     *
* routine must either:                                               *
*                                                                    *
*   1 - have the same AMODE / RMODE as CEEPIPI (i.e., ANY / 24), or  *
*                                                                    *
*   2 - the CALL macro must be removed and coded manually, using the *
*       appropriate mode switching instructions.                     *
*                                                                    *
*********************************************************************
PIPI2COB AMODE ANY
PIPI2COB RMODE 24
*
        EXTRN COBSUB            COBOL program is external
*
        STM   14,12,12(13)      Save caller's registers
        LR    12,15             Get base address
        USING PIPI2COB,12       Identify base register
*
        LA    15,SAVEAREA       Get this program's save area
        ST    13,4(,15)         Save caller's save area pointer
        ST    15,8(,13)         Save this program's save area pointer
        LR    13,15             Load standard save area Register 13
*
        LOAD  EP=CEEPIPI        Load CEEPIPI rtn dynamically
        ST    0,CEEPIPIA        Save the addr of CEEPIPI rtn
* ================================================================
*  Set up the parameter list for the COBOL program
* ================================================================
        LA    5,OP1             Get address of 1st parameter
        ST    5,PARM1             and store it in parm list
        LA    5,OP2             Get address of 2nd parameter
        ST    5,PARM2             and store it in parm list
* ================================================================
* Initialize a new LE environment
* ================================================================
        L     15,CEEPIPIA       Get address of CEEPIPI routine
        CALL  (15),(INITSUB,@CEXPTBL,@SRVRTNS,RUNTMOPT,TOKEN)
*
*                               User may want to check return code
*
* ================================================================
* Invoke the COBOL subprogram which is statically linked
* ================================================================
        L     15,CEEPIPIA       Get address of CEEPIPI routine
        CALL  (15),(CALLSUB,PTBINDEX,TOKEN,PARMPTR,SUBRETC,SUBRSNC,   X
              SUBFBC)           Invoke CEEPIPI
* ================================================================
* Terminate the LE environment
* ================================================================
        L     15,CEEPIPIA       Get address of CEEPIPI routine
        CALL  (15),(TERM,TOKEN,ENV_RC)    Invoke CEEPIPI
        DELETE EP=CEEPIPI       Delete CEEPIPI
*
* ================================================================
* Standard exit code
```

```
* ================================================================
SYSRET   L     13,4(,13)          Point to caller's save area
         RETURN (14,12),RC=(15)   Restore regs and return to caller
*
* ================================================================
* Data Areas and Constants
* ================================================================
OP1      DC    X'00100C'
OP2      DC    X'00200C'
*
PARMLIST DS    0F                 Parameter list
PARM1    DS    A
PARM2    DS    A
*
SAVEAREA DS    18F
CEEPIPIA DS    A                  Save the address of CEEPIPI routine
*
* ================================================================
* Parameter list passed to a CEEPIPI(INIT-SUB)
* ================================================================
INITSUB  DC    F'3'               Function code for init subprogram
@CEXPTBL DC    A(PPTBL)           Address of PIPI Table
@SRVRTNS DC    A(0)               No service routines
RUNTMOPT DC    CL255' '           No run-time options
TOKEN    DS    F                  Unique value returned
*
* ================================================================
* Parameter list passed to a CEEPIPI(CALL-SUB)
* ================================================================
CALLSUB  DC    F'4'               Function code for calling subpgm
PTBINDEX DC    F'0'               The row number of PIPI Table entry
PARMPTR  DC    A(PARMLIST)        Pointer to parameter list
SUBRETC  DS    F                  Subroutine return code
SUBRSNC  DS    F                  Subroutine reason code
SUBFBC   DS    3F                 Subroutine feedback token
*
* ================================================================
* Parameter list passed to a CEEPIPI(TERM)
* ================================================================
TERM     DC    F'5'               Function code for term subprogram
ENV_RC   DS    F                  Environment return code
*
* ================================================================
* PIPI Table
* ================================================================
PPTBL    CEEXPIT                  PIPI Table with index
         CEEXPITY COBSUB,COBSUB   Statically linked REENTRANT routine
         CEEXPITS
*
         END   PIPI2COB
```

Using CEEPIPI with Call_Main

```
PIPM2COB CSECT
*********************************************************************
* Since CEEPIPI must run as AMODE ANY / RMODE 24, this assembler     *
* routine must either:                                               *
*                                                                    *
*   1 - have the same AMODE / RMODE as CEEPIPI (i.e., ANY / 24), or  *
*                                                                    *
*   2 - the CALL macro must be removed and coded manually, using the *
*       appropriate mode switching instructions.                     *
*                                                                    *
*********************************************************************
PIPM2COB AMODE ANY
PIPM2COB RMODE 24
*
        EXTRN COBSUB            COBOL program is external
*
        STM   14,12,12(13)      Save caller's registers
        LR    12,15             Get base address
        USING PIPM2COB,12       Identify base register
*
        LA    15,SAVEAREA       Get this program's save area
        ST    13,4(,15)         Save caller's save area pointer
        ST    15,8(,13)         Save this program's save area pointer
        LR    13,15             Load standard save area Register 13
*
        LOAD  EP=CEEPIPI        Load CEEPIPI rtn dynamically
        ST    0,CEEPIPIA        Save the addr of CEEPIPI rtn
* ================================================================
*  Set up the parameter list for the COBOL program
* ================================================================
        LA    5,OP1             Get address of 1st parameter
        ST    5,PARM1             and store it in parm list
        LA    5,OP2             Get address of 2nd parameter
        ST    5,PARM2             and store it in parm list
* ================================================================
* Initialize a new LE environment
* ================================================================
        L     15,CEEPIPIA       Get address of CEEPIPI routine
        CALL  (15),(INITMAIN,@CEXPTBL,@SRVRTNS,TOKEN)
*
*                               User may want to check return code
*
* ================================================================
* Invoke the COBOL subprogram which is statically linked
* ================================================================
        L     15,CEEPIPIA       Get address of CEEPIPI routine
        CALL  (15),(CALLMAIN,PTBINDEX,TOKEN,RUNTMOPT,PARMPTR,SUBRETC, X
              SUBRSNC,SUBFBC)   Invoke CEEPIPI
* ================================================================
* Terminate the LE environment
* ================================================================
        L     15,CEEPIPIA       Get address of CEEPIPI routine
        CALL  (15),(TERM,TOKEN,ENV_RC)    Invoke CEEPIPI
        DELETE EP=CEEPIPI       Delete CEEPIPI
*
* ================================================================
* Standard exit code
```

```
* ================================================================
SYSRET   L     13,4(,13)          Point to caller's save area
         RETURN (14,12),RC=(15)   Restore regs and return to caller
*
* ================================================================
* Data Areas and Constants
* ================================================================
OP1      DC    X'00100C'
OP2      DC    X'00200C'
*
PARMLIST DS    0F                 Parameter list
PARM1    DS    A
PARM2    DS    A
*
SAVEAREA DS    18F
CEEPIPIA DS    A                  Save the address of CEEPIPI routine
*
* ================================================================
* Parameter list passed to a CEEPIPI(INIT-MAIN)
* ================================================================
INITMAIN DC    F'1'               Function code for init main program
@CEXPTBL DC    A(PPTBL)           Address of PIPI Table
@SRVRTNS DC    A(0)               No service routines
RUNTMOPT DC    CL255' '           No run-time options
TOKEN    DS    F                  Unique value returned
*
* ================================================================
* Parameter list passed to a CEEPIPI(CALL-MAIN)
* ================================================================
CALLMAIN DC    F'2'               Function code for calling main pgm
PTBINDEX DC    F'0'               The row number of PIPI Table entry
PARMPTR  DC    A(PARMLIST)        Pointer to parameter list
SUBRETC  DS    F                  Subroutine return code
SUBRSNC  DS    F                  Subroutine reason code
SUBFBC   DS    3F                 Subroutine feedback token
*
* ================================================================
* Parameter list passed to a CEEPIPI(TERM)
* ================================================================
TERM     DC    F'5'               Function code for term subprogram
ENV_RC   DS    F                  Environment return code
*
* ================================================================
* PIPI Table
* ================================================================
PPTBL    CEEXPIT                  PIPI Table with index
         CEEXPITY COBSUB,COBSUB   Statically linked REENTRANT routine
         CEEXPITS
*
         END   PIPM2COB
```

COBOL Example - COBSUB

```
  CBL  RENT
000100 IDENTIFICATION DIVISION.
000200   PROGRAM-ID. COBSUB.
000300*
000400 ENVIRONMENT DIVISION.
000500*
000600 DATA DIVISION.
000700   WORKING-STORAGE SECTION.
000800*
000900 LINKAGE SECTION.
001000   01 X PIC S9(5) COMP-3.
001100   01 Y PIC S9(5) COMP-3.
001200*
001300 PROCEDURE DIVISION USING X Y.
001400     COMPUTE X = Y + 1.
001500*
001600     GOBACK.
```