



# Code Optimization with IBM XL Compilers

June 2004

## Introduction

The IBM XL compiler family offers C, C++, and Fortran compilers built on an industry wide reputation for robustness, versatility and standards compliance. However, the XL compilers are about more than providing reliable, feature-rich functionality. The true strength of the compilers is in optimization, the ability to improve code generation, and produce fast-executing code on PowerPC™-based systems. Optimized code executes faster, using less machine resources, and makes you and your PowerPC systems more productive.

For example, consider the advantages of a compiler that properly exploits the inherent opportunities in the PowerPC architecture. A set of complex calculations that could take hours using unoptimized code, may be reduced to mere minutes when heavily 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 your development environment.

This document does not contain an exhaustive list of the many optimization capabilities of the XL compiler family. It 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

IBM's XL compiler family encompasses three core languages for three major operating systems on IBM PowerPC hardware.

Operating System and Machine Architecture	XL C/C++ Products	XL Fortran Product
Mac OS X on Apple G4 and G5 systems	XL C/C++ Advanced Edition	XL Fortran Advanced Edition
AIX™ on pSeries	C for AIX, VisualAge C++ for AIX	XL Fortran for AIX
Linux on pSeries and	VisualAge C/C++ for Linux	XL Fortran for Linux

iSeries		
---------	--	--

This document describes the optimizations available in the V8.1 XL Fortran and the V6.0 XL C and C++ products.

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

<http://www.ibm.com/software/awdtools/fortran/>  
<http://www.ibm.com/software/awdtools/ccompilers/>

Those web sites also contain information on contacting IBM if you have questions about the XL compiler family or this document. You may also contact the compiler team by sending e-mail to [compinfo@ca.ibm.com](mailto:compinfo@ca.ibm.com).

## XL Compiler History

The XL family of compilers is based on work at IBM that began in the mid-1980s with development of the IBM's 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 the IBM POWER and PowerPC range of systems.

The compiler design team at IBM works closely with the hardware design and operating system development teams. This ensures that the compilers take advantage of all the latest hardware and OS capabilities, and also that the compiler team can influence hardware and OS design to create performance-enhancing capabilities.

The XL family of compilers build performance-critical customer code every day and are key to the success of many of IBM's most performance-sensitive products such as AIX and DB2. IBM also relies on the XL family of compilers for producing industry-leading performance benchmarks results, such as SPEC.

The XL compiler family has evolved into a set of highly standards-compliant compilers. All core language standards for all three languages are supported, with some additional draft standard features also available. The XL compilers also support many important industry, IBM, and gcc/g++ language extensions such as the OpenMP Symmetric Multi-Processing (SMP) standard.

## 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 run-time libraries to support capabilities such as SMP and high-performance complex mathematical calculations.

Sharing common components ensures that performance-enhancing optimization techniques available to you in one language are available to you in all of them. 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 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 via IPA
- Profile-Directed Feedback (PDF) optimization
- User-directed optimization via directives and source-level intrinsic functions that give you direct access to PowerPC instructions
- Optimization of OpenMP programs and auto-parallelization capabilities to exploit SMP systems

## Optimization Levels

The optimizer includes five base optimization levels:

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

Note that the **-O1** level is not supported. Each of the optimization levels will be discussed in detail in subsequent sections of this document.

### ***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 to the optimization process as the **-O** level you choose.

The **Optimization Levels and Options** table below offers some options as additional optimization techniques. The table details compiler options implied by using a base optimization level; options you should use with each level; and some additional options that could be useful with each optimization level.

## Optimization Levels and Options

Base Optimization Level	Additional Options Implied by Base Optimization Level	Additional Recommended Options	Additional Options to Try with Base Optimization Level
<b>-O0</b>	None	<b>-qarch</b>	<b>-g</b>
<b>-O2</b>	<b>-qmaxmem=2048</b>	<b>-qarch</b> <b>-qtune</b>	<b>-qmaxmem=-1</b> <b>-qhot</b> <b>-qnostrict</b>
<b>-O3</b>	<b>-qnostrict</b> <b>-qmaxmem=-1</b>	<b>-qarch</b> <b>-qtune</b>	<b>-qhot</b> <b>-qpdf</b>
<b>-O4</b>	All of <b>-O3</b> plus: <b>-qhot</b> <b>-qipa</b> <b>-qarch=auto</b> <b>-qtune=auto</b> <b>-qcache=auto</b>	<b>-qarch</b> <b>-qtune</b> <b>-qcache</b>	<b>-qpdf</b>
<b>-O5</b>	All of <b>-O4</b> plus: <b>-qipa=level=2</b>	<b>-qarch</b> <b>-qtune</b> <b>-qcache</b>	<b>-qpdf</b>

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 **-qsmp=auto** can additionally be used with the base optimization levels if you desire automatic parallelization of your application.

Compilation time can also be a consideration when developing an application. In general, higher optimization levels take additional compilation time and resources. Specifying additional optimization options beyond the base optimization levels can also increase compilation time. For a large application with a long build time, compilation time can be an issue when choosing the appropriate optimization level and options for your build.

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**. The optimizer is sensitive to language standard rules. Even subtly non-conforming code can cause the optimizer to perform incorrect transformations, especially at the highest optimization levels. All of the XL compilers have options to limit optimizations for nonconforming code, but it is best practice to correct the code and not limit optimization opportunities.

### **Optimization Level 0**

At **-O0**, the XL compilers limit optimization transformations to your applications. Some limited optimization occurs at **-O0** even if you specify no other optimization levels or

options. However, limited optimization analysis generally results in the quicker compile times than other optimization levels.

This is the best level to use when debugging your code with a symbolic debugger. Specify **-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.

## **Optimization Level 2**

**-O2** performs a set of comprehensive low-level transformations generally limited to subprogram or compilation unit scopes with the possibility of some inlining. Optimization level 2 is the default setting of the **-O** compiler option, using **-O** implies **-O2**. This optimization level attempts to limit compilation time and resource requirements while performing as much low-level optimization as possible under those time and resource constraints. You can increase the memory resources available to some of the level 2 optimizations by providing a larger value to the **-qmaxmem** option than the default of 2048. The value of -1 makes unlimited memory available.

Optimization level 2 performs many transformations. These are some of the most important, and generally most applicable to application development:

- Global assignment of user variables to registers, also known as *graph coloring register allocation*.
- Strength reduction and effective use of addressing modes.
- Elimination of redundant instructions, also known as *common subexpression elimination*
- Elimination of instructions whose results are unused or that cannot be reached by a specified control flow, also known as *dead code elimination*.
- Value numbering (algebraic simplification).
- Movement of invariant code out of loops.
- Compile-time evaluation of constant expressions, also known as *constant propagation*.
- Control flow simplification.
- Instruction scheduling (reordering) for the target machine.
- Loop unrolling and software pipelining
- Elimination of unnecessary variable assignments, also known as *dead store elimination*.

Optimization level 2 preserves some limited debug information when you specify **-g**, and is the best choice for debugging optimized code. Optimization levels higher than **-O2** usually result in less accurate debug information being available to a debugger. However, along with **-g**, additional compiler options like **-qkeeparm** or directives (pragmas in C/C++) such as **SNAPSHOT** can assist in debugging optimized code at any optimization level.

## The **-qarch** and **-qtune** options at **-O2** and higher

The **-qarch** and **-qtune** options can be important to **-O2** and higher optimization levels. Choosing a hardware architecture target or family of targets allows the optimizer to make the best use of the hardware facilities available. 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. This allows you to compile for a general set of targets but have the code run best on a particular target. Details on the **-qarch** and **-qtune** options can be found in subsequent sections of this document.

## **Optimization Level 3**

The **-O3** optimization level is an intensified version of **-O2**. The compiler performs additional low-level transformations and removes limits on the optimization level 2 transformations, as **-qmaxmem** defaults to the -1 (unlimited) value. Optimizations encompass larger program regions and deepen to attempt more analysis. This level also performs 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.
- Increased optimization scope, typically to encompass a whole procedure.
- Specialized optimizations (those that might not help all programs).
- Optimizations that require large amounts of compile time or space.
- Implicit memory usage limits are eliminated.
- Implies **-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).

You can still gain most of the benefits of **-O3** and preserve 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 may also need **-qstrict** if your application is sensitive to floating-point exceptions or the order and manner in which floating-point arithmetic is evaluated is important. Generally, 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.

## Optimization Level 4

Optimization level 4 is an easy way of specifying **-O3** along with several other complementary optimization options. The most important of the additional options is **-qipa**. IPA optimization extends program analysis beyond individual files and compilation units to the entire application. IPA analysis can propagate values between compilation units 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. The **-O4** optimization level invokes IPA at the default IPA optimization level, level 1.

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** is the highest base optimization level and includes all **-O4** optimizations as well as running the IPA optimizer at level 2 instead of the default level 1. 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 it once your application is debugged and known to work at lower optimization levels.

**-O5**, as with **-O4**, implies **-qarch=auto**, **-qtune=auto**, and **-qcache=auto**. You may need to alter those defaults if the machine you are compiling on is not where you will execute the application.

## PowerPC Optimization Capabilities

The XL compiler family can fully exploit the potential of the PowerPC and related POWER series processors. Using an XL compiler, you can target any processor supported by the operating system on which you can use an XL compiler.

The AIX compilers target the full range of processors that AIX supports and the Linux compilers support range of PowerPC processors supported by the Linux pSeries and iSeries distributions. The Mac OS X compilers target the two latest processors available to Apple customers, the G4 and G5.

### **-qarch Option**

Using the correct **-qarch** suboption is the most important step in influencing chip-level optimization. The compiler uses that option to make both high- and low-level optimization decisions and trade-offs. It allows the compiler access to the full range of processor hardware instructions and capabilities when making code generation decisions. Even at low optimization levels such as **-O0** or **-O2**, specifying the correct target architecture can have a positive impact on performance. It allows the compiler to select more efficient machine instructions and generate instruction sequences that will perform best for a particular machine.

You must use **-qarch** correctly because it instructs the compiler to structure your application to execution 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 to be able to execute on a wider variety of machines, but still have as much architecture-specific optimization applied as possible. For example, an AIX or Linux compiler that is building an application that will only run on POWER4-based machines should use the **-qarch=pwr4** option. However, a Linux compiler that is building applications that will only run on 64-bit mode capable hardware but does not know which machines in particular, should use the **-qarch=ppc64** option to select the entire 64-bit PowerPC family of processors.

The default setting of **-qarch** at all optimization levels except **-O4** or **-O5**, selects the smallest subset of capabilities that all the processors have in common for that target operating system and compilation mode. For example, on Mac OS X, the default architecture setting is **ppcv** which selects a generic PowerPC implementation that includes the Apple Velocity Engine (AltiVec) instruction set. This setting generates code that correctly executes on G4 or G5 processors but will not exploit specific characteristics of the G5 chip. At optimization 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 your application will execute on that 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 compiler generates machine instructions consistent with your **-qarch** architecture choice. Using **-qtune** additionally allows the optimizer to perform transformations, such as instruction scheduling, so that the resulting code executes most efficiently on your chosen **-qtune** architecture. Since the **-qtune** option tunes code to run on one particular processor architecture, it supports specific processors as suboptions and does not support suboptions representing families of processors.

You should use **-qtune** to specify the most common or important processor where your application will execute. For example, if your AIX or Linux application will usually execute on a POWER4-based systems but will sometimes execute on POWER3-based systems, specify **-qtune=pwr4**. The code generated will execute more efficiently on POWER4-based systems but will still run correctly on Power3-based systems.

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, 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. 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 assumes you want to tune for that type of machine.

## ***-qcache Option***

This option allows you to describe 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, but not all of the cache characteristics of your target machine, specify the characteristics that you know. With the **-qcache=auto** suboption, the default at optimization levels **-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.

## ***Source-Level PowerPC 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 PowerPC 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

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 PowerPC instruction sequences through the **mc\_func** pragma.

## VMX Vector Instruction Programming Interface

Under the **-qaltivec** option, the Mac OS X C and C++ compilers additionally support the AltiVec programming interfaces available in 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 facilities of the Apple PowerPC processors. AltiVec vector capabilities allow your program to calculate arithmetic results on up to sixteen data items simultaneously.

## High-Order Transformation (HOT) Loop Optimization

The HOT optimizer in the XL compilers is a specialized loop transformation optimizer. HOT optimizations are activated by default at optimization levels **-O4** and **-O5**, but can be specified with levels **-O2** or **-O3** using the **-qhot** option. 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. While HOT works well with Fortran constructs, it also recognizes opportunities in C and C++ code compiled with the 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.

In addition to general loop transformation, the **-qhot** option supports suboptions that you can specify to enable particular transformations. These optimization techniques are discussed below.

### ***HOT 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. These mathematical routines use algorithms that calculate the results more efficiently than executing the original loop code. HOT vectorization does not use AltiVec/VMX PowerPC instructions and so it can even be used 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 could improve the efficiency of array-processing loops. Using this suboption can reduce cache misses and page faults that slow your array processing programs. Not all arrays will necessarily be padded, and the compiler can pad different arrays by different amounts, making the correct decisions to increase your application's performance. You can specify a padding factor to apply to 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.

## **Interprocedural Analysis (IPA) Optimization**

The IPA optimizer's primary purpose is whole-program analysis and optimization. IPA analyzes the entire program rather than a program on a file-by-file basis while running during the link step of an application build. On the link step, 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 the application, performing optimizations such as inlining between compilation units. Complex data flow analyses are performed 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 step 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. IPA's primary focus is link-step optimization but using the IPA optimizer only on the compile-step can still be very beneficial to your application.

Because 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 that 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 lower-level compilation-unit 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, and an executable program is created.

It is important that you specify a set of compilation options as consistent as possible when compiling and linking your application. This advice 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 is 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 you specify **-O4**, IPA is run at level one. If you specify **-O5**, IPA is run at level two. There is also a level zero which can lower compilation time, but performs a more limited analysis. Below are some of the important IPA transformations performed at each level.

### **-qipa=level=0**

- Performs automatic recognition of standard library functions (e.g. ANSI C, Fortran run-time)
- Localization of statically bound variables and procedures
- Partitioning and layout of code according to call affinity - expands the scope of the **-O2/-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), 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, and 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 such as program partitioning and object reference maps.

An important suboption that can speed compilation time is **-qipa=noobject**. You can lower 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 and this will create object files on the compile step that only the IPA link-time optimizer can use. The object files will be created more quickly because the low-level compilation unit optimizer is not invoked on the compile step when the **noobject** suboption is specified.

On AIX and Linux systems, you may also be able to reduce IPA optimization time by 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 multi-processor systems.

## Profile-Directed Feedback (PDF) Optimization

PDF is an optimization that is applied 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 the information collected. PDF is used by the XL compiler optimizer to discover information about your application such as the locations of heavily used or infrequently used blocks of code. Knowing the relative execution frequency of code provides opportunities to the optimizer to bias execution paths in favor of heavily used code. Program restructuring can also be performed 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 used to collect the PDF information be characteristic of the data your application will typically see. Using atypical data or insufficient data can lead to a false picture of the program and suboptimal program transformation. If you do not have sufficient data, you should not use PDF optimization.

The first step in PDF optimization is to compile your application with the **-qpdf1** option. Doing so will instrument your code with calls to a PDF run-time that will be linked with your program. You then need to execute your application with typical input data. You can do so as many times as you wish with as many data sets as you have. Each run will record information in data files. XL compilers supply the *cleanpdf* and *resetpdf* tools that assist you in managing PDF data.

After sufficient PDF data is collected, recompile your application with the **-qpdf2** option. The compiler will read 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 the XL compilers such as the standard **-O2** or **-O3** compilation unit optimizations, or the **-qhot**, or **-qipa** optimizations active at higher optimization levels.

The PDF optimization is most effective when applied 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.

## Symmetric Multiprocessing (SMP) Optimizations

*Note: Version 8 of the XL Fortran compiler on Mac OS X and Version 6 of the XL C/C++ compiler on Mac OS X provide SMP programming and optimization as a technology preview only. SMP is fully supported on other XL compiler platforms.*

The XL compilers support the OpenMP standard applicable to each language. Using OpenMP allows you to write portable code that is 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 run-time function interfaces you can use in your program to parallelize your application. The compilers include threadsafe libraries in support of OpenMP or for use with other SMP programming models.

The optimizer in the 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 run-time behavior of your application to maximize the SMP capabilities of your hardware. For basic SMP functionality, the **-qsmp=noopt** suboption allows you to transform an 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 is a powerful tool that enables automatic transformation of normal sequentially-executing code into parallel-executing code. Using 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 Fortran array language. If you do not specify **-qsmp=auto** or you specify **-qsmp=noopt**, automatic parallelization is turned off, and only constructs that are marked with prescriptive directives/pragmas are parallelized.

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 “\_r” (re-entrant) variations of the compiler name. For example, rather than use **xlc** or **xlF90** you must use **xlc\_r** or **xlF90\_r**. The different name signals to the compiler that you need to use an alternate set of compiler option defaults (e.g. **-qthreaded** is specified for you) and the correct set of threadsafe libraries will be used for linking the application. 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.

## Aliasing

The precision of compiler analyses is constrained by the apparent effects of direct or indirect memory access. 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 does analysis of 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 have a pervasive effect since they fundamentally improve the precision of compiler analysis.

Each language has its own rules as to what constitutes a valid or an invalid alias. Fortran 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 are applied, and the more likely the optimizer will be misled.

The XL compilers supply options that you can use to optimize programs with non-standard aliasing constructs. Using these options can result in poor-quality aliasing information being made available to the optimizer, and less than optimal code performance. It is recommended that you alter your source code where possible to conform to language rules. Doing so will make your code both more portable and more optimizable.

The **-qalias** option can be used for all three XL compiler languages to assert whether your application follows aliasing rules. For Fortran and C++, standards-conformant program aliasing is assumed by default. 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 non-standard 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. This exposes additional optimization opportunities because the optimizer performs more precise aliasing analysis in code involving pointers.

## Additional Performance Options

In addition to the compiler 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 and this is noted in the text.

### *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, C++)
  - The compiler will generate inline code for calls to the **alloca** function.
- **-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 stack (automatic) storage where possible; doing so may increase heap usage.
- **-qnostrict**: (all)
  - Default is **-qstrict** with **-O0** and **-O2**, **-qnostrict** with **-O3**, **-O4**, and **-O5**
  - **-qnostrict** allows the compiler to reorder floating-point calculations and instructions that can potentially cause exceptions.
  - Do not specify **-qstrict** unless your application relies on absolutely precise IEEE floating-point compliant results or the order of floating-point computations.
- **-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={overlap|nooverlap}**: (C, C++)
  - Specify whether aggregate assignments may have overlapping source and target locations.
  - Default is **overlap** with **cc** compiler invocations, **nooverlap** with **xlc** and **xlc** compiler invocations.

- **-qassert:** (all)
  - Default is **-qassert=deps:itercnt=10**
  - **deps** 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.
  - **itercnt** 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 being compiled in this compilation unit.
  - Using it can improve execution speed and reduce code footprint where the code has no C++ exception handling.
- **-qignerrno:** (C, C++)
  - Default is **-qnoignerrno**, for **-O3** or higher, the default is **-qignerrno**
  - Indicates that the value of errno is not needed by the program.
  - Can help in optimization of math functions that potentially set errno, such as sqrt.
- **-qlibansi:** (all)
  - Default is **-qno libansi**
  - 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.
  - Not needed when linking with **-qipa**.
- **-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** thus allowing the compiler to be more aggressive in eliminating register saves/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 non-compliant code is generated, it is to allow the compiler the freedom to exploit certain optimizations such as floating-point constant folding or to use efficient PowerPC instructions that combine operations. You can use the **-qfloat** option to prohibit these optimizations. Specify **-qfloat=subopt** where *subopt* is one of:

- **[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. You may need to disable maf instructions to produce results identical to those produced by non-PowerPC systems. Disabling maf instructions can result in significantly slower code.
- **[no]rrm:**
  - Specifies that rounding mode may not be round-to-nearest (default is **norm**) or may change across calls.
- **[no]rsqrt:**
  - Allows computation of a divide by square root to be replaced by a multiply of the reciprocal square root.

Note that AIX XL compilers which target older POWER and POWER2™ chipsets support additional **-qfloat** options.

## ***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.
- **-qreport:**
  - Specified as **-qreport=[hotlist | smplist]**.
  - Instructs the HOT or IPA optimizer to emit a report including pseudo-code 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.
  - The **-S** option is not supported with the Mac OS X XL compilers.

## **User-Directed Source-Level Optimizations**

The XL compilers support many source-level directives and pragmas that you can specify to influence the optimizer. Several have been mentioned in the previous sections. Following is an important subset of those that the XL compiler supports along with a

brief description of the purpose of each. Fortran uses directives to express these and C/C++ uses pragmas.

## **Fortran Directives**

- **ASSERT ( ITERCNT(*n*) | [NO]DEPS )**
  - Same behavior as the **-qassert** option but applicable to a single loop. This is much more useful because the characteristics of each loop can be described to the optimizer independently of the other loops in the program.
- **CACHE\_ZERO**
  - Inserts a *dcbz* (data cache block zero) instruction at the given address.
  - Useful when storing to contiguous storage as it avoids the level 2 cache store miss entirely.
- **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 - useful with sparse data.
- **PREFETCH**
  - **PREFETCH\_BY\_LOAD (*variable\_list*)**: issue dummy loads to cause the given variables to be prefetched into cache - useful to activate hardware prefetch.
  - **PREFETCH\_FOR\_LOAD (*variable\_list*)**: issue 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*)]**
  - Used to activate/deactivate compiler unrolling for the following loop.
  - Can be used to give a specific unroll factor.

## **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*)**
  - Asserts that the control path containing the pragma will be infrequently executed.
- **independent\_calls**
  - Equivalent to **CNCALL** in Fortran.

- **independent\_loop**
  - Equivalent to **INDEPENDENT** in Fortran.
- **isolated\_call (function\_list)**
  - Asserts that calls to the named functions do not have side effects.
- **iterations**
  - Equivalent to **ASSERT(ITERCNT)** in Fortran.
- **leaves (function\_list)**
  - Asserts that calls to the named functions will not return (e.g. exit).
- **permutation**
  - Equivalent to **PERMUTATION** in Fortran.
- **sequential\_loop**
  - Directs the compiler to execute the following loop in a single thread, even if the **-qsmp=auto** option is specified.

## Optimization-Friendly Programming

There are many ways that you can assist the XL compiler optimizer in maximizing the performance of your application that do not involve the use of compiler options or source-level directives. The following are some useful tips and techniques.

### *General Programming Tips*

- Use register-sized integers for scalars. In 32-bit mode, use 4-byte integers. In 64-bit mode, use 8-byte integers. However, for large arrays of integers, consider using 1- or 2-byte integers.
- Use the smallest floating-point precision appropriate to your computation. Use 16-byte floating-point or 32-byte complex data types only when you require extremely high precision.
- Obey all language aliasing rules. Try to avoid **-qassert=nostd** in Fortran and **-qalias=noansi** in C/C++.
- Use local variables wherever possible for loop index variables and bounds. In C/C++, avoid taking the address of loop indices and bounds.
- Keep array index expressions as simple as possible. Where indexing needs to be indirect, consider using the **PERMUTATION** directive/pragma.

### *Fortran Programming Tips*

- Use the **xlf90** or **xlf95** driver invocations where possible to ensure portability and maximize the use of automatic storage. If this is not possible, consider using the **-qnosave** option.
- When writing new code, use module variables rather than common blocks for global storage.
- Use modules to group related subroutines and functions.
- Use **INTENT** to describe usage of parameters.

- Limit the use of **ALLOCATABLE** variables and **POINTER** variables to situations which demand dynamic allocation.
- Use **CONTAINS** only to share thread local storage.
- Avoid the use of **-qalias=nostd** by obeying Fortran aliasing rules.
- When using array assignment or **WHERE** statements, pay close attention to the generated code with **-qlist** or **-qreport**. If performance is inadequate, consider using **-qhot** or rewriting array language in loop form.

## ***C/C++ Programming Tips***

- For C, use the **xlc** or **xlc\_r** invocations rather than **cc** or **cc\_r** when possible.
- Always include *string.h* when doing string operations and *math.h* when using the math library.
- Pass large class and structure parameters by address or reference, pass everything else by value where possible.
- Use unions and pointer type-casting only when necessary and try to follow ANSI type rules.
- If a class or structure contains a **double** variable, consider putting it first in the declaration. If this is not possible, consider using **-qalign=natural**.
- In C++, avoid virtual functions and virtual inheritance unless required for class extensibility. These are costly in object space and function invocation performance.
- Use **volatile** only for truly shared variables.
- Use **const** for globals, parameters and functions whenever possible.
- Do limited hand-tuning of small functions by defining them as **inline** in a header file.

## **Summary**

The IBM XL compiler family offers premier optimization capabilities on the AIX, Linux, and Mac OS X PowerPC 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 like Interprocedural Analysis (IPA), High-Order loop Transformations (HOT), Profile-Directed Feedback (PDF), Symmetric-Multiprocessing (SMP) optimizations, as well as a unique set of PowerPC optimizations that fully exploit the hardware architecture's capabilities. This optimization strength combines with the XL compiler family's robustness, capability, and standards conformance on multiple hardware and operating system platforms to produce a product set unmatched in the industry.

## ***Trial Versions and Purchasing***

You can download trial versions of the XL compilers for Mac OS X, Linux, and AIX at these IBM web sites:

<http://www.ibm.com/software/awdtools/fortran/xlfortran>

<http://www.ibm.com/software/awdtools/ccompilers>

Information on how to buy these compilers is also available at the above web sites. You can also obtain the Mac OS X versions of the XL compilers from Absoft© Corporation, an IBM OEM partner, at <http://www.absoft.com>.

### ***Contacting IBM***

IBM welcomes your comments. You can send them to [compinfo@ca.ibm.com](mailto:compinfo@ca.ibm.com) or mail them to this address:

XL Compiler Development  
Department 257  
Application Development Technology Centre  
Software Division Toronto Laboratory IBM Canada Ltd.  
8200 Warden Avenue  
Markham, Ontario  
Canada – L6G 1C7

### ***Copyright Notice***

© Copyright IBM Corp. 2004. All Rights Reserved.

IBM is trademark or registered trademark of International Business Machines Corporation in the U.S., other countries or both.