VS COBOL II Release 3.2 and Release 4.0 Performance Tuning

May 18, 1994

R. J. Arellanes

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

Abstract

This paper identifies some of the factors that can affect the performance of COBOL programs when using VS COBOL II. The tuning methods discussed here are intended to assist users in selecting from the various options which are available for compiling and executing COBOL programs with VS COBOL II Release 3.2 and Release 4.0. These methods provide the COBOL programmer with the opportunity to tune the run-time environment for better CPU time performance and better use of system resources.

Distribution Notice

| **Third Edition, May 1994**

| This edition applies to VS COBOL II Release 3.2 and Release 4.0, 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 the third edition are marked with change bars as indicated to the left of this paragraph.

Second Edition, August 1992

All changes made in the second edition are marked with change bars as indicated to the left of this paragraph.

First Edition, December 1990

Contents

Introduction

First, as background information, we will briefly discuss the role of the Language Performance Evaluation Department in order to acquaint the reader with its work. Then, we will look at some compile and run-time options that affect the run-time performance of a COBOL program. Next, we will look at some efficient COBOL coding techniques that can be used to further reduce 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 VS COBOL II.

Throughout this paper, all references to old COBOL will refer to both OS/VS COBOL and DOS/VS COBOL, unless otherwise indicated. Also, all references to MVS refer to MVS/SP, MVS/XA, and MVS/ESA; all references to CMS refer to CMS under VM/SP, VM/XA, and VM/ESA; and all references to VSE refer to VSE/ESA, unless otherwise indicated. Additionally, several items will have page number references after them. They are in **bold face**, enclosed in parentheses beginning with **APG:**, followed by **MVS** and the page numbers and/or **VSE** and the page numbers. These refer to pages in the VS COBOL II Release 3.2 Application Programming Guide for MVS and CMS, SC26-4045-4 and the VS COBOL II Release 3.2 Application Programming Guide for VSE, SC26-4697-0, respectively. The Application Programming Guides contains 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 get the most benefit from the information contained herein.

The Language Performance Evaluation Department

The Language Performance Evaluation Department at IBM is responsible for monitoring the performance of the various high level language products that IBM offers. In particular, our department monitors and analyzes the performance of COBOL, FORTRAN, PASCAL, and PL/I. Although we are not part of the development group, we do work very closely with them both in defining the performance characteristics of the products and in resolving performance problems that we encounter.

Most of our measurements are done on MVS batch and CMS, but we also measure the language products on CICS and IMS. In measuring the performance of the language products on MVS batch and CMS, 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 (Virtual and Total CPU Time on CMS and TCB Time on MVS), SIO or EXCP counts, virtual storage usage, and paging activity (CMS only). When measuring on CICS and IMS, we usually enlist the help of other departments that are more familiar with transaction environments in collecting and analyzing data from CICSPARS, IMSPARS, RMF, and SMF.

For a more detailed analysis of the performance of a specific application or if we encounter a performance problem, we use another set of tools to help us isolate the problem. With these tools, we can either modify the application program to put in calls to a timing routine or we can run the application program under an execution time sampler program which produces a histogram of where the CPU time is being spent. From these, we are usually able to isolate the problem to a specific area of the program (e.g., in the generated code or in a library routine). Additionally on CMS, we have other tools that collect data on SVC usage and instruction counts. Using all of these tools, we are able to identify and isolate a performance problem. After doing this, we present our findings to the appropriate development group for resolution of the problem.

In measuring a language product (for example, VS COBOL II), we have three different types of benchmarks: compile only, compile and execute, and kernels. The compile only set is designed to be compiled but not executed. They 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 are either subsets of real application programs or are entire application programs. These are somewhat representative of "real" customer application programs. Some are actual customer programs and the rest have been written by IBM. The kernel set is 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. These kernel programs will usually identify a performance problem with a specific type of statement.

Finally, we also work with the account SEs to help them resolve any performance problems that they have been unable to resolve themselves after working with the customer. We have been able to resolve most of these problems by using the techniques that will be identified throughout this paper. This paper is a summary of the performance experiences that we have had with VS COBOL II. By writing this paper, we hope to reach and benefit a wider audience than would otherwise be possible.

Tuning the Run-Time Environment

This section will focus on some of the options that are available for tuning an application as well as the overall VS COBOL II run-time environment for better performance without modifying the source code. This 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, setting up the COBPACKs, and placing the library routines in shared storage). But first, we will look at each of these external options in a little more detail. (**APG: MVS** pp 173-183; **VSE** pp 165-176)

Compiler options that affect run-time performance

This, by far, has been the cause of a vast majority of the performance problems. Many customers are not aware of the performance implications that many of the compiler options have at run time, in particular, the NUMPROC, TRUNC, SSRANGE, FASTSRT, OPTIMIZE, AWO, TEST, CMPR2, DYNAM, RESIDENT, and RENT options. Let's look at each of these options to see how they might affect the performance of an application program. (**APG: MVS** pp 173-178, 291; **VSE** pp 165-172, 269)

Note: Under VSE, since the VS COBOL II compiler is larger than the DOS/VS COBOL compiler, it is recommended that you place the compiler in the SVA in order to minimize the size of the partition required to compile your programs. If you do not do this, you may have to increase the partition size. Addition information on placing the compiler in the SVA can be found in IBM publication SC26-4696, "VS COBOL II Release 3.2 Installation and Customization for VSE".

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 old COBOL. Using NUMPROC(NOPFD) or NUMPROC(MIG) may also inhibit some other types of optimizations. 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, the use of NUMPROC(PFD) is recommended where possible. (**APG: MVS** pp 68, 175, 306-307; **VSE** pp 65-66, 169, 280-281)

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 adds a significant amount of degradation since typically some data conversions must be done which may require the use of some library subroutines. This 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. 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 (fullword or halfword). 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 where possible. (**APG: MVS** pp 177-178, 313-315; **VSE** pp 171-172, 282-284)

SSRANGE or NOSSRANGE

Using SSRANGE generates additional code to verify that all subscripts, indexes, and reference modification expressions are within the proper bounds. This code occurs at every reference to a subscripted or variable length data item, as well as at every reference modification expression, and can result in some degradation at run time. This degradation is substantially less than that of VS COBOL II Release 2 since the SSRANGE code is done in a library routine whereas the Release 3 SSRANGE code is done in-line. In general, if you only need to verify the subscripts 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, the use of NOSSRANGE is recommended. (**APG: MVS** pp 177, 312; **VSE** pp 171, 281-282)

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 was in effect. A list of requirements for using the FASTSRT option can be found in the Application Programming Guide. (**APG: MVS** pp 125-127, 174-175, 298; **VSE** pp 122-124, 168-169, 279)

OPTIMIZE or NOOPTIMIZE

As with old COBOL, VS COBOL II does some optimizations automatically. However, the ones that VS COBOL II does automatically are not necessarily the same as the ones that old COBOL does automatically. To ensure that you get good optimized code, you should use the OPTIMIZE compiler option. With the OPTIMIZE option in effect, you get all of the optimizations that old COBOL provided, plus more. These 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
- subscript optimization
- eliminating duplicate computations
- eliminating constant computations
- aggregating moves of contiguous, equal-sized items into a single move
- deleting unreachable code

Many of these optimizations were not available with old COBOL. NOOPTIMIZE is generally used during program development time when frequent compiles are necessary. NOOPTIMIZE also allows for easier debugging of the program since code is not moved and is required when using the TEST compiler option. OPTIMIZE requires more CPU time for compiles than NOOPTIMIZE, but generally produces more effi-

cient code at run time. For production runs, the use of OPTIMIZE is recommended. (**APG: MVS** pp 171-173, 175-176, 308; **VSE** pp 165-167, 169-170, 279)

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. (**APG: MVS** pp 41, 174, 293; **VSE** pp 39-40, 168, 277)

TEST or NOTEST

The TEST compiler option generates additional code so that the program can be run under the debugger. This can cause a significant performance degradation when used in a production environment since this additional code occurs at each COBOL statement. The TEST option should only be used when debugging an application. Additionally, with TEST, the OPTIMIZE option is disabled. For production runs, the use of NOTEST is recommended. (**APG: MVS** pp 177, 313; **VSE** pp 171, 295)

CMPR2 or NOCMPR2

The CMPR2 compiler option generates code that is compatible with code generated by VS COBOL II Release 2 (ANSI 1974 Standard) and does not allow the use of many of the new features available with VS COBOL II Release 3 (ANSI 1985 Standard). NOCMPR2 allows the full use of all supported 1985 Standard COBOL language features. Since additional features are available when using NOCMPR2 than when using CMPR2, there may be times when CMPR2 is faster than NOCMPR2.

Note: CMPR2 is provided as a migration aid to allow you to gradually convert your applications to the 1985 Standard support provided by NOCMPR2. All future enhancements to VS COBOL II will be provided only under NOCMPR2. Therefore, it is recommended that all applications be converted to NOCMPR2 as soon as possible. (**APG: MVS** pp 294, 433; **VSE** pp 273, 367)

DYNAM or NODYNAM

The DYNAM compiler option specifies that all subprograms invoked through the CALL 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-linkedited if the subprogram is changed. DYNAM also allows you to control the use of virtual storage in that when a subprogram is no longer needed, you can free the virtual storage used by that subprogram with the CANCEL statement. However, when using the DYNAM option, you pay a slight 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 slightly longer with DYNAM than with NODYNAM. (**APG: MVS** pp 174, 296; **VSE** pp 168, 277)

RESIDENT or NORESIDENT

The RESIDENT compiler option specifies that all COBOL library subroutines (except IGZEBST) are to be loaded dynamically at run time instead of being link-edited with the COBOL program. This allows for easier maintenance of the VS COBOL library since application programs would not have to be link-edited again after applying service updates to the library. Additional features supported with the RESIDENT compiler option are: running on CICS, running above the 16 MB line on an XA system, dynamic calls, debugging with the TEST compiler option, and sharing the library. The RESIDENT compiler option provides for full

function support of VS COBOL II, whereas the NORESIDENT compiler option does not. When using the RESIDENT option, you pay a slight performance penalty since with the first invocation of a library routine, there will be some additional overhead to locate and load the library routine; subsequent invocations of the library routine will not have this additional overhead. Some of this overhead can be reduced by tailoring the COBPACKs which we will address later. Although the individual application path length may be slightly longer when using the RESIDENT compiler option, the overall system performance will usually be better when the environment is properly tuned. (**APG: MVS** pp 176-177, 310; **VSE** pp 170-171, 278)

RENT or NORENT

Using the RENT compiler option causes the compiler to generate some additional code to ensure that the program is reentrant. This allows for the program to be placed in shared storage like the LPA or ELPA on MVS or the SVA on VSE. Also, the RENT option will allow the program to run above the 16 MB line on MVS/XA. Additionally, using RENT option causes the RESIDENT compiler option to be in effect. Because of producing reentrant code, there may be a slight increase in path length when using the RENT option. (**APG: MVS** pp 176, 308-309; **VSE** pp 170, 278-279)

Run-time options that affect run-time performance

Selecting the proper run-time options is another area that affects the performance of a COBOL application. Because of this, it is important for the system programmer responsible for installing and setting up the VS COBOL II environment to work with the application programmers so that the proper run-time options are set up correctly for your installation. We will next look at some of the options that can help to improve the performance of the individual application as well as the overall VS COBOL II run-time environment. (**APG: MVS** pp 178-180, 348-353; **VSE** pp 172-174, 323-328)

LIBKEEP

This option keeps the VS COBOL II library routines initialized and in storage for use by subsequent application programs in the address space or region. Using this option will reduce part of the initialization and termination overhead in a given address space or region because once the library routines have been loaded, they remain loaded until the address space or region terminates. Using STOP RUN does not cause this portion of COBOL initialization to be terminated. Using LIBKEEP improves the performance of applications where non-COBOL drivers repeatedly call COBOL main programs (for example, an IMS environment or an assembler driver). Using LIBKEEP does not affect the semantics of the COBOL main program (i.e., each COBOL main program will be entered in its initial state). However, LIBKEEP will force NOENDJOB processing when the run unit contains at least one old COBOL program. This means that all old COBOL programs, as well as all COBOL library routines, will remain in storage until the address space or region terminates. LIBKEEP can only be used with the RESIDENT compiler option in non-CICS environments. (**APG: MVS** pp 157, 178, 350; **VSE** pp 152, 172, 325)

RTEREUS

This option causes the VS COBOL II run-time environment to be initialized for reusability when the first COBOL program is invoked. The VS COBOL II 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 VS COBOL II 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 (using STOP RUN will result in control being returned to the caller of the routine that invoked the first COBOL program, cleaning up the COBOL run-time environment. However, if the LIBKEEP option is also in effect, the library routines will remain initialized and in storage.). 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 IMS environment or an assembler driver that repeatedly calls COBOL applications). The RTEREUS option will help this case. However, using the RTEREUS option does affect the semantics of the COBOL application: each COBOL program will now be considered as 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). In particular, 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. (**APG: MVS** pp 160, 179, 350; **VSE** pp 155, 173, 325)

Calling IGZERRE

Another way to set up a reusable run-time environment for VS COBOL II is by calling IGZERRE. This module can be invoked to explicitly initialize and terminate the VS COBOL II run-time environment. It allows a non-COBOL program to initialize the COBOL run-time environment, thereby effectively establishing itself as the main COBOL program. As a result of this, 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 ILBDSET0 (for DOS/VS COBOL) that accomplishes the same results as ILBOSTP0 and ILBDSET0, except that IGZERRE has been designed for the VS COBOL II run-time environment only. Using IGZERRE has the added benefits of supporting applications running above the 16 MB line on an XA system, allowing the application to terminate the COBOL run-time environment, and improving the performance of the application compared to using ILBOSTP0 or ILBDSET0 (using ILBOSTP0 or ILBDSET0 in a VS COBOL II environment will set up both the old COBOL and the VS COBOL II environments whereas using IGZERRE will only setup the VS COBOL II environment). The semantic changes and performance benefits of using this method are the same as when using the RTEREUS COBOL run-time option. (**APG: MVS** pp 160-162, 180-181; **VSE** pp 155-156, 174-175)

AIXBLD

This option allows the COBOL program to build the alternate indexes 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 and then using the NOAIXBLD run-time option. (**APG: MVS** pp 167, 178, 229, 349, 422; **VSE** pp 161, 172, 218, 324)

WSCLEAR

This option clears all external data records acquired by a program to binary zeros when the storage for the external data is allocated. Additionally, the working storage acquired by a RENT program is cleared to binary zeros (unless a VALUE clause is used on the data item) when the program is first called, or, for dynamic calls, when the program is cancelled and then called again; storage is not cleared on subsequent calls to the program. This can result in a slight amount of overhead at run time depending on the number of external data records in the program and the size of the working storage section. (**APG: MVS** pp 179-180, 350; **VSE** pp 173-174, 325)

SSRANGE

This 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 NOSSRANGE 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 only has an effect on a program that has been compiled with the SSRANGE compiler option. (**APG: MVS** pp 179, 349; **VSE** pp 173, 324)

STAE

This option allows COBOL to intercept an abnormal termination (abend), provide the abend information, and then terminate the COBOL run-time environment. NOSTAE prevents COBOL from intercepting the abend. In general, there will not be any significant impact on the performance of a COBOL application when using STAE. However, in the case of COBOL calling non-COBOL which then calls COBOL, the abend intercept routine is set upon each invocation of COBOL from the non-COBOL program and cancelled upon each return back to the non-COBOL program. This can result in some additional overhead (to set and cancel the intercept routine each time the COBOL program is called). Specifying NOSTAE will eliminate this overhead. (**APG: MVS** p 349; **VSE** p 324)

MIXRES

This option allows a mixture of RES and NORES programs to run within a single application (which can be a mixture of old COBOL and VS COBOL II) and only applies when the application contains at least one NORES program. MIXRES is provided as a migration aid to allow you to gradually convert your NORES applications to RES. Using MIXRES is slower than NOMIXRES for a NORES application, and you should consider re-compiling the application to be all RES if you encounter performance problems using MIXRES. (**APG: MVS** pp 150, 350, 357; **VSE** pp 145, 325, 331)

Space management tuning

Space management tuning can reduce the overhead involved in getting and freeing storage for the application program. If proper tuning is done, this can eliminate several unnecessary GETMAIN and FREEMAIN calls. To better understand the need for space management tuning, let's look at how the space manager works.

First of all, the space manager was designed to not keep any block of storage any longer than is necessary. This means that during the execution of a VS COBOL II 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 up 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. Let's suppose that this 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 the space manager to request 2K of storage. The space manager 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, it will call the space manager to indicate that it no longer needs this 2K of storage. The space manager, 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, this can result in a significant amount of CPU time degradation because of the amount of GETMAIN and FREEMAIN activity.

Fortunately, we have a way to compensate for this with VS COBOL II; it is called space management tuning. The SPOUT run-time option can help you in determining the values to use for any specific application program. You use the value returned by the SPOUT option as the size of the initial block of storage on the IGZTUNE macro for both the INIBLOW and the INIABOV parameters. This will prevent the above from happening in an all VS COBOL II environment. However, if the application contains a mixture of old COBOL programs and VS COBOL II programs that are being CALLed frequently, the SPOUT option may not indicate a need for additional storage. Increasing these initial values can also eliminate some space management activity in this mixed environment. (**APG: MVS** pp 180, 352-353; **VSE** pp 174, 327-328)

COBPACKs

If the application program has been compiled with the RESIDENT compiler option, tailoring the COBPACKs can also affect the performance by reducing the directory search and program fetch time to load the VS COBOL II library routines. The ideal COBPACKs are those that only have those library routines in them that are needed by the application program. This will reduce the amount of overhead required to load the COBPACKs as well as reduce the amount of virtual storage required by not having unused library routines loaded. When properly tuned, the number of library routines loaded can be reduced to a total of six. However, this may not be practical since the COBPACKs will generally be shared among several different applications and cannot be tuned for one specific application. In this case, the COBPACKs should have, as a minimum, all library routines that are common to all application programs. Others can be added as virtual storage is available. Additional information on using COBPACKs can be found in IBM publications SC26-4048, "VS COBOL II Release 3.2 Installation and Customization for MVS", SC26-4213, "VS COBOL II Release 3.2 Installation and Customization for CMS", and SC26-4696, "VS COBOL II Release 3.2 Installation and Customization for VSE".

Library in the LPA/ELPA (MVS) or SVA (VSE)

In addition to tailoring the COBPACKs, you can place them in the Link Pack Area (LPA) or Extended Link Pack Area (ELPA) on an MVS system or the Shared Virtual Area (SVA) on a VSE system. This will reduce the real storage requirements for the entire system for all VS COBOL II applications that are compiled with the RESIDENT compiler option since the COBPACKs can be shared by all applications instead of each application having its own copy of the library routines. If you want the COBPACKs to go above the 16 MB line, you must be careful to only include those library routines that have AMODE 31 and RMODE ANY in the COBPACKs (a list of the eligible library routines with their AMODE and RMODE attributes can be found in the appropriate VS COBOL II Installation and Customization manual). Placing the COBPACKs 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. When not using RTEREUS or LIBKEEP for a non-COBOL main program that is repeatedly calling COBOL, placing the library routines (IGZCPAC, IGZCPCO, IGZCTCO, IGZEINI, IGZEPCL, and IGZEPSU) in the shared area will reduce the overhead for loading them for each first-level COBOL program. Additional information on placing COBPACKs in the LPA or ELPA can be found in IBM publication SC26-4048, "VS COBOL II Release 3.2 Installation and Customization for MVS" and information on placing COBPACKs in the SVA can be found in IBM publication SC26-4696, "VS COBOL II Release 3.2 Installation and Customization for VSE".

Other factors that affect run-time performance

It is important to understand VS COBOL II's interaction with other products in order to obtain the best performance for the application. We will now look at some additional factors that should be considered for the application.

Mixing old COBOL and VS COBOL II

If the application program is an old COBOL program using the VS COBOL II library or if the application program has a mixture of old COBOL and VS COBOL II, there will be some degradation at run time since both the old COBOL environment and the VS COBOL II environment must be initialized and cleaned up. Converting the entire application to VS COBOL II will eliminate the need for setting up both environments. (**APG: MVS** p 181; **VSE** p 175)

Main program not COBOL

If the first program in the application is not COBOL, 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 using one of the following:

- you can use the run-time option LIBKEEP to keep the COBOL library routines initialized and in storage (this will eliminate the overhead of subsequent loading and deleting of the COBOL library routines)
- you can use the run-time option RTEREUS to initialize the COBOL environment for reusability, making all COBOL main programs become subprograms
- you can call IGZERRE from the first program of the application to make it appear as the COBOL main program and to initialize the COBOL environment for reusability
- you can 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)
- you can call ILBOSTP0 (when using OS/VS COBOL) or ILBDSET0 (when using DOS/VS COBOL) from the first program of the application to make it appear as the COBOL main program (this is provided for compatibility with old COBOL; calling IGZERRE is preferred over calling ILBOSTP0 or ILBDSET0)
- you can put IGZCPAC, IGZCPCO, IGZCTCO, IGZEINI, IGZEPCL, and IGZEPSU in the shared area (LPA/ELPA or SVA)

Make sure that you fully understand the implications of each one before you use them. (**APG: MVS** pp 147, 157-162, 181; **VSE** pp 143-144, 152-156, 175)

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.

For dynamic call identifier, use PIC X(8) for the data name. This is faster than using less than 8 bytes. (**APG: MVS** pp 133-134, 137-139; **VSE** pp 130-131, 134-136)

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 reducing 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 since it is does not have to be fetched from external storage. The RENT and RESIDENT compiler options are required for preloaded applications. (If you are using a release of IMS/VS prior to Version 3 Release 1, the data for the IMS application program must reside below the 16 MB line, and hence, the DATA(24) compiler option is also required if you are using IMS services.)

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 COBOL 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 either LIBKEEP or RTEREUS, it is recommended that the RENT and RESIDENT compiler options be used and the applications and library routines (IGZCPAC, IGZCPCO, IGZEINI,

IGZEPCL, IGZEPSU, and IGZCTCO) be preloaded when possible. (You can tailor the COBPACKs, IGZCPAC and IGZCPCO, to eliminate any library routines that you do not need. You can also remove all AMODE 24, RMODE 24 routines from the COBPACKs to allow the COBPACKs to reside above the 16 MB line. In this case, you should also preload any of the below the line routines that you need.) However, it is recommended that at least the LIBKEEP run-time option be used for better overall system performance. When using LIBKEEP, all COBOL library routines remain loaded in the IMS region after they are loaded by the first COBOL transaction.

Note: Using LIBKEEP causes all ENDJOB processing in OS/VS COBOL to be converted to NOENDJOB. Additionally, all GETMAINed areas acquired by the library routines are retained.

If your IMS RES applications include a large number of non-preloaded subprograms that are dynamically called from COBOL programs, you should consider writing a BLDL user exit routine to maintain directory information look-aside tables to improve run-time performance. By doing this, load time for the subprograms is reduced because the look-aside table is searched, instead of the DASD directory, each time the subprogram is dynamically called. Additional information on using BLDL user exits can be found in IBM publication SC26-4048, "VS COBOL II Release 3.2 Installation and Customization for MVS".

Additional information on using VS COBOL II in an IMS environment can be found in IBM publication G320-9538, "Programming Language Considerations in an IMS Environment - VS COBOL II". (**APG: MVS** pp 181-182, 200-202, 347)

CICS

The RESIDENT, RENT, and NODYNAM compiler options are required for an application running under CICS.

VS COBOL II supports static calls to VS COBOL II subprograms containing EXEC CICS commands. Under CICS 1.7 and later, VS COBOL II supports dynamic calls to VS COBOL II subprograms that do not have CICS commands or dependencies. Under CICS 2.1 and later, VS COBOL II supports dynamic calls to VS COBOL II subprograms that have CICS commands or dependencies. Static calls are done with the CALL literal statement and dynamic calls are done with the CALL identifier statement. Old COBOL does not have this support. Converting EXEC CICS LINKs to COBOL CALLs can improve transaction response time and reduce virtual storage usage. Additional information on some of these improvements can be found in IBM publication GG09-1001, "VS COBOL II Improvements Using CICS and MVS/XA".

VS COBOL II does not support calls to or from old COBOL programs in a CICS environment. In this case, EXEC CICS LINK must be used. (**APG: MVS** pp 182, 198-199, 347; **VSE** pp 175-176, 189-191, 322)

If you are using the COBOL CALL statement to call a program that has been translated with the CICS translator, you must pass DFHEIB 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 DFHEIB and DFHCOMMAREA on the CALL statement.

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. For additional information on the TRUNC option, please refer to the compiler options section of this paper. **Note:** This is the most up-to-date information for using TRUNC with CICS applications and may be different than recommended in the Application Programming Guide. (**APG: MVS** p 193; **VSE** p 186)

Under VSE, since VS COBOL II uses more virtual storage than DOS/VS COBOL, increasing the partition size may also help.

Efficient COBOL Coding Techniques

This section will focus on some of the options that are available for tuning a program for better performance by modifying the source code. This can produce the higher performing code since the coding style as well as the data types can have a significant impact on the performance of the application. This usually has a far greater impact than that of tuning the application via compiler and run-time options, but 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.

Data Files

Planning how the files will be created and used is an important factor in determining the most efficient file characteristics for the application. Some of the characteristics that affect the performance of file processing are: the file organization, the access method, the record format, and the blocksize. Some of these are discussed in more detail below. (**APG: MVS** p 19-26; **VSE** p 17-24)

QSAM or SAM files

When using QSAM (MVS) or SAM (VSE) 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 3.1 or later, you can specify the BLOCK CONTAINS 0 clause for any new files that you are creating and omit the BLKSIZE parameter in your JCL for this file and the system will determine the optimal blocksize for you. This should significantly reduce 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. (**APG: MVS** pp 26, 166; **VSE** pp 24, 160)

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 can reduce the number of calls to Data Management Services to handle the I/Os. (**APG: MVS** pp 41, 166; **VSE** pp 39-40, 160)

VSAM files

When using VSAM files, increase the number of data buffers for sequential access or index buffers 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.

Sequential access is the most efficient, dynamic access the next, and random access is the least efficient. Random access results in an increase in I/O activity because VSAM must access the index for each request. **Note:** This is the most up-to-date information for using VSAM files and may be different than recommended in the Application Programming Guide.

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. (**APG: MVS** pp 21, 167, 229, 242-243; **VSE** pp 19, 161, 218, 231-232)

Data Types

When using binary (COMP) data items, the use of the SYNCHRONIZED clause ensures 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 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):

- 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**)

Note: Using 9 digits is slower than using 10 digits. (**APG: MVS** pp 73-74; **VSE** pp 70-71)

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 these guidelines for data types be followed. (**APG: MVS** pp 70-71; **VSE** pp 67-68)

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. (**APG: MVS** pp 71, 167; **VSE** pp 68, 161)

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 the most efficient code since this uses an integral multiple of bytes in storage for the data item. (**APG: MVS** pp 165-167; **VSE** pp 159-161)

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 9 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. (**APG: MVS** pp 167-168; **VSE** pp 161-162)

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
8 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 4 or fewer digits for the data item may also offer some added reduction in CPU time since halfword arithmetic can be used. (**APG: MVS** pp 94-96, 111, 167-170; **VSE** pp 91-93, 108, 161-164)

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 that 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 a fixed-length data item prior to a period of high-frequency use can reduce some of this overhead. (**APG: MVS** pp 167, 170; **VSE** pp 161, 164)

Program Design

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. (**APG: MVS** p 163; **VSE** p 157)

After deciding on the algorithm, look at the data structures and data types. Ensure that they are both 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 using the DISPLAY data item. (**APG: MVS** p 163; **VSE** p 157)

Examine the coding style. Ensure that the program is well structured, utilizing the structured coding constructs that are available with VS COBOL II. Avoid using the GO TO statement (in particular, avoid using 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.

Factor expressions where possible, especially in loops. The optimizer does not do the factoring for you. 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). 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. (**APG: MVS** pp 52-58, 163-165; **VSE** pp 50-56, 157-159)

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 only need to be checked in 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.

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.

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. (**APG: MVS** pp 108-109, 111; **VSE** pp 105-106, 108)

Performance Improvements

VS COBOL II has made some performance improvements in some specific areas of the compiled code and library routines. Here is a brief summary of these improvements by release.

Release 3.2 provides the following improvements over Release 3.1:

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

Release 3.1 provided the following improvements over Release 3.0:

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

Release 3.0 provided the following improvements over Release 2:

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

A Performance Checklist

The following list of questions will help to isolate the problem area if a performance related problem arises with VS COBOL II. Most of the performance related problems that have been reported in the past have fallen under one or more of the categories below. Each of the categories below do not always apply to all operating environments. Most of these have been discussed earlier in this paper, so we will not go into any additional detail 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?
2. If the program is compiled with RES, how are the COBPACKs defined?
3. Is the program compiled with old COBOL using the VS COBOL II Library?
4. Does the application have a mixture of old COBOL and VS COBOL II programs?
5. If the problem is on MVS, what are the JOBLIB and STEPLIB datasets and where are the COBOL Libraries in the search order?
6. Is the application called by any other non-COBOL programs?
7. 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?
8. What run-time options were used? For a non-COBOL driver repeatedly calling COBOL, was LIBKEEP, RTEREUS, or a CALL to IGZERRE used?
9. What Space Management Tuning was used? The SPOUT run-time option can help you to determine the correct values to use.
10. For IMS, was the application and/or library preloaded? Was a BLDL user exit used?
11. 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 COMP-4, data types used (USAGE DISPLAY, INDEX, COMP-n), etc.)?
12. What release of VS COBOL II was used? Has all current maintenance been applied? If the most current release of VS COBOL II is not being used, you should try it before reporting the problem to IBM since the problem may have already been addressed.
13. What environment and release was used (IMS, CICS, CMS under VM/SP, CMS under VM/XA, CMS under VM/ESA, MVS, MVS/XA, MVS/ESA, VSE/ESA)? Has current maintenance been applied?
14. If using SORT, what release of SORT was used? Has current maintenance been applied? Was the FASTSRT compiler option used?

Summary

This paper has identified some of the factors for tuning the performance of a VS COBOL II application through the use of compile time options, run-time options, and efficient program coding techniques. Additionally, it has identified some factors for tuning the overall run-time environment of VS COBOL II. A variety of different tuning tips was provided for each of the above. The primary focus was on tuning the application using the compile time 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 are relatively low, as is the skill level required, since 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 VS COBOL II application program and run-time environment for better CPU time performance and better use of system resources. The COBOL programmer has many compile time 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. Coding Examples

Using IGZERRE

```
*************************************************************************
*                                                                       *
* The  IGZERRE  module  can  be invoked  to set up the COBOL Run-time *
* Environment before the first COBOL program is called.  Invoking     *
* IGZERRE  will explicitly  drive COBOL's  initialization and termin- *
* ation functions.                                                      *
*                                                                       *
* For the RES environment, LOAD/DELETE of IGZERRE must be done by the *
* user.  This load module must remain loaded until after the COBOL    *
* run-time environment has been terminated (either by IGZERRE termi-  *
* nation or a STOP RUN).                                                *
*                                                                       *
* For the NORES environment, there must be at least one VS COBOL II   *
* program in the application if the IGZERRE initialization or         *
* termination functions are to be explicitely used. Also, the module  *
* ILBOSRV must be link-edited with the application.                     *
*                                                                       *
* 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 Runtime
*                                  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 on MVS/ESA in 24-bit mode.  *
*  IGZERRE will always return via BSM 0,14, if running on MVS/ESA.    *
*************************************************************************
*
         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
         BALR  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 - COBOL already initialized (initialization only)             *
*   8 - Invalid function code (not 1 or 2)                          *
*  12 - ILBOSTP0 not part of load module (NORES only)               *
*  16 - COBOL not initialized (termination only)                    *
*  20 - COBTEST in use (RES 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
         BALR  14,15               and branch to it
*
*********************************************************************
* Call or re-call as many COBOL programs as needed.  Be aware that  *
* they will be treated as sub-programs because the environment will *
* not be refreshed between calls. Also, the programs must be self-  *
* initializing.                                                     *
*********************************************************************
*
* ================================================================
*  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
         BALR  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)
*
```

```
* ===================================================================
*   Data Constants and Parameter Lists
* ===================================================================
COBPGM   DC    V(COBPGM)          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 DC    18F'0'             Standard Save Area
IGZERREA DC    A(0)               Address of IGZERRE
*
         END   RRE2COB
```

COBOL Example - COBPGM

```
000100 IDENTIFICATION DIVISION.
000200   PROGRAM-ID. COBPGM.
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.
```