Performance Tuning with Enterprise COBOL

Mike Chase
mike.chase@ca.ibm.com

Compiler Optimization Developer
Lead Developer, IBM watsonx Code Assistant for Z
Code Optimization Advice

Contents

- Introduction
- Migration Strategies
- Compiler Options
- Runtime/LE Options
- Coding Practices

Introduction

Goals of this presentation

Practical advice

This presentation focuses on the most important things you can do as a COBOL developer to improve the performance of COBOL applications and discusses the advantages and drawbacks of those changes.

Your time is valuable. Time you spend on performance is time you can't spend on something else. This presentation will help you get the best bang for your buck.

Migration

This presentation focuses on improving performance for programs compiled with Enterprise COBOL 6.

Much of the advice also applies to earlier versions of COBOL.

For details on migrating from earlier versions of COBOL, please see the Enterprise COBOL 6 Migration Webinar.

Comparisons

Overall performance comparisons use the COBOL 6.5 GA on an IBM z17 machine.

Comparisons for some options use older Enterprise COBOL 6 GAs on older hardware.

Unless otherwise specified, performance comparisons are based on IBM's internal benchmark suite, and represent the geometric mean of the performance improvement.

Caveats

The ability of the compiler to improve the performance of a program depends on many factors. IBM cannot guarantee performance improvements on every program.

This presentation aims to be clear about the conditions and limitations of the performance improvements it describes.

Introduction
A new approach to COBOL compilation

In the past ...

Computers were slow.

Memory was expensive and was measured in KB.

Compiler optimizations were constrained by space and time.

Programs are compiled once

Now ...

Computers are fast.

Memory is fast and is measured in GB.

and run many times.

Introduction
A new approach to COBOL compilation

Prior to Enterprise COBOL 5...

The compiler was designed to run quickly.

The compiler traded optimization quality for compilation speed.

The compiler was an independent project, with no connection to the rest of IBM's compiler technology.

Now ...

The compiler is designed to make compiled programs run quickly.

The compiler trades compilation speed for optimization quality.

The compiler shares code with IBM's enterprise-grade compiler for Java.

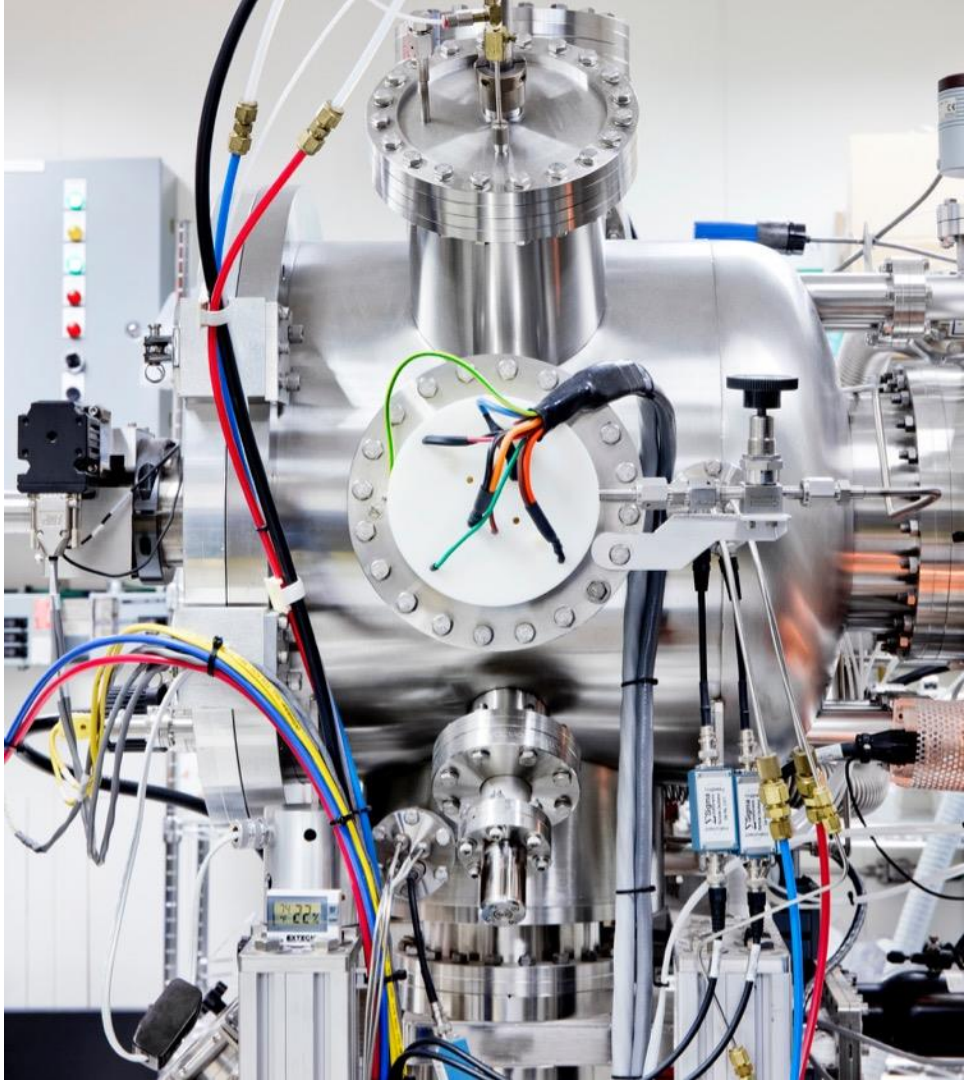
Advancements

New optimizations

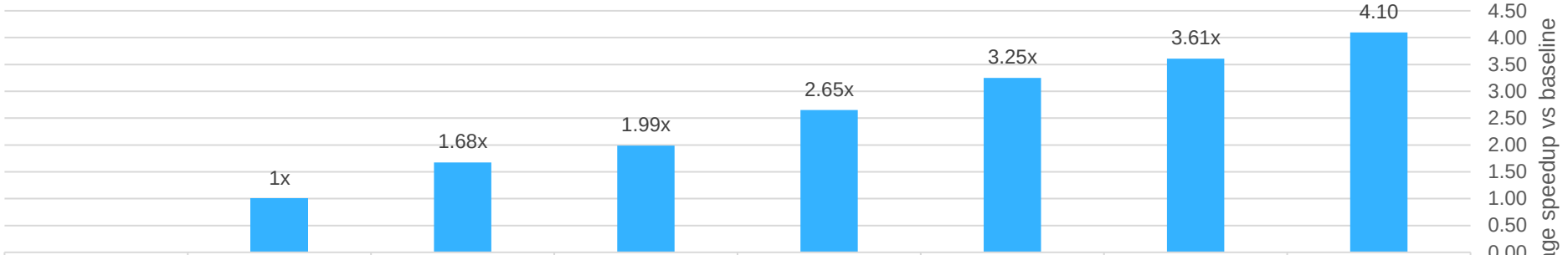
The new compiler implements many new optimizations, including global optimizations that look at the entire program.

Hardware exploitation

The new compiler takes full advantage of the new instructions on the latest IBM Z mainframes.



History of Enterprise COBOL for z/OS Performance
for compute intensive and I/O bound COBOL applications



Hardware	IBM zEC12	IBM z13	IBM z13	IBM z14	IBM z15	IBM z16	IBM z17
Compiler	COBOL 4.2	COBOL 5.2	COBOL 6.1	COBOL 6.2	COBOL 6.3	COBOL 6.4	COBOL 6.5
GA year*	2009	2015	2016	2017	2019	2022	2025
Performance options**	OPT(FULL)	ARCH(11) OPT(2)	ARCH(11) OPT(2)	ARCH(12) OPT(2)	ARCH(13) OPT(2)	ARCH(14) OPT(2)	ARCH(15) OPT(2)
Average CPU usage reduction vs baseline	-	40%	50%	62%	69%	72%	76%

4.10x speedup from hardware and compiler improvements

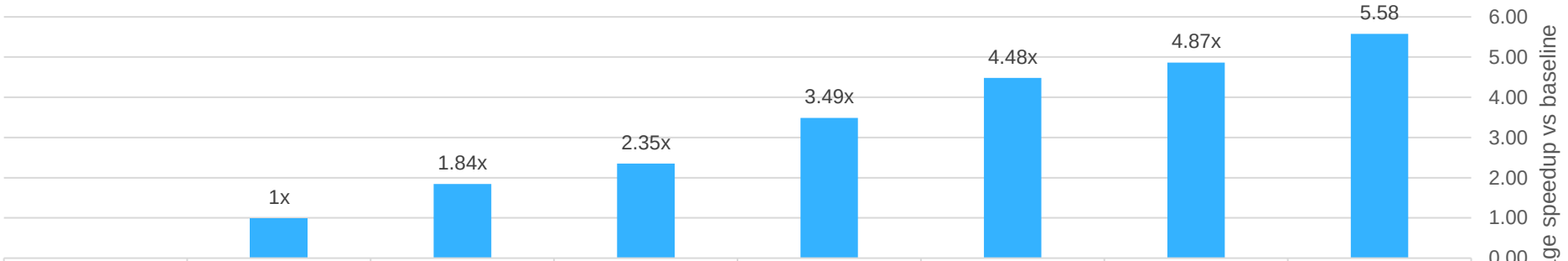
Enterprise COBOL 4.2 on zEC12 → Enterprise COBOL 6.5 on z17

based on internal benchmarks for compute intensive and I/O bound COBOL applications

* Compiler GA year. Compiler releases generally align with hardware releases.
** Recommended performance options for the respective hardware

Disclaimer: The performance improvements are based on the geometric mean of IBM internal measurements on IBM z17 running a z/OS 3.1 LPAR with 1 CP and 80GB Central Storage, IBM z16 running a z/OS 2.4 LPAR with 1 CP and 80GB Central Storage, IBM z15 running a z/OS 2.4 LPAR with 1 CP and 80GB Central Storage, IBM z14 running a z/OS 2.3 LPAR with 2 CP and 128GB Central Storage, IBM z13 running a z/OS 2.3 LPAR with 1 CP and 64GB Central Storage, IBM zEC12 running a z/OS 2.2 LPAR with 2 CP and 64GB Central Storage. All benchmarks compiled with IBM Enterprise COBOL for z/OS 5/6 use the options STGOPT, AFP(NOVOLATILE), HGPR(NOPRESERVE), and LIST. All benchmarks compiled with IBM Enterprise COBOL for z/OS 4.2 use the option LIB. Performance results for customer applications will vary, depending on the source code, the compiler options specified, and other factors.

History of Enterprise COBOL for z/OS Performance
for decimal and floating point compute intensive COBOL applications



Hardware	IBM zEC12	IBM z13	IBM z13	IBM z14	IBM z15	IBM z16	IBM z17
Compiler	COBOL 4.2	COBOL 5.2	COBOL 6.1	COBOL 6.2	COBOL 6.3	COBOL 6.4	COBOL 6.5
GA year*	2009	2015	2016	2017	2019	2022	2025
Performance options**	OPT(FULL)	ARCH(11) OPT(2)	ARCH(11) OPT(2)	ARCH(12) OPT(2)	ARCH(13) OPT(2)	ARCH(14) OPT(2)	ARCH(15) OPT(2)
Average CPU usage reduction vs baseline	-	46%	58%	71%	78%	79%	82%

5.58x speedup from hardware and compiler improvements

Enterprise COBOL 4.2 on zEC12 → Enterprise COBOL 6.5 on z17

based on internal benchmarks for decimal and floating point compute intensive COBOL applications

* Compiler GA year. Compiler releases generally align with hardware releases.
** Recommended performance options for the respective hardware

Disclaimer: The performance improvements are based on the geometric mean of IBM internal measurements on IBM z17 running a z/OS 3.1 LPAR with 1 CP and 80GB Central Storage, IBM z16 running a z/OS 2.4 LPAR with 1 CP and 80GB Central Storage, IBM z15 running a z/OS 2.4 LPAR with 1 CP and 80GB Central Storage, IBM z14 running a z/OS 2.3 LPAR with 2 CP and 128GB Central Storage, IBM z13 running a z/OS 2.3 LPAR with 1 CP and 64GB Central Storage, IBM zEC12 running a z/OS 2.2 LPAR with 2 CP and 64GB Central Storage. All benchmarks compiled with IBM Enterprise COBOL for z/OS 5/6 use the options STGOPT, AFP(NOVOLATILE), HGPR(NOPRESERVE), and LIST. All benchmarks compiled with IBM Enterprise COBOL for z/OS 4.2 use the option LIB. Performance results for customer applications will vary, depending on the source code, the compiler options specified, and other factors.

Migration strategies

Migration strategies

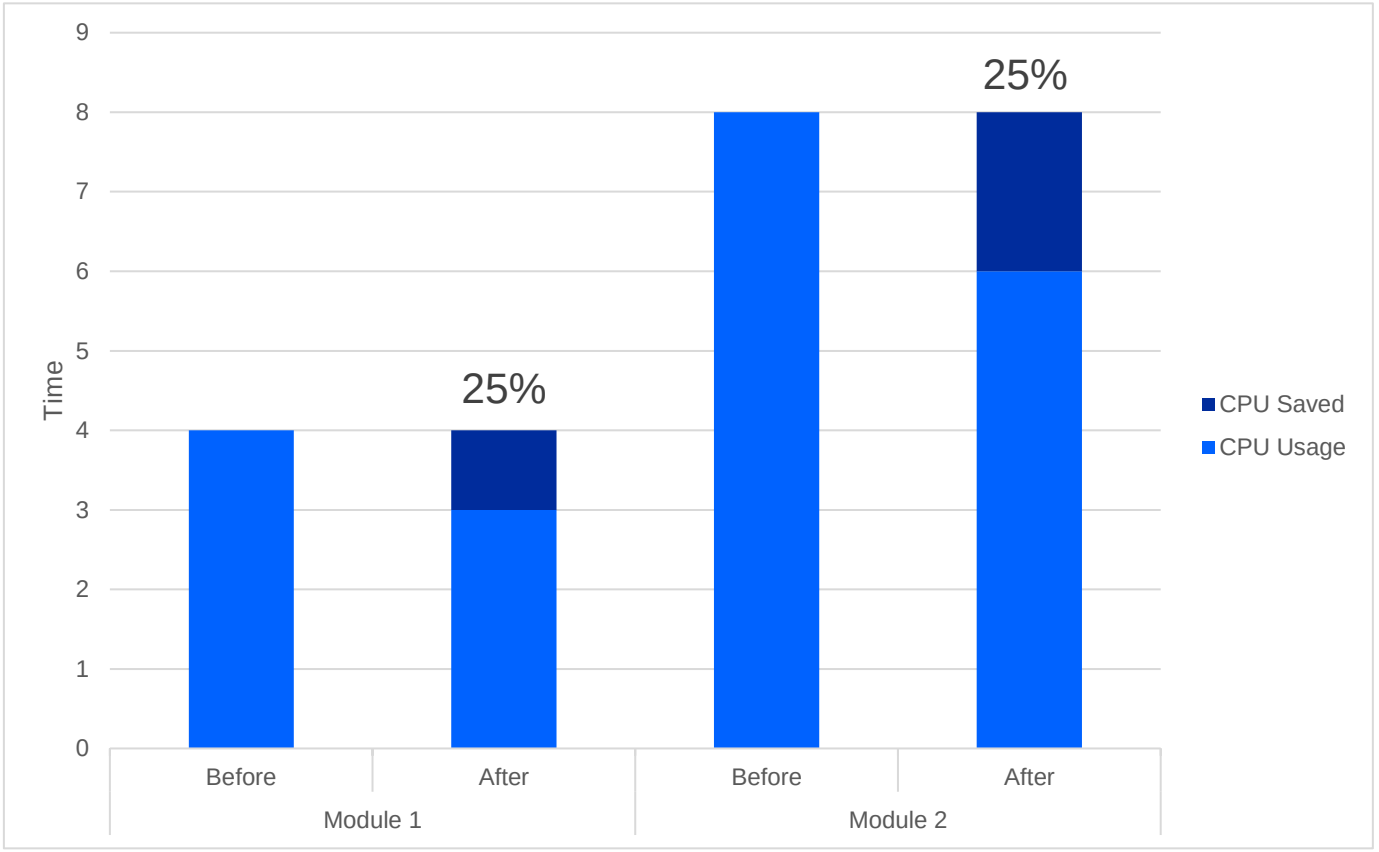
Two simple principles …

1. The more time you spend doing something, the more you gain from speeding it up.
2. A compiler can only speed up code that it compiles.

Prioritizing Migration

The more time you spend doing something, the more you gain from speeding it up.

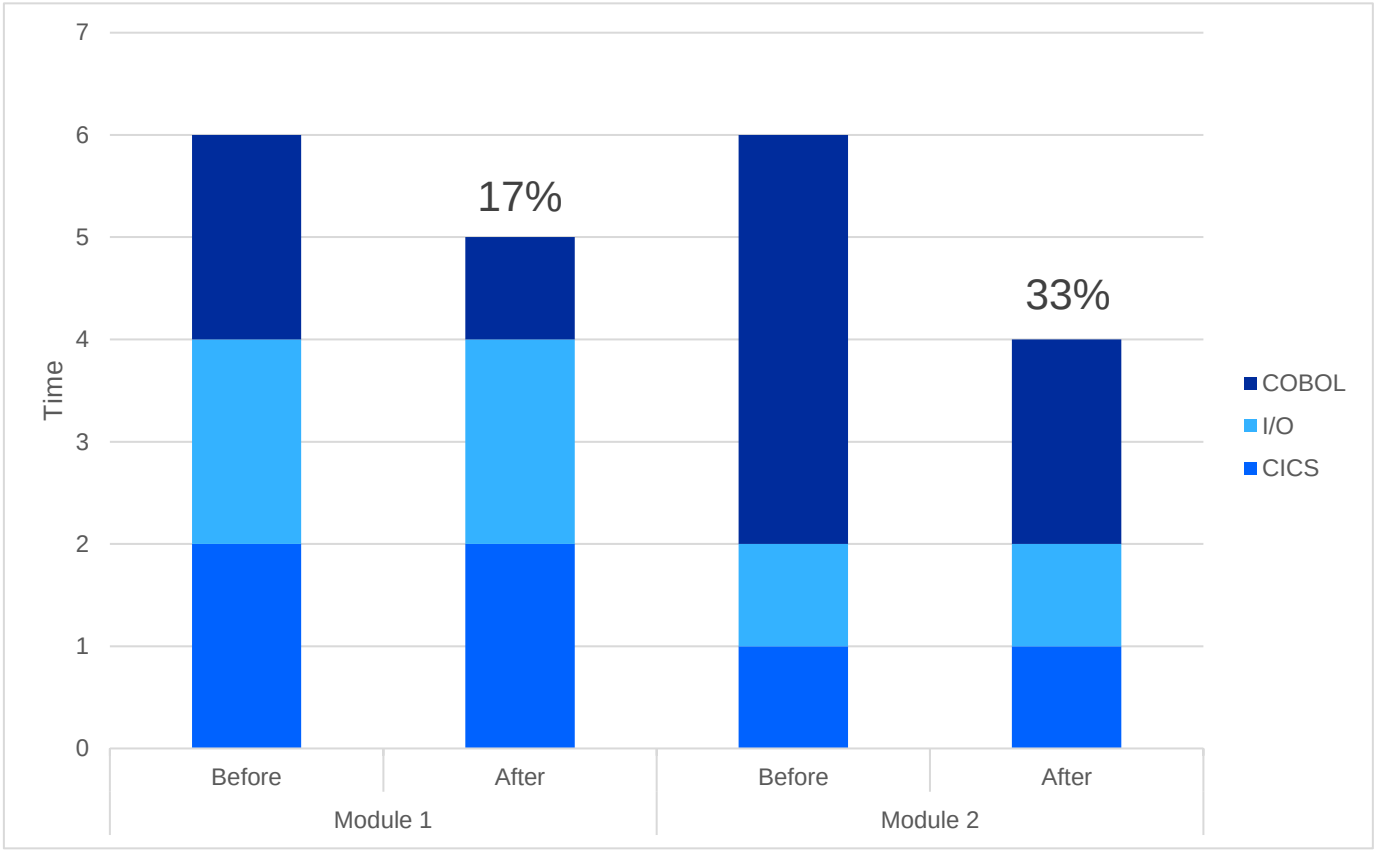
- A 25% speedup saves twice as much time on a program that takes twice as long.
- Prioritize the hot spots in your workload.



Prioritizing Migration

A compiler can only speed up the code that it compiles.

- The compiler does not speed up file I/O.
- The compiler does not speed up middleware: Db2, CICS, etc…
- Modules that spend more time executing COBOL code will see the largest benefit.



Migration Strategies
Tips and tricks

Start with batch

Batch programs often spend more time in COBOL code.

Transactional programs are often just a thin layer between other components.

Profile!

The best way to find your hotspots is by measuring them.

IBM Application Performance Analyzer for z/OS is a good tool for this; IBM and other vendors offer other tools as well

More information is available in a whitepaper: COBOL Applications: Techniques to Make Them More Efficient

Be methodical

Take a baseline before you start making changes, and make sure to compare apples to apples.

Small helper routines that are called from many different places are often an easy place to make an improvement.

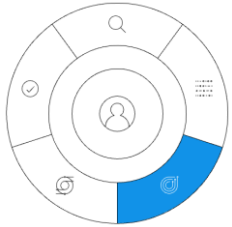
Give us feedback!

We can't measure your code.

Let us know what kind of improvements you are seeing.

If you are not seeing improvements, and you have profile data showing that your COBOL code is a performance hotspot, **we are very interested in hearing from you.**

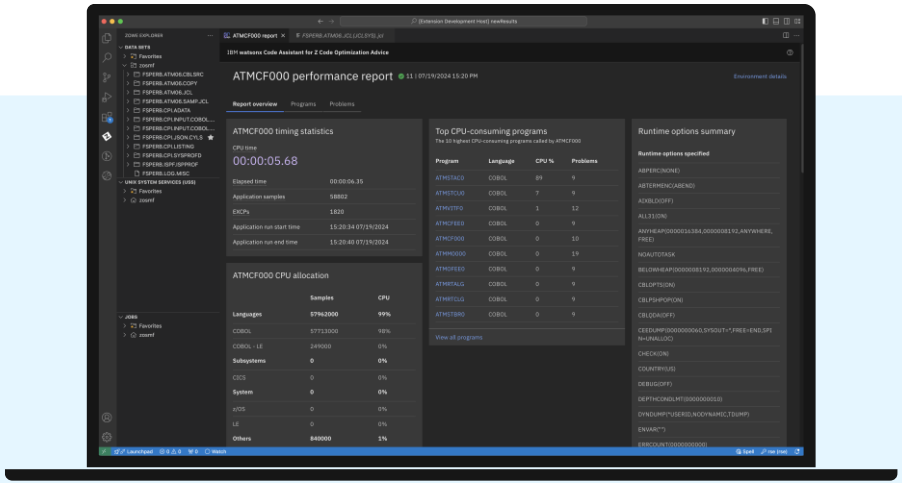
Code Optimization Advice: Improve your COBOL code performance with prioritized insights



Provides performance insight
Conducts comprehensive analysis of COBOL modules using both static and dynamic analysis, enabling quick identification and resolution of performance issues.

Provides prioritized and actionable recommendations
Ranks performance issues based on impact, enabling developers to focus on high-priority tasks for maximum efficiency.

Identifies performance issues in source
Access source files and copybooks at the exact line of code to apply recommended fixes and enhancements



Key Values

Boost productivity by enabling quick identification and resolution of performance issues

Reduce skill gap by allowing developers of all skill levels to independently resolve performance issues

Quick resolution of performance issues in COBOL source

Shift-left strategy to proactively prevent performance problems before they reach production

Migration Strategies
What if I don’t want to recompile/migrate to COBOL 6?

Use ABO

Automatic Binary Optimizer optimizes previously-compiled COBOL programs without the source code

Latest version is ABO 2.3, which targets hardware up to the IBM z17

ABO can help you see performance gains faster than recompiling

Which takes less effort?

ABO-optimized modules have the same behavior, even when programs use invalid data

Less testing is needed than when recompiling older programs with COBOL 6

ABO requires no source code changes and has very few options; besides ARCH, the existing compiler options are left alone

When Should I Use ABO?

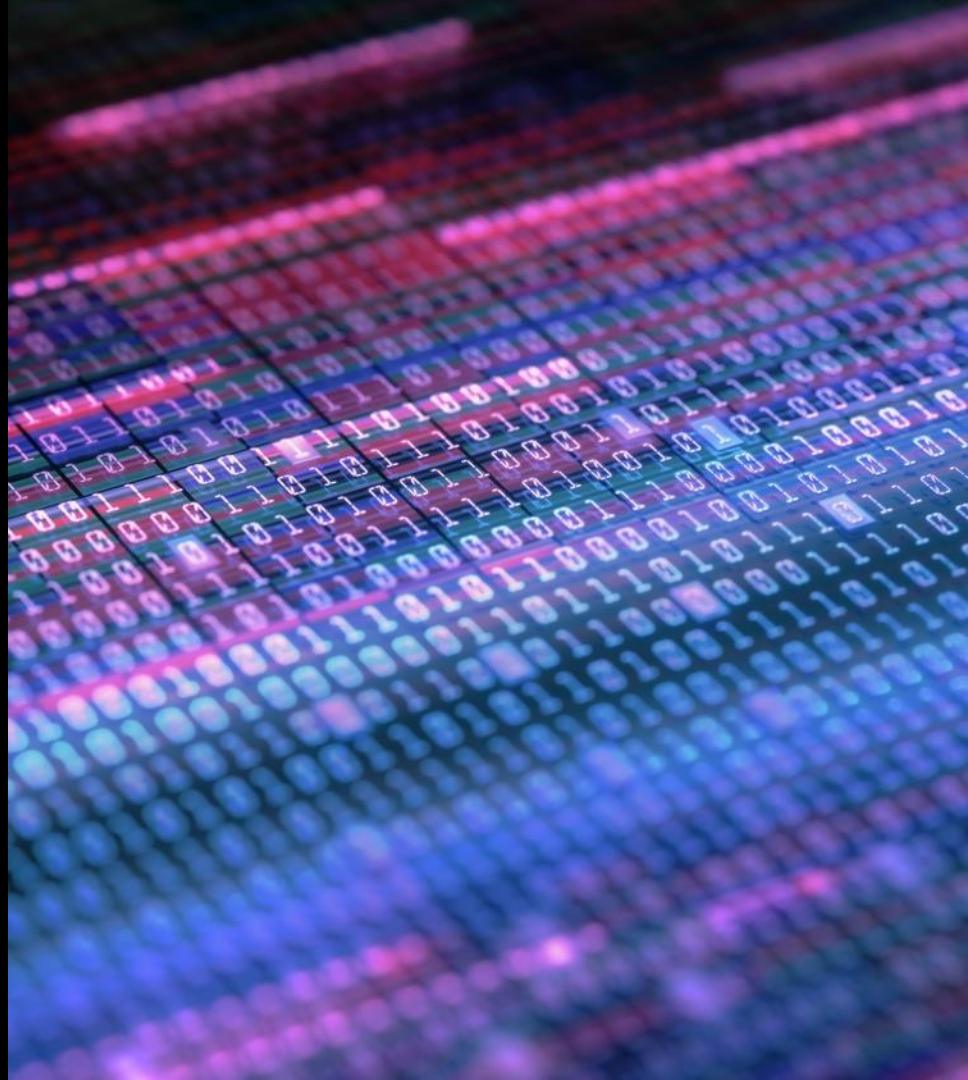
ABO is a faster, easier, alternative to recompiling with COBOL 6 and will give comparable performance gains.

Programs optimized with ABO have comparable performance to programs recompiled with COBOL 6, using OPT(2) and the same ARCH option as ABO and keeping all other compiler options the same as when the program was previously compiled.

To get an extra performance boost, recompile with COBOL 6 and follow the performance tuning advice in this webinar

Performance results for customer applications will vary, depending on the source code, the compiler options specified, and other factors. Find the full disclaimer here.

Compiler options:
Low hanging fruit



Compiler options
OPT

Best Performance

OPT(2)

OPT(0)

OPT(0) uses a minimal set of optimizations.

OPT(0) is not recommended for performance-critical applications.

OPT(0) guarantees that instructions from separate statements will not be interspersed: each statement is completely finished before starting the next statement.

OPT(1)

OPT(1) uses a subset of the compiler's optimizations. It omits some optimizations and runs weaker versions of others.

OPT(1) lies between the compilation speed of OPT(0) and the performance of OPT(2).

OPT(2)

OPT(2) uses all of the compiler's most powerful optimizations.

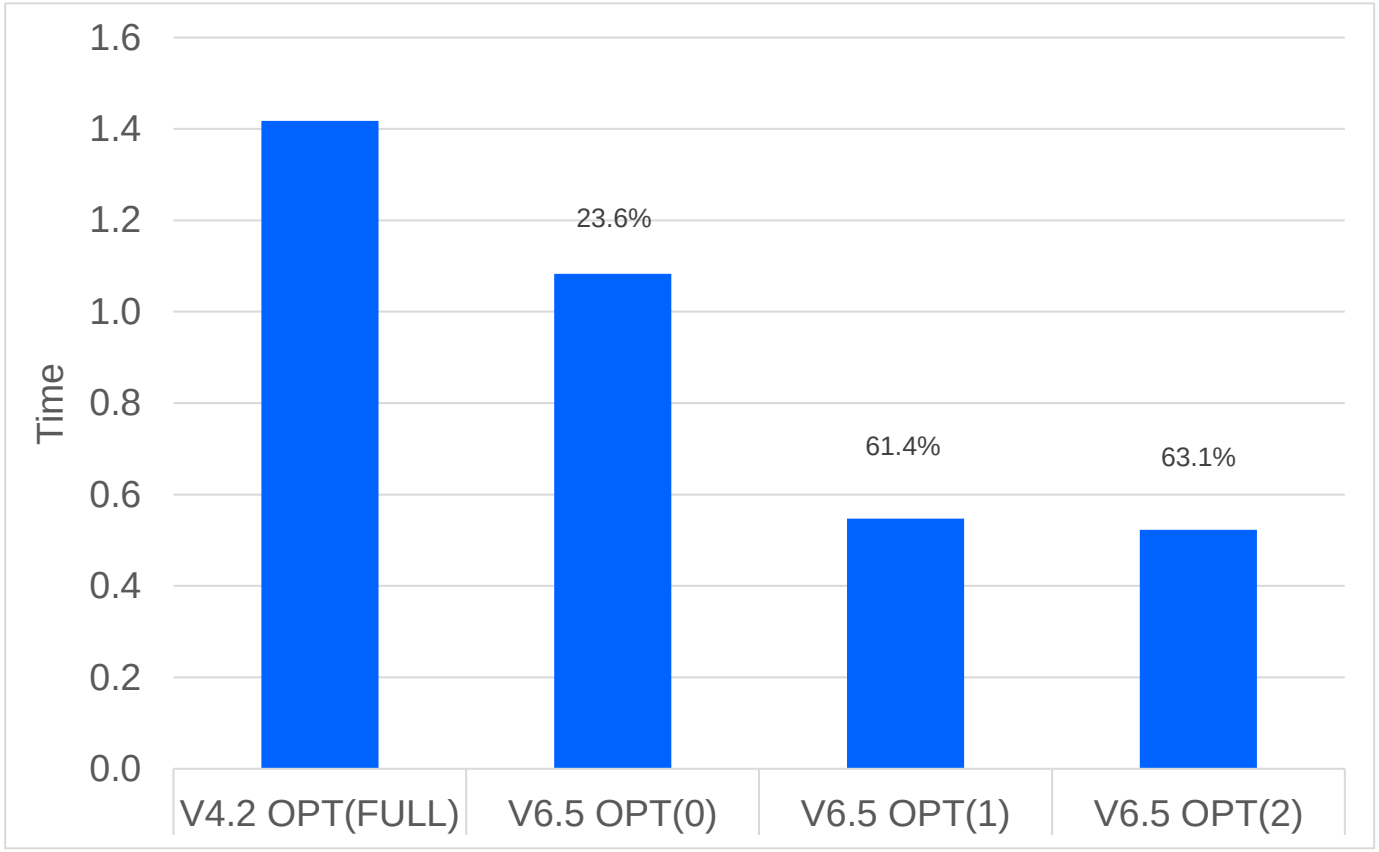
OPT(2) is the best choice for performance-critical applications.

Compiler options
OPT

Performance comparison

This comparison is across IBM's internal suite of benchmarks.

A program in our benchmark suite was **84%** faster with OPT(1) over OPT(0) and an additional **49%** faster with OPT(2) over OPT(1) (for a **92%** improvement with OPT(2) over OPT(0)).



Compiler options
ARCH

Best performance

As high as possible.

Select the ARCH level corresponding to the **lowest** level machine on which your program must run.

For example: a customer using IBM z17 machines in production and IBM z16 machines for disaster recovery should use ARCH(14), corresponding to IBM z16.

Hardware exploitation

Every new release of IBM Z mainframes adds new instructions to the hardware.

ARCH limits the compiler's instruction choices to those that exist on a target machine.

By setting the correct ARCH level, you take full advantage of the capabilities of your mainframe.

Free lunch

Increasing the ARCH level does not affect compilation time.

Increasing the ARCH level does not affect the ability to debug your programs.

Caveat: exploitation of decimal floating-point instructions only occurs at OPT(1) and above.

Biggest wins

ARCH(12): The addition of the vector packed decimal facility in IBM z14 provides an additional boost to decimal arithmetic.

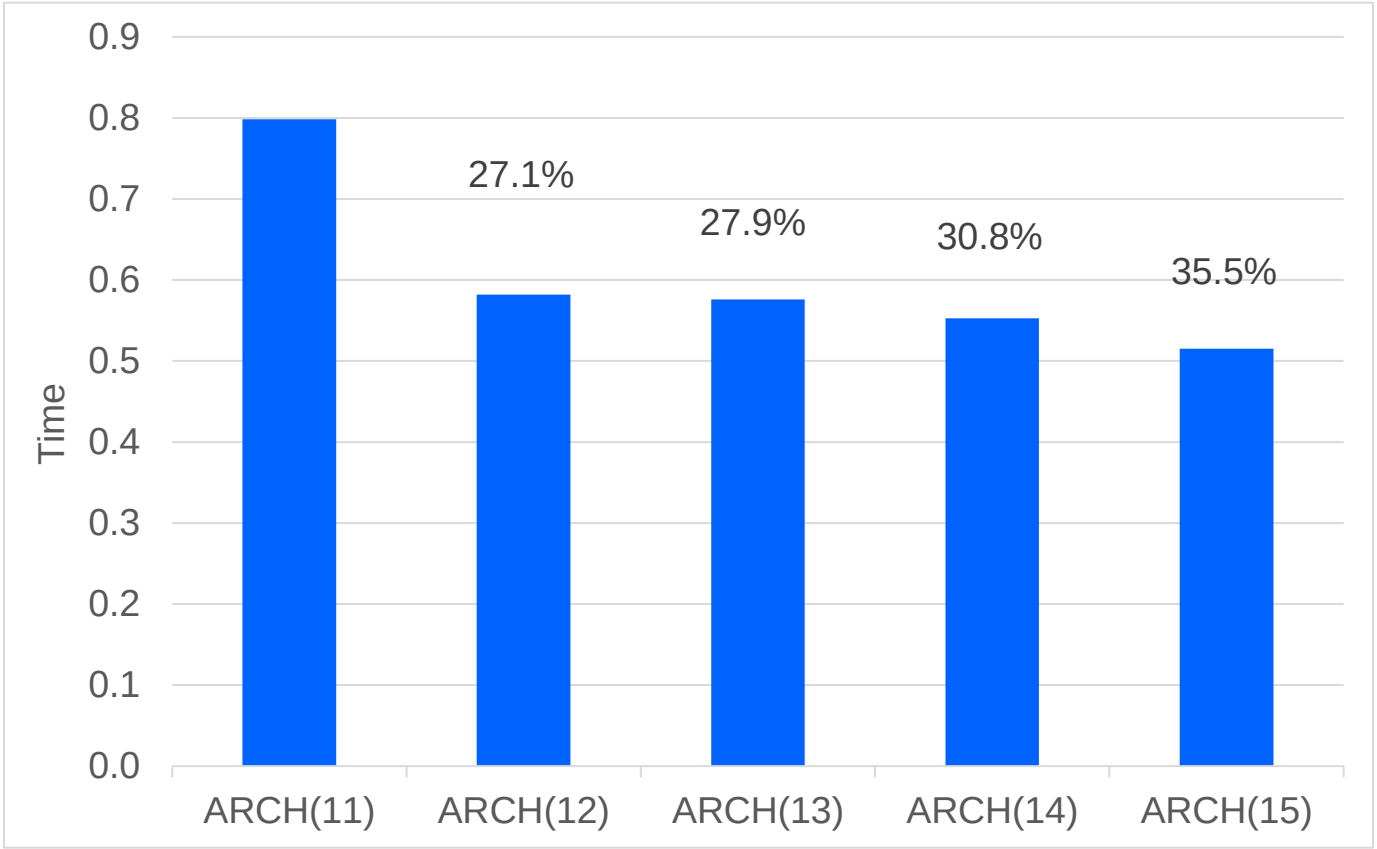
ARCH(10): The new decimal floating point instructions introduced for IBM zEC12 significantly improve the performance of decimal arithmetic.

Compiler options
ARCH

Performance comparison

This comparison is across IBM's internal suite of benchmarks, compiled with COBOL 6.5 at OPT(2), run on an IBM z17. Comparison is relative to ARCH(11).

An individual program in our benchmark suite improved by **84%** using ARCH(15) compared to ARCH(11)



Compiler options
TUNE

Best performance

Select the TUNE level corresponding to the machine on which your program run **most often**.

For example: a customer using IBM z17 machines in production and IBM z16 machines for disaster recovery must use ARCH(14), corresponding to IBM z16, but should also use TUNE(15), corresponding to IBM z17, for best performance in production.

Hardware exploitation

TUNE instructs the compiler to make the best performance choices on a target machine, limited by the available instructions.

By setting the correct TUNE level, you take full advantage of the capabilities of the mainframe where your application runs most often.

TUNE must always be at least as high as ARCH.

With ARCH set to match older hardware, the average gain of matching TUNE to newer production hardware is 1-2% over matching TUNE to ARCH and the older hardware

Free lunch

Increasing the TUNE level does not affect compilation time.

Increasing the TUNE level does not affect the ability to debug your programs.

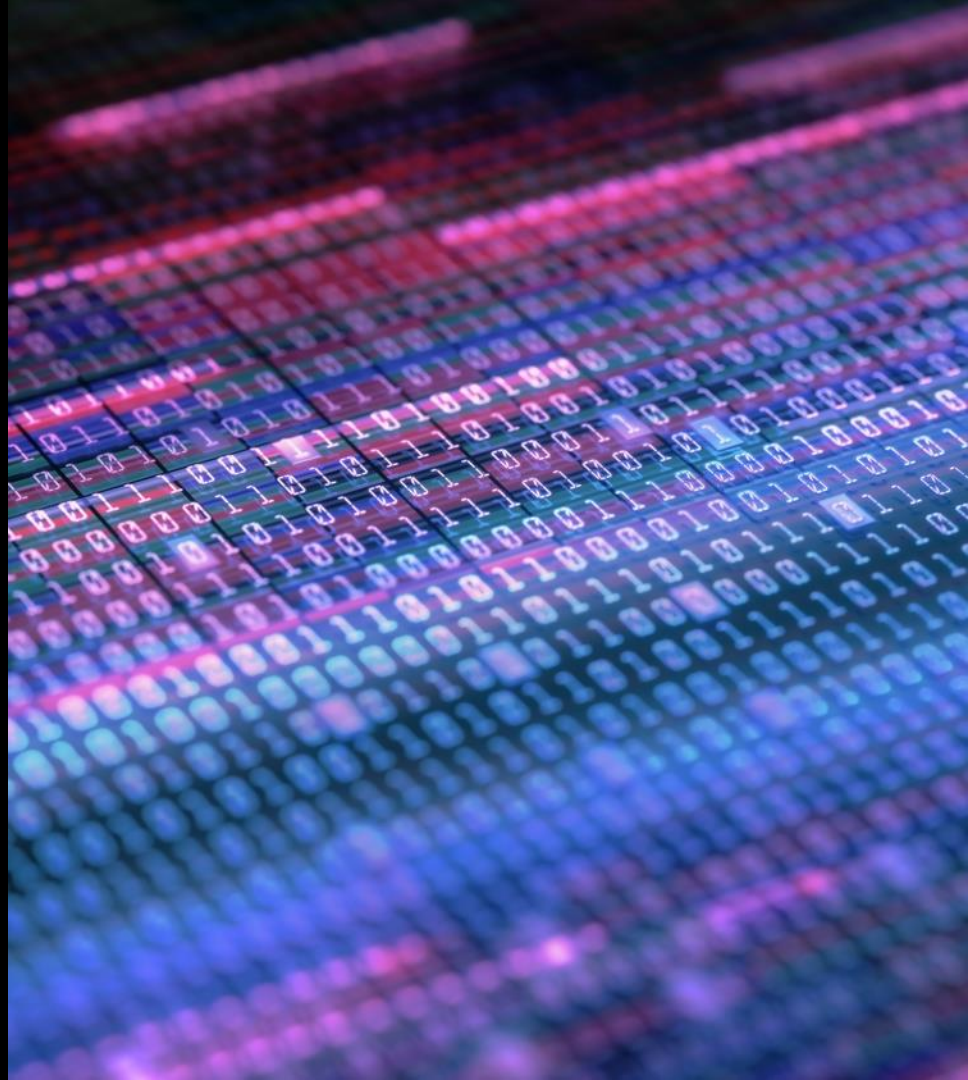
Biggest wins

On IBM z14, the compiler sometimes inserts stores from vector packed registers to temporary storage for performance. This is only useful on an IBM z14, and the extra instructions hurt performance on other hardware.

ARCH(12),TUNE(13) allows applications to safely run on an IBM z14 but get better production, avoiding the stores to temps, on an IBM z15 or z16.

A program in our benchmark suite was **22%** faster with ARCH(12),TUNE(13) than ARCH(12),TUNE(12) on an IBM z15

Compiler options:
Some assembly required



Compiler options
INVDATA

Best performance

NOINVDATA

If necessary, test for bad data with NUMCHECK.

What is bad data?

Not all arrangements of bits are valid decimal values. Consider the number -123. There are multiple ways to represent it. The programmer selects the representation of a data item using the PICTURE clause.

Each of these representations has constraints on what the data can look like.

Zoned decimal

Ex: PIC S9(3)

F1 F2 D3

This number has **three digits**. They must be between 0 and 9.

It also has a **sign code**. It must be between A and F. In this case, D means the number is negative.

It also has **zone bits**. These must always be F.

Biggest wins

Ex: PIC S9(3) COMP-3

12 3D

This number also has **three digits** and a **sign code**. However, it does not have any **zone bits**.

Compiler options
INVDATA

Why does it matter?

There are many ways to generate code for a statement.

They all do the same thing for valid data. They may behave differently for invalid data.

The new compiler may make different, faster choices than the old compiler.

INVDATA and INVDATA(FORCENUMCMP,NOCLEANSIGN) restrict the compiler's ability to make those faster choices.

Example:

```
==================
01 GRP.
  02 VAR PIC 9(3).

MOVE LOW-VALUES TO GRP.
IF VAR = 0 THEN
...
==================
```

Because VAR is unsigned, it should look like:

F0 F0 F0

If the **zone bits** and the **sign code** are all guaranteed to be F, then the compiler can compare VAR to 0 directly.

Because we moved LOW-VALUES to VAR, it looks like:

00 00 00

This is not bitwise-equal to 0. The compiler must first convert VAR to packed decimal and set the sign code to F before comparing

Best strategy

NOINVDATA will give the best performance.

If your application has problems with bad data, use NUMCHECK to locate and fix them.

Note: As we add new optimizations to the compiler, the performance gap between NOINVDATA and the other options will only continue to grow.

Compiler options
NUMPROC

Best performance

NUMPROC(PFD)

If necessary, test with NUMPROC(PFD) and NUMCHECK(ZON,PAC)

What does it do?

COBOL defines *preferred* sign codes for decimal values:

- C for positive numbers.
- D for negative numbers.
- F for unsigned numbers.

Sign codes of A, B, and E are valid, but *non-preferred.*

Arithmetic operations always produce the preferred sign.

If NUMPROC(PFD) is

specified, the compiler can assume that all data has the preferred sign.

Why does it matter?

The compiler can generate more efficient code.

For example, if C is the only positive sign code, values can be compared using a bitwise comparison.

If values might exist with a sign code of A, a more expensive comparison must be used.

Across a suite of compute-intensive benchmarks, NUMPROC(PFD) is **7.5%** faster than NOPFD.

Best strategy

NUMPROC(PFD) should only be used after verifying that your data conforms to the rules for preferred sign.

Compiler options
TRUNC

Best performance

TRUNC(STD) or TRUNC(OPT)

If necessary, test with TRUNC(OPT) and NUMCHECK(BIN).

Data items coming from Db2 or CICS may not conform to a PIC clause; use COMP-5 for those data items.

What does it do?

TRUNC determines the behaviour of BINARY data items that exceed their PICTURE clause.

TRUNC(STD): Truncate according to the PICTURE clause.

TRUNC(BIN): Truncate according to the underlying binary data type.

TRUNC(OPT): The compiler can assume that the PICTURE clause is never exceeded.

Performance?

TRUNC(BIN) is the slowest. Intermediate results grow quickly and may require larger data types.

TRUNC(STD) requires a fixup after each operation that might exceed the picture clause. These fixups are heavily optimized in the new compiler.

TRUNC(OPT) combines the benefits of both BIN and STD, but the programmer must avoid truncation.

Best strategy

Avoid TRUNC(BIN).

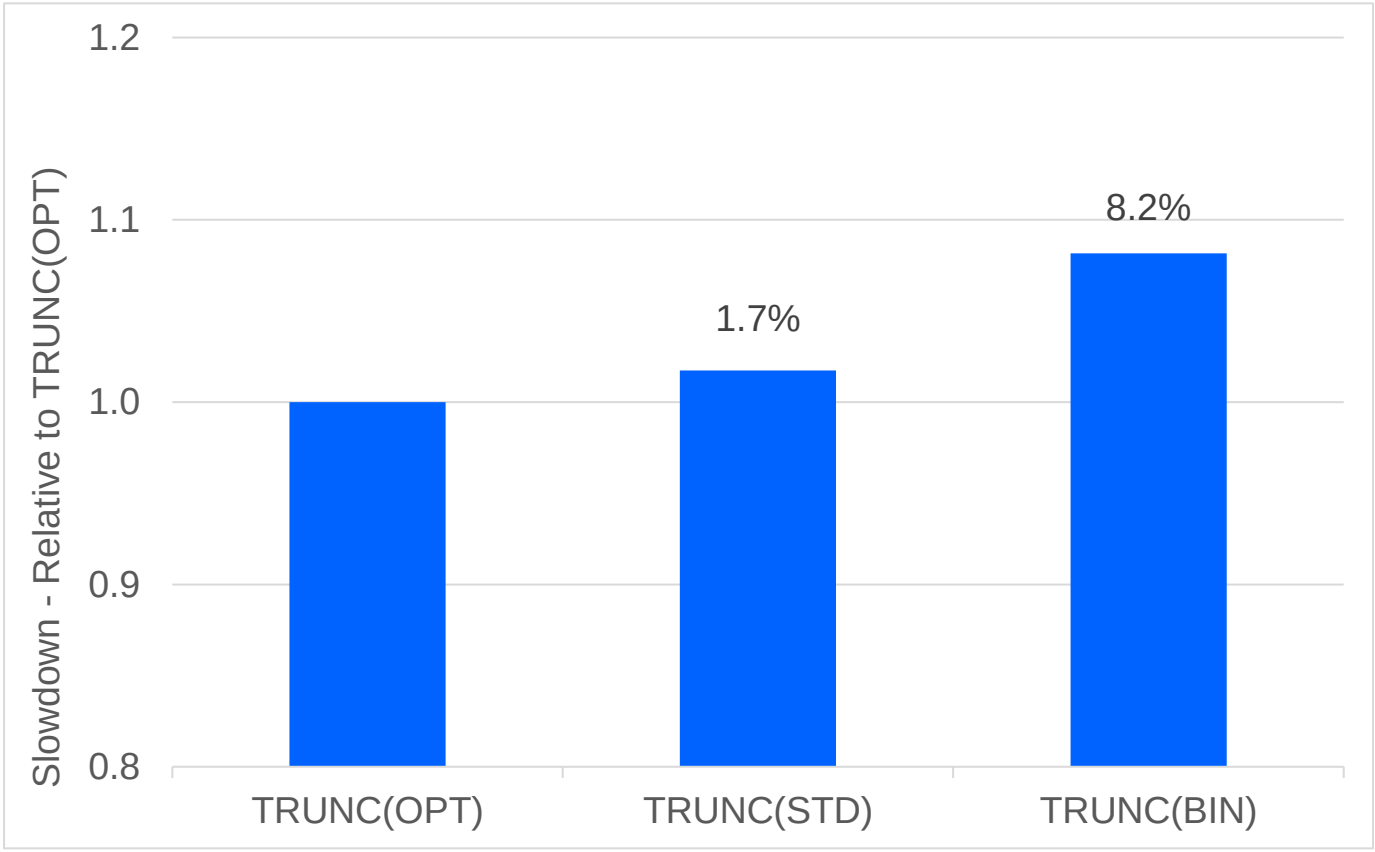
TRUNC(STD) is often good enough.

TRUNC(OPT) may be faster in some cases, but usually not enough to matter.

Compiler options

TRUNC

Performance comparison



Compiler options
STGOPT

Best performance

STGOPT

What does it do?

STGOPT allows the compiler to eliminate unreferenced data items in WORKING-STORAGE and LOCAL-STORAGE.

Prior to Enterprise COBOL 5, this was controlled using OPT(STD) vs OPT(FULL).

Benefits

Memory consumption: Copybooks often introduce unreferenced data items into programs. Using STGOPT can reduce the memory consumption of your application.

Data locality: Moving frequently used data items closer together may make it easier for the hardware to have your data available when the program needs it, which can improve performance.

Caveats

The reduction in memory consumption from STGOPT is more significant than the reduction in CPU usage.

Do not use STGOPT for programs containing unreferenced eyecatchers.

Do not use STGOPT for programs depending on unreferenced data items to lay out memory correctly.

The VOLATILE clause can be used to prevent the compiler from eliminating specific data items.

Compiler options
AWO

Best performance

AWO

What does it do?

The APPLY WRITE-ONLY clause optimizes buffer and device space allocation for QSAM files that have standard sequential organization, have variable-length records, and are blocked.

This allows the program to combine more records together into a single EXCP.

The AWO compiler option adds an implicit APPLY WRITE-ONLY clause for every physical sequential, variable-length, blocked file in the program.

Benefits

In one example program, AWO is **90%** faster than NOAWO and uses **98%** fewer EXCPs.

Compiler options
BLOCK0

Best performance

BLOCK0

What does it do?

BLOCK CONTAINS 0 can be specified for QSAM files.

This allows the operating system to determine the best size for the block.

The BLOCK0 compiler option adds an implicit BLOCK CONTAINS 0 clause for every QSAM file in the program that does not specify RECORDING MODE U and does not already have a BLOCK CONTAINS clause.

Benefits

Using the best block size can reduce the number of physical I/O transfers, resulting in fewer EXCPs.

In one example program, BLOCK0 is **90%** faster than NOBLOCK0 and uses **98%** fewer EXCPs.

Compiler options
FASTSRT

Best performance

FASTSRT

What does it do?

In SORT USING and SORT GIVING statements, FASTSRT allows the DFSORT product to perform I/O directly on the input and output files, instead of returning control to COBOL after each record is processed.

This eliminates significant overhead.

Benefits

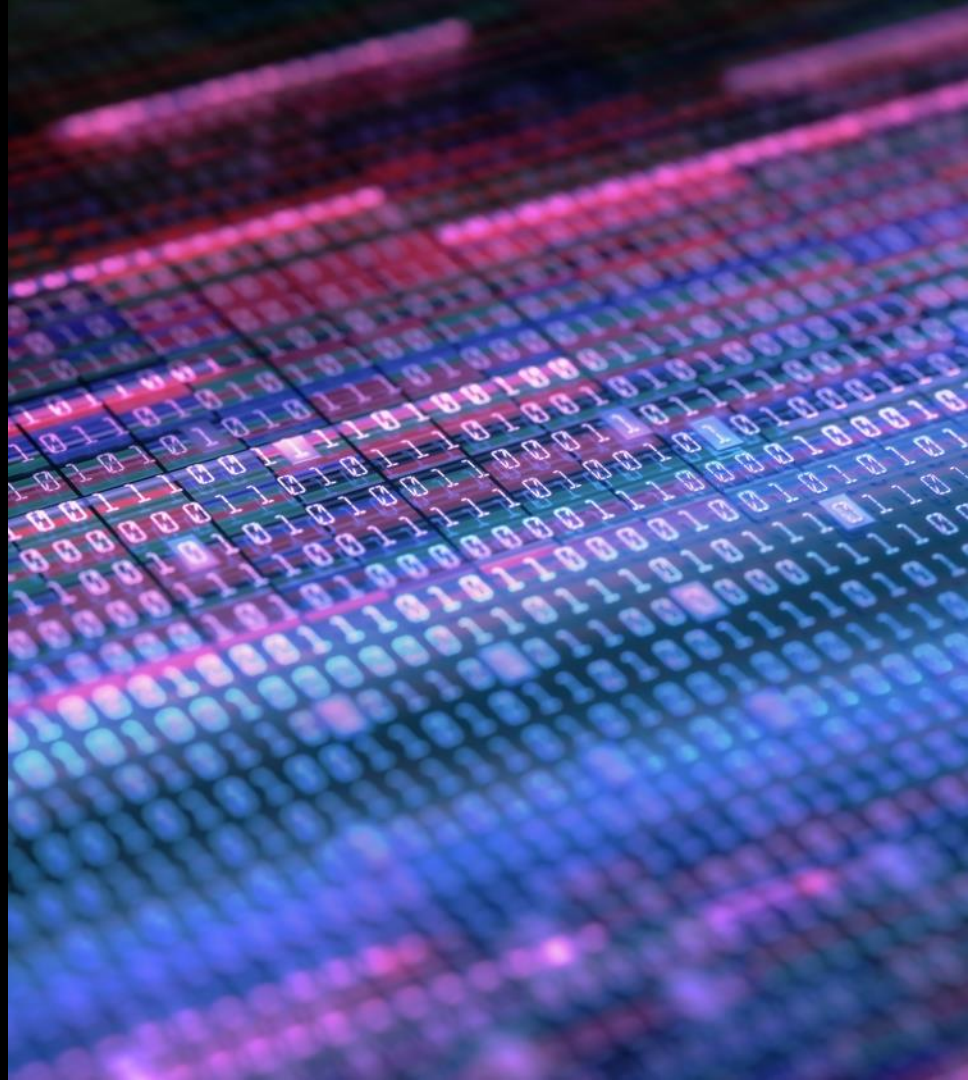
In an example program that processes 100,000 records, FASTSRT is **45%** faster than NOFASTSRT, and uses **4,000** fewer EXCPs.

Caveats

You cannot use the DFSORT options SORTIN or SORTOUT if you use FASTSRT. The FASTSRT compiler option does not apply to line-sequential files you use as USING or GIVING files.

If you specify file status and use FASTSRT, file status is ignored during the sort.

Compiler options:
Options to avoid



Compiler options
SSRANGE

Best performance

NOSSRANGE

What does it do?

SSRANGE inserts code that verifies that subscripts and indexes are within the proper range.

Performance

Across our benchmarks, SSRANGE is **7%** slower than NOSSRANGE.

Best strategy

Use SSRANGE during testing to find bugs. Fix the bugs, then use NOSSRANGE in production.

Compiler options
NUMCHECK

Best performance

NONUMCHECK

Note: The z17 hardware introduces instructions that greatly reduce the impact of NUMCHECK. Across our benchmarks, NUMCHECK was only **3.6%** slower than NONUMCHECK when using OPT(2),ARCH(15) and running on a z17.

But, check your minimum ARCH level!

What does it do?

NUMCHECK inserts code that verifies that numeric data items used as senders have valid data.

NUMCHECK(ZON) inserts tests for DISPLAY items.

NUMCHECK(PAC) inserts tests for COMP-3 items.

NUMCHECK(BIN) inserts tests for BINARY items.

Performance

Across our benchmarks:

NUMCHECK(ZON) is **15%** slower than NONUMCHECK.

NUMCHECK(PAC) is **10%** slower than NONUMCHECK.

NUMCHECK(BIN) is **13%** slower than NONUMCHECK.

Best strategy

Use NUMCHECK during testing.

Do not use NUMCHECK in production.

Compiler options
PARMCHECK

Best performance

NOPARMCHECK

What does it do?

PARMCHECK inserts a buffer immediately following WORKING-STORAGE.

Before a CALL, a bit pattern is written into the buffer.

Following a CALL, the buffer is checked to ensure it is unchanged.

A change means the CALL resulted in a write past the end of the caller's WORKING-STORAGE section.

Performance

A program repeatedly calling another empty program took **22%** longer with PARMCHECK than with NOPARMCHECK.

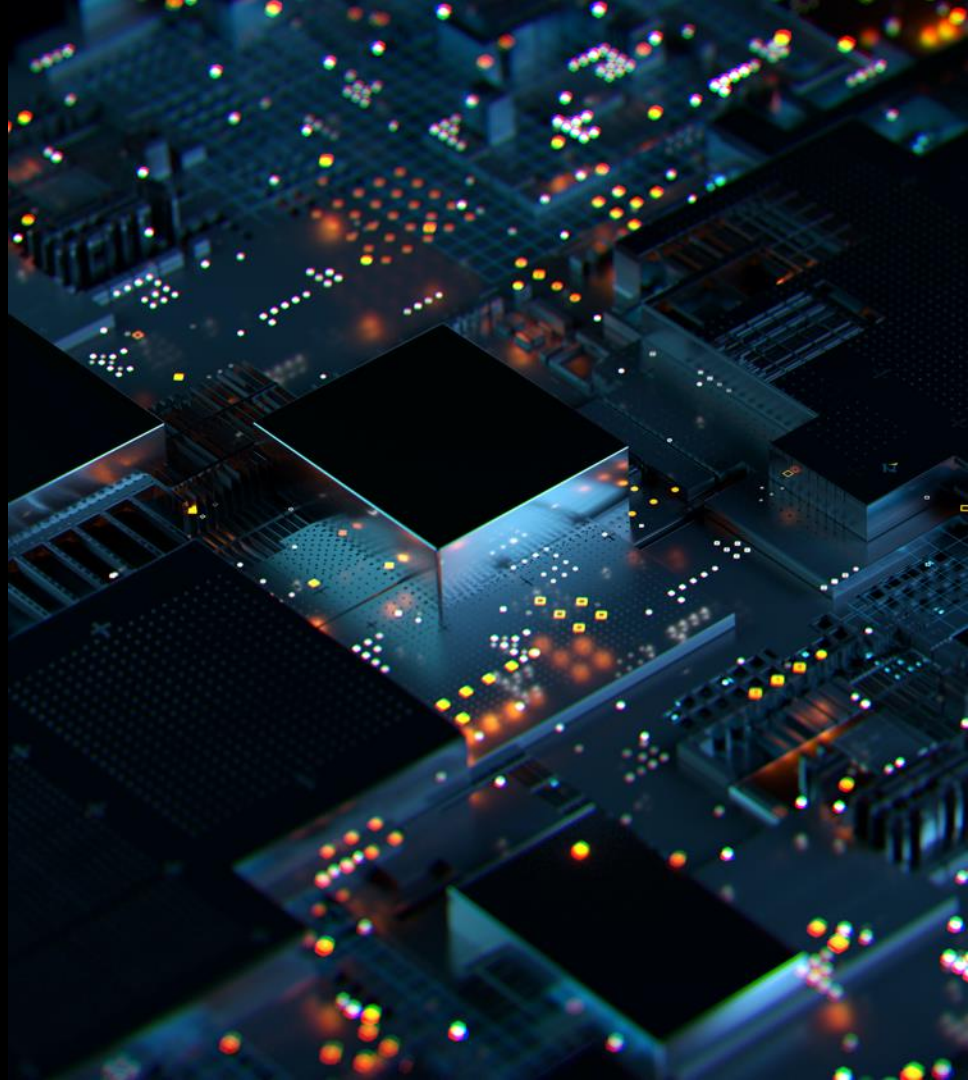
Best strategy

Use IBM Developer for z Systems **Scanning COBOL Programs for Compatibility** feature.

If you don't have IDz, use PARMCHECK during testing.

Do not use PARMCHECK in production.

Runtime / LE Options



Runtime / LE options
Storage management tuning

Best performance

HEAP

ANYHEAP

BELOWHEAP

STACK

LIBSTACK

RPTSTG

What is it?

Storage management is designed to keep a block of storage only as long as is necessary.

If the last block of storage allocated for a program does not contain enough free space to satisfy a storage request by a library routine, the library routine will issue a GETMAIN and FREEMAIN to acquire and release the storage.

If that library routine is called frequently, performance will be degraded.

Best strategy

Use RPTSTG(ON) to measure the storage requirements of your application.

Use the values reported by RPTSTG as the size of the initial blocks for the HEAP, ANYHEAP, BELOWHEAP, STACK, and LIBSTACK options.

In a transactional environment, the LE storage tuning user exit allows you to set storage values for your main programs without having to link-edit the values into your load modules.

Runtime / LE options
First program not LE-conforming

The problem

If the first program in an application is not LE-conforming, the COBOL environment must be initialized and terminated each time a main program is invoked.

This can cause a significant performance degradation.

This is most likely if the first program is written in Assembler.

Best solutions

Rewrite the first program in COBOL.

Replace the first program with a COBOL stub program that calls the first program.

Alternatives

Use the CEEENTRY and CEETERM macros in the first program to make it an LE-conforming program.

Call CEEPIPI from the first program to initialize and terminate the LE environment.

Use the runtime option RTEREUS to initialize the runtime environment for reusability.

Use LRR (library routine retention).

Place library routines in the LPA / ELPA.

Benefits

The overhead of calling an empty COBOL program is **99%** smaller if the COBOL environment has already been initialized.

Runtime / LE options
Options to avoid in production

Best performance	Best strategy
DEBUG	These options add overhead to the execution of your program.
INTERRUPT	
RPTSTG	Use these options for debugging and tuning when necessary but avoid them in production.
STORAGE	
TEST	

Coding practices

Coding practices
Data types

Using the best type for your data items is one of the easiest ways to speed up the computation in a program.

Binary

BINARY, COMP-4, COMP-5, INDEXED BY

Binary arithmetic is the native language of the hardware. Computations done in binary are faster than any other type.

Some operations require their operands to be converted to binary:

- Indexing/subscripting a table
- Reference modification
- Object of ODO clause

Use binary data items whenever possible. Prefer COMP/COMP-4 to COMP-5.

Packed decimal

PACKED-DECIMAL, COMP-3

Packed-decimal arithmetic is like a second language. The hardware knows how to work with it, but it is slower than binary arithmetic.

Use packed decimal for data items with decimal points, or larger values that do not fit in binary items.

Zoned decimal

DISPLAY, NATIONAL, numeric-edited, unspecified

Zoned-decimal arithmetic is like a foreign language. The hardware has to translate it to do anything.

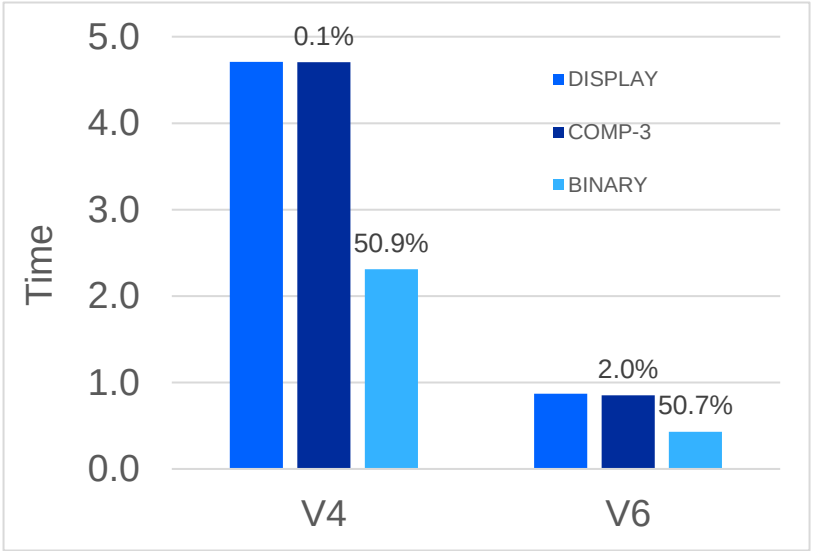
Use zoned decimal for data items that will be used in DISPLAY statements.

Coding practices
Zoned decimal

Avoid using zoned decimal data items for computation whenever possible

Example

```
============================
Working-Storage Section.
01 A PIC 9(8).
01 B PIC 9(8).

Procedure Division.
    Perform Varying A
        from 1 by 1
        until A = 10000000
            Add 1 to B
    End-Perform.
============================
```



Coding practices
Packed decimal

Avoid using packed decimal data items with an even number of digits

Packed decimal

Consider the following data item:

```
01 EVEN-PACK PIC 9(4)
COMP-3
```

It is represented as:

01 23 4C

The **upper half** of the high-order byte is unused and must always be 0. Not only does this waste space, but it requires the compiler to generate additional instructions to set it to 0 after EVEN-PACK is used as a receiver.

Example

```
======================
01  A PIC 9(?) Value 0.
01  B PIC 9(?) Value 0.

Perform Varying A
    from 1 by 1
    until A = 100000
        Add 1 TO B
End-Perform.
======================
```

Performance

The example program is **16%** faster, compiled with ARCH(11), when A and B are PIC 9(7) than when A and B are PIC 9(6).

Coding practices
Binary

Where possible, use COMP over COMP-5.

Use COMP-5 for binary values set by other products: IMS, DB2, C, etc.

What is the difference?

COMP and COMP-5 differ based on their handling of truncation.

COMP data items obey their PIC clause. If a value larger than the PIC clause is moved into a COMP item, it will undergo decimal truncation.

COMP-5 data items are truncated according to the size of the underlying binary value:

9(1) to 9(4)	9(5) to 9(9)	9(10) to 9(18)
2 bytes	4 bytes	8 bytes
0 to 65,535	0 to 4,294,967,295	0 to 18,446,744,073,709,551,615

Intermediate results

Because COMP-5 data items can contain values that exceed their PIC clause, the compiler must assume that intermediate results of computations using those items are larger.

If an operation that involves binary operands requires intermediate results longer than 18 digits, the compiler must convert the operands to packed decimal before performing the operation.

This can be much slower. In some cases, it requires calls to library routines.

Performance

If operands do not exceed 9 digits, COMP-5 is comparable to COMP.

If operands exceed 9 digits, computations involving COMP data items can be **99.5% faster** than the equivalent operations using COMP-5.

Coding practices
Occurs Depending On

Avoid creating variably located data items:

- **Place ODO tables at the end of level-01 groups.**
- **Avoid nesting ODO tables where possible.**

Variably located data items

Some data items are not always located at the same location in working storage.

If a data item follows a variable-length table in a level-01 item, but is not subordinate to it, then the address of that data item depends on the size of the variable-length table.

If a table contains variable-length elements, the address of an element in that table depends on the length of the elements.

Variably located data items require address calculations whenever they are accessed.

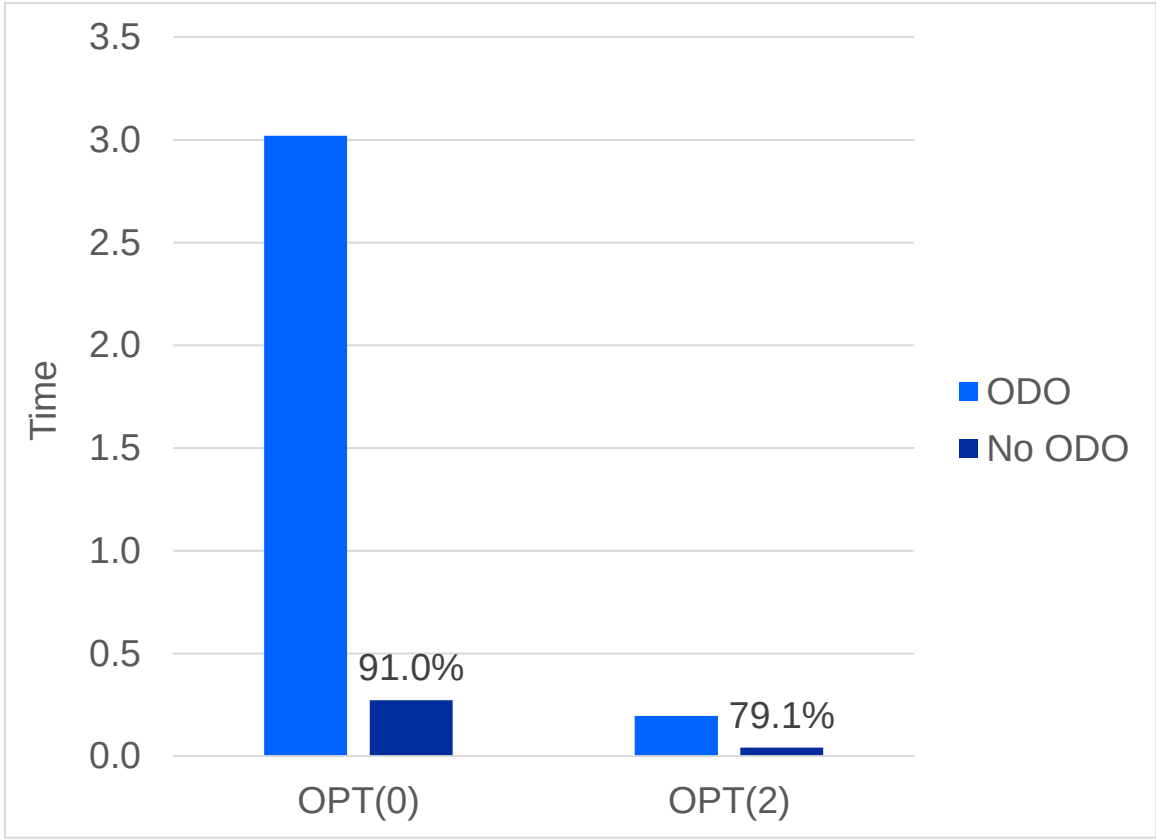
Example

```
================================================
01 FIELD-A.
    02 COUNTER-A                          PIC 99.
    02 TABLE-A.
        03 RECORD-A OCCURS 1 TO 5 TIMES
            DEPENDING ON COUNTER-A        PIC X(3).
    02 EMPLOYEE-NUMBER                    PIC X(5).

01 FIELD-B.
    02 COUNTER-B                          PIC 99.
    02 TABLE-B-1 OCCURS 5 TIMES
                INDEXED BY IDX.
        03 TABLE-ITEM                     PIC 99.
        03 TABLE-B-2 OCCURS 1 TO 3 TIMES
                DEPENDING ON COUNTER-B.
            04 DATA-NUM                   PIC 99.
================================================
```

Coding practices
Occurs depending on

Example

```
========================================
01  GROUP-1.
  02  SIZE-1 PIC 99 value 50.
  02  SIZE-2 PIC 99 value 50.
  02  SIZE-3 PIC 99 value 50.
  02  TABLE-1 PIC 99 OCCURS 1 TO 99 TIMES
              DEPENDING ON SIZE-1.
  02  TABLE-2 PIC 99 OCCURS 1 TO 99 TIMES
              DEPENDING ON SIZE-2.
  02  TABLE-3 PIC 99 OCCURS 1 TO 99 TIMES
              DEPENDING ON SIZE-3.
  02  A PIC 9(6) COMP-3.
  02  B PIC 9(6) COMP-3.
  02  C PIC 9(6) COMP-3.

[...]
PERFORM 100000000 TIMES
    ADD A TO B GIVING C
END-PERFORM
========================================
```



Coding practices
Calls

Static calls are faster than dynamic calls.

However, dynamic calls may still be best for your program.

What is the difference?

If DYNAM is not specified, a CALL to a literal will cause the program with that name to be bound into the same load module as the calling program.

If DYNAM is specified, or if a CALL is to a data item or function pointer instead of a literal, then the target of the call will not be bound into the same load module. Instead, the target will be determined by the COBOL runtime when the program is executed.

The first dynamic call to a program will be significantly slower than subsequent calls.

Performance

When calling an empty program, a static call is **74% faster** than an equivalent dynamic call.

Other considerations

If an application contains programs that are not called every time the application runs, using dynamic calls can avoid ever loading the unused programs into memory.

Dynamic calls are faster than static calls in applications that contain both COBOL 4/earlier programs and COBOL 5/6 programs. Static calls are faster in pure COBOL 4/earlier or pure COBOL 5/6 programs.

Coding practices
RULES(NOLAXPERF)

The compiler can automatically identify many opportunities for code improvement

Example

```
=====================================
Working-Storage Section.
01 Bad-Loop    PIC 9(4).
01 Bad-Arith-1 PIC 9(4).
01 Bad-Arith-2 PIC 9(4).
01 Small-Item  PIC X(10).
01 Large-Item  PIC X(1000).

Procedure Division.
    Perform varying Bad-Loop
    from 1 by 1 until Bad-Loop = 100
        Add Bad-Arith-1 to Bad-Arith-2
        Move Small-Item to Large-Item
    End-perform.
=====================================
```

Opportunities

Loops:
Bad-Loop is a loop counter. It should be defined as COMP or COMP-3.

Arithmetic:
Bad-Arith-1 and Bad-Arith-2 are used in arithmetic. They should be defined as COMP or COMP-3.

Excess padding:
Moving a PIC X(10) to a PIC X(1000) requires 990 bytes of padding to be filled with spaces.

Resources

Need Help With COBOL Performance?

- IBM watsonx Code Assistant for Z Optimize can help you find the hotspots in your COBOL applications and identify and prioritize performance tuning opportunities
  - For additional information see the wca4z website, or click-through demo (click Optimize Mainframe Code to get started)

- An Enterprise COBOL developer can answer specific questions you have about your COBOL Migration, as well as COBOL performance questions or other questions about Enterprise COBOL.
  - Contact an expert here

- Mainframe Application Modernization Services – COBOL Upgrade Service from IBM Consulting can help you recompile applications previously compiled with older IBM COBOL compilers with Enterprise COBOL for z/OS 6.4 or 6.5. Optimization of COBOL modules with Automatic Binary Optimizer for z/OS is also available.
  - For additional information, see the IBM Consulting website.

Thank you

Mike Chase
Compiler Optimization Developer
Lead Developer, IBM watsonx Code Assistant for Z Code Optimization Advice
—
mike.chase@ca.ibm.com
ibm.com

Performance Claims

IBM Enterprise COBOL for z/OS 6.5 (applies here and here)

The performance improvements are based on the geometric mean of IBM internal measurements on IBM z17 running a z/OS 3.1 LPAR with 1 CP and 80GB Central Storage, IBM z16 running a z/OS 2.4 LPAR with 1 CP and 80GB Central Storage, IBM z15 running a z/OS 2.4 LPAR with 1 CP and 80GB Central Storage, IBM z14 running a z/OS 2.3 LPAR with 2 CP and 128GB Central Storage, IBM z13 running a z/OS 2.3 LPAR with 1 CP and 64GB Central Storage, IBM zEC12 running a z/OS 2.2 LPAR with 2 CP and 64GB Central Storage. All benchmarks compiled with IBM Enterprise COBOL for z/OS 5/6 use the options STGOPT, AFP(NOVOLATILE), HGPR(NOPRESERVE), and LIST. All benchmarks compiled with IBM Enterprise COBOL for z/OS 4.2 use the option LIB. Performance results for customer applications will vary, depending on the source code, the compiler options specified, and other factors.

IBM Automatic Binary Optimizer for z/OS 2.3 (applies here)

The performance improvements are based on the geometric mean of IBM internal measurements on IBM z17 running a z/OS 3.1 LPAR with 1 CP and 80GB Central Storage, and IBM zEC12 running a z/OS 2.3 LPAR with 1 CP and 80GB Central Storage. All benchmarks optimized with IBM Automatic Binary Optimizer for z/OS 2.3 use the new ARCH(15) option and default settings for all other options. The input COBOL benchmarks modules optimized by IBM Automatic Binary Optimizer for z/OS were all compiled by Enterprise COBOL 4.2. All benchmarks compiled with IBM Enterprise COBOL for z/OS 4.2 use the options OPT(STD), LIB. Performance results for customer applications will vary, depending on the source code, the compiler options specified, and other factors.