Code optimization with the IBM XL Compilers

Introduction

The IBM® XL compiler family offers C, C++, and Fortran compilers built on an industry wide reputation for robustness, versatility and standards compliance. The true strength of the XL compilers comes through in its optimization and the ability to improve code generation. Optimized code executes with greater speed, using less machine resources, making you more productive.

For example, consider the advantages of a compiler that properly exploits the inherent opportunities in Power Architecture®. A set of complex calculations that could take hours using unoptimized code, can be reduced to mere minutes when fully optimized by an XL compiler.

Built with flexibility in mind, the XL compiler optimization suites give you the reassurance of powerful, no-hassle optimization, 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.

XL compiler family

The IBM XL family encompasses three core languages, (C, C++, and Fortran), for AIX® and Linux, on IBM Power®, and Blue Gene®.

You can find more information about the XL compiler family, including downloadable trial versions, on the Web, at:

- http://www.ibm.com/software/awdtools/xlcpp/
- http://www.ibm.com/software/awdtools/fortran/

XL compiler history

The XL family of compilers has grown from work at IBM that began in the mid-1980s with development of the earliest Power-based AIX systems. Since then, the compilers have been under continuous development, with special attention to producing highly optimized applications that fully exploit IBM Power Systems™. The XL compilers also share optimization components with several key IBM mainframe System z® 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 take advantage of 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 XL family of compilers is used to build performance-critical customer code, and is key to the success of many performance-sensitive IBM products such as AIX, DB2® and Lotus® Domino®. IBM XL family of compilers are also used for publishing benchmark results, such as Standard Performance Evaluation Corporation (SPEC) results.

Optimization technology overview

Optimization techniques for the XL compiler family are built on a foundation of common components and techniques that are then customized for the C, C++, and Fortran languages. All three language parser components emit an intermediate language that is processed by the interprocedural analysis (IPA) optimizer and the optimizing code generator. The languages also share a set of language-independent high-performance runtime libraries to support capabilities such as symmetric multiprocessing (SMP) and high-performance complex mathematical calculations.

Sharing common components ensures that performance-enhancing optimization are techniques available in all three languages available to you. At the highest optimization levels, the IPA optimizer can combine and optimize code from all three languages simultaneously when linking an application.

Below are some of the optimization highlights that the XL family of compilers offers:

- Five distinct optimization levels, as well as many additional options that allow you to tailor the optimization process for your application
- Code generation and tuning for specific hardware chipsets
- Interprocedural optimization and inlining using IPA
- Profile-directed feedback (PDF) optimization on AIX and Linux
- User-directed optimization with directives and source-level intrinsic functions that give you direct access to Power, Vector Multimedia eXtension (VMX) and Vector Scalar eXtension (VSX) instructions on AIX and Linux, and Quad Processing eXtension (QPX) on Blue Gene/Q
- Optimization of OpenMP programs and auto-parallelization capabilities to exploit SMP systems
- Automatic parallelization of calculations using vector machine instructions and high-performance mathematical libraries

For more information, see http://www.ibm.com/software/awdtools/xlcpp/ and http://www.ibm.com/software/awdtools/fortran/.

Optimization levels

The optimizer includes five base optimization levels (**-O1** level is not supported):

- **-O0**, almost no optimization, best for debugging
- **-O2**, strong low-level optimization that benefits most programs
- **-O3**, intense low-level optimization analysis and base-level loop analysis
- **-O4**, all of **-O3** plus detailed loop analysis and basic whole-program analysis at link time
- **-O5**, 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 **-O** level you choose.

The **Optimization levels and options** table details compiler options implied by using each level, options you should use with each level, and some useful additional options.

Optimization levels and options

Base optimization level	Additional options implied by level	Additional recommended options
-O0	None **-qsimd=auto** 1	**-qarch**
-O2	**-qmaxmem=8192** **-qsimd=auto** 1	**-qarch** **-qtune**

Optimization levels and options

-O3	**-qnostrict** **-qmaxmem=-1** **-qhot=level=0** **-qsimd=auto** [1]	**-qarch** **-qtune**
-O4	All of **-O3** plus: **-qhot** **-qipa** **-qarch=auto** **-qtune=auto** **-qcache=auto** **-qsimd=auto** [1]	**-qarch** **-qtune** **-qcache**
-O5	All of **-O4** plus: **-qipa=level=2**	**-qarch** **-qtune** **-qcache**

Notes:

1. **-qsimd=auto** is only applicable on Blue Gene/Q.

While the above table provides a list of the most common compiler options for optimization, the XL compiler family offers optimization facilities for almost any application. For example, you can also use **-qsmp=auto** with the base optimization levels if you desire automatic parallelization of your application.

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

It is important to test your application at lower optimization levels before moving on to higher levels, or adding optimization options. If an application does not execute correctly when built with **-O0**, it is unlikely to execute correctly when built with **-O2**. Even subtly non-conforming code can cause the optimizer to perform incorrect transformations, especially at the highest optimization levels. All XL compilers have options to limit optimizations for non-conforming code, but it is best practice to correct code and not limit your optimization opportunities. One such option is **-qalias=noansi** (XL C/C++), which is particularly useful in tracking down unexpected application behavior due to a violation of the ANSI aliasing rules.

Optimization level 0 (-O0)

At **-O0**, XL compilers minimize optimization transformations to your applications. Some limited optimization occurs at **-O0** even if you specify no other optimization options. However, limited optimization analysis generally results in a shorter compile time than other optimization levels.

-O0 is the best level to specify when debugging your code with a symbolic debugger. Additionally, include **-g** on the command line to instruct the compiler to emit debugging information. Whenever possible, ensure that your application executes correctly after compiling with **-O0** before increasing your optimization level. When debugging SMP code, you can specify **-qsmp=noopt** to perform only the minimal transformations necessary to parallelize your code and preserve maximum debug capability.

Optimizing at level 2 (-O2)

After successfully compiling, executing, and debugging your application using **-O0**, recompiling at **-O2** opens your application to a set of comprehensive low-level transformations that apply to subprogram or compilation unit scopes and can include some inlining. Optimizations at **-O2** attempt to find a balance between increasing performance while limiting the impact on compilation time and system resources. You can increase the memory available to some of the optimizations in the **-O2** portfolio by providing a larger value for the **-qmaxmem** option. Specifying **-qmaxmem=-1** lets the optimizer use memory as needed without checking for limits but does not change the transformations applied to your application at **-O2**.

A note about tuning

Choosing the right hardware architecture target or family of targets becomes even more important at **-O2** 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 **-qtune** option can direct the compiler to emit code consistent with the architecture choice, but will execute optimally on the chosen tuning hardware target.

The **-O2** option can perform a number of beneficial optimizations, including:

- Common subexpression elimination - Removes redundant expressions.
- Constant propagation - Evaluates constant expressions at compile time.
- Dead code elimination - Eliminates instructions that a particular control flow does not reach, or that generate an unused result.
- Dead store elimination - Eliminates unnecessary variable assignments.
- Graph coloring register allocation - Globally assigns user variables to registers.
- Instruction scheduling for the target machine.
- Loop unrolling and software pipelining - Simplifies loop control flow and reorders loop instructions to create looser data dependencies.
- Strength reduction and effective use of addressing modes.
- Value numbering - Simplifies algebraic expressions, by eliminating redundant computations.

Even with **-O2** optimizations, some useful information about your source code is made available to the debugger if you specify different levels of **-g**. Higher optimization levels can transform code to an extent to which debug information is no longer accurate. Use that information with discretion.

Optimization level 3 (-O3)

-O3 is an intensified version of **-O2**. The compiler performs additional low-level transformations and removes limits on **-O2** transformations, as **-qmaxmem** defaults to the -1 (unlimited) value. Optimizations encompass larger program regions and deepen to attempt more analysis. By default, **-O3** implies **-qhot=level=0**, which introduces basic high-level analysis of and loop transformation. **-O3** can perform transformations that are not always beneficial to all programs, and attempts several optimizations that can be both memory and time intensive. However, most applications benefit from this extra optimization. Some general differences with **-O2** are:

- Better loop scheduling and transformation
- 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 **-qnostrict**, which allows some reordering of floating-point computations and potential exceptions

Because **-O3** implies the **-qnostrict** option, certain floating-point semantics of your application can be altered to gain execution speed. These typically involve precision trade-offs 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-multiple-and-add instruction.

You can still gain most of the benefits of **-O3** while preserving precise floating-point semantics by specifying **-qstrict**. This is only necessary if absolutely precise floating-point computational accuracy, as compared with **-O0** or **-O2** results, is important. You can also specify**-qstrict** 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 **-qstrict**, 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.

Additional suboptions have been added to **-qstrict**. XL C for AIX, V10.1, XL C/C++ for AIX, V10.1, XL C/C++ for Linux, V10.1, XL Fortran for AIX, V12.1 and XL Fortran for Linux, V12.1 introduced additional **-qstrict** options for fine-grain control of strict IEEE conformance. XL C for AIX, V11.1, XL C/C++ for AIX, V11.1, XL C/C++ for Linux, V11.1, XL Fortran for AIX, V13.1 and XL Fortran for Linux, V13.1 introduced a new suboption, **vectorprecision**, which allows more control over optimizations and transformations that violate strict program semantics. The new suboption disables vectorization in loops where it might produce different results in vectorized iterations than in non-vectorized ones.

Optimization level 4 (-O4)

-O4 is a way to specify **-O3** with several additional optimization options. The most important of the additional options is **-qipa=level=1** 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 **-O4** on the compilation and the link steps of your application build. 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 **-O3** 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.

-O4 implies other optimization options beyond IPA:

- **-qhot** enables a set of high-order transformation optimizations that are most effective when optimizing loop constructs.
- **-qarch=auto** and **-qtune=auto** are enabled, and assume that the machine on which you are compiling is the machine on which you will execute your application. If the architecture of your build machine is incompatible with the machine that will execute the application, you will need to specify a different **-qarch** option after the **-O4** option to override **-qarch=auto**.
- **-qcache=auto** assumes that the cache configuration of the machine on which you are compiling is the machine on which you will execute your application. If you are executing your application on a different machine, specify correct cache values, or use **-qnocache** to disable the **auto** suboption.

Optimization level 5 (-O5)

-O5 is the highest base optimization level including all **-O4** optimizations and setting **-qipa** to level 2. That change, like the difference between **-O2** and **-O3**, broadens and deepens IPA optimization analysis and performs even more intense whole-program analysis. **-O5** can consume the most compile time and machine resource of any optimization level. You should only use **-O5** once you have finished debugging and your application works as expected at lower optimization levels.

If your application contains both C/C++ and Fortran code compiled using XL compilers, you can increase performance by compiling and linking the code with the **-O5** option.

-O5, as with **-O4**, implies **-qarch=auto**, **-qtune=auto**, and **-qcache=auto**.

Processor optimization capabilities

The AIX and Linux compilers target the full range of Power processors that those operating systems support. Both AIX and Linux compilers support the VMX vector instruction sets. XL C for AIX, V11,1, XL C/C++ for AIX, V11.1, XL C/C++ for Linux, V11.1, XL Fortran for AIX, V13.1, and XL Fortran for Linux, V13.1, and later compilers support VSX vector instruction sets available in chips such as POWER7® processors.

IBM Blue Gene is a family of supercomputers optimized for bandwidth, scalability and the ability to handle large amounts of data.

-qarch option

Using the correct **-qarch** suboption is the most important step in influencing chip-level optimization. The compiler uses the **-qarch** option to make both high and low-level optimization decisions and trade-offs. The **-qarch** 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.

-qarch instructs the compiler to structure your application to execute on a particular set of machines that support the specified instruction set. The **-qarch** option features suboptions that specify individual processors and suboptions that specify a family of processors with common instruction sets or subsets. The choice of processor gives you the flexibility of compiling your application to execute optimally on a particular machine or on a variety of machines, but still have as much architecture-specific optimization applied as possible.

For example, compiling applications with the XL compilers that will only run on POWER7 systems, use **-qarch=pwr7**. For compiling AIX applications that will only run on 64-bit mode capable hardware, use **-qarch=ppc64** to select the entire 64-bit PowerPC® and Power family of processors.

The default setting of **-qarch** at **-O0**, **-O2** and **-O3** selects the smallest subset of capabilities that all the processors have in common for the target operating system and compilation mode. The default for **-qarch** is platform dependent. On AIX, the default architecture setting in 32-bit mode is **ppc**, which selects a small instruction set common to all processors supported by AIX. This setting generates code that correctly executes on all supported AIX systems but will not exploit many common instruction subgroups more modern Power and PowerPC processors support. At levels **-O4** or **-O5** or if **-qarch=auto** is specified, the compiler will detect the type of machine on which you are compiling and assume that your application will execute on the same type of machine architecture.

-qtune option

The **-qtune** 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 **-qtune** setting depends on the setting of the **-qarch** option. If the **-qarch** option selects a particular machine architecture, the range of **-qtune** suboptions that are supported is limited by the chosen architecture, and the default tune setting will be compatible with the selected target processor. If instead, the **-qarch** option selects a family of processors, the range of values accepted for **-qtune** is expanded across that family, and the default is chosen from a commonly used machine in that family. Using **-qtune** allows the optimizer to perform transformations, such as instruction scheduling, so that resulting code executes most efficiently on your chosen **-qtune** architecture. Since the **-qtune** option tunes code to run on one particular processor architecture, it does not support suboptions representing families of processors.

-qtune=balanced is the default when the default **-qarch** setting is specified. **-qtune=balanced** instructs the compiler to tune generated code for optimal performance across a range of recent processor architectures, including POWER7.

Use **-qtune** to specify the most common or important processor where your application executes. For example, if your application usually executes on a POWER7-based system but sometimes executes on POWER6-based systems, specify **-qtune=pwr7**. The code generated executes more efficiently on POWER7-based systems but can run correctly on POWER6-based systems.

With the **-qtune=auto** suboption, which is the default for optimization levels **-O4** and **-O5**, the compiler detects the machine characteristics on which you are compiling, and tunes for that type of machine.

-qcache option

-qcache describes to the optimizer the memory cache layout for the machine where your application will execute. There are several suboptions you can specify to describe cache characteristics such as types of cache available, their sizes, and cache-miss penalties. If you do not specify **-qcache**, the compiler will make cache assumptions based on your **-qarch** and **-qtune** option settings. If you know some cache characteristics of the target machine, you can still specify them. With the **-qcache=auto** suboption, the default at**-O4** and **-O5**, the compiler detects the cache characteristics of the machine on which you are compiling and assumes you want to tune cache optimizations for that cache layout. If you are unsure of your cache layout, allow the compiler to choose appropriate defaults.

Source-level optimizations

The XL compiler family exposes hardware-level capabilities directly to you through source-level intrinsic functions, procedures, directives, and pragmas. XL compilers offer simple interfaces that you can use to access Power instructions that control low-level instruction functionality such as:

- Hardware cache prefetching/clearing/sync
- Access to FPSCR register (read and write)
- Arithmetic (e.g. FMA, converts, rotates, reciprocal SQRT)
- Compare-and-trap
- VMX and VSX vector data types and instructions (on AIX and Linux)
- QPX vector data types and instructions (on Blue Gene/Q)

The compiler inserts the requested instructions or instruction sequences for you, but is also able to perform optimizations using and modelling the instructions' behavior. For the XL C and C++ compilers, you can additionally insert arbitrary POWER® instruction sequences through the **mc_func** pragma or inline assembler facilities.

The vector instruction programming interface

Under the **-qaltivec** option, XL C and C++ compilers for AIX and Linux additionally support the AltiVec programming interfaces originally defined by the Mac OS X GCC compiler. This allows source-level recognition of the vector data type and the more than 100 intrinsic functions defined to manipulate vector data. These interfaces allow you to program source-level operations that manipulate vector data using the VMX or VSX facilities of PowerPC VMX processors such as POWER7. Vector capabilities allow your program to calculate arithmetic results on up to sixteen data items simultaneously.

XL Fortran for AIX and Linux also supports VMX and VSX instructions and a VECTOR data type to access vector programming interfaces.

On Blue Gene/Q platforms, new data types and intrinsic functions are introduced to support the QPX instructions. The QPX data type is of four double-precision floating-point values. It is automatically enabled when you compile your program for the Blue Gene/Q architecture.

The **-qsimd=auto** option enables automatic generation of vector instructions for processors that support them. It replaces the **-qenablevmx** option, which has been deprecated.

High-order transformation (HOT) loop optimization

The HOT optimizer is a specialized loop transformation optimizer. HOT optimizations are active by default at **-O4** and **-O5**, as well at a reduced intensity at **-O3**. You can also specify full HOT optimization at level **-O2** and **-O3** using the **-qhot** option. At **-O3**, basic HOT optimization is performed by default by using **-qhot=level=0**. Loops typically account for the majority 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.

HOT is especially adept at handling Fortran 90-style array language constructs and performs optimizations such as elimination of intermediate temporary variables and fusion of statements. HOT also recognizes opportunities in code compiled with XL C and C++ compilers.

In all three languages, you can use pragmas and directives to assist the HOT optimizer in loop analysis. Assertive directives such as INDEPENDENT or CNCALL allow you to describe important loop characteristics or behaviors that the HOT optimizer can exploit. Prescriptive directives such as UNROLL or PREFETCH allow you to direct the HOT optimizer's behavior on a loop-by-loop basis. You can additionally use the **-qreport** compiler option to generate information about loop transformations. The report can assist you in deciding where pragmas or directives can be applied to improve performance.

HOT short vectorization using VMX or VSX

IBM XL compilers for AIX and Linux support vector multimedia extension (VMX) and vector scalar extension (VSX) instructions. When targeting VMX and VSX Power processors such as POWER7, specifying **-qsimd=auto** allows the optimizer to transform code into VMX or VSX instructions. VMX and VSX machine instructions can execute up to sixteen operations in parallel. The most common opportunity for this transformation is with loops that iterate over contiguous array data performing calculations on each element. You can use the NOSIMD directive in Fortran or the equivalent pragma in C/C++ to prevent the transformation of a particular loop.

HOT long vectorization

By default, if you specify **-qhot** with no suboptions (or an option like **-O4** that implies **-qhot**), the **-qhot=vector** suboption is enabled. The **vector** suboption can optimize loops in source code for operations on array data by ensuring that operations run in parallel where applicable. The compiler uses standard machine registers for these transformations and does not restrict vector data size, supporting both single- and double-precision floating-point vectorization. Often, HOT vectorization involves transformations of loop calculations into calls to specialized mathematical routines supplied with the compiler through the MASS libraries. These mathematical routines use algorithms that calculate the results more efficiently than executing the original loop code. As HOT long vectorization does not use VMX or VSX Power instructions, it can be used on all types of systems.

HOT array size adjustment

An array dimension that is a power of two can lead to a decrease in cache utilization. The **-qhot=arraypad** suboption allows the compiler to increase the dimensions of arrays where doing so can improve the efficiency of array-processing loops. Using this suboption can reduce cache misses and page faults that slow your array processing programs. Padding will not necessarily occur for all arrays, and the compiler can pad different arrays by different amounts. You can specify a padding factor for all arrays, which would typically be a multiple of the largest array element size. Array padding should be done with discretion. The HOT optimizer does not check for situations where array data is overlaid, as with Fortran EQUIVALENCE or C union constructs, or with Fortran array reshaping operations.

HOT fast scalar math routines

The XLOPT library contains faster versions of certain math functions that are normally provided by the operating system or in the default runtime. With **-qhot=fastmath**, the compiler replaces calls to the math functions with their faster counterparts in XLOPT library. This option requires **-qstrict=nolibrary** to be in effect.

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 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.

Since the XL compiler family shares optimization technology, object files created using IPA on the compile step with the XL C, C++, and Fortran compilers can all be analyzed by IPA at link time. Where program analysis shows objects were built with compatible options, such as **-qnostrict**, IPA can perform transformations such as inlining C functions into Fortran code, or propagating C++ constant data into C function calls.

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 **-O2** or **-O3** 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 **-qipa** 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 **-qipa** 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 **-qipa** without a level, or **-O4**, IPA is run at level one. If you specify **-O5**, 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:

-qipa=level=0

- Automatic recognition of standard library functions such as ANSI C, and Fortran runtime routines
- Localization of statically bound variables and procedures
- Partitioning and layout of code according to call affinity. This expands the scope of the **-O2** and **-O3** low-level compilation unit optimizer

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

-qipa=level=1

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

-qipa=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 **-qipa** option has many other suboptions available for fine-tuning the optimizations applied to your program. There are several suboptions to control inlining including:

- How much to do
- Threshold levels
- Functions to always or never inline
- Other related actions

Other suboptions allow you to describe the characteristics of given subprograms to IPA — pure, safe, exits, isolated, low execution frequency, and others. The **-qipa=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 **-qipa=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 **-qipa=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 also reduce IPA optimization time using the **-qipa=threads** suboption. The threads suboption allows the IPA optimizer to run portions of the optimization process in parallel threads, which can speed up the compilation process on multiprocessor systems.

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. XL compilers use 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 first step in PDF optimization is to compile your application with the **-qpdf1** 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. XL compilers supply the **cleanpdf** and **resetpdf** tools that assist you in managing PDF data. The **mergepdf** utility can be useful when combining the results of different PDF runs and assigning relative weights to their importance in optimization analysis.

After you collect sufficient PDF data, recompile or simply relink your application with the **-qpdf2** option. The compiler reads the PDF data files and make the information available to all levels of optimization that are active. PDF optimization can be combined with other optimization techniques in XL compilers such as the standard **-O2** or **-O3** compilation unit optimizations, or the **-qhot**, or **-qipa** optimizations active at higher optimization levels.

In addition, **-qpdf=level=0|1|2** supports multiple-pass profiling, cache miss, block counter, call counter and extended value profiling. You can use **-qpdf1=exename** to specify the name of the generated PDF file according to the output file name specified with the **-o** parameter. You can also use **-qpdf=defname** to revert the PDF file to its default file name.

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.

Note: Profile-directed feedback (PDF) optimization is not supported on Blue Gene/Q.

Symmetric multiprocessing (SMP) optimizations

IBM XL C for AIX, V12.1, XL C/C++ for AIX, V12.1, XL C/C++ for Linux, V12.1, XL C/C++ for Blue Gene/Q, V12.1, XL Fortran for AIX, V14.1, XL Fortran for Linux, V14.1, and XL Fortran for Blue Gene/Q, V14.1 support the OpenMP API Version 3.1 specification. Using OpenMP allows you to write portable code compliant to the OpenMP parallel-programming standard and enables your application to run in parallel threads on SMP systems. OpenMP consists of a defined set of source-level pragmas, directives, and runtime function interfaces you can use to parallelize your application. The compilers include threadsafe libraries for OpenMP support, or for use with other SMP programming models.

The optimizer in XL compilers includes threadsafe optimizations specific to SMP programming and particular performance enhancements to the OpenMP standard. The **-qsmp** compiler option has many suboptions that you can use to guide the optimizer when analyzing SMP code. You can set additional SMP specific environment variables to tune the runtime behavior of your application in a way that maximizes the SMP capabilities of your hardware. For basic SMP functionality, the **-qsmp=noopt** suboption allows you to transform your application into an SMP application, but performs only the minimal transformations required in order to preserve maximum source-level debug information.

The **-qsmp=auto** suboption enables automatic transformation of normal sequentially-executing code into parallel-executing code. This suboption allows the optimizer to automatically exploit the SMP and shared memory parallelism available in IBM processors. By default, the compiler will attempt to parallelize explicitly coded loops as well as those that are generated by the compiler for array language. If you do not specify **-qsmp=auto**, or you specify **-qsmp=noopt**, automatic parallelization is turned off, and parallelization only occurs for constructs that you mark with prescriptive directives or pragmas.

The **-qsmp** compiler option additionally supports suboptions that allow you to guide the compiler's SMP transformations. These include suboptions that control transformations of nested parallel regions, use of recursive locks, and what task scheduling models to apply.

When using **-qsmp** or any other parallel-based programming model, you must invoke the compiler with one of the threadsafe (_r) variations of the compiler name. For example, rather than use **xlc** you must use **xlc_r**. **_r** tells the compiler to use an alternate set of compiler option defaults like **-qthreaded**. You can use the **_r** versions of the compiler even when you are not generating code that executes in parallel. However, especially for XL Fortran, code and libraries will be used in place of the sequential forms that are not always as efficient in a single-threaded execution mode.

IBM Mathematics Acceleration Subsystem (MASS) libraries

Starting with XL C/C++, V7.0 and XL Fortran, V9.1, these compiler products began shipping the IBM MASS libraries of mathematical intrinsic functions specifically tuned for optimum performance on Power architectures.

The MASS libraries include scalar, vector and SIMD functions. They are thread-safe, and support both 32-bit and 64-bit compilations, and offer improved performance. The MASS vector libraries contain intrinsic functions that can be used with C, C/C++ or Fortran applications.

The MASS scalar library, `libmass.a`, contains an accelerated set of frequently used math intrinsic functions in the AIX system library `libm.a`.

Table 1. Libraries included in the MASS library

Mass vector library	**Tuned for processor**
libmassv.a	
libmassvp7.a	POWER7
libmassvp6.a	POWER6®
libmassvp5.a	POWER5
libmassvp4.a	POWER4
libmassvp3.a	POWER3

`libmassv.a`, contains vector functions that will run on all models in the IBM Power Systems family, while `libmassvp3.a` and `libmassvp4.a` each contain a subset of `libmassv.a` functions that have been specifically tuned for the POWER3 and POWER4 processors, respectively. `libmassvp5.a` contains functions that have been tuned for POWER5, `libmassvp6.a` contains functions tuned for POWER6, and `libmassvp7.a` contains functions tuned for POWER7.

The MASS SIMD library `libmass_simdp7.a` or `libmass_simdp7_64.a` contains a set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions.

Basic Linear Algebra Subprograms (BLAS)

BLAS is a set of high-performance algebraic functions. Four BLAS functions are shipped with each XL compiler in the `libxlopt` library:

- `sgemv` (single-precision) and `dgemv` (double-precision), which compute the matrix-vector product for a general matrix or its transpose
- `sgemm` (single-precision) and `dgemm` (double-precision), which perform combined matrix multiplication and addition for general matrices or their transposes

Aliasing

The apparent affects of direct or indirect memory access can often constrain the precision of compiler analyses. 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 analyses possible aliases at all optimization levels, but analysis of these apparent references is best when using the **-qipa** option. Options such as **-qalias** and directives or pragmas such as CNCALL and INDEPENDENT can fundamentally improve the precision of compiler analysis.

IBM XL compiler 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.

XL compilers supply 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 **-qalias** option for all three XL compiler languages to assert whether your application follows aliasing rules. For Fortran and C++, standards-conformant program aliasing is the default assumption. For the C compiler, the invocations that begin with "xlc" assume conformance, and the invocations that begin with "cc" do not. The **-qalias** option has suboptions that vary by language. Suboptions exist for both the purpose of specifying that your program has nonstandard aliasing, and for asserting to the compiler that your program exceeds the aliasing requirements of the language standard. The latter set of suboptions can remove barriers to optimization that the compiler must assume due to language rules. For example, in the XL C/C++ compiler, specifying **-qalias=typeptr** allows the optimizer to assume that pointers to different types never point to the same or overlapping storage. For the XL C compiler, 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.

Additional performance options

In addition to the options already introduced, the XL compilers have many other options that you can use to direct the optimizer. Some of these are specific to individual XL compilers rather than the entire XL compiler family. Those options that apply to all languages have their name followed by (all).

Optimizer guidance options

-qcompact (all)
Default is **-qnocompact**. Prefers final code size reduction over execution time performance when a choice is necessary. Can be useful as a way to constrain the **-O3** and higher optimization levels.

-ma (C only)
The compiler will generate inline code for calls to the `alloca` library function.

-qinline (all)
Attempts to inline procedures instead of generating calls to those procedures, for improved performance. Its suboption **level=***number* provides guidance to the compiler about the relative value of inlining. The values you specify for *number* range from 0 and 10 inclusive. 0 means no inlining and 10 means maximum inlining. The default value is 5.

-qminimaltoc (all)
Default is **-qnominimaltoc**. Controls the generation of the table of contents (TOC). **-qminimaltoc** does not need to be applied to the whole program.

-qprefetch (all)
Instructs the compiler to insert prefetch instructions automatically where there are opportunities to improve code performance.

-qro,-qroconst (C, C++)
Directs the compiler to place string literals (**-qro**), or constant values in read-only storage (**-qroconst**).

-qsmallstack (all)
Default is **-qnosmallstack**. Instructs the compiler to minimize the use of stack (automatic) storage where possible; doing so can increase heap (dynamically allocated) usage.

-qunroll (all)
Default is **-qnounroll** (unless optimizing). Independently controls loop unrolling. It is turned on implicitly with any optimization level higher than **-O0**. You can specify suboptions that determine the aggressiveness of automatic loop unrolling.

Program behavior options

-qaggrcopy=ovrlap|noovrlap (C, C++)
Specifies whether aggregate assignments may have overlapping source and target locations. Default is **ovrlap** with **cc** compiler invocations, **noovrlap** with **xlc** and **xlC** compiler invocations.

-qassert (all)
Provides information about the characteristics of your code that can help the compiler fine-tune optimizations. For example, for the XL Fortran compiler, the **deps** suboption indicates that at least one loop has a memory dependence or conflict from iteration to iteration. For improved performance, try **-qassert=nodeps** when no loops in the compilation unit carry a dependence around loop iterations. The **itercnt** suboption modifies the default assumptions about the expected iteration count of loops; normally the optimizer will assume ten iterations for a typical loop.

-qnoeh (C++)
Default is **-qeh**. Asserts that no throw is reachable from the code compiled in this compilation unit. Using this option can improve execution speed and reduce code footprint where the code has no C++ exception handling.

-qignerrno (C, C++)
Default is **-qnoignerrno**. For **-O3** and up, default is **-qignerrno**. Indicates that the value of errno is not needed by the program. Can help optimization of math functions that may set `errno`, such as `sqrt`.

-qlibansi (all)
Default is **-qnolibansi**. Specifies that calls to ANSI standard function names will be bound with conforming implementations. Allows the compiler to replace the calls with more efficient inline code or at least do better call-site analysis.

-qproto (C)
Asserts that procedure call points agree with their declarations even if the procedure has not been prototyped. Useful for well-behaved K&R C code.

-qnounwind (all)
Default is **-qunwind**. Asserts that the stack will not be unwound in such a way that register values must be accurately restored at call points. Most Fortran applications can use **-qnounwind**, which allows the compiler to be more aggressive in eliminating register saves and restores at call points.

Floating-point computation options

The **-qfloat** option provides precise control over the handling of floating-point calculations. XL compiler default options result in code that is *almost* IEEE 754 compliant. Where the compiler generates non-compliant code, it is allowed to exploit certain optimizations such as floating-point constant folding, or to use efficient Power instructions that combine operations. You can use **-qfloat** to prohibit these optimizations. Some of the most frequently applicable **-qfloat** suboptions:

[no]fenv
Specifies whether the code depends on the hardware environment and whether to suppress optimizations that could cause unexpected results due to this dependency. When **-qnofenv** is in effect, the compiler assumes that the program does not depend on the hardware environment, and that aggressive compiler optimizations are allowed.

[no]fold
Enables compile time evaluation of floating-point calculations. You may need to disable folding if your application must handle certain floating-point exceptions such as overflow or inexact.

[no]maf
Enables generation of combined multiple-add instructions. In some cases you must disable **maf** instructions to produce results identical to those on non-PowerPC systems. Disabling **maf** instructions can result in significantly slower code.

[no]rrm
Specifies that the rounding mode is not always round-to-nearest. The default is **norrm**. The rounding mode can also change across calls.

[no]rsqrt
Speeds up some calculations by replacing division by the result of a square root with multiplication by the reciprocal of the square root.

[no]single
Allows single-precision arithmetic instructions to be generated for single-precision floating-point values. If you wish to preserve the behavior of applications compiled for earlier architectures, where floating-point arithmetic was performed in double-precision and then truncated to single-precision, use **-qfloat=nosingle:norndsngl**.

[no]rngchk
Specifies whether range checking is performed for input arguments for software divide and inlined square root operations. **norngchk** instructs the compiler to skip range checking, allowing for increased performance where division and square root operations are performed repeatedly within a loop.

[no]hscmplx
Speeds up operations involving complex division and complex absolute value. **nohscmplx** is the default.

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.

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

-qlistfmt
Generates reports in HTML and XML formats. Compiler reports contain information about optimizations performed by the compiler and missed optimization opportunities. These reports contain information about some optimizations performed by the compiler and some missed optimization opportunities for inlining, loop transformations, data reorganization and profile-directed feedback. This information can be used to understand the application code and to tune the code for better performance.

-qreport
-qreport=[hotlist | smplist] in XL Fortran, or **-qreport** in XL C/C++. Instructs the HOT or IPA optimizer to emit a report including pseudocode along with annotations describing what transformations, such as loop unrolling or automatic parallelization, were performed. Includes data dependence and other information such as program constructs that inhibit optimization.

-S Invokes the disassembly tool supplied with the compiler on the object file(s) produce by the compiler. This produces an assembler file that is compatible with the system assembler.

User-directed source-level optimizations

XL compilers support many source-level directives and pragmas that you can specify to influence the optimizer. Several have been mentioned in previous sections. The following two sections contain an important subset that XL compilers support, including a brief description of each Fortran directive and C/C++ pragma.

XL Fortran directives

ASSERT (ITERCNT(*n*) | [NO]DEPS)
Identical behavior to the **-qassert** option but applicable to a single loop. Allows the characteristics of each loop to be analyzed by the optimizer independently of the other loops in the program.

CACHE_ZERO
Zeros the data cache block for a variable or list of variables using the dcbz hardware instruction. This avoids having to store the zero values back to main memory preventing level 2 store cache misses.

CNCALL
Asserts that the calls in the following loop do not cause loop-carried dependences.

INDEPENDENT
Asserts that the following loop has *no* loop-carried dependences. Enables locality and parallel transformations.

PERMUTATION (*names*)
Asserts that elements of the named arrays take on distinct values on each iteration of the following loop. This is useful with sparse data.

PREFETCH_BY_LOAD (*variable_list*)
Issues dummy loads that cause the given variables to be prefetched into cache. This is useful to activate hardware prefetch.

PREFETCH_FOR_LOAD (*variable_list***)**
Issues a *dcbt* instruction for each of the given variables.

PREFETCH_FOR_STORE (*variable_list*)
Issue a *dcbtst* instruction for each of the given variables.

UNROLL
Specified as **[NO]UNROLL [(*n*)]** to turn loop unrolling on or off. You can specify a specific unroll factor.

XL C/C++ pragmas

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

execution_frequency (very_low | very_high)
Asserts that the control path containing the pragma will be infrequently executed.

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 [(*n*)]** to turn loop unrolling on or off. You can specify a specific unroll factor.

Summary

The IBM XL compiler family offers premier optimization capabilities on AIX and Linux, and Blue Gene platforms. You can control the type and depth of optimization analysis through compiler options, which allow you choose the levels and kinds of optimization 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), symmetric multiprocessing (SMP) optimizations, as well as a unique set of Power optimizations that fully 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.

Trial versions and purchasing

You can download trial versions of the XL compilers for AIX and Linux at these IBM Web sites:

- www.ibm.com/software/awdtools/xlcpp/
- www.ibm.com/software/awdtools/fortran/

Information on how to purchase the XL compilers is also available at the Web sites above.

Contacting IBM

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

June 2012

References in this document to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.