Optimizing C code at optimization level 2

Overview

Getting maximum results from optimizing at any level often requires collaboration between the compiler and coding practices. The highly advanced optimizer found in the XL family of compilers is most effective when applied to thoroughly debugged, standards compliant source code. How you write your code can be as important as the optimization options you use. This paper describes coding techniques that may improve optimization of your programs at level 2 and recommends using compiler options to help you get the best performance from your application code. The default settings quoted in this paper are applicable to XL C for AIX®, V10.1 and XL C/C++ for AIX, V10.1.

Quick optimization technology overview

The compilers of the XL compiler family are built on a foundation of common components that are then customized for the C, C++, and Fortran languages. Sharing common components ensures that performance-enhancing optimizations are available to you in all three languages. For example, during Interprocedural Analysis (IPA), which can only take place at one of the higher optimization levels, the optimizer can combine and optimize code from all three languages simultaneously when linking an application.

Some optimization highlights of the XL family of compilers are as follows:

- 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 processors
- Interprocedural analysis and optimization, using IPA
- Optimization toggles and switches (**-qalias**, **-qstrict**, **-qfloat**...)
- Simplified automatic parallelization and VMX enablement through the compiler's support of source-level intrinsic functions and user-applied directives
- Profile-directed feedback (PDF) optimization

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 good whole-program analysis at link time
- **-O5**, all of **-O4** and detailed whole-program analysis at link time

The five optimization levels allow you to choose from minimal optimization to intense program analysis that provides benefits even across programming languages.

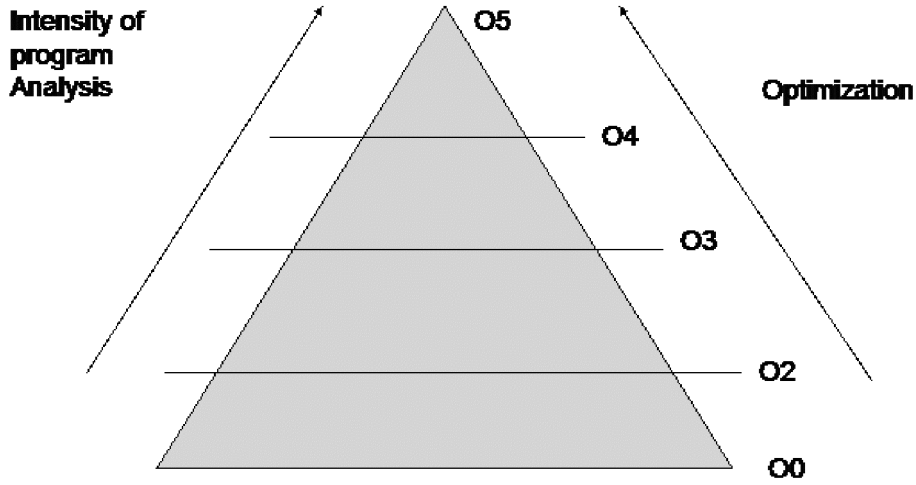

Figure 1: Optimization levels and intensity of program analysis.

The scope of optimization analyses can range from local basic block to subprogram to file-level to whole-program analysis. The higher the optimization level, the more intense the program analysis becomes as increasingly sophisticated optimization techniques are applied to your code. (See figure 1).

Before you optimize

As you begin the optimization process, consider that not all optimization levels and techniques suit all applications. Experimenting with different optimization levels and techniques can help you find the balance between increasing performance while limiting the impact on compilation time and system resources.

Although it would be highly desirable, no way exists to reduce both code size and compile time. Likewise, there is usually no way to increase optimization, while reducing compile time. To achieve improved optimization you usually must trade-off against an increase in compile time, a reduction in debugging capability, or an increase in code size. (See figure 2). For example, at higher levels of optimization, the optimizer can trade numeric precision for execution speed. If this effect is not desired, you can specify compiler options to prevent such trade-offs albeit at the cost of reducing the degree to which you can optimize. Other options such as **-qsmallstack** or **-qcompact** allow you to bias optimization decisions in favor of smaller stack space or program size.

Usually after analyzing which is the best way to proceed you will opt for either reducing memory space required or execution time. In most cases the decision is to opt for reducing time.

All XL compilers have options to limit optimizations for non-conforming code, but it is better to correct the code and not limit your optimization opportunities. For non-conforming source code, specify the appropriate options to inform the compiler which rules the source is breaking. Further, it is not advisable

to plunge straight in and try to optimize at level 5. Unless you are certain that your code is perfect, you may wait two hours for your code to compile only to find that it does not execute as expected. Prematurely optimizing at level 5 will leave you with errors that are time-consuming and difficult to trace.

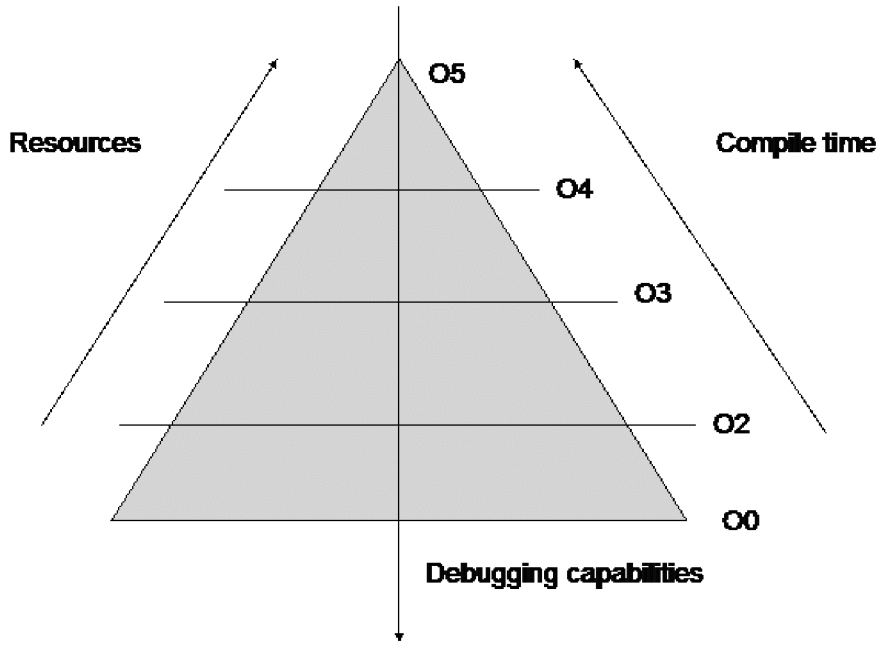

Figure 2: Trade-off among resources, debugging capabilities and compile time

The XL compiler provides a number of ways to verify how far your code conforms to the language standard including: **-qcheck** which generates code that performs certain types of runtime checking and **-qextchk** (with the linker **-btypchk** option) which generates link-time type checking information and checks for compile-time consistency. Also, **-qinitauto** initializes uninitialized automatic variables to a specific value, for debugging purposes.

| The compiler follows the type-based aliasing rule in the C/C++ standards when the **-qalias=ansi**
| compiler option is in effect (which is by default). Older code will often run into problems with
| type-based aliasing (also known as the ANSI aliasing rule).

To get additional informational messages about potential problems in your program, use the **-qinfo** compiler option. For example, **-qinfo=all** enables all diagnostic messages for all groups.

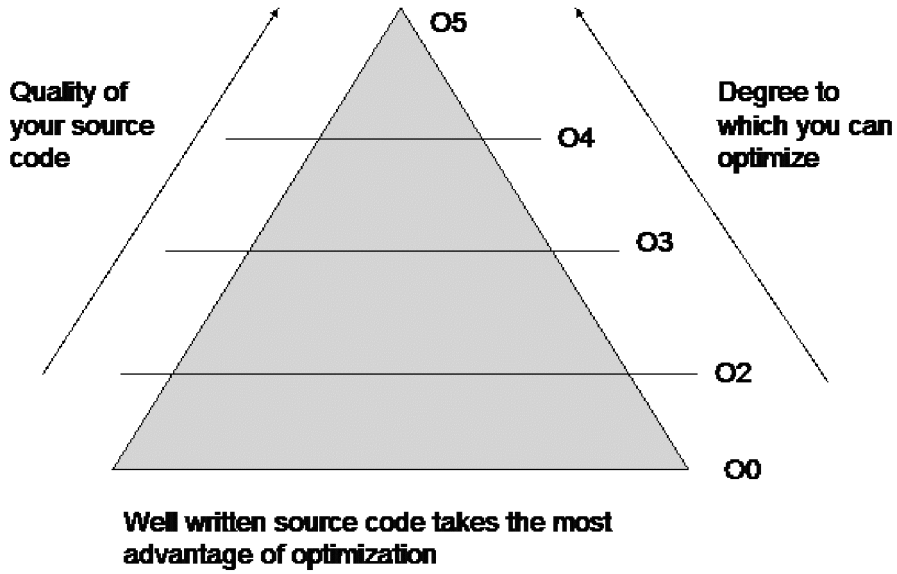

Figure 3: The relationship between the quality of your source code and ability to optimize.

Getting the best optimization is an iterative process of moving forward, then slowly backing off when you encounter problems. Problems in the code may prevent the compiler from reaching the highest optimization level. (This is shown in figure 3). Begin at level 0 and make sure any code issues are fixed.

If an application executes correctly but its performance has not improved, then you can optimize your application at the next level of optimization.

The XL compilers offer sub-levels of each optimization level that allow code to be compiled and run at a higher or lower level of optimization. For example, the XL compiler optimization levels specify packages of optimization such that the lower optimization level is a subset of the optimizations of the higher level.

You can move forward slowly testing sub-levels of each optimization to make sure there are no problems. When you encounter compile-time or run-time problems, you can reiterate at a lower optimization level. (See figure 4). In this manner, you can isolate problematic areas in your source code.

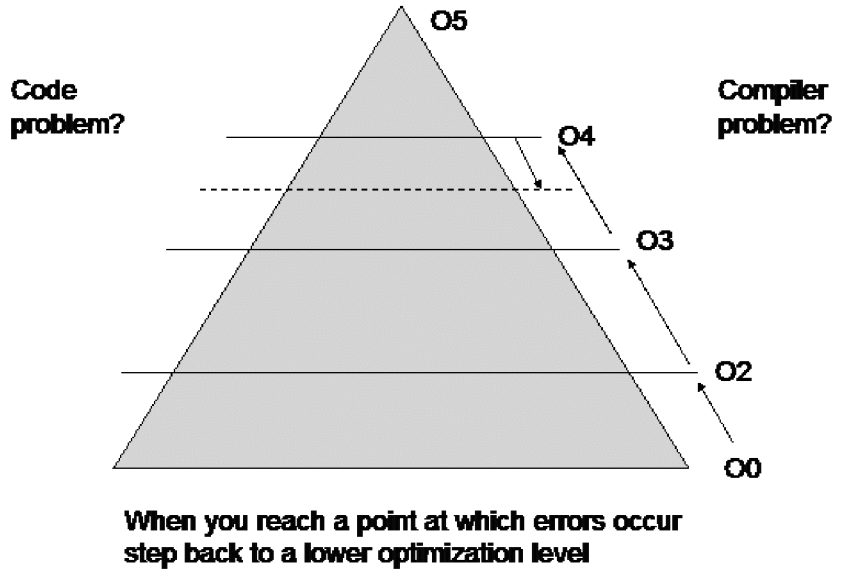

Figure 4: Optimization levels and sub-levels.

Optimizing at level 2

Table 1 lists the reasons to optimize code at level 2.

Table 1. Features and benefits of optimization level 2

Feature	Benefit
Common subexpression elimination	Eliminates duplicated (redundant) or unreachable (unused) code.
Dead code elimination	Eliminates instructions that a particular control flow does not reach, or that generated 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	Structures code to take advantage of a given microprocessor or architecture family.
Loop unrolling and software pipelining	Moves invariant code out of loops and simplifies control flow.
Strength reduction and effective use of addressing modes	Replaces costly operations with less expensive operations reducing the number of clock cycles. For example, x/8 is replaced by x right shift 3.
Value numbering	Simplifies algebraic expressions, by eliminating redundant computations.

The *Optimization level 2 and options* table details compiler options implied by using level 2 and options we recommend you should use at level 2.

Table 2. Optimization level 2 and options

Base optimization level	Additional option implied by level	Additional recommended options
-O2	-qmaxmem=8192	• -qarch • -qtune • -qmaxmem=-1 • –qhot=level=0 • -g • -qsmp=auto • -qlinedebug

An example of the difference between compiling with no optimization and using optimization level 2

When you compile code with no optimization, the default compile follows all the source code instructions literally. For each iteration of the loop, there would be a new load and a new store of the variable y.

```
y = 0.0;
for (j=0; j < jlim ; j++ )
{
      y+= a[j] * b[j]
 }
```

When you compile the same code with optimization level 2, the compiler recognizes that the value of y does not need to be stored until the loop is completed, and that intermediate values could be kept in registers. Even if the loads and stores are cached, the optimization could lead to an order of magnitude or better improvement in the performance of this loop.

Optimizations at -O2 attempt to find a balance between increasing performance while limiting the impact on compilation time and system resources.

Optimizing C code at level 2 using compiler options

You can instruct the compiler to use memory without checking for limits, target your application to a particular machine, optimize loops and array language, emit debug information, and turn off language standards checking.

By default, some techniques the optimizer uses to improve performance, such as loop unrolling and array vectorization, may also make the program larger. For systems with limited storage, you can use **-qcompact** to reduce the expansion that takes place. If your program has many loop and array language constructs, using the -qcompact option will affect your application's overall performance. You may want to restrict using this option to those parts of your program where optimization gains will remain unaffected.

Conforming to C standard aliasing rules

If your code does not conform to C standard aliasing rules (different types are aliases for the same storage location), you can use the **-qalias=noansi** compiler option. When **noansi** is in effect, the optimizer makes worst-case aliasing assumptions. It assumes that a pointer of a given type can point to an external object or any object whose address is already taken, regardless of type.

Functions with names identical to those of library functions

If your C application does not define functions with names identical to those of library functions, compile with **-qlibansi**. This option assumes that all functions with the name of an ANSI C library function are, in fact, system functions. When **libansi** is in effect, the optimizer will receive information about the behavior of a given function, such as whether or not it has any side effects, allowing the optimizer to generate better code.

Increasing memory available to the optimizer

If you find the compiler requires more memory to optimize your application, this message displays:

```
1500-030: (I) INFORMATION; functionname: Additional optimization may be attained
by recompiling and specifying the MAXMEM option with a value greater than value.
```

You need to specify -qmaxmem=-1 which allows the compiler to use memory as needed without checking for limits.

Emitting Debug information

Debugging optimized programs presents special usability problems. For example, loops are unrolled and the values assigned by expressions are consolidated. There is no longer a correspondence between the line numbers for these statements in the optimized source and the line numbers in the original source thus preventing symbolic debugging.

You can instruct the compiler to emit debug information using the **-g** compiler option. This option turns off inlining unless you explicitly request the compiler to inline functions using the -qinline compiler option.

To produce abbreviated debugging information in a smaller object size, you can use the **-qlinedebug** compiler option.

Optimizing loops

The **-qhot** compiler option performs high-order loop analysis and transformations (HOT) during optimization. It is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. This compiler option will always attempt to optimize loops, regardless of the suboptions you specify.

Note that most loop optimizations work only on countable loops or on loops with a single entry and exit point and an iteration count that can be determined before the loop begins. Also, many loop optimizations, including outer unroll-and-jam and interchange, require perfect loop nesting. This means there can be no intervening code between loops.

Use the option **-qhot=level=0** at -O2 to perform a subset of the high-order transformations and set the default to novector:nosimd:noarraypad. The result is:

- **novector** disables the conversion of loop array operations into calls to MASS library routines
- **nosimd** disables the conversion of loop array operations into calls to vector instructions
- **noarraypad** disables the increase of the dimensions of arrays

Improving performance with interprocedural analysis (IPA)

Compiler option **-qipa** enables interprocedural analysis, a two-step process that should be applied to as much of your program as possible and that may significantly improve your program's performance. Specifying -qipa automatically sets the optimization level to level 2. For additional performance benefits, you can also specify the -Q option, which instructs the compiler to attempt to inline functions instead of generating calls to those functions. **-qipa** can help reduce excessive call overhead found in programs with too many small functions and methods. However, it should be borne in mind that -qipa can cause incorrect but previously functioning programs to fail. For information on side-effects and how to use -qipa, see "Getting the most from -qipa" in *Optimization and Programming Guide*.

Targeting a specific architecture

All IBM® Power System servers share a common set of instructions, but may also include additional instructions unique to a given processor or processor family. If you want maximum performance on a specific architecture and will not be using the program on other architectures, use the appropriate **-qarch** option. If you want to generate code that can run on more than one architecture, specify a **-qarch** suboption that supports a group of architectures.

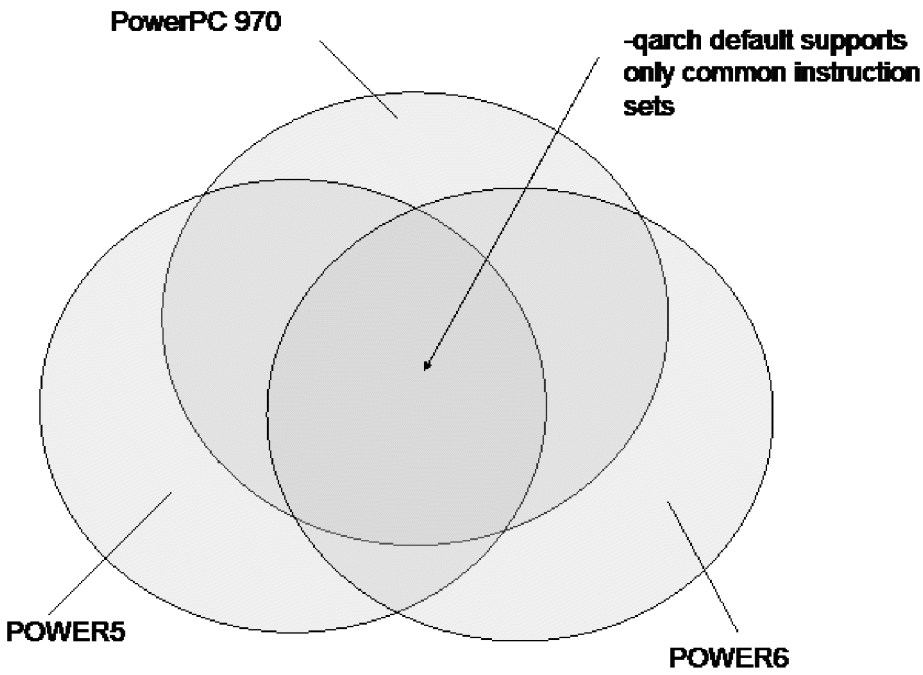

Figure 5: Default setting of -qarch at -O2 selects the smallest subset of capabilities that all the processors have in common for that target operating system and compilation mode

The default for **-qarch** is platform dependent. On AIX, the default architecture setting in 32-bit mode is **-qarch=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 does not exploit certain common instruction subgroups supported on more modern POWER® and PowerPC® processors. The **-qarch=ppc**

suboption causes the compiler to be slower on integer divides and multiply operations and to produce single-precision instructions to be used with single-precision data.

By default, the **-qarch** setting produces code using only instructions common to all supported architectures, with resultant settings of **-qtune** and **-qcache** that are relatively general. The choice of **-qarch** suboption controls the default setting of the -qtune option. Because the defaults for **-qarch** also affect the defaults for **-qtune** and**-qcache**, the **-qarch** option is often all that is needed. To tune performance for a particular processor set or architecture, you may need to specify different settings for one or more of these options. The natural progression to try is to use **-qarch**, and then add **-qtune**, and then add **-qcache**.

If your application will run on the same architecture on which you are compiling, use the **-qarch=auto** option, which automatically detects the specific architecture of the compiling machine and generates code to take advantage of instructions available only on that machine (or on a system that supports the equivalent processor architecture). Be sure to add **qtune=auto**, as this option will generate code that runs on all of the architectures in the specified group, but the instruction sequences will be those with the best performance on the architecture of the compiling machine.

Specify with **-qarch** the smallest family of machines possible that will be expected to run your code correctly. For example, on an XL compiler building an application that will only run on POWER5-based machines use **-qarch=pwr5**. Object code produced with **-qarch=pwr5** will also run on POWER5+™ and POWER6™. For an AIX compiler building applications that will only run on 64-bit mode capable hardware but does not know which machines in particular, use **-qarch=ppc64** to select the entire 64-bit PowerPC family of processors.

If you are running in a production environment, then use **-qarch=suboption**, where the suboption is the specific processor that will produce object code that will run only on that platform. For example, use **-qarch=pwr6** to produce object code containing instructions that will only run on the POWER6 hardware platforms. Currently this is the only suboption that provides support for decimal floating point instructions.

If your application will run on all the POWER and PowerPC processors, use **-qarch=pwr3**. **-qarch=pwr3** produces object code containing instructions that will run on any POWER3™, POWER4™, POWER5™, POWER5+, POWER6, or PowerPC 970 hardware platform.

If your application requires decimal floating point instructions, then use the **-qfloat=nosingle:norndsngl** compiler option or **-qarch=pwr6**.

If your application requires square root instruction support, then use **-qarch=ppc64grsq**.

Biasing optimization decisions for executing an application on a particular architecture

The **-qtune** option directs the optimizer to bias optimization decisions for executing an application on a particular architecture. **-qtune** does not prevent the application from running on other architectures nor does it imply anything about an application's ability to run correctly on a given machine. 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.

Note: These defaults may change over time

Using **-qtune** allows the optimizer to perform transformations, such as instruction scheduling, so that resulting code executes most efficiently on your chosen **-qtune** architecture (the code will still execute on other machines specified by the **-qarch** setting). Since the **-qtune** option tunes code to run on one particular processor architecture, it does not support suboptions representing families of processors.

To instruct the compiler to generate code tuned for optimal performance across a range of recent processor architectures, including POWER6, use the **-qtune=balanced** compiler option. This suboption is the default when the default **-qarch=ppc** setting is specified.

You should use **-qtune** to specify the most common or important processor where your application will execute.

For example:

1. if your application will usually execute on a POWER6-based system but will sometimes execute on POWER5-based systems, specify -qtune=pwr6. The code generated will execute more efficiently on POWER6-based systems but will run correctly on POWER5-based systems
2. if 90% of users are using POWER6, but 10% of users are using POWER5, you can do **–qarch=pwr5 –qtune=pwr6**. If 2/3 are using POWER5 and 1/3 are using POWER6, then do **–qarch=pwr5**, **-qtune=balanced**

To instruct the compiler to tune generated code for optimal performance for the platform on which the application was compiled, use the **-qtune=auto** suboption. Using this suboption, the compiler detects the machine characteristics on which you are compiling, and tunes for that type of machine.

If you are not sure what option to specify, try **-qtune=pwr3**. This option will tune the optimization for the POWER3 hardware platforms and with **-qarch=pwr3** will produce object code containing instructions that will run on most hardware platforms, for example, any POWER3, POWER4, POWER5, POWER5+, POWER6, or PowerPC 970 hardware platform.

Describing cache characteristics

Compiler 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 makes 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 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.

Tuning the performance of your application with profile-directed feedback (PDF)

You can use **-qpdf1** and **-qpdf2** to tune the performance of your application for a typical usage scenario. The compiler optimizes the application based on an analysis of how often branches are taken and blocks of code are executed. The PDF process is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production. **-qipa** can also benefit when used in conjunction with PDF.

To learn more about PDF, see "Using profile-directed feedback" in *Optimization and Programming Guide*.

Optimizing C code at level 2 by rewriting code

It is important to avoid excessively hand-optimizing your code which can confuse the optimizer (and other developers) and may require you to hand-tune your code to retarget for new hardware technologies. However, there are coding techniques that can improve the performance of your code, including:

- At optimization level 2 the compiler focuses on optimization opportunities within functions and basic blocks. Therefore, grouping related functions in the same file and making as many *static* variables as possible assists the optimizer by more clearly delineating its scope. When using pointers, try to adhere as closely as possible to language rules as a matter of practice. Also, simplify your array indexing by using language facilities where possible
- To assist the compiler in optimizing loops, try to use counted DO loops or canonical *for* loops, which are usually countable. Ensure perfect loop nesting by splitting loops so that no other code is executed within the containing loop.

For in-depth information on optimizing your applications, see "Coding your application to improve performance" in *Optimization and Programming Guide*. You will find information on improving your program's performance of input and output, considerations when writing a function or calling a library function, techniques to improve memory management, how to optimize variables, how to better manipulate strings, and how to optimize expressions and program logic.

Summary

The optimizer found in the XL compiler family will do a lot of the performance enhancement work for you. However, there is much to be gained from fully understanding the relationship between the quality and language standards compliance of your source code and the optimization capabilities of an XL compiler. Before using compiler options you should verify how compliant your code is with the language standard. The options and suboptions you choose to use will be, to some degree, influenced by how compliant your source code is. The effectiveness of many options will also be influenced by your code's compliance.

Another factor to consider is how specifically you can identify the architectures your code will run on. The more precise your knowledge of the target architecture the more advantage you will derive from **-qarch**, **-qtune** and **-qcache**. If possible, try and use -qarch to target the most specific architecture possible for your code and use -qtune and -qcache to enhance optimization on your chosen architecture.

This paper has identified some key options and suboptions that may greatly enhance the performance of your application. Most applications can achieve a significant portion of their potential performance improvement using just basic optimization options. Understanding the key options and their suboptions will allow you to reap the full potential of the IBM XL compiler optimizer. In addition to taking advantage of XL compiler options and suboptions following certain programming techniques will greatly enhance the optimization of your code.

September 2009

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.

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at ″Copyright and trademark information″ at www.ibm.com/legal/copytrade.shtml