Code optimization with the IBM z/OS XL C/C++ compiler

Introduction

The IBM® z/OS® XL C/C++ compiler is built on an industry-wide reputation for robustness, versatility, and high level of standards compliance. The true strength of the z/OS XL C/C++ compiler comes through in its optimization and the ability to improve code generation. Optimized code runs with greater speed, by using less machine resources, making you more productive.

For example, consider the advantages of a compiler that properly uses the inherent opportunities in z/Architecture®. A set of complex calculations that might take hours by using unoptimized code, can be reduced to mere minutes when fully optimized by the z/OS XL C/C++ compiler.

Built with flexibility in mind, the z/OS XL C/C++ compiler optimization suites give you the reassurance of powerful, no-hassle optimization, which is coupled with the ability to tailor optimization levels and settings to meet the needs of your application and development environment.

This document introduces the most important capabilities and describes the compiler options, source constructs, and techniques that you can use to maximize the performance of your application. Compiler options are described under the z/OS batch format. You can also specify the options in other ways according to your environment. For example, on the z/OS UNIX System Services shell command line, the **`xlc`** utility uses the **`-q`** format and the **`c89`** utility uses the compatible **`-W<phase>`** format.

z/OS XL C/C++ compiler

The z/OS XL C/C++ compiler helps you to create and maintain critical business applications that are written in C or C++, to maximize application performance, and to improve developer productivity. z/OS XL C/C++ compiler transforms C or C++ source code to fully exploit your existing IBM Z hardware and tap into the new IBM z14™ (z14) through compiler options, built-in functions, performance-tuned libraries, and language constructs that simplify system programming and boost application runtime performance.

The current IBM Z mainframe has a long history that started in mid-1960s with a number of compilers that enable developing mainframe applications. The first IBM z/OS C compiler was introduced in late 1980s followed by the z/OS XL C/C++ compiler. Since then, the z/OS XL C/C++ compiler has been under continuous development, with special attention to producing highly optimized applications that fully exploit z/Architecture. The z/OS XL C/C++ compiler also shares optimization components with several key compilers such as the IBM XL C/C++ for AIX® and XL C/C++ for Linux compilers allowing for shared enhancements between compilers.

The compiler design team at IBM works closely with the hardware and operating system development teams. This allows the compilers to exploit the latest hardware and operating system capabilities, and the compiler team can influence hardware and operating system design for creating performance-enhancing capabilities.

The optimization technology in the z/OS XL C/C++ compiler is used to build performance-critical customer code, and is key to the success of many performance-sensitive IBM products such as IMS™, CICS®, and DB2®.

z/OS V2R3 XL C/C++ exploits new instructions in the IBM z14™ hardware. The compiler provides **ARCH(12)** and **TUNE(12)** options to help you exploit new instructions that are available on z14. Vector programming support, including language extensions and built-in functions allow users to exploit the Vector Facility for z/Architecture -- single instruction, multiple data (SIMD) instructions. The z/OS XL C/C++ compiler can use the instructions that are supported by the vector enhancement facility 1 and the vector packed decimal facility when the **ARCH(12)** and **VECTOR** options are in effect.

The **VECTOR** option provides potential performance improvements in the following aspects: fixed point decimal operations, built-in library functions, operations on binary floating-point float, double, and long double data types, and automatic SIMDization or vectorization. Specifying the **VECTOR(TYPE)** suboption enables the compiler to support `vector` data types in addition to `__vector` data types. The **VECTOR(AUTOSIMD)** suboption enables the compiler to generate code, when possible, using the SIMD instructions. SIMD instructions calculate several results at one time, which is faster than calculating each result sequentially.

z/OS V2R3 XL C/C++ demonstrates on average 13% reduction in CPU time for floating point intensive applications on z14 over the same applications that are built with z/OS V2R2 XL C/C++ on z14. z/OS V2R3 XL C/C++ also demonstrates on average 8% reduction in CPU time for compute intensive applications on z14 over the same applications that are built with z/OS V2R2 XL C/C++ on z14.[1]

For more information about the z/OS XL C/C++ compiler, see Marketplace page for IBM z/OS XL C/C++ at https://www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos.

Optimization technology overview

Optimization techniques for the z/OS XL C/C++ compiler are built on a foundation of common components and techniques that are then customized for the C/C++ language. The C/C++ language parser components emit an intermediate language that is processed by the interprocedural analysis (IPA) optimizer and the optimizing code generator. The z/OS XL C/C++ compiler also ships a set of Mathematical Acceleration Subsystem (MASS) and Automatically Tuned Linear Algebra Software (ATLAS) libraries for high-performance mathematical computing.

The z/OS XL C/C++ compiler supports several levels of increasingly aggressive code transformations. Advanced optimization techniques, such as interprocedural analysis (IPA) and profile-directed feedback (PDF), are available at high levels of optimization that can result in significant performance improvements. IPA analyzes and optimizes your application as a whole, rather than on a file-by-file basis. PDF generates information that instructs the optimizer to focus on tradeoffs that favor code that runs more frequently.

The z/OS XL C/C++ compiler offers the following optimization highlights:

- Five distinct optimization levels with many options to tailor the optimization process for your applications
- Code generation and tuning for specific hardware
- Interprocedural optimization by using IPA
- Profile-directed feedback (PDF) optimization
- User-directed optimization with directives
- Source-level intrinsic functions that give you direct access to IBM Z hardware

1. The performance improvements are based on internal IBM lab measurements. All CPU intensive integer and floating-point benchmarks were compiled in 31-bit addressing mode and built using **XPLINK**, **HGPR**, **O3**, **HOT** and **IPA LEVEL(2)** with **PDF** compiler options. The benchmarks compiled with the z/OS V2R2 XL C/C++ compiler were built using the **ARCH(11) TUNE(11)** options; the benchmarks compiled with the z/OS V2R3 XL C/C++ compiler were built using the **ARCH(12) TUNE(12)** options; All benchmarks were executed on a z/OS V2R2 dedicated LPAR with 1 CP and 8GB Central Storage on z14. Performance results for specific applications will vary, depending on the source code, the compiler options specified, and other factors.

Optimization levels

The optimizer includes five base optimization levels (**-O1** level is not supported). The **xlc** utility option is listed first followed by the z/OS batch option.

-O0 or NOOPT
Almost no optimization, best for getting the most debugging information.

-O2 or OPT(2)
Strong low-level optimization that benefits most programs.

-O3 or OPT(3)
Intense low-level optimization analysis.

-O4 or OPT(3), HOT, IPA(LEVEL(1))
All of **-O3** plus detailed loop analysis and basic whole-program analysis at link time.

-O5 or OPT(3), HOT, IPA(LEVEL(2))
All of **-O4** and detailed whole-program analysis at link time.

Optimization progression

While increasing the level at which you optimize your application can provide an increase in performance, other compiler options can be as important as the optimization level you choose.

The following table lists compiler options that are implied by using each optimization level, options you can use with each level, and some other useful options.

Table 1. Optimization levels and options

Base optimization level - xlc utility option	Base optimization level - z/OS batch option	Additional options implied by optimization level	Additional recommended options
-O0	**OPT(0)** or **NOOPT**	None	**ARCH(n)**
-O2	**OPT(2)**	None	**ARCH(n)** **INLINE** (to tune inlining) **TUNE(n)**
-O3	**OPT(3)**	**NOSTRICT**	**ARCH(n)** **TUNE(n)**
-O4	**OPT(3)** **HOT** **IPA(LEVEL(1))**	All of **OPT(3)** plus: **HOT** **IPA(LEVEL(1))**	**ARCH(n)** **TUNE(n)** **PDF**
-O5	**OPT(3)** **HOT** **IPA(LEVEL(2))**	All of **OPT(3)** plus: **HOT** **IPA(LEVEL(2))**	**ARCH(n)** **TUNE(n)** **PDF**

While table 1 provides a list of the most common compiler options for optimization, the z/OS XL C/C++ compiler offers optimization facilities for almost any application. For example, you can also use **HOT** with the base optimization levels if you want high-order loop analysis and transformations (HOT) during optimization.

In general, higher optimization levels take more compilation time and resources. Specifying additional optimization options beyond the base optimization levels might increase compilation time. For an application with a long build time, compilation time might be an issue when choosing the right optimization level and options.

It is important to test your application at lower optimization levels before moving to higher levels, or adding optimization options. If an application does not run correctly when built with **NOOPT**, it is unlikely to run correctly when built with **OPT(2)**. Even subtly nonconforming code can cause the optimizer to perform incorrect transformations, especially at the highest optimization levels. The z/OS XL C/C++ compiler has options to limit optimizations for nonconforming code, but it is a best practice to correct code and not limit your optimization opportunities. One such option is **ANSIALIAS**, which is particularly useful in tracking down unexpected application behavior due to a violation of the ANSI aliasing rules.

Optimization level -O0 or OPT(0) or NOOPT

When you compile with **NOOPT**, which is the default optimization level, the compiler checks the application source code for algorithmic correctness, exposes problems such as uninitialized variables and improper casting (using the **INFO** option), and saves all debug information (using the **DEBUG** option). The intermediate code that is generated from the compiler front-end passes directly to the back-end, where basic optimizations such as redundant code elimination and constant folding are performed before binding. Specifying **ARCH** with **NOOPT** sets the target architecture for the application and can result in better performance as the compiler exploits features of that target architecture. The **ARCH** and **TUNE** options instruct the compiler to generate code for optimal execution on a specific processor or architectural family.

Optimization level -O2 or OPT(2)

Recompiling at **OPT(2)** exposes the source code to comprehensive low-level transformations that apply to the subprogram or compilation unit scope. At **OPT(2)** the compiler back-end performs, for example, more optimizations on loops, and identifies and removes unnecessary code constructs and redundant computations. The compiler strives to balance improved performance while limiting the impact on compilation time and system resources. The target architecture, indicated by **ARCH** and **TUNE**, is more important at **OPT(2)** as the compiler optimizes to exploit the features of the hardware.

The compiler at **OPT(2)** performs several beneficial optimizations, including:

Inlining
: Inlining replaces certain function calls with the actual code of the function being performed.

Value numbering
: Value numbering involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

Straightening
: Straightening is rearranging the program code to minimize branching logic and to combine physically separate blocks of code.

Common expression elimination
: Common expressions recalculate the same value in a subsequent expression. The duplicate expression can be eliminated by using the previous value. This is done even for intermediate expressions within expressions.

Code motion
: If variables used in a computation within a loop are not altered within the loop, it might be possible to perform the calculation outside of the loop and use the results within the loop.

Strength reduction
: Less efficient instructions are replaced with more efficient ones. For example, in array addressing, an add instruction replaces a multiply.

Constant propagation
Constants used in an expression are combined and new ones generated. Some mode conversions are done, and compile-time evaluation of some intrinsic functions takes place.

Instructions scheduling
Instructions are reordered to minimize execution time.

Dead store elimination
The compiler eliminates stores when the value stored is never referred to again. For example, if two stores to the same location have no intervening load, the first store is unnecessary, and is therefore removed.

Dead code elimination
The compiler may eliminate code for calculations that are not required. Other optimization techniques may cause code to become dead.

Graph coloring register allocation
The compiler uses a global register allocation for the whole function, thereby allowing variables to be kept in registers rather than in memory.

Even with **OPT(2)** optimizations, some useful information about your source code is made available to the debugger if you specify **DEBUG**. Higher optimization levels can transform code to an extent to which debug information is no longer accurate. Use that information with discretion.

The **LEVEL** suboption of the **DEBUG** option controls the amount of debug information that is created. For more information, see the z/OS XL C/C++ User's Guide.

Optimization level -O3 or OPT(3)

Recompiling at **OPT(3)** provides more intense low-level transformations that removes many of the limitations present at **OPT(2)**. Optimizations at **OPT(3)** encompass larger program regions and attempt more in-depth analysis, while balancing the trade-offs in compilation time and memory resources. While not all applications contain opportunities for the optimizer to provide a measurable increase in performance, most applications can benefit from this kind of analysis.

OPT(3) is an intensified version of **OPT(2)**. The compiler performs additional low-level transformations and removes limits on **OPT(2)** transformations by turning **STRICT** (preserve floating point semantics) off. Optimizations encompass larger program regions and deepen to attempt more analysis. By default, **OPT(3)** can perform transformations that are not always beneficial to all programs, and attempts several optimizations that can be both memory and compile time intensive. However, most applications benefit from this extra optimization. Some general differences with **OPT(2)** are:

- Increased optimization scope, typically to encompass a whole procedure
- Specialized optimizations that might not help all programs
- Optimizations that require large amounts of compile time or space
- Elimination of implicit memory usage
- Activation of **NOSTRICT**, which allows some reordering of floating-point computations and potential exceptions

Because **OPT(3)** implies the **NOSTRICT** option, certain floating-point semantics of your application can be altered to gain execution speed. These typically involve precision tradeoffs such as the following:

- Reordering of floating-point computations
- Reordering or elimination of possible exceptions (for example, division by zero or overflow)
- Combining multiple floating-point operations into single machine instructions; for example, replacing an add then multiply with a single more accurate and faster float-multiply-and-add instruction

You can still gain most of the benefits of **OPT(3)** while preserving precise floating-point semantics by specifying **STRICT**. This is only necessary if a particular level of floating-point computational accuracy, as compared with **NOOPT** or **OPT(2)** results, is important. You can also specify **STRICT** if your application is sensitive to floating-point exceptions, or if the order and manner in which floating-point arithmetic is evaluated is important. Largely, without **STRICT**, the difference in computed values on any one source-level operation is very small compared to lower optimization levels. However, the difference can compound if the operation involved is in a loop structure, and the difference becomes additive.

Optimization level -O4 or OPT(3), HOT, IPA(LEVEL(1))

Optimization at **-O4** builds on that of **OPT(3)** by triggering interprocedural analysis (IPA), which strives to optimize the entire application as a unit. With IPA specified on both the compile and bind steps, the compiler performs multiple iterations through the optimizer and the back-end, with tradeoffs at compile time, especially on the bind step. Applications that contain many frequently used routines are most likely to benefit from IPA. Profile-directed feedback (PDF), which iteratively refines a profile of how often branches are taken and blocks of code are executed in the application, requires IPA. PDF is designed to tune an application for a particular usage scenario. The **-O4** optimization process begins with the front-end emitting an intermediate language which IPA analyzes and transforms. The optimized intermediate language is then processed by the low-level optimizing back-end for further optimization and object code creation.

Optimization at **-O4** is a way to specify **OPT(3)** with several additional optimization options. The most important of the additional options is **IPA** which performs interprocedural analysis (IPA).

IPA optimization extends program analysis beyond individual files and compilation units to the entire application. IPA analysis can propagate values and inline code from one compilation unit to another. Global data structures can be reorganized or eliminated, and many other transformations become possible when the entire application is visible to the IPA optimizer.

To make full use of IPA optimizations, you must specify **IPA** on the compilation and the link steps of your application build (which is done when specifying **-O4**). At compilation time, important optimizations occur at the compilation-unit level, as well as preparation for link-stage optimization. IPA information is written into the object files produced. At the link step, the IPA information is read from the object files and the entire application is analyzed. The analysis results in a restructured and rewritten application, which subsequently has the lower-level **OPT(3)** style optimizations applied to it before linking. Object files containing IPA information can also be used safely by the system linker without using IPA on the link step.

Optimization at **-O4** implies other optimization options beyond IPA. For example, **HOT** enables a set of high-order transformation optimizations that are most effective when optimizing loop constructs.

Optimization level -O5 or OPT(3), HOT, IPA(LEVEL(2))

The highest optimization level currently available to the z/OS XL C/C++ compiler is **-O5**. This initiates aggressive optimization and the highest level of interprocedural analysis currently available. The **-O5** optimization process begins with the front-end emitting an intermediate language which IPA analyzes and transforms. The optimized intermediate language is then processed by the low-level optimizing back-end for further optimization and object code creation.

Compilation at **-O5** builds on the optimizations at **-O4**, adding deeper whole-program analysis. The most aggressive transformations are available at **-O5**, and the compiler makes full use of loop optimizations and the assumptions resulting from IPA.

Optimization at **-O5** is the highest base optimization level including all **-O4** optimizations and setting **IPA** to level 2. That change, like the difference between **OPT(2)** and **OPT(3)**, broadens and deepens IPA optimization analysis and performs even more intense whole-program analysis. Optimization at **-O5** can

consume the most compile time and machine resource of any optimization level. You should only use optimization **-O5** once you have finished debugging and your application works as expected at lower optimization levels.

PDF can further tune the performance of your application resulting in even faster runtime code.

Compiling at NOOPT, OPT(2), or OPT(3)

The following diagram illustrates the process of compiling at **NOOPT**, **OPT(2)**, or **OPT(3)**.

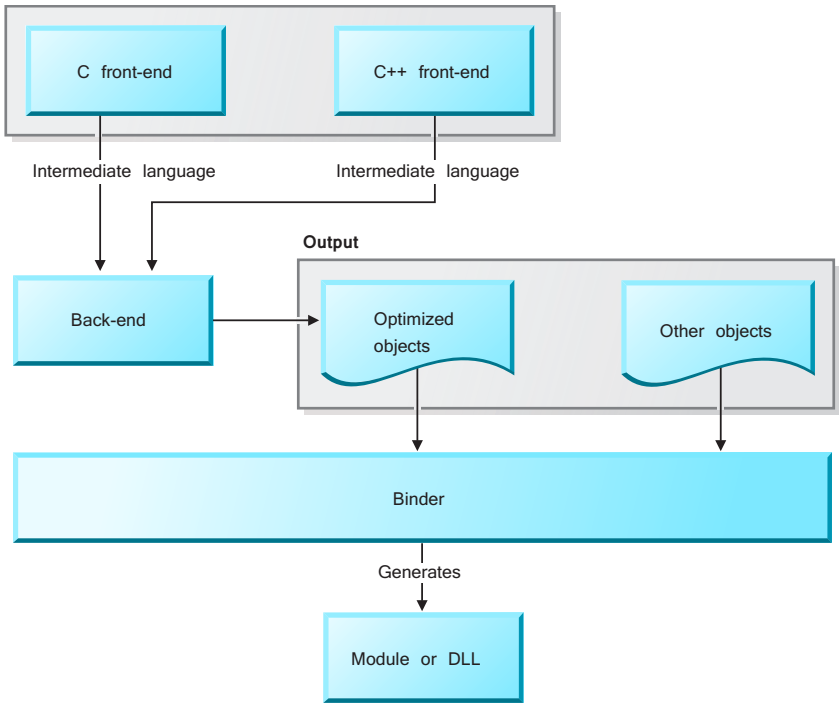

Figure 1. Compiling at NOOPT, OPT(2), or OPT(3)

Compiling at -O4 or -O5

The following diagram illustrates the process of compiling at **-O4** or **-O5**.

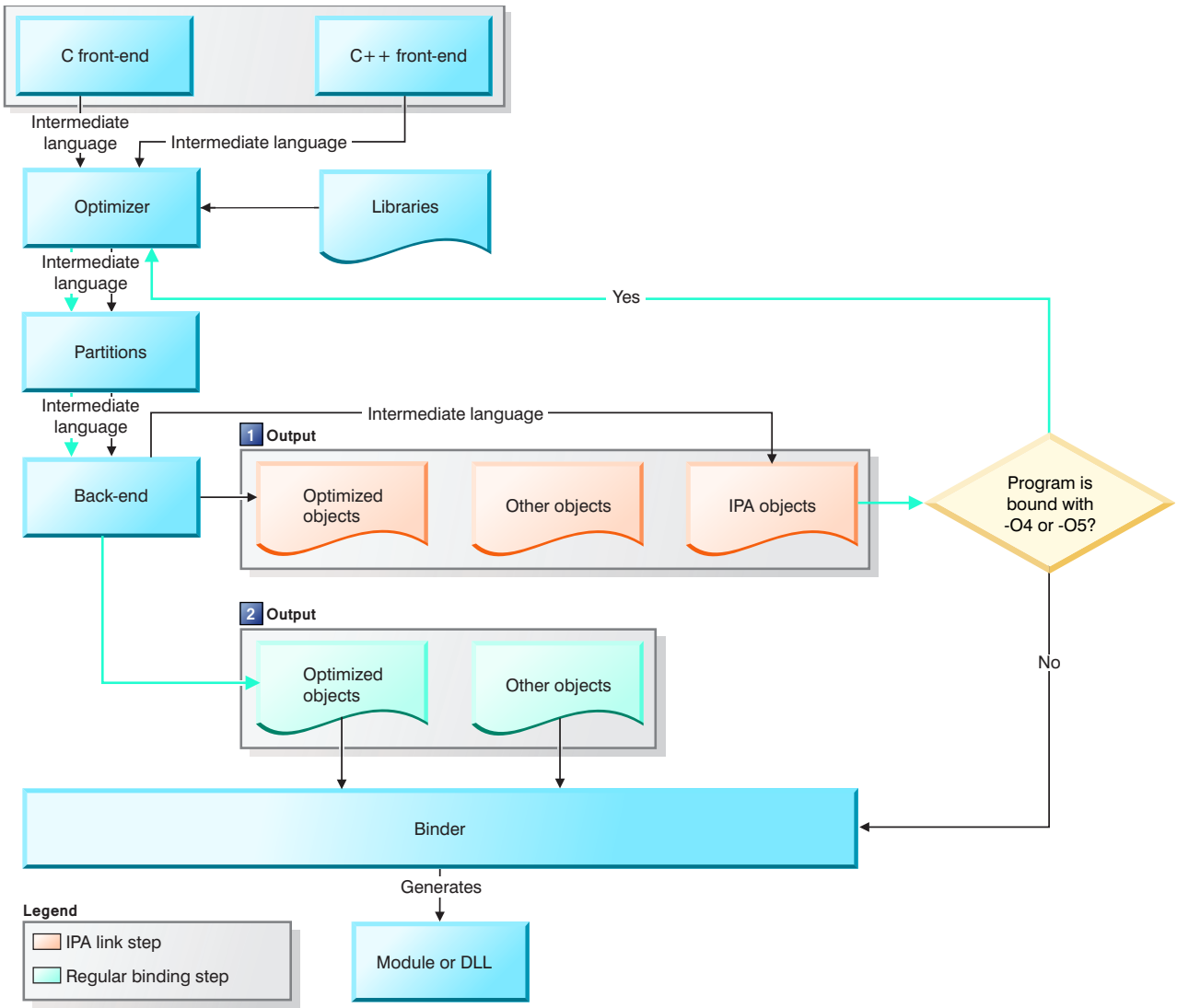

Figure 2. Compiling at -O4 and -O5

Processor optimization capabilities

The z/OS XL C/C++ compiler targets the full range of IBM Z processors.

ARCH option

Using the correct **ARCH** option is the most important step in influencing chip-level optimization. The compiler uses the **ARCH** option to make both high and low-level optimization decisions and trade-offs. The **ARCH** option allows the compiler to access the full range of processor hardware instructions and capabilities when making code generation decisions. Even at low optimization levels specifying the correct target architecture can have a positive impact on performance.

ARCH instructs the compiler to structure your application to execute on a particular set of machines that support the specified instruction set and later. The **ARCH** option features suboptions that specify individual processors. The choice of processor gives you the flexibility of compiling your application to execute

optimally on a particular machine or on any higher-level architecture machines, but still have as much architecture-specific optimization applied as possible.

For example, compiling applications with the z/OS XL C/C++ compiler to produce code that uses instructions available on the z14 models, use **ARCH(12)**. For compiling z/OS C/C++ applications that will only run on 64-bit mode capable hardware, use **ARCH(5)** to select the entire 64-bit z/Architecture family of processors.

The default setting of **ARCH** is 10, which selects the instruction set common to all processors supported by z/OS V2R3. This setting produces code that uses instructions available on the 2827-xxx (IBM zEnterprise® EC12 (zEC12)) and 2828-xxx (IBM zEnterprise BC12 (zBC12)) models in z/Architecture mode.

TUNE option

The **TUNE** option directs the optimizer to bias optimization decisions for executing the application on a particular architecture, but does not prevent the application from running on other architectures. The default **TUNE** setting depends on the setting of the **ARCH** option. If the **ARCH** option selects a particular machine architecture, the range of **TUNE** suboptions that are supported is limited by the chosen architecture, and all architectures above that level. Using **TUNE** allows the optimizer to perform transformations, such as instruction scheduling, so that resulting code executes most efficiently on your chosen **TUNE** architecture.

TUNE(10) is the default. **TUNE(10)** generates code that is optimized for the 2827-xxx (IBM zEnterprise EC12 (zEC12)) and 2828-xxx (IBM zEnterprise BC12 (zBC12)) models.

Use **TUNE** to specify the most common or important processor where your application executes. For example, if your application usually executes on z14 models but sometimes executes on z13™ models, use **ARCH(11) TUNE(12)**. The code generated executes more efficiently on z14 models but can run correctly on z13 models.

Choosing the right hardware architecture target or family of targets becomes even more important at **OPT(2)** and higher. This allows you to compile for a general set of targets but have the code run best on a particular target. If you choose a family of hardware targets, the **TUNE** option can direct the compiler to emit code consistent with the architecture choice, but will execute optimally on the chosen tuning hardware target.

Source-level optimizations

The z/OS XL C/C++ compiler exposes hardware-level capabilities directly to you through source-level intrinsic functions, procedures, directives, and pragmas. The compiler offers simple interfaces that you can use to access z/Architecture instructions that control low-level instruction functionality such as:

- Hardware cache prefetching
- Arithmetic (e.g. FMA, converts, rotates)
- Compare-and-trap

Using the built-in functions, the compiler inserts the requested instructions or instruction sequences for you, but is also able to perform optimizations using and modelling the instructions' behavior. In addition, z/Architecture instruction sequences can be inserted with inline assembly statements.

High-order transformation (HOT) loop optimization

The High-order transformation (HOT) optimizer is a specialized loop transformation optimizer. HOT optimizations are active by default at **-O4** and **-O5** optimization. You can also specify HOT optimization at **OPT(2)** and **OPT(3)** using the **HOT** option. Loops typically account for most of the execution time of most

applications and the HOT optimizer performs in-depth analysis of loops to minimize their execution time. Loop optimization techniques include: interchange, fusion, unrolling of loop nests, and reducing the use of temporary arrays. The goals of these optimizations include:

- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers (TLBs). Increasing memory locality reduces cache/TLB misses.
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of processor resources through reordering and balancing the usage of instructions with complementary resource requirements. Loop computation balance typically involves load/store operations balanced against floating-point computations.

Interprocedural analysis (IPA) optimization

The IPA optimizer's primary focus is whole-program analysis and optimization. IPA analyzes the entire program at once rather than on a file-by-file basis. This analysis occurs during the link step of an application build when the entire program, including linked-in libraries, is visible to the IPA optimizer. IPA can perform transformations that are not possible when only one file or compilation unit is visible at compilation time.

IPA link-time transformations restructure your application, performing optimizations such as inlining between compilation units. Complex data flow analysis occur across subprogram calls to eliminate parameters or propagate constants directly into called subprograms. IPA can recognize system library calls because it acts as a pseudo-linker resolving external subprogram calls to system libraries. This allows IPA to improve parameter usage analysis or even eliminate the call completely and replace it with more efficient inline code.

In order to maximize IPA link-time optimization, the IPA optimizer must be used on both the compile and the link step. IPA can only perform a limited program analysis at link time on objects that were not compiled with IPA, and must work with greatly reduced information. When IPA is active on the compile step, program information is stored in the resulting object file, which IPA reads on the link step when the object file is analyzed. The program information is invisible to the system linker, and the object file can be used as a normal object and be linked without invoking IPA. IPA uses the hidden information in the object to reconstruct the original compilation and is then able to completely reanalyze the subprograms in the object in the context of their actual usage in the application.

The IPA optimizer performs many transformations even if IPA is not used on the link step. Using IPA on the compile step initiates optimizations that can improve performance for each individual object file even if the object files are not linked using the IPA optimizer. Although IPA's primary focus is link-step optimization, using the IPA optimizer only on the compile-step can still be very beneficial to your application.

IPA's link-time analysis facilitates a restructuring of the application and a partitioning of it into distinct units of compatible code. After IPA optimizations are completed, each unit is further optimized by the optimizer normally invoked with the **OPT(2)** or **OPT(3)** options. Each unit is compiled into one or more object files, which are linked with the required libraries by the system linker, producing an executable program.

It is important that you specify a set of compilation options as consistent as possible when compiling and linking your application. This applies to all compiler options, not just **IPA** suboptions. The ideal situation is to specify identical options on all compilations and then to repeat the same options on the IPA link step. Incompatible or conflicting options used to create object files or link-time options in conflict with compile-time options can reduce the effectiveness of IPA optimizations. For example, it can be unsafe to inline a subprogram into another subprogram if they were compiled with conflicting options.

IPA suboptions

The IPA optimizer has many behaviors which you can control using the **IPA** option and suboptions. The most important part of the IPA optimization process is the level at which IPA optimization occurs. By default, the IPA optimizer is not invoked. If you specify **IPA** without a level, or **OPT(IPA(LEVEL(1))**, IPA is run at level one. If you specify **OPT(IPA(LEVEL(2))**, IPA is run at level two. Level zero can reduce compilation time, but performs a more limited analysis. Some of the important IPA transformations at each level:

IPA(LEVEL(0))

- Automatic recognition of standard library functions such as ANSI C
- Localization of statically bound variables and procedures
- Partitioning and layout of code according to call affinity. This expands the scope of the **OPT(2)** and **OPT(3)** low-level compilation unit optimizer

This level can be beneficial to an application, but cost less compile time than higher levels.

IPA(LEVEL(1))

- Level 0 optimizations
- Procedure inlining
- Partitioning and layout of static data according to reference affinity

IPA(LEVEL(2))

- Level 0 and level 1 optimizations
- Whole program alias analysis
- Disambiguation of pointer references and calls
- Refinement of call side effect information
- Aggressive intraprocedural optimizations
- Value numbering, code propagation and simplification, code motion (into conditions, out of loops), and redundancy elimination
- Interprocedural constant propagation, dead code elimination, pointer analysis
- Procedure specialization (cloning)

In addition to selecting a level, the **IPA** option has many other suboptions available for fine-tuning the optimizations applied to your program.

The **IPA(LIST)** suboption can show you brief or detailed information concerning IPA analysis like program partitioning and object reference maps.

An important suboption that can speed compilation time is **IPA(NOOBJECT)**. You can reduce compilation time if you intend to use IPA on the link step and do not need to link the object files from the compilation step without using IPA. Specify the **IPA(NOOBJECT)** option on the compile step to create object files that only the IPA link-time optimizer can use. This creates object files more quickly because the low-level compilation unit optimizer is not invoked on the compile step.

You can additionally use the **REPORT** compiler option to produce pseudo-C code listing files that show how sections of code have been optimized with HOT, IPA compile, and IPA link.

Profile-directed feedback (PDF) optimization

PDF is an optimization the compiler applies to your application in two stages. The first stage collects information about your program as you run it with typical input data. The second stage applies transformations to your application based on that information. The compiler uses PDF to get information such as the locations of heavily used or infrequently used blocks of code. Knowing the relative execution frequency of code provides opportunities to bias execution paths in favor of heavily used code. PDF can perform program restructuring in order to ensure that infrequently-executed blocks of code are less likely to affect program path length or participate in instruction cache fetching.

It is important that the data sets PDF uses to collect information be characteristic of data your application will typically see. Using atypical data or insufficient data can lead to a faulty analysis of the program and suboptimal program transformation. If you do not have sufficient data, PDF optimization is not recommended.

The following diagram illustrates PDF process.

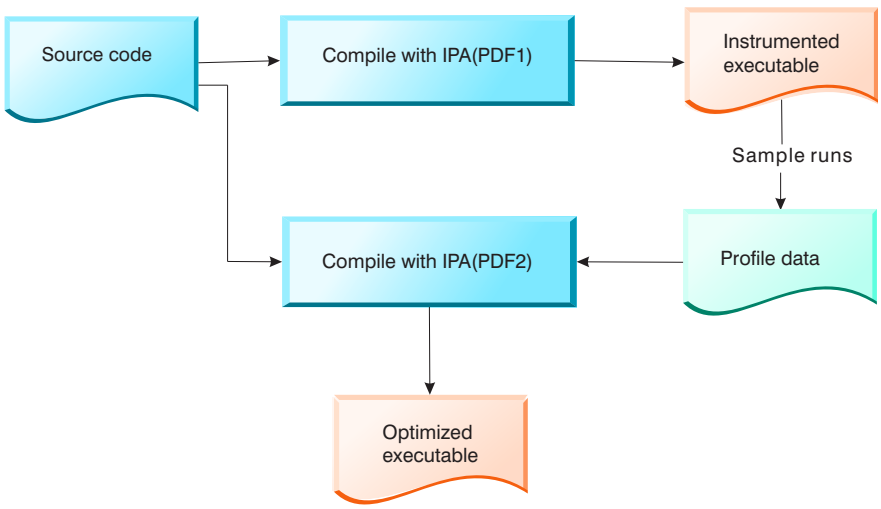

Figure 3. Profile-directed feedback (PDF) process

The first step in PDF optimization is to compile and link your application with the **IPA(PDF1)** option. Doing so instruments your code with calls to a PDF runtime library that will link with your program. Then execute your application with typical input data as many times as you wish with as many data sets as you have. Each run records information in data files.

After you collect sufficient PDF data, recompile or simply relink your application with the **IPA(PDF2)** option. The compiler reads the PDF data files and makes the information available to all levels of optimization that are active. PDF optimization requires IPA. It can be combined with other optimization techniques such as the standard **OPT(2)** or **OPT(3)** compilation unit optimizations, or the **HOT** optimizations that are active at higher optimization levels.

PDF optimization is most effective when you apply it to applications that contain blocks of code that are infrequently and conditionally executed. Typical examples of this coding style include blocks of error-handling code and code that has been instrumented to conditionally collect debugging or statistical information.

High performance libraries

The z/OS XL C/C++ compiler is shipped with a set of libraries for high-performance mathematical computing:

Mathematical Acceleration Subsystem (MASS) libraries

The Mathematical Acceleration Subsystem (MASS) consists of a library of scalar C functions, a vector library, and a SIMD library. These libraries include mathematical intrinsic functions that are tuned specifically for zEC12, z13, or z14, which provide improved performance over the corresponding standard system math library functions.

z/OS V2R3 MASS vector functions on z14 demonstrate throughput improvements averaging 12.4x, and up to 37x, over the corresponding z/OS V2R2 runtime math library functions on z13.[2]

Automatically Tuned Linear Algebra Software (ATLAS) libraries

The Automatically Tuned Linear Algebra Software (ATLAS) is a set of high-performance, processor-tuned linear algebra libraries. The ATLAS libraries contain all the Basic Linear Algebra Subprograms (BLAS) and a subset of the Linear Algebra Package (LAPACK) routines with interfaces that are provided for C versions of the routines across platforms and architectures.

The following C/C++ compiler options are required to compile and link a program that uses ATLAS functions: **FLOAT(IEEE)**, **ROUND(N)**, **ARCH(10)** or higher, **VECTOR**, and **TARGET(zOSV2R1)** or higher.

A key subset of z/OS V2R3 ATLAS functions on z14 demonstrate throughput improvements averaging up to 16% over the corresponding functions on z13.[3]

Aliasing

The apparent affects of direct or indirect memory access can often constrain the precision of compiler analysis. Memory can be referenced directly through a variable, or indirectly through a pointer, function call or reference parameter. Many apparent references to memory are false, and constitute barriers to compiler analysis. The compiler analyzes possible aliases at all optimization levels, but analysis of these apparent references is best when using the **IPA** option. Options such as **ANSIALIAS** can fundamentally improve the precision of compiler analysis.

The C/C++ language rules are well defined for what can and cannot be done with arguments passed to subprograms. Failure to follow language rules that affect aliasing will often mislead the optimizer into performing unsafe or incorrect transformations. The higher the optimization level and the more optional optimizations you apply, the more likely the optimizer will be misled.

The z/OS XL C/C++ compiler supplies options that you can use to optimize programs with nonstandard aliasing constructs. Specifying these options can result in poor-quality aliasing information, and less than optimal code performance. It is recommended that you alter your source code where possible to conform to language rules.

You can specify the **ANSIALIAS** option to assert whether your application follows the type-based aliasing rules defined in the ISO C and C++ standards. For example, the **c89** and **c99** invocations assume ANSI aliasing conformance creating additional optimization opportunities as the optimizer performs more precise aliasing analysis in code with pointers. To help diagnose areas with aliasing violations, the **INFO(ALS)** option is very useful. This option causes diagnostic messages to be emitted where the compiler believes there may be an ANSI aliasing violation.

---

2. This claim is based on results from internal lab measurements on similarly configured dedicated LPARs (1 CP and 131 GB central storage) on z13 and z14 using 64-bit XPLINK vector MASS. The benefit is demonstrated using a subset of MASS vector functions. The performance improvements achieved will vary depending on the workload, vector length, and other factors.

3. This claim is based on results from internal lab measurements running on similarly configured dedicated LPARs (1CP and 131 GB central storage) on z13 and z14 IPLed with z/OS V2.2 using 64-bit XPLINK ATLAS. The benefit is demonstrated using a key ATLAS single-threaded function and taking a geometric mean over a set of problem sizes. The performance improvements achieved will vary depending on the workload, problem size, and other factors.

Additional performance options

In addition to the options already introduced, the z/OS XL C/C++ compiler has many other options that you can use to direct the optimizer. Some of these are specific to C compile, C++ compile, and IPA link rather than the entire z/OS XL C/C++ compiler.

Optimizer guidance options

COMPACT
Default is **NOCOMPACT**. Prefers final code size reduction over execution time performance when a choice is necessary. Can be useful as a way to constrain the **OPT(3)** and higher optimization levels.

INLINE Attempts to inline procedures instead of generating calls to those procedures, for improved performance.

PREFETCH
Instructs the compiler to insert prefetch instructions automatically where there are opportunities to improve code performance.

UNROLL Default is **NOUNROLL** (unless optimizing). Independently controls loop unrolling. It is turned on implicitly with any optimization level higher than **NOOPT**. You can specify suboptions that determine the aggressiveness of automatic loop unrolling.

VECTOR Controls whether the compiler enables the vector programming support and automatically takes advantage of vector/SIMD instructions.

Program behavior options

AGGRCOPY(OVERL)
Specifies whether aggregate assignments may have overlapping source and target locations. Default is **NOOVERL**.

ASSERT(RESTRICT)
Enables optimizations for restrict qualified pointers. Optimizations based on restrict qualified pointers will occur unless you explicitly disable them with the option **ASSERT(NORESTRICT)**.

CHECKNEW (C++ only)
Controls whether a null pointer check is performed on the pointer that is returned by an invocation of the throwing versions of operator `new` and operator `new[]`.

IGNERRNO
For **NOOPT** and **OPT(2)**, the default option is **NOIGNERRNO**. For **OPT(3)**, the default option is **IGNERRNO**. Indicates that the value of `errno` is not needed by the program. Can help optimization of math functions that might set `errno`, such as `sqrt`.

LANGLVL(CHECKPLACEMENTNEW)
This option controls whether a null pointer check is performed on the pointer that is returned by an invocation of the reserved forms of the placement operator `new` and operator `new[]`. The **LANGLVL(NOCHECKPLACEMENTNEW)** option is especially beneficial if the calls to placement operator `new` and operator `new[]` are inside loops or in functions which are called frequently.

LIBANSI
Default is **NOLIBANSI**. Indicates whether or not functions with the name of an ANSI C library function are in fact ANSI C library functions and behave as described in the ANSI standard. Allows the compiler to replace the calls with more efficient inline code or at least do better call-site analysis.

Floating-point computation options

The **FLOAT(IEEE|HEX)** option provides precise control over the handling of floating-point calculations. For Metal C and 64-bit programs the compiler by default generates binary (IEEE754 compliant) floating-point numbers and instructions. For 31-bit programs the compiler by default generates hexadecimal floating-point formatted numbers and instructions. Where the compiler can generate non-compliant code, it is allowed to exploit certain optimizations such as floating-point constant folding, or to use efficient instructions that combine operations. You can use **FLOAT** to prohibit these optimizations. Some of the most frequently applicable **FLOAT** suboptions for IEEE mode are:

FOLD Enables compile time evaluation of floating-point calculations. You might need to disable folding if your application must handle certain floating-point exceptions such as overflow or inexact.

MAF Enables generation of combined multiple-add instructions. In some cases you must disable MAF instructions to produce results identical to those performed at compile time or on other types of computers. Disabling MAF instructions can result in significantly slower code.

RRM Specifies that the rounding mode is not always round-to-nearest. The default is **NORRM**. The rounding mode can also change across calls.

Diagnostic options

The following options can assist you in analyzing the results of compiler optimization. You can examine this information to see if expected transformations have occurred.

LIST Generates an object listing that includes pseudo-assembly representations of the generated code and text constants.

REPORT Instructs the HOT or IPA optimizer to emit a report containing pseudo C code.

User-directed source-level optimizations

The z/OS XL C/C++ compiler supports many source-level pragmas that you can specify to influence the optimizer. Several have been mentioned in previous sections. The following section contain an important subset that the z/OS XL C/C++ compiler supports, including a brief description of each C/C++ pragma.

z/OS XL C/C++ pragmas

disjoint (*variable_list***)**
Asserts that none of the named variables or pointer dereferences share overlapping areas of storage.

isolated_call (*function_list***)**
Asserts that calls to the named functions do not have side effects.

leaves (*function_list***)**
Asserts that calls to the named functions will not return.

unroll Specified as `[no]unroll` to turn loop unrolling on or off. You can use `unroll(<n>)` to specify a specific unroll factor.

Summary

The z/OS XL C/C++ compiler offers premier optimization capabilities on z/OS. You can control the type and depth of optimization analysis through compiler options, which allow you choose the levels and kinds of optimization that are best suited to your application. IBM's long history of compiler development gives you control of mature industry-leading optimization technology such as interprocedural analysis (IPA), high-order transformations (HOT), profile-directed feedback (PDF), as well

as a unique set of optimizations that exploit the hardware architecture's capabilities. This optimization strength combines with robustness, capability, and standards conformance to produce a product set unmatched in the industry.

Purchasing

Information about purchasing the z/OS XL C/C++ compiler is available at the Marketplace page for IBM z/OS XL C/C++.

Contacting IBM

IBM welcomes your comments. You can send them to `compinfo@ca.ibm.com`.

August 2017