Enterprise COBOL Version 5 Release 1

Performance Tuning Guide

Enterprise COBOL Version 5 Release 1

Performance Tuning Guide

Note
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 67.

First Edition, June 2014

This edition applies to IBM Enterprise COBOL Version 5 Release 1 running with the Language Environment component of z/OS Version 2 Release 1, and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Tables

Chapter 1. Introduction

This paper identifies key performance benefits and tuning considerations when using IBM® Enterprise COBOL for z/OS® Version 5 Release 1.

First, this paper gives an overview of the major performance features and options in this latest release of the compiler followed by performance improvements for several specific COBOL statements. Next, it provides tuning considerations for many compiler and runtime options that affect the performance of a COBOL application. Coding techniques to get the best performance are examined next with a special focus on any coding recommendations that have changed when using Version 5.

The final section examines some causes of increased program object size and studies the object size impact of the various new TEST suboptions as well as discussing the related issue of why PDSEs are now required for programs compiled using Version 5.

Note on version naming

In this paper, "IBM Enterprise COBOL for z/OS Version 5 Release 1" is shortened to "V5", when the recommendations or performance measurements are relevant to all Version 5 releases up to and including the March 2014 PTF Level.

Performance comparisons to earlier releases were always to "IBM Enterprise COBOL for z/OS Version 4 Release 2", shortened to "V4". In most cases, the recommendations also apply to earlier releases.

If a particular section only applies strictly to the March 2014 PTF Level or later, the terminology "V5.1.1" is specifically used.

Performance measurements

The performance measurements in this paper were made on the 2827 IBM zEnterprise® zEC12 system. 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.

Referenced IBM publications

Throughout this paper, all references to OS/VS COBOL refer to OS/VS COBOL Release 2.4, and all references to MVS™ (except in product names) refer to z/OS, unless otherwise indicated. Additionally, several items have page number references after them, 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:

Table 1. Abbreviation and IBM publication and number

Abbreviation	IBM Publication and Number
COB PG	Enterprise COBOL for z/OS Programming Guide Version 5 Release 1, SC14-7382-00

Table 1. Abbreviation and IBM publication and number (continued)

Abbreviation	IBM Publication and Number
COB LRM	Enterprise COBOL for z/OS Language Reference Version 5 Release 1, SC14-7381-00
COB MIG	Enterprise COBOL for z/OS Migration Guide Version 5 Release 1, GC14-7383-00
LE PG	z/OS V2R1 Language Environment® Programming Guide, SA38-0682-00
LE REF	z/OS V2R1 Language Environment Programming Reference, SA38-0683-00
LE CUST	z/OS V2R1 Language Environment Customization, SA38-0685-01
LE MIG	z/OS V2R1 Language Environment Runtime Application Migration Guide, GA32-0912-00

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.

When later versions of these manuals become available, they should be checked for updated recommendations. While the page number references may change for later versions, the information should still be in the same general section of the manuals.

Chapter 2. Why recompile with V5?

Recompiling your applications will leverage the advanced optimizations and z/Architecture® exploitation capabilities in Enterprise COBOL V5, and provide a solid framework to deliver release to release performance improvements for COBOL on System z®.

Improved performance is delivered through:

Architecture exploitation

COBOL V5 introduces a new ARCH option (short for architecture) in order to get the most out of your hardware investment, and to fully realize the performance benefits of the new compiler.

The default setting for ARCH is 6, and other supported values are 7, 8, 9 and 10.

For more information on the facilities available at each level, and the mapping of these ARCH levels to specific hardware models, see **COB PG** pp 317.

Each successive ARCH level allows the compiler to exploit more and more facilities in your hardware leading to the potential for increased performance. To illustrate the benefits from a COBOL application perspective, each ARCH level will be examined in greater detail below.

ARCH(6)

Hardware Feature: Long displacement instructions

Why This Matters For COBOL Performance: COBOL programs often work with a large amount of WORKING-STORAGE, LOCAL-STORAGE and LINKAGE SECTION data. The long displacement instructions reduce the need for initializing Base Locator cells and tying up registers for this purpose.

Instead, the compiler makes use of the much larger reach of the long displacement instructions to access 256 times as much data as the standard displacement instructions used exclusively in earlier releases of the compiler.

Hardware Feature: 64-bit G format instructions

Why This Matters For COBOL Performance: Whenever BINARY (and its synonym types of COMP and COMP-4) data exceeds 9 decimal digits, then the standard instruction set that operates on 32-bit registers can no longer contain the full range of values. In earlier releases of the compiler, this meant converting to another data type (such as packed decimal) or maintaining pairs of 32-bit registers. Both of these solutions add extra overhead and reduce performance.

Even if your BINARY data is declared with 9 or fewer decimal digits, intermediate arithmetic results can exceed 9 decimal digits and might require a conversion from

a 32-bit to a 64-bit representation. The conversion is needed because some 10 decimal-digit values and all values greater than 10 decimal digits cannot be fully encoded in the 32-bit two's complement representation that is used for BINARY data.

For example, a multiplication of two PIC 9(5) digit BINARY data items results in an intermediate value of 10 digits, as the source operand digit values of five must be added to arrive at the intermediate precision.

When using addition the intermediate precision is one greater than the highest of the operand precision values. This means that adding a PIC 9(9) value to a PIC 9(1) value results in an intermediate precision of 10 digits.

In V5, the 64-bit G format instruction set is used to hold values with up to 19 decimal digits and therefore a type conversion or using pairs of register is no longer needed and performance is dramatically increased. Above the 64-bit limit type conversions are still required, but the performance of these cases has also been improved in V5. See "BINARY (COMP or COMP-4)" on page 51 for a more in depth discussion of BINARY data and interaction with TRUNC suboptions.

ARCH(7)

Hardware Feature: 32-bit immediate form instructions for a range of arithmetic, logical and compare operations

Why This Matters For COBOL Performance: When your application contains binary data involved in arithmetic or compares, particularly if the data exceeds 5 digits, there is considerable opportunity for the compiler to take advantage of these new ARCH(7) 32-bit immediate form instructions.

Using a lower architecture setting or compiling with V4 would only allow the use of at most 16 bits worth of immediate data in a single instruction. Any larger values required storing the data in the literal pool, or using multiple other instructions in order to construct the immediate value in a register.

Both alternatives are less efficient in time and space.

With ARCH(7), immediate values up to 32 bits can be embedded directly in the instruction text with no need to reference the literal pool or generate other instructions that will increase path length. The result is generally smaller and faster code and less literal pool usage (saving the space and any delay in retrieving the data).

ARCH(8)

Hardware Feature: Decimal Floating Point (DFP)

Why This Matters For COBOL Performance: Decimal Floating Point is a natural fit for the packed decimal (COMP-3) and external decimal (DISPLAY) types that are ubiquitous in most COBOL applications. Using ARCH(8) and some OPTIMIZE setting above 0 enables the compiler to convert larger multiply and divide operations on any type of decimal operands to DFP, in order to avoid an expensive callout to a library routine.

This is possible as the hardware precision limit for DFP is much greater than is allowed in the packed decimal multiply and divide instructions.

The overhead of converting to DFP means it is not suitable for all decimal arithmetic that would not need a library call. However, the ARCH(10) option described later in this section enables much greater use of DFP to improve performance.

Hardware Feature: Larger Move Immediate Instructions

Why This Matters For COBOL Performance: MOVEs of literal data and VALUE clause statements are very common in many COBOL applications. Lower ARCH settings and all earlier compiler releases only contained support for moving a single byte of literal data in a single instruction, for example, by using the MVI - Move Immediate Instruction.

Any larger literal data required storing the constant value in the literal pool and using a memory move instruction in order to initialize the data item. This was less efficient in time and space than being able to embed larger immediate values directly in the instruction text.

With ARCH(8), several new move immediate instruction variants are available to move up to 16 bytes of sign extended data using one or two of these new instructions.

Also, these instructions are exploited regardless of the data type, so binary, internal/external decimal, alphanumeric and even floating point literals take advantage of these more efficient instructions.

ARCH(9)

Hardware Feature: Distinct Operands Instructions

Why This Matters For COBOL Performance: Updating a data item or index to a new value while retaining the original value occurs frequently in many contexts in a typical COBOL application. One instance is when processing a table as some base value for the table is updated in order to access the various elements within the table. Under lower ARCH settings or in all earlier compiler releases, almost all instructions available that took two operands in order to produce a result would also overwrite the input first operand with the result.

For example: a conceptual operation such as:

```
C = A + B
```

Implemented with a pre ARCH(9) instruction variant would conceptually have to perform the operation as:

```
A = A + B
C = A
```

This means if the original value of A is required in another context, it must first be saved:

```
T = A
T = T + B
C = T
```

With ARCH(9), the distinct-operands facility is exploited to take advantage of the new variants of many arithmetic, shift and logical instructions that will not destructively overwrite the first operand.

So the operation can be implemented in a more straightforward way:

C = A + B

That removes the need for extra instructions to save the original value as it is naturally preserved with the distinct operand instruction form. This feature reduces path length leading to better performance.

ARCH(10)

Hardware Feature: Improved Decimal Floating Point (DFP) Performance

Why This Matters For COBOL Performance: Using ARCH(8) and an OPTIMIZE setting greater than 0 already enables the compiler to make use of DFP to improve performance of packed and external decimal arithmetic in some particular instances. ARCH(10) goes further by adding very efficient instructions to convert between DISPLAY (in particular unsigned and trailing signed overpunch zoned decimal) types and DFP.

These ARCH(10) instructions lower the overhead for using DFP for arithmetic on zoned decimal data items and enable the compiler to make much greater use of DFP to improve performance.

Instead of converting zoned decimal data items to packed decimal format in order to perform arithmetic, the compiler will convert zoned decimal data directly to DFP format and then back again to zoned decimal format after the computations are complete. This generally results in better performance, as the DFP instructions operate on in-register (compared to in-memory) data that is more efficiently handled by the hardware in many cases.

Advanced optimization

In addition to deep architecture exploitation, Enterprise COBOL V5 improves performance of your application by employing a suite of advanced optimizations. In V4, the OPTIMIZE option has three settings; however, the kind and number of optimizations enabled in V5 is quite different.

Specifying OPTIMIZE(1) or OPTIMIZE(2) enables a range of general and COBOL specific optimizations.

For example, specifying OPTIMIZE(1) enables optimizations including:

- Strength reduction of complex and expensive operations, such as:
  - Reducing decimal multiply and divide by powers of ten, to simpler and better performing decimal shift operations
  - Reducing binary multiply and divide by powers of two to less expensive shift operations
  - Reducing exponentiation operations with a constant exponent to series of multiplications
  - Refactoring and redistributing arithmetic
- Eliminate common sub expressions, so computations are not duplicated

- Inline out-of-line PERFORM statements to save the branching overhead and expose other optimization opportunities for the surrounding code
- Coalesce sequential stores of constant values to a single larger store to reduce path length
- Coalesce individual loads/stores from/to sequential storage to a single larger move operation to reduce path length and reduce overall object size
- Simplify code to remove unneeded computations
- Remove unreachable code
- Propagate the VALUE OF clause literal over the entire program for data items that are read but never written
- Move nested programs inline to reduce CALL overhead and expose other optimization opportunities for the surrounding code
- Compute constant expressions, including the full range of arithmetic, data type conversions and branches, at compile-time
- Use a better performing branchless sequence for conditionally setting level-88 variables
- Convert some packed and zoned decimal computations to use better performing Decimal Floating Point types

When specifying OPTIMIZE(2), all the optimizations above are enabled plus additional optimizations, including:

- Instruction scheduling to expose instruction level parallelism to improve performance
- Propagate values and ranges of values over the entire program to expose constants and enable simpler sequences of instructions to be used
- Propagate sign values, including the "unsigned" sign encoding, over the entire program to eliminate redundant sign correction
- Allocate global registers for accessing indexed tables, and control PERFORM 'N' TIMES looping constructs to reduce path length
- Remove redundant sign correction operations globally. For example, if a sign correction for a data item in a loop is dominated by one outside of a loop to the same data item, then these sign-correcting instructions in the loop will be removed

Enhanced functionality

In addition to the performance improvements offered on your existing programs through architecture exploitation and advanced optimizations, COBOL V5 also offers enhanced functionality in several areas.

The section of “Changes in IBM Enterprise COBOL for z/OS, Version 5 Release 1” in **COB MIG** contains a complete list of new and changed functions. Some highlights are:

- XML GENERATE enhancements to provide more flexibility and control over the form of the XML document being generated
- XML parsing enhancing improvements through a new special register, XML-INFORMATION
- Support added for UNBOUNDED tables and groups to enable top-down mapping of data structures between XML and COBOL applications
- A new set of Unicode intrinsic functions

Chapter 3. Prioritizing your application for migration to V5

In order to prioritize your migration effort to V5, this section describes a number of specific COBOL statements and data type declarations that typically perform better with V5 versus earlier releases of the compiler. This is not meant to be an exhaustive list, but instead demonstrate some specific known cases where V5 performs reliably well.

See **COB MIG** pp 28, "Prioritizing Your Applications", for related information about migrating to maintain correctness of your application.

All performance measurements are compared to running the same program on the same machine level but compiled with V4. In all cases, the V4 programs were compiled with OPTIMIZE(FULL) and other options left at their default settings, except when ARITH(EXTEND) was required for the data and literal data items that contained more than 18 digits.

COMPUTE

A significant number of COMPUTE, ADD, SUBTRACT, MULTIPLY, DIVIDE statements show improved performance in V5.

Under "**Data types**" in the following examples, *italics* are used to indicate the variant tested for performance, but all data types listed would demonstrate similar performance results.

Larger decimal multiply/divide

Statement: COMPUTE (* | /), MULTIPLY, *DIVIDE*

Data types: COMP-3, *DISPLAY*, NATIONAL

Options: OPT(1 | 2), ARCH(8 | 9 |10)

Conditions: When intermediate results exceed the limits for using the hardware packed decimal instructions. This occurs at around 15 digits depending on the particular operation.

V4 behavior: Call to runtime routine

V5 behavior: Inline after converting to DFP

Source Example:

```
1 z14v2 pic s9(14)v9(2)
1 z13v2 pic s9(13)v9(2)

Compute z14v2 = z14v2 / z13v2.
```

Performance: V5 is 75% faster using ARCH(10), and 32% faster using ARCH(8 | 9)

Zoned decimal (DISPLAY) arithmetic

Statement: COMPUTE (+ | - | * | /), ADD, SUBTRACT, MULTIPLY, *DIVIDE*

Data types: DISPLAY

Options: OPT(1 | 2), ARCH(10)

Conditions: In all cases

V4 behavior: Inline using packed decimal instructions

V5 behavior: Inline after converting to DFP

Source Example:

```
1 z12v2 pic s9(12)v9(2)
1 z11v2 pic s9(11)v9(2)
Compute z12v2 = z12v2 / z11v2
```

Performance: V5 is 56% faster

Divide by powers of ten (10,100,1000,..)

Statement: COMPUTE (/), *DIVIDE*

Data types: *COMP-3*, DISPLAY, NATIONAL

Options: Default

Conditions: Divisor is a power of 10 (e.g. 10,100,1000,...)

V4 behavior: Use packed decimal divide (DP) instruction

V5 behavior: Model as decimal right shift

Source Example:

```
1 p8v2a pic s9(8)v9(2) comp-3
1 p8v2b pic s9(8)v9(2) comp-3

Compute p8v2b = p8v2a / 100
```

Performance: V5 is 79% faster

Multiply by powers of ten (10,100,1000,..)

Statement: COMPUTE (/), *MULTIPLY*

Data types: COMP-3, *DISPLAY*, NATIONAL

Options: Default

Conditions: Multiply is a power of 10 (e.g. 10,*100*,1000,...)

V4 behavior: Use packed decimal multiply (MP) instruction

V5 behavior: Model as decimal left shift

Source Example:

```
1 z5v2 pic s9(5)v9(2)
1 z7v2 pic s9(7)v9(2)

Compute z7v2 = z5v2 * 100
```

Performance: V5 is 80% faster

Decimal exponentiation

Statement: COMPUTE (**)

Data types: COMP-3, *DISPLAY*, NATIONAL

Options: Default

Conditions: In all cases

V4 behavior: Call to runtime routine

V5 behavior: Call to a more efficient runtime routine

Source Example:

```
1  R     PIC  9V9(8) value 0.05.
1  NF    PIC  9999 value 300.
1  EXP   PIC  9(23)v9(8).

COMPUTE EXP = (1.0 + R) ** NF.
```

Performance: V5 is 92% faster

Complex COMPUTE statements that have independent components

Statement: A mix of COMPUTE (+ | - | * | / | **) operations comprised of smaller independent arithmetic operations. For example, COMPUTE = (A + B) / (C – D) where the dividend (A + B) and divisor (C – D) are independent.

Data types: All

Options: OPT(2)

Conditions: In most cases

V4 behavior: Code is generated for the operations in the source order

V5 behavior: The optimizer will schedule independent instructions to expose instruction level parallelism, hide latencies and improve performance

Source Example:

```
1 z7v2a pic s9(7)v9(2).
1 z7v2b pic s9(7)v9(2).
1 z7v2c pic s9(7)v9(2).

ADD 1 TO z7v2a z7v2b z7v2c
```

Performance: V5 is 7% faster. Note that the exact same instructions are generated as V4 in this case, and it is only how these instructions are scheduled that causes the performance improvement.

Decimal scaling and divide

Statement: COMPUTE (/), *DIVIDE*

Data types: *COMP-3*, DISPLAY, NATIONAL

Options: Default

Conditions: When the divisor value and the decimal scaling cancel out. In the example below, the divide operation necessitates a decimal left shift by 2, and since the divide by 100 is modelled as the decimal right shift by 2, these operations cancel out.

V4 behavior: Use packed decimal shift (SRP) and divide (DP) instructions

V5 behavior: Divide and decimal scaling are cancelled out so instructions equivalent to a simple MOVE operation are generated

Source Example:

```
1 p9v0 pic s9(9) comp-3
1 p10v2 pic s9(10)v9(2) comp-3.

COMPUTE p10v2 = p9v0 / 100
```

Performance: V5 is 90% faster

TRUNC(STD) binary arithmetic

Statement: COMPUTE (+ | - | * | /), *ADD*, SUBTRACT, MULTIPLY, DIVIDE

Data types: BINARY, *COMP*, COMP-4

Options: TRUNC(STD)

Conditions: In all cases

V4 behavior: Use an expensive divide operation to correct digits back to PIC specification

V5 behavior: Only use divide when actually required (in cases of overflow).

Source Example:

```
1 b5v2a pic s9(5)v9(2) comp.
1 b5v2b pic s9(5)v9(2) comp.

COMPUTE b5v2a = b5v2a + b5v2b
```

Performance: V5 is 21% faster

Large binary arithmetic

Statement: COMPUTE (+ | - | * | /), *ADD*, SUBTRACT, MULTIPLY, DIVIDE

Data types: BINARY, *COMP*, COMP-4

Options: TRUNC(STD)

Conditions: Intermediate results exceed 9 digits

V4 behavior: Arithmetic performed piecewise and converted to packed decimal

V5 behavior: Arithmetic performed in 64-bit registers

Source Example:

```
1 b8v2a pic s9(8)v9(2) comp.
1 b8v2b pic s9(9)v9(2) comp.

Compute b8v2a = b8v2a + b8v2b.
```

Performance: V5 is 91% faster

Negation of decimal values

Statement: COMPUTE (-), SUBTRACT

Data types: *COMP-3*, DISPLAY, NATIONAL

Options: Default

Conditions: In all cases

V4 behavior: Treat as any other subtract from zero

V5 behavior: Recognize as a special case negate operation

Source Example:

```
1 p7v2a pic s9(7)v9(2) comp-3.
1 p7v2b pic s9(7)v9(2) comp-3.

Compute p7v2b = - p7v2a.
```

Performance: V5 is 61% faster

INSPECT

INSPECT REPLACING ALL on 1 byte operands

Statement: INSPECT REPLACING ALL

Data types: PIC X

Options: Default

Conditions: In all cases

V4 behavior: Uses general translate instruction as in all cases

V5 behavior: Handle short cases with a simple test and move

Source Example:

```
1 ITEM PIC X(1)

INSPECT ITEM REPLACING ALL ' ' BY '.'.
```

Performance: V5 is 56% faster

Consecutive INSPECTs on the same data item

Statement: More than one consecutive INSPECT REPLACING ALL on the same data item

Data types: PIC X

Options: OPT(1 | 2)

Conditions: When the compiler can prove, according to the rules of INSPECT, that the optimization will not alter the result.

V4 behavior: Generate separate operations for each INSPECT operation

V5 behavior: Coalesce the separate INSPECTs into a single INSPECT operation

Source Example:

```
1 ITEM PIC X(15)

INSPECT ITEM REPLACING ALL QUOTE BY SPACE.
INSPECT ITEM REPLACING ALL LOW-VALUE BY SPACE.
```

Performance: V5 is 29% faster

MOVE

VALUE clause and initializing groups

Statement: MOVE and VALUE IS

Data types: All types

Options: OPT(1 | 2)

Conditions: Initializing data items with literals

V4 behavior: Series of separate and sequential move instructions

V5 behavior: Coalesces literals and generates fewer move instructions

Source Example:

```
01 WS-GROUP.
  05 WS-COMP3     COMP-3 PIC S9(13)V9(2).
  05 WS-COMP      COMP   PIC S9(9)V9(2).
  05 WS-COMP5     COMP-5 PIC S9(5)V9(2).
```

```
  05 WS-COMP1     COMP-1.
  05 WS-ALPHANUM  PIC X(11).
  05 WS-DISPLAY   PIC 9(13) DISPLAY.
  05 WS-COMP2     COMP-2.

Move +0 to WS-COMP5
           WS-COMP3
           WS-COMP
           WS-DISPLAY
           WS-COMP1
           WS-COMP2
           WS-ALPHANUM.
```

Performance: V5 is 63% faster

SEARCH

SEARCH ALL

Statement: SEARCH ALL

Options: Default

Conditions: In all cases

V4 behavior: Call to runtime routine

V5 behavior: Call to a more efficient runtime routine

Source Example:

```
SEARCH ALL table
   AT END
     statements
   WHEN conditions
     statements
```

Performance: V5 is 49% faster

TABLES

Indexed TABLEs

Statement: Accessing data items in indexed TABLEs

Options: OPT(2)

Conditions: In all cases

V5 behavior: An efficient sequence is used to access indexed table elements by caching the offset to the start of the table in a globally available register versus having to reload this each time

Source Example:

```
1  TAB.
   5 TABENTS OCCURS 40 TIMES INDEXED BY TABIDX.
      10 TABENT1    PIC X(4) VALUE SPACES.
      10 TABENT2    PIC X(4) VALUE SPACES.
```

```
IF TABENT1 (TABIDX) NOT = TABENT2 (TABIDX)
    statements
END-IF
```

Performance: V5 is 17% faster

Chapter 4. How to tune compiler options to get the most out of V5

Enterprise COBOL V5 offers a number of new and substantially changed compiler options that can affect performance. This section highlights these options and gives recommendations on the optimal settings in order to achieve the best possible performance for your application.

Recommended compiler option set for best performance is: OPT(2), ARCH(x)

These options improve performance through:

- Maximum level of optimization - OPT(2)
- Deepest architecture exploitation – ARCH(x), where x = 6 | 7 | 8 | 9 | 10. Set the value as high as possible in accordance with the recommendations in this document and COBOL V5 PG.

Additional settings for maximum performance applicable to some users are: STGOPT, AFP(NOVOLATILE), HGPR(NOPRESERVE)

These options improve performance through:

- Removal of unreferenced data items – STGOPT
- Omitting of saves/restores for floating point and high word registers – AFP and HGPR

Note: There are some important prerequisites for using these additional options as discussed below, and in Chapter 17, Compiler Options of **COB PG**. Read and understand these options settings completely before using.

In short, these restrictions are:

- STGOPT - If your program relies upon unreferenced level 01 or level 77 data items (e.g. "eye-catchers"), STGOPT cannot be used (as STGOPT might remove these items)
- AFP(NOVOLATILE) - Requires a CICS® Transaction Server V4.1 or later
- HGPR(NOPRESERVE) – must only be set when the caller is Enterprise COBOL, Enterprise PL/I or z/OS XL C/C++ compiler-generated code

Next we will discuss the considerations when setting these and other performance-related compiler options.

AFP

Default

AFP(VOLATILE)

Recommended

AFP(NOVOLATILE)

Reasoning

Better performance as no code has to be generated to save on entry and restore on exit the Additional Floating Point (AFP) registers 8-15. Using the

recommended AFP setting is particularly important to improve the performance of relatively small COBOL programs that are entered many times.

The use of AFP(NOVOLATILE) over AFP(NOVOLATILE) reduces the overhead of a program call by 9% at OPT(2). Note this was measured in an otherwise empty COBOL program to emphasize the performance cost of this option and would be less of an overall degradation in a more substantial callee program.

Considerations

Specifying AFP(NOVOLATILE) requires a CICS Transaction Server V4.1 or later.

(**COB PG**: pp 316)

ARCH

Default

ARCH(6)

Recommended

ARCH(x) where x is the lowest level of hardware your application will have to be run on, including any disaster recovery systems, in order to achieve the best performance.

Reasoning

Higher ARCH settings enable the compiler to exploit features of the corresponding and all earlier hardware models in order to improve performance.

Considerations

None besides matching to hardware level

By varying only ARCH and keeping the other options at their best recommended settings, the following performance improvements were measured over a set of IBM internal performance benchmarks:

Table 2. ARCH levels with average improvement

ARCH Levels	Average % Improvement
ARCH(7) vs. ARCH(6)	1.6%
ARCH(8) vs. ARCH(7)	0.75%
ARCH(9) vs. ARCH(8)	1.03%
ARCH(10) vs. ARCH(9)	0.83%

When moving from ARCH(6) directly to ARCH(10), the average performance gain on these same set of benchmarks is 4.14%.

Note that only the ARCH compiler option was changed for the numbers above, and the underlying hardware was an IBM zEnterprise EC12 machine in all cases. This means the performance gains are strictly from compiler improvements in optimizing the COBOL applications tested.

These benchmarks are a mix of computational intensive and also I/O intensive applications, so some individual benchmarks improve dramatically and others less so or not at all.

For example, three of the more computationally intensive benchmarks improved by 17%, 21% and 34% respectively comparing ARCH(6) to ARCH(10).

Whereas other benchmarks that spend the majority of time performing I/O operations and not very much time in the compiler generated code are not affected by the ARCH compiler option and performance doesn't change significantly.

For reference, the mapping between ARCH settings and hardware models is provided below:

Table 3. ARCH settings and hardware models

ARCH	Hardware Models
ARCH(6)	2084-xxx models (z990) 2086-xxx models (z890)
ARCH(7)	2094-xxx models (IBM System z9® EC) 2096-xxx models (IBM System z9 BC)
ARCH(8)	2097-xxx models (IBM System z10® EC) 2098-xxx models (IBM System z10 BC)
ARCH(9)	2817-xxx models (IBM zEnterprise z196 EC) 2818-xxx models (IBM zEnterprise z114 BC)
ARCH(10)	2827-xxx models (IBM zEnterprise EC12) 2828-xxx models (IBM zEnterprise BC12)

(**COB PG**: pp 317)

ARITH

Default

ARITH(COMPAT)

Recommended

Use ARITH(EXTEND) only if the larger maximum number of digits enabled by this option is required (31 instead of 18). Otherwise, use ARITH(COMPAT) as it can result in better performance in some cases.

Reasoning

In addition to allowing larger variables to be declared, ARITH(EXTEND) also raises the maximum number of digits maintained for intermediate results. These larger intermediate results require different and sometimes more expensive runtime library routines to be used in order to deal with larger sizes.

For example, the comp-1 floating point exponentiation:

```
COMPUTE C = A ** B
```

Is 67% faster when using ARITH(COMPAT) compared to ARITH(EXTEND).

Similarly for the comp-3 multiplication where all data items are declared with 18 digits:

```
COMPUTE D = A * B * C
```

Is 24% faster when using ARITH(COMPAT) compared to ARITH(EXTEND).

AWO

Default

NOAWO

Recommended

AWO, unless the written record is required to be updated on disk as soon as possible

Reasoning

A large reduction of EXCPs is possible by combining records written together in a block, resulting in faster file output operations, and lower CPU usage.

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.

Notes:

- The APPLY WRITE-ONLY clause can be used on the physical sequential, variable-length, blocked files in the program instead of using the AWO compiler option. However, to obtain the full performance benefit, the APPLY WRITE-ONLY clause would have to be used on every physical sequential, variable-length, blocked file in the program. When used this way, the performance benefits will be the same as using the AWO compiler option.
- The AWO compiler option has no effect on a program that does not contain any physical sequential, variable-length, blocked files.

As a performance example, one test program using variable-length blocked files and AWO was 90% faster than NOAWO. This faster processing was the result of using 98% fewer EXCPs to process the writes.

(**COB PG**: pp 12-13, 307-308, 672)

BLOCK0

Default

NOBLOCK0

Recommended

BLOCK0

Reasoning

Blocked I/O can reduce the number of physical I/O transfers, resulting in fewer EXCPs. The BLOCK0 compiler option changes the default for QSAM files from unblocked to blocked (as if the BLOCK CONTAINS 0 clause were specified for the files), and thus gain the benefit of

system-determined blocking for output files. BLOCK0 activates an implicit BLOCK CONTAINS 0 clause for each file in the program that meets all of the following criteria:

- The FILE-CONTROL paragraph either specifies ORGANIZATION SEQUENTIAL or omits the ORGANIZATION clause.
- The FD entry does not specify RECORDING MODE U.
- The FD entry does not specify a BLOCK CONTAINS clause.

As a performance example, one test program using BLOCK0 that meets the above criteria was 90% faster than a corresponding one using NOBLOCK0, and used 98% fewer EXCPs.

(**COB PG** pp 307-308, 672)

DATA(24) and DATA(31)

Default

DATA(31)

Recommended

DATA(31), if the program doesn't need to call and pass parameters to AMODE 24 subprograms.

Reasoning

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 runtime 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:

- For NORENT programs, the RMODE option determines where non-EXTERNAL data is allocated.
- See QSAM buffers for additional information on QSAM file buffers.
- See ALL31 for information on where EXTERNAL data is allocated.
- LOCAL-STORAGE data is not affected by the DATA option. The STACK runtime option and the AMODE of the program determine where LOCAL-STORAGE is allocated.

Note that while it is not expected to impact the performance of the application, it does affect where the program's data is located.

(**COB PG**: pp 43-44, 172-173, 314-315, 341-342, 408, 431, 464, 467, 673)

DYNAM

Default

NODYNAM

Considerations

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 relinked 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.

As a performance example of using CALL literal in a CALL intensive program (measuring CALL overhead only), the overhead associated with the CALL using DYNAM was around 100% slower than NODYNAM. The result is affected by the number of calls to the same program. A larger number of calls tend to amortize more the overhead cost of loading the subprogram.

For additional considerations using call literal and call identifier, see "Using CALLs" on page 43.

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 320-321, 410, 412, 429, 440, 450-453, 455-456, 673)

FASTSRT

Default

NOFASTSRT

Recommended

FASTSRT, if COBOL file error handling semantics is not needed during the sort processing.

Reasoning

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.

As a performance example, one test program that processed 100,000 records was 45% faster when using FASTSRT compared to using NOFASTSRT and used 4,000 fewer EXCPs.

(**COB PG**: pp 225-228, 231, 322, 673)

HGPR

Default

HGPR(PRESERVE)

Recommended

HGPR(NOPRESERVE)

Reasoning

Better performance as no code has to be generated to save on entry and restore on exit the high halves of the 64-bit GPRs. Using the recommended HGPR setting is particularly important to improve the performance of relatively small COBOL programs that are entered many times.

The use of HGPR(NOPRESERVE) over HGPR(PRESERVE) reduces the overhead of a program call by 29% at OPT(2). Note this was measured in an otherwise empty COBOL program to emphasize the performance cost of this option and would be less of an overall degradation in a more substantial callee program.

Considerations

The PRESERVE suboption is necessary only if the caller of the program is not Enterprise COBOL, Enterprise PL/I, or z/OS XL C/C++ compiler-generated code.

(**COB PG**: pp 338)

MAXPCF

MAXPCF is a new option in V5 to automatically reduce the amount of optimization for large and complex programs that may require excessive compilation time or excessive storage requirements.

The MAXPCF option is intended to allow large programs to compile successfully but at the cost of reduced optimization. However, if it is possible, it is recommended to restructure your large applications into smaller separate programs.

A number of new "global" optimizations have been added to the V5 compiler release. These optimizations are termed "global" as they attempt to find synergies and improve performance across an entire program instead of just "locally" within a statement or a linear set of statements in a section. Because these global optimizations must analyze the statements and data items of the entire program, they sometimes require significant amounts of storage and time.

For this reason, and to generally benefit software maintenance activities, it is strongly recommended to break large programs into smaller separate programs linked together by static calls (to minimize overhead versus using dynamic calls). These smaller programs will have a much better chance of not requiring a downgrade of optimization by the MAXPCF option. This will also likely result in faster compile times requiring less storage, and the final compiled and linked application will have been subject to the full suite of optimizations available in the V5 compiler.

(**COB PG**: pp 343)

NUMPROC

Default

NUMPROC(NOPFD)

Recommended

When your numeric data exactly conforms to the IBM system standards detailed in the **COB PG** pp 348, use NUMPROC(PFD) to improve the performance of your application.

Reasoning

NUMPROC(PFD) improves performance as the compiler no longer has to generate code to correct input sign configurations. This is particularly important when your application contains unsigned internal decimal and zoned decimal data, as this type of data requires correction before use in any arithmetic or compare statements, in addition to also correcting after certain arithmetic, move and compare statements.

A benchmark that contains many types of arithmetic improves by 4% when using NUMPROC(PFD) compared to NUMPROC(NOPFD)

OPTIMIZE

Default

OPT(0)

Recommended

OPT(2)

Reasoning

Maximum level of optimization generally results in the fastest performing code to be generated by the compiler.

Considerations

OPT(2) compiles generally use more memory and take longer to complete compared to using OPT(1) or OPT(0).

Compile-time data gathered from a set of benchmarks show that on average OPT(2) takes 1.8 times longer than OPT(1) and 3.4 times longer than OPT(0) (comparing CPU time). For very large test cases, the compile-time trade off can be worse than the average.

In addition, debuggability can be reduced as compiler optimizations such as instruction scheduling and dead code removal is more advanced at this setting.

Note that unreferenced level 01 and level 77 items are no longer deleted with this highest OPT setting as was the case in V4 with OPT(FULL). This means programs that could not use OPT(FULL) previously can specify OPT(2). See “STGOPT” on page 25 for more information.

Although both V4 and V5 offer three levels of OPT specifications, the names and more importantly the underlying optimizations enabled have changed.

For example, an important difference between V4 and V5 is that the highest setting in V4 of OPT(FULL) was the suite of OPT(STD) optimizations plus the removal of unreferenced data items and the corresponding code to initialize their VALUE clauses.

In contrast, the highest setting in V5 is OPT(2) and this contains the suite of OPT(1) optimizations plus additional optimizations to improve performance, such as globally propagating values and sign state information, better register allocation for accessing indexed tables and low level instruction scheduling.

As detailed in **COB PG** pp 352 (Table 48), the V4 OPT settings are currently tolerated, but none of the V4 settings map to OPT(2). For example, OPT(FULL) specified with V5 is mapped to OPT(1) and STGOPT.

SSRANGE

Default

NOSSRANGE

Recommended

For best performance, NOSSRANGE is recommended. Specifying SSRANGE will cause extra code to be generated to detect out of range storage references.

Reasoning

The extra checks enabled by SSRANGE can cause significant performance degradations for programs with index, subscript and reference modification expressions in performance sensitive areas of your program. If only a few places in your program require the extra range checking, then it might be faster to code your own checks instead of using SSRANGE which will enable checks for all cases.

Note that in V5, there is no longer a runtime option to disable the compiled-in checks. So specifying SSRANGE will always result in the range checking code to be used at runtime. A benchmark that makes moderate use of subscripted references to tables slows down by 20% when SSRANGE is specified.

STGOPT

Default

NOSTGOPT

Recommended

STGOPT

Reasoning

This is a new option in V5 that is now orthogonal to OPT. In V4, the STGOPT behavior to remove unreferenced data items and the corresponding code to initialize their VALUE clauses is implied when going from OPT(STD) to OPT(FULL). In V5, that behavior is now specified independently. Over a set of benchmark programs, the use of STGOPT results in an average 2.8% reduction in the size of the object file at OPT(2), and a maximum reduction of 11.8%.

Considerations

The same considerations that applied in V4 to specifying OPT(FULL) should be used in deciding to use STGOPT in V5. That is, if your program relies upon unreferenced level 01 or level 77 data items, then neither OPT(FULL) nor STGOPT should be used.

TEST and OPT Interaction

See **COB PG** pp 364 –366 for a full discussion of the TEST option and suboptions including a discussion of performance versus debugging capability tradeoffs.

To summarize the performance tradeoffs of programs compiled with TEST options:

- NOTEST performs better than TEST(NOEJPD)
- TEST(NOEJPD) performs significantly better than TEST(EJPD)

TEST(EJPD) enables the JUMPTO and GOTO commands and therefore puts severe restrictions on the amount optimization performed by the compiler. The NOEJPD suboption limits the compiler to in-statement optimizations to allow the JUMPTO and GOTO commands to work properly.

Note: Debug Tool now allows GOTO/JUMPTO with TEST(NOEJPD). It is not always reliable, but might be called a "Dirty GOTO", which is much like debuggers provided by other software vendors.

TEST(NOEJPD) also restricts the optimizer, but much less so than TEST(EJPD). The NOEJPD suboption allows the viewing of data items at statement boundaries and this restricts the optimizer in removing some dead code and dead stores as well as inhibiting instruction scheduling performance improvements.

The table below shows average execution time performance numbers for an IBM internal benchmark that simulates a large number of financial transactions in an automated teller machine.

The numbers were produced using ARCH(10) and at OPT(1) and OPT(2), using different TEST suboptions. The percentages show the performance degradations of TEST(NOEJPD) or TEST(EJPD) over NOTEST.

Table 4. Performance degradations of TEST(NOEJPD) or TEST(EJPD) over NOTEST

OPT level	TEST(NOEJPD) % degradation versus NOTEST	TEST(EJPD) % degradation versus NOTEST
OPT(1)	7.9%	42.1%
OPT(2)	2.9%	66.2%

As expected, this demonstrates the much larger impact on performance of TEST(EJPD) versus TEST(NOEJPD).

THREAD

Default

NOTHREAD

Recommended

NOTHREAD

Reasoning

The THREAD option requires additional locking in the generated code and the COBOL runtime library, which can impact performance if the program is not running in an multi-threaded environment.

This applies not just to Enterprise COBOL V5, but also applies to previous Enterprise COBOL compilers.

The THREAD option indicates that a COBOL program is to be enabled for execution in an environment that has multiple POSIX threads or PL/I tasks. In order to do so, the compiler inserts locks in various places in the generated code to protect the execution. This can impact performance of a THREAD compiled program in comparison with a corresponding NOTHREAD program.

It is recommended that the NOTHREAD option is used unless the program requires it. The compiler default is NOTHREAD.

One example where the compiler needs to insert locks to protect is I/O. When the THREAD option is used, all I/O verbs (OPEN, READ, WRITE, REWRITE, CLOSE, etc) are protected by locks. In a measurement, we observed a performance degradation of 10% due to the THREAD option.

TRUNC

As in V4, there are three possible settings for the TRUNC option: BIN, STD and OPT.

The recommended option for best performance continues to be TRUNC(OPT), as this allows the compiler the most freedom in determining the most efficient code to generate. For additional information on determining which TRUNC option to specify, see **COB PG** pp 368, "TRUNC" option.

The cost of using TRUNC(STD) has been improved compared to V4, as the divide instruction used to truncate the result back to the number of digits in the PICTURE clause of the BINARY receiving data item is only conditionally executed in V5. The compiler inserts a runtime check for overflow and will branch around the divide if no truncation is required.

However, better performance is still possible when using TRUNC(OPT) as no runtime overflow checks or divide instructions are required at all.

TRUNC(BIN) will often result in poorer performance, and is usually the slowest of the three TRUNC suboptions. Although no divides (conditional or otherwise) are required in order to truncate results, the full 2, 4 or 8 byte value is considered significant and therefore intermediate results grow that much more quickly and require conversions to larger or more complex data types.

For example, adding two BINARY PIC(10) values with TRUNC(BIN) requires conversion to packed decimal and a library call.

But if TRUNC(STD) or TRUNC(OPT) is used instead, the maximum intermediate result size is 11 digits and no conversions to another type or library call is required, performance is much better.

In one program with a significant amount of binary arithmetic setting TRUNC(BIN) results in a 76% slowdown compared to TRUNC(STD). This performance difference is due to the runtime library calls required for the larger intermediate result sizes.

Another program that also contains a significant amount of binary arithmetic improves by 4% when specifying TRUNC(OPT) compared to TRUNC(STD).

In V4, this same benchmark improves by 39% when specifying TRUNC(OPT) compared to TRUNC(STD).

Note that these are only relative improvements when compiling with the same release and only changing the TRUNC setting. Comparing between releases shows that V5 in this program compared to V4 performs 15% faster in the TRUNC(OPT) case, 46% faster in the TRUNC(STD) case, and 82% faster in the TRUNC(BIN) case.

See "BINARY (COMP or COMP-4)" on page 51 for a more detailed discussion and study of BINARY data and interaction with TRUNC suboptions.

Program residence and storage considerations

Compiler option

The following compiler options can affect where the program resides (above/below the 16 MB line), which in turn can affect the location of WORKING-STORAGE section, and I/O file buffers and record areas.

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 42-44, 173, 285-286, 291, 296, 341-342, 412, 415, 431-432, 437, 464, 467, 482, 491, 495, 499, 674, 676)

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. While it is not expected to impact the performance of the application, it can affect where the program and its WORKING-STORAGE are located.

(**COB PG**: pp 42-44, 173, 341, 342-343, 431, 467, 674)

Location of Storage

WORKING-STORAGE

COBOL WORKING-STORAGE is allocated from the Language Environment heap storage when the COBOL program, compiled with the RENT option, is in one of the following cases:

- Running in CICS environments
- Enterprise COBOL V4.2 or earlier releases
- COBOL V5.1.1 in a program object that contains only COBOL programs (V5.1.1, V4.2 or earlier) and assembly programs. There is no Language Environment interlanguage call within the program object. There is no COBOL V5.1.0 programs compiled before the V5.1.1 PTF.
- COBOL V5.1 in a program object where the main entry point is COBOL V5.1 or later releases. In this case, the program object can contain Language Environment interlanguage calls, with COBOL statically linking with C, C++ or PL/I. All COBOL V5.1 programs within such program object (even if they are not the main entry point) have their WORKING-STORAGE allocated from heap storage.
- COBOL V5.1 program compiled with the DATA(24) option.

COBOL WORKING-STORAGE is not allocated from the Language Environment heap storage for cases other than the above.

LOCAL-STORAGE

COBOL LOCAL-STORAGE is always allocated from the Language Environment stack storage. It is affected by the LE STACK runtime option.

EXTERNAL variables

External variables in an Enterprise COBOL program are always allocated from the Language Environment heap storage.

QSAM buffers

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 compiled with RENT and DATA(31) or compiled with NORENT and RMODE(ANY)
- The program is executing in AMODE 31
- The ALL31(ON) and HEAP(,,ANYWHERE) runtime options are used (for EXTERNAL files)
- The file is not allocated to a TSO terminal
- The file is not spanned external, spanned with a SAME RECORD clause, or spanned opened as I-O and updated with REWRITE

(**COB PG**: pp 44, 159-161, 172-173, 672)

VSAM buffers

VSAM buffers can be allocated above the 16 MB line if 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.

Chapter 5. Runtime options that affect runtime performance

Selecting the proper runtime options 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 runtime options are set up correctly for your installation. It is also important to understand these options so that you can set the appropriate options for specific programs, applications, and regions that require fast performance. The individual LE runtime options can be set using any of the supported methods for setting that individual option. Below examines some of the options that can help to improve the performance of the individual application, as well as the overall LE runtime environment.

Note: In the following option description, if an option setting is different between CICS and non-CICS, the setting will be qualified by text in parentheses. Otherwise the same setting applies to both CICS and Non-CICS.

(**LE PG**: pp 119-130; **LE REF**: pp 3-6, 9-104; **LE CUST**: pp 19-30, 45-144)

AIXBLD

Default
OFF (Non-CICS); N/A (CICS)

Recommended
OFF (Non-CICS); N/A (CICS)

Considerations
The AIXBLD option allows alternate indexes to be built at run time. However, this may adversely affect the runtime 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 AIXBLD runtime option. Note that AIXBLD is not supported when VSAM data sets are accessed in RLS mode.

(**LE REF**: pp 11-12; **LE CUST**: pp 50; **COB PG**: pp 204-205, 208, 648-649)

ALL31

Default
ON

Recommended
ON, unless there are AMODE(24) routines in the application

Considerations
The ALL31 option allows LE to take advantage of the knowledge 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 runtime 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) runtime option and compile the program with either the DATA(31) and RENT compiler options or with the RMODE(ANY) and NORENT compiler options. Note that when using ALL31(OFF), you must also use STACK(,,BELOW).

Notes:

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

As a performance example (measuring CALL overhead only), a test program using ALL31(ON) was equivalent to ALL31(OFF).

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 runtime routines.

(**LE REF**: pp 12-14; **LE CUST**: pp 50-52; **COB PG**: pp 40-42, 180-181, 460-461, 472; **COB MIG**: pp 24-26)

CBLPSHPOP

Default

ON

Recommended

N/A (Non-CICS); ON (CICS), if compatible behavior with VS COBOL II is required in the EXEC CICS condition handling commands. If compatible behavior with VS COBOL II is not required, or if the program does not use any of the EXEC CICS condition handling commands, the OFF setting is recommended

Considerations

The CBLPSHPOP option controls whether CICS PUSH HANDLE and CICS POP HANDLE commands are issued when a COBOL subroutine is called.

This option only applies to the CICS environment. The CBLPSHPOP option is used to avoid compatibility problems when calling COBOL subroutines that contain CICS CONDITION, AID, or ABEND condition handling commands.

- When CBLPSHPOP is OFF and you want to handle these CICS conditions in your COBOL subprogram, you will need to issue your own CICS PUSH HANDLE before calling the COBOL subprogram and CICS POP HANDLE upon return. Otherwise, the COBOL subroutine will inherit the caller's settings and upon return, the caller will inherit any settings that were made in the subprogram. This behavior is different from that of VS COBOL II.

- When CBLPSHPOP is ON, you will receive the same behavior as with the VS COBOL II run time when using CICS condition handling commands. However, the performance of calls will be impacted.

For performance considerations using CBLPSHPOP, see "CICS" on page 46.

(**LE PG**: pp 119-120, 408; **LE REF**: pp 18-19; **LE CUST**: pp 57; **COB PG**: pp 420-422; **COB MIG**: pp 8)

CHECK

The CHECK runtime option is ignored for applications compiled with Enterprise COBOL V5.

If the compile time option SSRANGE is specified, range checks are generated by the compiler and checks are always executed at run time. The compiled-in range checks cannot be disabled.

DEBUG

Default
OFF

Recommended
OFF

Considerations
The DEBUG option activates the COBOL batch debugging features specified by the USE FOR DEBUGGING declarative. This might 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:

- When not using the USE FOR DEBUGGING declarative, on the average, DEBUG was equivalent to NODEBUG.
- When using the USE FOR DEBUGGING declarative, a test program measured was 900% slower when using DEBUG compared to using NODEBUG.

Note: The program in this test had WITH DEBUGGING MODE clause on the SOURCE-COMPUTER paragraph, and contained a USE FOR DEBUGGING ON a paragraph name in the procedure division. This paragraph is empty (that is containing just an EXIT statement), and is performed many times in a loop. The paragraph in the declarative section is also empty (just an EXIT statement). The purpose is to give an indication on the overhead due to transferring of control to the USE FOR DEBUGGING declarative.

(**LE REF**: pp 24; **LE CUST**: pp 63; **COB PG**: pp 372-373, 644)

INTERRUPT

Default
OFF (Non-CICS); N/A (CICS)

Recommended
OFF (Non-CICS); N/A (CICS)

Considerations
The INTERRUPT option causes attention interrupts to be recognized by Language Environment. When you cause an interrupt, Language Environment can give control to your application or to Debug Tool.

Performance considerations using INTERRUPT: On the average, INTERRUPT(ON) is 1% slower than INTERRUPT(OFF), with a range of equivalent to 20% slower.

(**LE REF**: pp 49; **LE CUST**: pp 88-89)

RPTOPTS

Default
OFF

Recommended
OFF

Considerations
The RPTOPTS option allows you to get a report of the runtime 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 programs are repeatedly invoked.

(**LE REF**: pp 69-70; **LE CUST**: pp 108)

RPTSTG

Default
OFF

Recommended
OFF

Considerations
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: The degradation in a call intensive program was measured to be more than 200%.

Note: The program did nothing except repeatedly calling a number of subprograms, which were empty (that is, containing only a GOBACK statement).

(**LE REF**: pp 71-72; **LE CUST**: pp 108-109)

RTEREUS

Default

OFF (Non-CICS); N/A (CICS)

Recommended

OFF (Non-CICS); N/A (CICS)

Considerations

The RTEREUS option causes the LE runtime environment to be initialized for reusability when the first COBOL program is invoked.

The LE runtime 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 runtime environment initialization will be bypassed. Most of the runtime 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 runtime environment).

Because of the effect that the STOP RUN statement has on the runtime 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, a non-LE-conforming 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). This means that storage that is acquired during the execution of the application will not be freed. Since the storage is not freed, RTEREUS cannot be used with SVC LINK since after return from the SVC LINK, the program LINKed to will be deleted by the operating system, but the COBOL control blocks will still be initialized and in storage. Therefore, RTEREUS may not be applicable to all environments.

Performance considerations using RTEREUS (measuring CALL overhead only): One testcase (a non-LE-conforming 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.

(**LE REF**: pp 72-74; **LE CUST**: pp 110-112, 174-175; **LE MIG**: pp 19; **COB PG**: pp 322, 357, 445, 501; **COB MIG**: pp 248-249)

STORAGE

Default

NONE,NONE,NONE,0K

Recommended

NONE,NONE,NONE,0K

Considerations

The STORAGE option specifies the heap allocations or stack 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 (see note below), 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. In any case, 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.

The WORKING-STORAGE is affected by the STORAGE option in the following categories of RENT programs:

- When the program runs in CICS environment
- When the program is compiled with Enterprise COBOL V4.2 or earlier
- When the program object (where the program resides) contains only programs compiled with COBOL V5.1.1 or later, or compiled with COBOL V4.2 or earlier compilers; (i.e. there is no Language Environment interlanguage calls within the program object)
- When the primary entry point of the program object is a program compiled with Enterprise COBOL V5.1.1 or later; (i.e. having LE interlanguage calls within the program object is allowed)

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. You should not use STORAGE(,,00) to initialize variables for your application. Instead, you should change your application to initialize their own variables. You should not use STORAGE(,,00) in any performance-critical application.

Performance considerations using STORAGE:

- On the average, STORAGE(00,00,00) was 11% slower than STORAGE(NONE,NONE,NONE), with a range of equivalent to 133% 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 5 % 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 9% slower.
- For a call intensive program, STORAGE(NONE,NONE,00) can degrade more than 100%, depending on the number of calls.

  Note: 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 79-81; **LE CUST**: pp 118-121; **LE MIG**: pp 19)

TEST

Default

NOTEST(ALL,*,PROMPT,INSPPREF)

Recommended

NOTEST(ALL,*,PROMPT,INSPPREF)

Considerations

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.

(**LE REF**: pp 87-89; **LE CUST**: pp 127-129)

TRAP

Default

ON,SPIE

Recommended

ON,SPIE

Considerations

The TRAP option allows LE to intercept an abnormal termination (abend), provide the abend information, and then terminate the LE runtime 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. In addition, LE uses condition handling internally and requires TRAP(ON). 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).

When using the SPIE suboption, LE will issue an ESPIE to handle program interrupts. When using the NOSPIE suboption, LE will handle program interrupts via an ESTAE.

Performance considerations using TRAP: On the average, TRAP(ON) was equivalent to TRAP(OFF).

(**LE PG**: pp 197-198, 407, 427; **LE REF**: pp 96-98; **LE CUST**: pp 137-140, 186; **COB PG**: pp 173, 200, 215, 238; **COB MIG**: pp 219)

VCTRSAVE

Default

OFF (Non-CICS); N/A (CICS)

Recommended

OFF (Non-CICS); N/A (CICS)

Considerations

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**: pp 100-101; **LE CUST**: pp 142-143)

Chapter 6. COBOL and LE features that affect runtime performance

COBOL and Language Environment have several installation and environment tuning features that can enhance the performance of your application.

The following information describes 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.

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 unused, 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 unused, 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) runtime 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 runtime 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)
```

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

```
ANYHEAP(4K,4080,ANYWHERE,FREE)
BELOWHEAP(4K,4080,FREE)
HEAP(4K,4080,ANYWHERE,KEEP,4K,4080)
LIBSTACK(32,4000,FREE)
STACK(4K,4080,ANYWHERE,KEEP,4K,4080)
```

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 reduced by reducing the default storage options and by moving some of the library routines above the line.

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

(**LE PG**: pp 169-192; **LE REF**: pp 17-18, 36-37, 40-42, 49, 75-77, 79-81; **LE CUST**: pp 52-53, 54-56, 74-77, 79-81, 113-116, 118-121, 129-133, 156, 167-168, 198-215; **COB MIG**: pp 97, 179-180, 217-218)

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 156, 198-216)

Using the CEEENTRY and CEETERM macros

To improve the performance of non-LE-conforming Assembler calling COBOL, you can make the Assembler program LE-conforming. This can be done using the CEEENTRY and CEETERM macros provided with LE.

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.

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

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

  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 also "First program not LE-conforming" on page 46 for additional performance considerations comparing using CEEENTRY and CEETERM with other environment initialization techniques.

(**LE PG**: pp 448-453)

Using preinitialization services (CEEPIPI)

LE preinitialization services (CEEPIPI) can also be used to improve the performance of non-LE-conforming Assembler calling COBOL.

LE preinitialization services let an application initialize the LE environment once, execute multiple LE-conforming programs, 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 for an example of using CEEPIPI to call a COBOL subprogram and Using CEEPIPI with Call_Main for an example of using CEEPIPI to call a COBOL main program.

Performance considerations using CEEPIPI (measuring CALL overhead only):

- One testcase (a non-LE-conforming 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.

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 "First program not LE-conforming" on page 46 for additional performance considerations comparing using CEEPIPI with other environment initialization techniques.

(**LE PG**: pp 481-524)

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 non-LE-conforming assembler drivers that repeatedly call LE-conforming languages and for IMS/TM regions. LRR is not supported under CICS. See "IMS" on page 49 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 runtime 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 library routine retention (LRR)" for an example of using LRR.

Performance considerations using LRR:

- One testcase (a non-LE-conforming Assembler calling COBOL) using LRR was 96% faster than using not using LRR.

  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 "First program not LE-conforming" on page 46 for additional performance considerations comparing using LRR with other environment initialization techniques.

(**LE PG**: pp 444-447; **LE CUST**: pp 169)

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 data set.

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.

(**LE CUST**: pp 9, 11-12, 247-269; **LE MIG:** pp 3)

Using CALLs

You should consider storage management tuning for all CALL intensive applications.

With static CALLs (call literal with NODYNAM), 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 (call literal with DYNAM or call identifier), each subprogram is link-edited separately from the others. They are brought into storage only if they are needed. This is the better way of managing complicated applications. 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.

Another aspect is program loading. Since a dynamic CALL subprogram is brought into storage when it is first needed, it is not loaded into storage at the beginning together with the caller program. There is an overhead in terms of program load processing. In general, it is beneficial to use dynamic calls when the call structure of an application is complicated, the size of the subprograms is not small, and not all subprograms are called in a particular run of the application.

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

- Static CALL literal was on average 40% faster than dynamic CALL literal.
- Static CALL literal was on average 52% faster than dynamic CALL identifier.
- Dynamic CALL literal was on average 20% faster than dynamic CALL identifier.

Note: These measurements are only for 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 457-472)

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.

There is an overhead in initializing all WORKING-STORAGE variables with VALUE clauses. The performance impact depends on the number and sizes of such variables.

(**COB PG**: pp 5, 14, 173, 200, 457, 458-459; **COB LRM**: pp 84, 102)

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 (an LE-conforming Assembler repeatedly calling COBOL) using IS RECURSIVE was 15 % 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 4, 17, 469; **COB LRM**: pp 98, 101-102)

Chapter 7. Other product related factors that affect runtime performance

It is important to understand COBOL's interaction with other products in order to enhance the performance of your application.

This section describes some product related factors that should be considered for the application.

Using ILC with Enterprise COBOL

Interlanguage communication (ILC) applications are applications built of two or more high-level languages (such as COBOL, PL/I, or C) and frequently assembler.

ILC applications run outside of the realm of a single language environment, which creates special conditions, such as how the data from each language maps across load module boundaries, how conditions are handled, or how data can be passed and received by each language.

While LE fully supports ILC applications, there can be significant performance implications when using them with COBOL applications. One area to look at is condition handling.

COBOL normally either ignores decimal overflow conditions or handles them by checking the condition code after the decimal instruction. However, when languages such as C or PL/I are in the same application as COBOL, these decimal overflow conditions will now be handled by LE condition management since both C and PL/I set the Decimal Overflow bit in the program mask. This can have a significant impact on the performance of the COBOL application if decimal overflows are occurring during arithmetic operations in the COBOL program.

Performance considerations for a COBOL program using COMP-3 (PACKED-DECIMAL) data types in 100,000 arithmetic statements that cause a decimal overflow condition:

- The C or PL/I, ILC case was over 100 times slower than the COBOL only, non ILC case, measured in CPU time.

Notes:

- The C or PL/I program does not need to be called. Just the presence of C or PL/I in the load module will cause this degradation for decimal overflow conditions.
- When XML GENERATE or XML PARSE statements are in the program, the C runtime library will be initialized. Hence, the decimal overflow bit in the program mask will be set even though you do not explicitly have a C or PL/I program in the application. This will also cause the same degradation for decimal overflow conditions.

(**COB PG**: pp 4, 17, 469; **COB LRM**: pp 98, 101-102)

First program not LE-conforming

If the first program in the application is non LE-conforming, and if this program is repeatedly calling COBOL, there can be a significant degradation because 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 sub 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 runtime option RTEREUS to initialize the runtime environment for reusability, making all COBOL main programs become subprograms.
- Use the Library Routine Retention (LRR) function (similar to the function provided by the LIBKEEP runtime option in VS COBOL II).
- Call CEEPIPI main 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.
- Place the LE library routines in the LPA or ELPA. The list of routines to put in the LPA or EPLA is release dependent and is the same routines listed under the IMS preload list considerations.

(**LE PG**: pp 439-480)

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.

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 runtime 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 40) to modify the default behavior of this automatic storage tuning.

(**LE CUST**: pp 165-167)

The RENT compiler option is required for an application running under CICS. Additionally, if the program is run through the CICS translator or co-processor (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 344-345, 472)

Enterprise COBOL supports static and dynamic calls to Enterprise COBOL and VS COBOL II (with the RES option) subprograms containing CICS commands or dependencies. Note that Enterprise COBOL 4.2 and earlier releases also support calls to VS COBOL II with the NORES option. 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. Enterprise COBOL does not support 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. 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 648-649)

If you are using the COBOL CALL statement to call a program that has been translated with the CICS translator or has been compiled with the CICS co-processor, 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 runtime 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 runtime. The CBLPSHPOP setting can be changed dynamically by using the CLER transaction.

(**LE PG**: pp 119-120, 406-409; **LE REF**: p 18; **LE CUST**: p 57; **COB PG**: pp 413-414, 420-422)

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 compiler option, see "TRUNC" on page 27.

(**COB PG**: pp 357-359, 418-419; **COB MIG**: pp 82, 201-203)

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 "TRUNC" on page 27.

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

For the best performance, make sure that you use codepages that are compatible to avoid unnecessary conversions. For example, if your DB2 database uses codepage 037, but you use the CODEPAGE(1140), SQL, SQLCCSID compiler options, the performance can be slower than using either CODEPAGE(037), SQL, SQLCCSID or CODEPAGE(1140), SQL, NOSQLCCSID since the first set of options require conversions to match the codepage but the second and third set of options do not require such conversions.

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

(**COB PG**: pp 344-345, 357-359, 427-429, 472)

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 an INPUT PROCEDURE phrase or an OUTPUT PROCEDURE phrase for your sort files, the FASTSRT option has no impact to the INPUT PROCEDURE or the OUTPUT PROCEDURE. However, if you have an INPUT PROCEDURE phrase with a GIVING phrase or a USING phrase with an

OUTPUT PROCEDURE phrase, FASTSRT will still apply to the USING or GIVING part of the SORT statement. The complete list of requirements is contained in the **COB PG**.

Performance considerations using DFSORT:

- One program that processed 100,000 records is 50% faster when using FASTSRT compared to using NOFASTSRT.

(**COB PG:** pp 225-228, 231, 322, 673)

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 435-444 472, 648-649; **COB MIG**: pp 211-212, 218-219)

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 runtime 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 steps:

1. In your startup JCL or procedure to bring up the IMS dependent region, specify the PREINIT=xx parameter, where xx is the 2-character suffix of the DFSINTxx member in your IMS PROCLIB data set.
2. Include the name CEELRRIN in the DFSINTxx member of your IMS PROCLIB data set.
3. 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 data set. If you do this, use your module name in place of CEELRRIN above.

(**LE PG**: pp 444-447; **LE CUST**: pp 169-170)

Warning: If the RTEREUS runtime option is used, the top level COBOL programs of all applications must be preloaded.
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 modules:

- For all COBOL applications: CEEBINIT, IGZCPAC, IGZCPCO, CEEEV005, CEEPLPKA, IGZETRM, IGZEINI, IGZCLNK, CEEV004, IGZXDMR, IGZXD24, IGZXLPIO, IGZXLPKA, IGZXLPKB, IGZXLPKC
- If the application also contains VS COBOL II programs: IGZCTCO, IGZEPLF, and IGZEPCL

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

Other than the COBOL library modules listed above, you should also preload any of the below the line routines that you need. A list of the below the line routines can be found in the Language Environment Customization manual.

(**LE CUST**: pp 249-250)

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

The TRUNC(OPT) compiler option can be used if the following conditions are satisfied:

- 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. For additional information on the TRUNC compiler option, see “TRUNC” on page 27.

(**COB PG**: pp 357-359)

Chapter 8. Coding techniques to get the most out of V5

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.

BINARY (COMP or COMP-4)

BINARY data and synonyms COMP and COMP-4 are the two's complement data representation in COBOL.

BINARY data is declared with a PICTURE clause as with internal or external decimal data but the underlying data representation is as a halfword (2 bytes), a fullword (4 bytes) or a doubleword (8 bytes).

The compiler option TRUNC(OPT | STD | BIN) determines if and how the compiler corrects values back to the declared picture clause and how much significant data is present when accessing a data item.

Although the overall general performance considerations for BINARY data and the TRUNC option from V4 and earlier versions still apply in V5, some of the relative performance differences for the various TRUNC suboptions have changed (sometimes dramatically). These changes may impact coding and compiler option choices.

To quantify the relative and absolute performance differences a series of addition operations on binary data items were executed in a loop on an EC12 machine. The 4 tests below contain the same type and number of arithmetic operations but have a varying number of digits:

- TEST 1: 8 additions with one each of 1 through 8 digits
- TEST 2: 8 additions each with 9 digits
- TEST 3: 8 additions with one each of 10 through 17 digits
- TEST 4: 8 additions each with 18 digits

The tests were then compiled varying the TRUNC option.

The first experiment specified the TRUNC(STD) compiler option.

TRUNC(STD) instructs the compiler to always correct back to the specified PICTURE clause and allows the compiler to assume that loaded values only have the specified number of PICTURE clause digits.

Table 5. Performance differences results of four test cases when specifying TRUNC(STD)

TRUNC(STD)	V4 versus TEST 1	V5 versus TEST 1	V5 versus V4
TEST 1: 1-8 digits	0%	0%	73%
TEST 2: 9 digits	-85%	-11%	84%
TEST 3: 10-17 digits	-357%	-27%	93%
TEST 4: 18 digits	-874%	-29%	96%

These results demonstrate that:

- V5 outperforms V4 for all lengths when using TRUNC(STD).

- Although performance slows as the number of digits increases for both V4 and V5 it slows much more gradually and to much lower overall amount using V5 compared to V4.

The second experiment specified the TRUNC(BIN) compiler option. Specifying this option is equivalent to using the COMP-5 type for all BINARY data.

TRUNC(BIN) instructs the compiler to allow values to only correct back to the underlying data representation (two, four or eight bytes) instead of back to the specified PICTURE clause. This option also requires the compiler to assume that loaded values can have up to two, four or eight bytes worth of significant data.

Table 6. Performance differences results of four test cases when specifying TRUNC(BIN)

TRUNC(BIN)	V4 versus TEST 1	V5 versus TEST 1	V5 versus V4
TEST 1: 1-8 digits	0%	0%	64%
TEST 2: 9 digits	-50%	0%	76%
TEST 3: 10-17 digits	-3413%	-7736%	20%
TEST 4: 18 digits	-3414%	-7752%	20%

These results demonstrate that:

- V5 also outperforms V4 for all lengths when using TRUNC(BIN).
- V5 shows no slow down when testing at 9 digits.
- There is a dramatic reduction in performance for both V4 and V5 (but more so for V4 in absolute terms) when the length is increased beyond 9 digits. This is due to the TRUNC(BIN) requirement that input data may contain up to the full integer half/full/double word of data (and extra data type conversions and library routines are required).

The third and final experiment specified the TRUNC(OPT) compiler option. TRUNC(OPT) is a performance option. The compiler assumes that input data conforms to the PICTURE clause and then allows the compiler the freedom to manipulate data in either of the folowing ways that are most optimal:

- Correcting back to the PICTURE clause as with TRUNC(STD) or
- Only correcting back to the two, four or eight byte boundary as with TRUNC(BIN)

Table 7. Performance differences results of four test cases when specifying TRUNC(OPT)

TRUNC(BIN)	V4 versus TEST 1	V5 versus TEST 1	V5 versus V4
TEST 1: 1-8 digits	0%	0%	5%
TEST 2: 9 digits	-305%	0%	77%
TEST 3: 10-17 digits	-121%	0%	57%
TEST 4: 18 digits	-7285%	-184%	96%

These results demonstrate that:

- V5 also outperforms V4 for all lengths when using TRUNC(OPT).
- V5 shows no slow down until the 18 digit case but still vastly outperforms V4 at this longest length.

Note: Use the TRUNC(OPT) only if you are sure the data being moved in the binary areas conforms to the PICTURE clause otherwise unpredictable results could occur. See "TRUNC" in **COB PG** for more information.

Across all the TRUNC options and data item lengths just presented V5 outperforms V4. These improvements are due to the following reasons:

- The use of 64-bit G form instructions enables much more efficient code for > 8 digit cases
- More efficient library routines for the very large TRUNC(BIN) cases

Chapter 3, "Prioritizing your application for migration to V5," on page 9 has a specific example of binary double word arithmetic (Large Binary Arithmetic) that demonstrates the performance improvement for this type of operation relative to version 4 of the compiler. In this example V5 is considerably faster than V4 and earlier compiler releases.

The relative performance differences across the different TRUNC options have been smoothed out compared to V4. This is primarily due to the compiler inserting a runtime test for overflow. If no overflow is possible then an expensive divide hardware instruction is avoided.

If your data is known to conform to the PICTURE clause then TRUNC(OPT) remains the best overall option to choose but relatively speaking it improves less over TRUNC(STD) than in V4 and overall absolute performance is better with either option in V5.

Although TRUNC(BIN) enables more efficient code when storing out a COMPUTE or MOVE result it continues to significantly harm the performance when these data items are used as input to arithmetic statements (as the compiler must assume the max 2,4,8 byte size). V5 optimizes the correction code for TRUNC(STD) so the performance benefit of TRUNC(BIN) has been reduced slightly.

It might be better to only specify COMP-5 for select data items versus using TRUNC(BIN). For example, performance will usually be improved if data items in COMPUTE statements in particular are not specified with COMP-5.

DISPLAY

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "Avoid using USAGE DISPLAY data items for computations (especially in areas that are heavily used for computations)".

Using V5 and the options ARCH(10) and OPT(1 | 2) enables the compiler to efficiently convert DISPLAY operands to Decimal Floating Point (DFP). This optimization improves the performance of DISPLAY data items in computations.

Comparing data USAGE DISPLAY:

```
1 A pic s9(17).
1 B pic s9(17).
1 C pic s9(18).
```

To COMP-3:

```
1 A pic s9(17) COMP-3.
1 B pic s9(17) COMP-3.
1 C pic s9(18) COMP-3.
```

For the statement:

```
ADD A TO B GIVING C.
```

In V4 using COMP-3 is 22% faster than using DISPLAY; while in V5 using COMP-3 is only 4% faster than using DISPLAY. So the DISPLAY performance in this case has improved by 18% in V5 compared to V4.

Although performance comparisons will vary for different sizes of data and different types of computational statements the performance of DISPLAY data items in computations has generally improved in V5 when using ARCH(10) and OPT(1 | 2).

PACKED-DECIMAL (COMP-3)

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "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".

Using V5 and the options ARCH(8 | 9 | 10) and OPT(1 | 2) enables the compiler to generate inline decimal floating-point (DFP) code for some of these larger multiplication and division operations. Starting from 5.1.1, the maximum intermediate result size supported for this optimization was raised from 31 to 34 digits.

Although there is some overhead in this conversion to DFP it less of a penalty than having to invoke a library routine.

This is also true for external decimal (DISPLAY and NATIONAL) types that are always converted by the compiler to packed decimal for COMPUTE statements.

Fixed-point versus floating-point

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "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".

In V5 it is still true that floating point exponentiation is much faster than fixed point exponentiation; however the relative cost of each type of exponentiation has changed from V4 to V5.

Consider the following code example:

```
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 original, fixed point exponentiation, is 93% faster in V5 compared to V4.

The forced to floating point exponentiation is 48% faster in V5 compared to V4.

However, because floating point exponentiation remains many times faster then fixed point exponentiation, it is still recommended to use floating point exponentiation whenever possible.

Factoring expressions

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "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."

The V5 compiler will factor expressions as a part of optimization and no longer requires the factoring to be done at the source code level as was recommended in V4.

Symbolic constants

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "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".

This remains a valid and important recommendation with the one difference that the V5 compiler now tolerates the data item being re-initialized to an identical value as was specified in the VALUE clause. In this case, the V5 compiler recognizes that the value of the data item has not changed from its initial value and will still treat it as a constant.

Performance tuning considerations for Occurs Depending On tables

Usually the relative ordering of data item declarations does not have a significant or easily predictable impact on performance.

However, if your program contains Occurs Depending On (ODO) tables, the specific group layout of data items that follow an ODO table might lead to greatly degraded performance when accessing certain other variables.

Consider an ODO table declared as below:

```
01 TABLE-1.
   05 X PIC S9.
   05 Y OCCURS 3 TIMES
        DEPENDING ON X PIC X.
   05 Z PIC S9.
```

Because the size of item Y in TABLE-1 depends on another data-item, any subsequent non-subordinate items in the same level-01 record are variably located items, such as item Z in the previous example.

Any load or store to the variably located item Z requires additional code to be generated by the compiler to determine the location of Z based on the current value of X. If ODO tables are nested then multiple extra computations are required.

However, by always ending a record after the ODO table, all variables declared after the table will no longer be variably located and access to these variables will be much more efficient. The following example adds a new level 01 record after the ODO table and before any other variables are declared:

```
01 TABLE-1.
   05 X PIC S9.
   05 Y OCCURS 3 TIMES
        DEPENDING ON X PIC X.
01 WS-VARS.
   05 Z PIC S9.
```

Using PERFORM

Enterprise COBOL allows you to use the PERFORM verb in two basic ways: you may write an inline PERFORM or an out-of-line PERFORM.

An inline PERFORM is preferable from a performance perspective, because at all optimization levels control flow is straightforward. In addition, with V5 program objects, Debug Tool is capable of skipping over the contents of an out-of-line PERFORM. However, it is generally not desirable to replicate large or complicated code sequences simply to have inline PERFORMs.

In general, the executed code for an out-of-line PERFORM includes the following steps:

1. Establishing the program address where control will return when the PERFORM is completed and saving that address in a compiler generated LOCAL-STORAGE data item.
2. Branching to the start of the PERFORMed range.
3. Executing the PERFORMed range.
4. Branching back indirectly via the compiler generated data item mentioned in the first step.

In addition, the logic associated with phrases such as those for specifying the number of iterations or testing conditions is also executed.

At optimization levels above OPT(0), the compiler will attempt to remove some of the out-of-line branching code. If necessary, it will replicate code sequences to achieve this. This replication is limited to a maximum size for a PERFORM range and to a total maximum size for the whole program. There are no configuration options to control these maximum values.

This PERFORM inlining optimization can be done on a per PERFORM statement basis. However, the nature of the range being PERFORMed must have certain characteristics in order to be a candidate.

In essence, out-of-line PERFORM statements should resemble procedure calls to have the best chance to be optimized. And, of course, that implies that the range being performed should resemble a procedure.

Typically, a procedure has a single entry point and control always returns ultimately to the caller. Therefore, in a performed range, all branching (except for additional PERFORM statements that code in the range itself might execute) should remain within the range. Similarly, the program should not contain branches (other than the PERFORMs of the range) from outside the performed range to statements inside it. For example, assuming that we have sequential sentences A, B and C in order in the program, the compiler does not optimize the following PERFORMs as the second PERFORM essentially branches into the middle of the first PERFORMs range:

```
PERFORM A THROUGH C
PERFORM B THROUGH C
```

Overlapping performed ranges in general can also inhibit the PERFORM inlining optimization as well as other global optimizations that tend to work best on more straightforward control flow constructs. Assuming that, in addition to sentences A, B and C, we also have (immediately following C) sentence D. The following statements result in overlapping performed ranges:

```
PERFORM A THROUGH C
PERFORM B THROUGH D
```

Recursively performed ranges are also not recommended in COBOL as these will also inhibit optimizations. For example, the following is not recommended:

```
A.            IF COND THEN PERFORM A.
```

Recursion can be more subtle than this case. Ranges A and B might recursively call each other and this would inhibit optimization.

In general, any branching between code in the main program and code in declarative sections (except the branching that happens as part of the natural flow of the COBOL program) is an impediment to optimization. And this is certainly true of branching in the form of PERFORM statements.

You should write the COBOL code that most naturally expresses the required logic. Sometimes, however, you can achieve the same thing in a number of ways, especially in utility routines. For example, the following code expresses logic one way:

```
LOCAL-STORAGE SECTION.
 01 ACTION PIC 9.

 PROCEDURE DIVISION.

 MOVE 1 TO ACTION
 PERFORM A

 MOVE 2 TO ACTION
 PERFORM A

 MOVE 1 TO ACTION
 PERFORM A


A.      IF ACTION = 1 DISPLAY "X" ELSE DISPLAY "Y".
```

In a case like this, it is likely beneficial to specialize the performed range as follows:

```
PERORM A1

PERFORM A2

PERFORM A1

A1. DISPLAY "X".
A2. DISPLAY "Y".
```

In this specific case, the optimizer will be able to achieve the same effect. It will start by replicating the statement in A at each PERFORM statement. Then it will have to spend compilation resources at each PERFORM statement to realize that, in each context, it can clearly identify whether or not ACTION = 1. Furthermore, in similar code patterns, there may be cases where the programmer knows something about the use of a utility range that the optimizer is not able to deduce.

Using 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).

You can have the system determine the optimal block size 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. You can also omit the BLOCK CONTAINS clause for the file and use the BLOCK0 compiler option to achieve the same effect. This should significantly improve the file processing time (both in CPU time and elapsed time).

Performance considerations using I/O buffers for a program that reads 14,000 records and wrote 28,000 records with no BLOCK CONTAINS clause and no BLKSIZE in the JCL:

- Using BLOCK0 was 90% faster and used 98% fewer EXCPs than NOBLOCK0.

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.

Performance considerations using I/O buffers for a program that reads 14,000 records and wrote 28,000 records with no blocking:

- Using DCB=BUFNO=1 took 0.452 CPU seconds
- Using DCB=BUFNO=5 took 0.129 CPU seconds
- Using DCB=BUFNO=10 took 0.089 CPU seconds
- Using DCB=BUFNO=25 took 0.067 CPU seconds

Refer to Chapter 4.1 for a discussion on the location of QSAM buffers.

(**COB PG**: pp 44, 159-161, 172-173, 672)

Using 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” on page 20.

(**COB PG**: pp 11-13, 307)

Using HFS files

You can process byte-stream HFS files as ORGANIZATIONAL SEQUENTIAL files using QSAM and specifying the PATH=fully-qualified-pathname and FILEDATA=BINARY options on the DD statement or using an environment variable to define the file.

You can process text HFS files as ORGANIZATION SEQUENTIAL files using QSAM and specifying the PATH=fully-qualified-pathname and FILEDATA=TEXT on the DD statement or as ORGANIZATION LINE SEQUENTIAL and specifying the PATH=fully-qualified-pathname on the DD statement. Performance data using HFS files is not available at this time.

(**COB PG**: pp 145-147, 174, 207-212; **COB LRM**: pp 141-144)

Using 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. In order give an idea of the differences in using SEQUENTIAL, RANDOM, and DYNAMIC access for sequential operations on an INDEXED file, we provide the measurements that were obtained from running a COBOL program that uses an ORGANIZATION IS INDEXED file on our test system; this may not be representative of the results on your system. The COBOL program does 10,000 writes and 10,000 reads. The ratios of CPU time, elapsed time and EXCP counts are shown, with ACCESS IS SEQUENTIAL used as the base line 100%.

Table 8. CPU time, elapsed time and EXCP counts with different access mode

Access mode	CPU Time (seconds)	Elapsed (seconds)	EXCP counts
ACCESS IS SEQUENTIAL	100%	100%	100%
ACCESS IS DYNAMIC with READ NEXT	134%	143%	193%
ACCESS IS DYNAMIC with READ	713%	1095%	7189%
ACCESS IS RANDOM	1405%	3140%	15190%

Note: For the DYNAMIC with READ and the RANDOM cases, the record key of the next sequential record was moved into the data buffer prior to the READ.

If you use alternate indexes, it is more efficient to use the Access Method Services to build them than to use the AIXBLD runtime 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.

Refer to Chapter 4.1 about the location of VSAM buffers.

To improve VSAM performance, you can use system-managed buffering (SMB) when possible. To use SMB, the data set must use System Management Subsystem (SMS) storage and be in Extended format (DSNTYPE=xxx in the data class, where xxx is some form of extended format). Then you can use one of the following, depending on the record access type needed:

1. AMP='ACCBIAS=DO': optimize for only random record access
2. AMP='ACCBIAS=SO': optimize for only sequential record access
3. AMP='ACCBIAS=DW': optimize for mainly random record access with some sequential access
4. AMP='ACCBIAS=SW': optimize for mainly sequential record access with some random access

Refer to Chapter 33. Tuning your program in Programing Guide for additional coding techniques and best practices.

(**COB PG**: pp 197-199, 203-205, 676-677)

Chapter 9. Program object size and PSDE requirement

Why you might see larger program object sizes when using V5?

Enterprise COBOL V5 contains a number of advanced optimizations that trade-off smaller and more efficient generated code for a larger literal data section.

This trade-off, done to improve program performance, is the primary cause of object size increases you might see when comparing programs compiled with V5 compared to earlier releases.

One example of this optimization is when using INITIALIZE on a group:

```
01  WS-GROUP.
   05  WS-DISP    PIC 99.
   05  WS-COMP3   PIC 9v99 comp-3.
   05  WS-X4      PIC X(4).
   05  WS-COMP    PIC 9(5) comp.
   05  WS-COMP1   COMP-1.

  ...

  INITIALIZE WS-GROUP.
```

In V4 and earlier releases the generated code is:

```
0000111  INITIALIZE
    00036A  5820 C004             L     2,4(0,12)             CBL=1
    00036E  D201 A000 2098        MVC   0(2,10),152(2)        (BLW=0)+0      PGMLIT AT +136
    000374  D201 A002 2096        MVC   2(2,10),150(2)        (BLW=0)+2      PGMLIT AT +134
    00037A  D203 A004 2092        MVC   4(4,10),146(2)        (BLW=0)+4      PGMLIT AT +130
    000380  4130 0000             LA    3,0(0,0)
    000384  5030 A008             ST    3,8(0,10)             (BLW=0)+8
    000388  5030 A00C             ST    3,12(0,10)            (BLW=0)+12
```

This sequence contains 34 bytes for its 7 instructions and requires 8 bytes from the literals section.

In comparison, V5 at OPT(1 | 2) generates this code:

```
000111:            INITIALIZE WS-GROUP.
    000238  D20F 8098 3230     000111           MVC     152(16,R8),560(R3)    # _$CONSTANT_AREA+560
```

This sequence has only 6 bytes for its single instruction but requires 16 bytes from the literals section.

In terms of runtime performance improvements the V5 sequence is 59% faster.

This optimization is not limited to INITIALIZE, but is performed by V5 at OPT(1) when any sequential series of stores of literals is found. In these cases, a larger literal comprised of all the literal values being stored is created at compile time, and placed in the literals section. Then, a corresponding wider store instruction is generated to initialize these multiple sequential storage locations at once.

At OPT(2), a more advanced version of this optimization is performed as well that handles initializing MOVES that are out of order with respect to the declared order of the variables.

Components of object size increases

To demonstrate the components and scale of object size increases on a wider scale, consider these statistics obtained by compiling a large selection of COBOL tests in our performance verification suite of applications.

Table 9. V5/V4 ratio and geometric mean

	Geometric Mean	
V5/V4 ratio	**V5 OPT(1) versus V4 OPT(STD)**	**V5 OPT(2) versus V4 OPT(FULL)**
Executable code size	1.17	1.22
Program literals size	3.63	4.61
Overall object	1.36	1.48

So although the overall object size is now larger with V5, the executable code size itself in the object has grown by a smaller ratio. This difference in the growth ratio between the overall object size and the executable code size portion is due to the large growth in the literals section with V5. Although the program literals section size has grown by a large ratio, the overall size of the literals section to executable code section is relatively small (it is typically around one-tenth the size).

This explains why the much larger literals section size ratio only leads to a more modest object size growth.

The primary reason for this much larger literals section size was described in "Why you might see larger program object sizes when using V5?" on page 61.

The usual reason for this extra executable code growth over V4 is a more advanced version of an optimization in V5 to inline some out-of-line PERFORM statements to their calling site. This inlining has a number of advantages for program performance. One is to avoid the overhead of dispatching and returning from the out-of-line PERFORM, and the other is to expose the PERFORMd statements and used variables to the surrounding statements. This latter reason often allows the optimizer, even at OPT(1), to discover and exploit synergies and to eliminate redundancies in order to improve performance.

There are other reasons as well why the executable size may be larger in some cases compared to V4:

- Use of higher ARCH instructions that are usually 6 bytes versus 4 bytes for many lower arch instructions. For example:
  - Using more than one ARCH(8) move immediate instruction instead of one in memory move
  - Exploiting Decimal Floating Point for packed/zoned decimal arithmetic
- Various V5 optimizations over and above V4 results in more generated code but shorter path length and better performance. For example:
  - More advanced INSPECT inlining
  - Conditionally inlining some complex conversions
  - Conditionally correcting decimal precision for binary data
- V4 used "base locator" pointers accessed by 4 byte load, but in V5, fewer base locators are used, but 6 byte long displacement instructions are used instead
- V5 has a higher unroll threshold than V4 when deciding to use multiple MVCs for large copies to avoid a more expensive MVCL instruction

Impact of TEST suboptions on program object size

As detailed in the Compiler Options section of the **COB PG**, the TEST and NOTEST options have several new suboptions in V5. The SOURCE/NOSOURCE and DWARF/NODWARF suboptions directly control if extra information used for debugging should or should not be included in the program object, and therefore can change the size of the object significantly.

The TEST option by itself and the suboption EJPD will also affect the program object size, but making it either greater or lesser, as these options may change the amount and types of optimizations performed by the compiler (so the resulting code and literal data areas may be smaller or larger).

Note that although the added debugging information will affect the size of the resulting program object, this will not affect the LOADed size.

Since the debugging information is in NOLOAD class segments, these parts of the program object are not loaded when the program is run, unless Debug Tool or Fault Analyzer or CEEDUMP processing explicitly requests it. So, unlike COBOL V4 and earlier versions, the size of the program object related to debugging information does not affect LOAD times or execution performance.

Since each suboption will impact the object size in a different way, let's examine each suboption separately. Results will be shown for OPTIMIZE(0) and OPTIMIZE(1) for each case, and all other options are kept at their default settings.

The size comparisons were gathered from a large selection of COBOL tests in our performance verification suite of applications.

First, let's compare the NOTEST suboption DWARF/NODWARF. The DWARF setting will cause basic DWARF diagnostic information to be included in the object.

Table 10. NOTEST(DWARF) % size increase over NOTEST(NODWARF)

Average Size	**NOTEST(DWARF) % size increase over NOTEST(NODWARF)**
OPTIMIZE(0)	96.1%
OPTIMIZE(1)	105.7%

So at both OPTIMIZE settings measured the overall object size roughly doubles when specifying DWARF overall the default NODWARF setting.

For TEST, let's first look at the option by itself versus NOTEST. The differences in object size in this case are due to a few reasons:

- The first major reason for the size increase is that TEST always causes full DWARF debugging information to be included in the object
- The second major reason for the size increase is that TEST by default enables the SOURCE suboption, so the generated DWARF debug information includes the expanded source code
- The third reason, and this generally matters less than the previous two reasons, is that TEST slightly inhibits optimization, and this may result in object size increases or decreases depending on the characteristics of the program

Table 11. TEST % size increase over NOTEST

Average Size	TEST % size increase over NOTEST
OPTIMIZE(0)	216.7%
OPTIMIZE(1)	237.8%

Next, let's look at the impact of the SOURCE/NOSOURCE suboption. This increase is directly related to the size of your expanded source file as it will be included in the DWARF debug information when TEST(SOURCE) is specified.

Despite the object size increase it causes, the advantage of specifying SOURCE is that since the DWARF information will contain the expanded source, a separate compiler listing will not be required by IBM Debug Tool.

Table 12. TEST(SOURCE) % size increase over TEST(NOSOURCE)

Average Size	TEST(SOURCE) % size increase over TEST(NOSOURCE)
OPTIMIZE(0)	27.5%
OPTIMIZE(1)	27.0%

Finally, let's look at the impact to object size from toggling the TEST suboption EJPD/NOEJPD. This option does not change the amount or type of DWARF debug information included in the object, but only impacts the amount and types of optimizations performed by the compiler in order to meet the debugging requirements of EJPD.

Table 13. TEST(EJPD) % size increase over TEST(NOEJPD)

Average Size	TEST(EJPD) % size increase over TEST(NOEJPD)
OPTIMIZE(0)	0%
OPTIMIZE(1)	4.0%

The 0% change at OPTIMIZE(0) makes sense, as this lowest level of optimization is already low enough to be not restricted by the extra debugging requirements of EJPD.

At OPTIMIZE(1), the more restrictive EJPD setting generally inhibits optimizations that would have resulted in smaller, and likely faster performing, executable code.

Why does COBOL V5 use PDSEs for executables?

As detailed in the **COB MG**, section "Changes in compiling with Enterprise COBOL Version 5.1", COBOL V5 executables must reside in a PDSE and can no longer be in a PDS.

This section describes some of the rationale for this change in behavior.

First, here is background information regarding PDS. When using PDS, customers reported problems in several areas:

- The need for frequent compressions
- Loss of data due to the directory being overwritten

- Performance impact due to a sequential directory search
- Performance delay if member added to beginning of directory
- When PDS went into multiple extents

In addition, PDS data sets cannot share update access to members without an enqueue on the entire dataset. More seriously though, a PDS library has to be taken down in order to perform compression to reclaim member space, or for a directory reallocation to reclaim wasted spaced (also known as gas).

Both of these can cause application downtime in production systems and are therefore very undesirable.

PDSEs, which were introduced in 1990, were designed to eliminate or at least reduce these problems and for the most part they have been successful. The initial rollout of PDSEs was rocky, and due to these problems long ago, many sites continue to avoid PDSEs to this day.

On the other hand, many other sites have moved their COBOL load libraries to PDSEs, and the process to do is fairly mechanical. For example:

- Allocate new PDSE datasets with new names
- Copy Load Modules into PDSEs - these are converted to Program Objects
- Rename PDSs, then rename PDSEs

In fact, Enterprise COBOL has required program objects, therefore, PDSE for executables since 2001 for features such as long program names, object-oriented programs and for DLLs using the binder instead of the prelinker.

Only PDSEs (and z/OS USS files) can contain program objects, and this allows program management binder to solve some long standing existing problems using these program object features.

For example, once the 16M text size limit of load modules was hit, the only solution was an expensive redesign or refactoring of the program in order to make it smaller. With program objects the text size limit is increased to 1G.

This extra space also allows the COBOL compiler to perform more advanced optimizations that may increase program literal area, and ultimately object, size (with the goal of course of improving runtime performance) such as those described in “Why you might see larger program object sizes when using V5?” on page 61. There are other advantages as well for COBOL using program objects:

- QY-con requires program objects
- Condition-sequential RLD support requires program objects (leading to a performance improvement for bootstrap invocation)
- Program objects can get page mapped 4K at a time for better performance
- Common reentrancy model with C/C++ requires program objects
- Looking into potential for the future XPLINK requires program objects and will be used for AMODE 64

A related issue is the different sharing rules across a SYSPLEX system. Unlike PDS libraries, PDSE data sets cannot be shared across SYSPLEX systems. Therefore, if existing pre-V5 PDS based COBOL load libraries are being shared, then V5 PDSE based load libraries can be moved using the following process:

- One SYSPLEX can be the writer/owner of master PDSE load library (development SYSPLEX)
- When PDSE load library is updated, push the new copy out to production SYSPLEX systems with XMIT or FTP
- The other SYSPLEX systems would then RECEIVE the updated PDSE load library

Notices

This information was developed for products and services offered in the U.S.A.

IBM might not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information in softcopy, the photographs and color illustrations might not appear.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.html.

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, 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

Permission is granted to distribute this paper to IBM customers. IBM retains all other rights to this paper, including the right for IBM to distribute this paper to others.

Printed in USA