Interested in tuning your C/C++ applications for Linux™ on POWER™? This article compares the optimization options for both Linux on POWER C/C++ compilers: GCC and IBM XL C/C++. This paper also reviews tactics, such as Interprocedural Analysis, Profile Directed Feedback, and High Order Transformations, which are used by one or both of the compilers to extract higher performance from the Power architecture.

Introduction

The C/C++ compiler translates your C/C++ code into uniform and optimized machine code. By default, compilers perform only basic local optimizations or no optimization at all. By turning on optimization options, the execution time of your program can sometimes be reduced substantially without any recoding effort. Modern compilers often produce optimized code as good as or better than developers can code by hand, and this is accentuated in code written for portability. Instead of maintaining multiple code bases with extensive platform specific modification, Linux developers often keep one code base and let the compilers handle the dirty details of optimization. By leveraging the optimization options of the compiler, developers can keep portable code performing well on many platforms.

The Linux on POWER platform offers more than one option to produce binary C/C++ code. In addition to supporting both 32- and 64-bit runtime environments simultaneously, Linux on POWER has two compiler collections. The GNU Compiler Collection, or GCC, is consistent with other Linux implementations with specific exceptions for the POWER architecture. GCC is the leading compiler for portability but also features a number of performance enhancements for optimizing code. The IBM XL C/C++ compiler for Linux on POWER is derived from the high performance compiler for AIX but uses the GNU linker and assembler to create ELF objects that are fully compatible with objects produced by GCC. This document provides side-by-side comparisons of how these two compilers are controlled, overviews of what the compilers are capable of, in terms of optimization, and tips for writing code that is more easily optimized with either of these compilers.

Compilers for Linux on POWER

GCC is a robust compiler aimed at world class quality with emphasis on portability across platforms and open source development. You can read the GCC mission statement at the GCC

May 10, 2005
Web site. (See Related topics.) In addition to GCC, IBM's high performance, standards-based compiler, XL C/C++ is available for Linux on POWER. XL C/C++ is designed to optimize at several levels: expression, basic block, procedure, and whole program. In addition, applications that rely on floating point and loop operations often benefit significantly from XL C/C++. The GCC and XL C/C++ manuals are available online. (See Related topics.)

Both GCC and XL C/C++ provide blanket optimization levels (-O3, for example) and specific optimization flags (-floop-optimize, for example). Other types of blanket and specific options are provided by both compilers, such as architecture flags. These options are reviewed here, side by side, to provide a cross reference for developers.

## Optimization level

Compilers provide various optimization levels that increase application performance, but often at the expense of longer compile time, larger program size, and complete debugging support. Also, the compiler makes certain assumptions about some of the statements in the code that can potentially be optimized by rewriting that section of the code. Therefore, it is important to always verify your output with testing as you increase the level of compiler optimization.

### Comparison of basic optimization blanket options

Table 1 compares the basic optimization levels of GCC and XL C/C++. Use this table as a concise reference, but see the compiler manuals for which specific options are enabled by each level of the blanket optimization flags.

### Table 1. Basic optimization levels of GCC and XL C/C++

<table>
<thead>
<tr>
<th>GCC Level</th>
<th>GCC Description</th>
<th>XL C/C++ Level</th>
<th>XL Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>No optimizations performed. (Default if no flag is supplied.)</td>
<td>-qopt=0 or -O0¹</td>
<td>Perform only quick local optimizations. (Default if no flag is supplied.)</td>
</tr>
<tr>
<td>-O or -O1</td>
<td>Perform simple optimization attempting to reduce both the code size and execution time. Debugging is still available at this level.</td>
<td>-O</td>
<td>Comprehensive low-level optimization with partial debugging support. These optimizations do not affect program correctness.</td>
</tr>
<tr>
<td>-O2</td>
<td>Performs nearly all supported optimizations that do not involve a space-speed trade-off. <em>Interprocedural Analysis</em></td>
<td>-O2</td>
<td>(Same as -O)</td>
</tr>
</tbody>
</table>

¹ February 2005 XL C/C++ V7.0 Update

If no optimization flag is specified, GCC does no optimization at all (same as -O0), while XL C/C++ still performs some small optimizations.

Note that the default optimization level, -O, defaults to -O1 for GCC and -O2 for XL C/C++. At the -O level, GCC begins some simple optimizations, such as merging common constants across compile objects; debugging is still fully functional with GCC at -O. XL C/C++ -O optimizations imply a more thorough optimization, comparable to what GCC does at -O2, including branch
optimization, strength reduction, and extensive modification to loops and ordering. Debugging support is partially compromised at the -O level with XL C/C++ because code is rearranged to optimize branching. However, XL C/C++ preserves partial debugging support at -O.

At the -O2 level, GCC is optimizing with every algorithm that will not increase executable size. Therefore, GCC is not using loop unrolling and function inlining methods at -O2. Contrast this with XL C/C++, which uses unrolling techniques at -O2.

Comparison of advanced optimization blanket options

Table 2 summarizes advanced optimization blanket option, -O3, for both compilers.

Table 2. Advanced optimization blanket option (-O3)

<table>
<thead>
<tr>
<th>GCC Level</th>
<th>GCC Description</th>
<th>XL C/C++ Level</th>
<th>XL Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O3</td>
<td>Executable size sacrificed for performance. Inclusive of all -O2 optimization options, plus: Turns on a new web construction pass, -fweb, as well as inlining functions and making more efficient use of registers.</td>
<td>-O3</td>
<td>More aggressive than -O2. Semantics of program can change, as function inlining, extensive loop unrolling, and scheduling optimizations are implemented in code.</td>
</tr>
</tbody>
</table>

At -O3, GCC enables all the options from -O2, along with more advanced methods, such as inlining functions, renaming registers, and other scheduling improvements. Of particular note are -frename-registers and -fweb. The -frename-registers flag makes use of unallocated registers to avoid false dependencies in code scheduling. The -fweb option takes an integrated, higher level view of optimization, as it considers the interactions of register reallocation, loop unrolling, scheduling modifications, and more. This feature is similar to Interprocedural Analysis (see "Interprocedural Analysis").

XL C/C++ at the -O3 level aggressively applies optimization algorithms to your code. Compile times can be lengthened considerably, and the semantics of the program can be changed. At this level, like GCC, XL C/C++ is inlining functions, unrolling loops aggressively, and improving scheduling. However, XL C/C++ is tailored to the POWER architecture more tightly than GCC. These improvements are areas in which XL C/C++ starts to emerge with a clear performance advantage.

Optimization levels exclusive to XL C/C++

-03 is the last stop for GCC blanket optimization options, but XL C/C++ continues with two more levels advised for very performance intensive applications, -O4 and -O5.

Table 3. XL C/C++ advanced optimization blanket options -O4 and -O5

<table>
<thead>
<tr>
<th>GCC Level</th>
<th>GCC Description</th>
<th>XL C/C++ Level</th>
<th>XL Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>N/A</td>
<td>-O4</td>
<td>Inclusive of -O3 optimizations but begins hardware specific</td>
</tr>
</tbody>
</table>
tuning, IPA, and high order transformations.
- Hardware-specific optimization
  (-qarch=auto, -qtune=auto, -qcache=auto).
- Interprocedural analysis.
- High-order transformations.

Interprocedural Analysis, or IPA, enables the compiler to perform the whole program analysis. It is available in three levels of aggressiveness (level=0,1,2). We discuss more about IPA under "Interprocedural Analysis".

High-order transformations are enabled with the -O4 flag (or separately with -qhot). High-order transformations are optimization techniques specifically designed for improving performance of loops. They take advantage of the effective use of caches, translation look-aside buffers, and data pre-fetching capabilities provided by the hardware. They also improve the utilization of processor resources through instruction reordering and balancing. For more information, please consult the Compiler Reference - XL C/C++ Advanced Edition V7.0 for Linux. (See Related topics.)

The -O4 level is essentially the same as -O3, plus -qhot and -qipa=level=1.

The -O5 level extends the benefits featured in -O4 but, again, is more aggressive. The IPA level is raised to level 2, and high-order transformations are delayed until link time, when the entire program can be analyzed holistically. In sum, GCC blanket optimization options perform many of the same optimizations as with XL C/C++, but the algorithms in XL C/C++ are generally more refined and specifically tailored to the POWER architecture and its strengths.

Optimizing for a system architecture
Both GCC and XL C/C++ can generate code for the full range of POWER processors. The difficulty can be in determining how specifically you want to compile your code and which processor to aim for. There are two types of architecture flags: one that specifies the instructions to be used for the processor, and another that specifies ordering and scheduling details for the processor. With GCC, the first type of flag is -mcpu, where a CPU name is specified with the flag, and the second is -mtune, where slightly more general processor families are specified. These are analogous to the -qarch and -qtune flags with XL C/C++. These architecture options are summarized in the following table:

Table 4. Architecture options

<table>
<thead>
<tr>
<th>GCC Option</th>
<th>GCC Behavior</th>
<th>XL C/C++ Option</th>
<th>XL Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mcpu</td>
<td>Controls which family's instruction set is used to compile the binary. powerpc is the default argument. Other relevant options for IBM eServer hardware are common.</td>
<td>-qarch</td>
<td>Controls which family's instruction set is used to compile the binary. At less than -O4, the default argument is ppc64grsq. Above -O3, the default is auto.</td>
</tr>
</tbody>
</table>
The PowerPC® architecture has two branches of its processor family: POWER and PowerPC. The ontogeny of the chips is documented elsewhere, but it is important to note that GCC considers all recent POWER CPUs to be PowerPC architecture, whereas XL C/C++ is more specific. The default argument to the -mcpu and -mtune options for GCC is powerpc. There are many more options documented in the GCC manual, but the relevant ones for IBM eServer hardware are:

- common
- power3
- power4
- power5
- 970
- powerpc64

The common flag will tell GCC to emit code to run on any POWER or PowerPC processor. The power3, power4, power5, and 970 flags are specific to those chips, and the powerpc64 is the 64-bit version of the default.

XL C/C++ uses a slightly different convention, with -qtune and -qarch. The auto argument makes specificity much easier, as this option tells the compiler to discover the CPU type and produce code for that CPU. The default flag is controlled by the -O level selected, such that -O4 and -O5 use the auto argument, and all other levels use ppc64grsq. Do not be deceived by this flag, it has no bearing on whether or not the binary produced is 32- or 64-bit. This flag does, however, specify a 64-bit architecture on which to run the code. 32-bit binaries can benefit from running on 64-bit hardware with Linux on POWER due to the preservation of the full 64-bit register space in both 32-bit and 64-bit runtime modes. This means even 32-bit applications will use some 64-bit instructions, if you compile with a 64-bit -qarch flag. If you intend for your XL compiled code to deploy on older 32-bit processors, use a more general arch flag, such as -qarch=ppc.

The -qinlglue option is automatically enabled when -qtune=pwr4, -qtune=pwr, -qtune=ppc970, or -qtune=auto is enabled on a system that uses one of these architectures, along with -q64 specified. The -qinlglue option generates fast external linkage by inlining the glue code. As defined by the XL C/C++ manual, glue code is used for passing control between two external functions or when functions are called through the pointer.

More recently, additional architecture flags have been added to support the Vector Multimedia eXtension®, or VMX, features of PowerPC chips. GCC supports several processors with VMX...
instructions, whereas XL C/C++ only supports the PowerPC 970 and PowerPC 970FX. With GCC, the VMX extensions are named for the Motorola trademark AltiVec®, instead of the IBM trademark VMX.

Table 5. Additional architecture options

<table>
<thead>
<tr>
<th>GCC Option</th>
<th>GCC Behavior</th>
<th>XL C/C++ Option</th>
<th>XL Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>-maltivec</td>
<td>Enables the built-in functions, which allow access to the VMX instructions.</td>
<td>-qenablevmx</td>
<td>Enables the VMX instruction set. For SUSE SLES 9 and Red Hat Enterprise Linux 4, this option is enabled by default for ppc970, but not for Red Hat Enterprise Linux 3.</td>
</tr>
<tr>
<td>-mabi=altivec</td>
<td>Enables the ABI extensions for VMX. -mabi=altivec is required for -maltivec to have an effect.</td>
<td>-qaltivec</td>
<td>Enables compiler support for AltiVec data types. -qaltivec has effect only when -qarch is set or implied to be ppc970.</td>
</tr>
</tbody>
</table>

Both compilers require two separate flags to make use of VMX extensions for vectorized code. GCC requires -maltivec, which will enable the built-in functions for using the VMX instructions, and -mabi=altivec will instruct the compiler to accept the ABI extensions for VMX. XL C/C++ has two similar flags: -qenablevmx corresponds to -maltivec, and -qaltivec is analogous to -mabi=altivec. However, take note that with XL C/C++, the -qarch flag must be set to -qarch=ppc970 in order to use the VMX instructions.

It is also worth mentioning that XL C/C++ does attempt some automated vectorization and optimization of scalar code when the -qhot flag is used. GCC does not attempt to vectorize any code automatically.

Interprocedural Analysis (IPA)

Both XL C/C++ and GCC offer Interprocedural Analysis, a whole program approach that optimizes across different files. IPA often results in significant performance improvements, but be warned that IPA will increase compile time. As so, you may want to use IPA only in the final stage of performance tuning.

IPA optimization with GCC

Starting with GCC Version 3.4, IPA is available by using the -funit-at-a-time option. Among enterprise distributions of Linux for POWER, only Red Hat Enterprise Linux Version 4 currently provides this level of GCC. GCC implements the following IPA optimizations²:

- Removal of unreachable functions and variables
- Discovery of functions with static linkage whose address is never taken
- Reordering of functions in topological order of the call graph
- Out-of-order inlining heuristics that allow limiting overall compilation unit growth

²See "GCC 3.4 Changes, New Features, and Fixes" for a complete list of optimizations. (See Related topics.)
IPA optimization with XL C/C++

IPA is enabled with XL C/C++ by using the -qipa option or implicitly by using the -O4 option. It is possible to enable IPA on the compile step only or with both the compile and link steps. IPA with both the compile and link steps is referred to as "whole program mode," since IPA can analyze across objects in this configuration. You may want to try both options to explore the benefits to your code. The following IPA optimizations are implemented in XL C/C++:

- Localization of statically bound variables and procedures
- Partitioning and layout of procedures according to calling relationships
- Function inlining
- Partitioning and layout of static data according to reference affinity
- Global alias analysis, specialization, and interprocedural data flow

The full set of suboptions and syntax is described under the -qipa entry in the "Compiler Reference - XL C/C++ Advanced Edition V7.0 for Linux." (See Related topics.)

Since IPA can generate significantly larger object files than traditional compilations, ensure that there is sufficient free space in the /tmp directory, or use the TMPDIR environment variable to specify a different directory with sufficient free space.

Profile-Directed Feedback (PDF)

PDF is a two step compilation process that improves the performance of your application when applied to typical workloads. PDF optimizes the application based on analysis of how often branches are taken and blocks of code are executed. PDF performs further procedural level optimization, such as directing register allocations, instruction scheduling, and basic block rearrangement.

PDF optimization with GCC

To use PDF with GCC, first compile your source with the -fprofile-generate option, and then run the application with a data set representing your typical workload. You can repeat this process any number of times, and GCC will combine the data from each profile with each iteration. Finally, compile again with the -fprofile-use option. GCC will use the profile generated to reorder the program to run more efficiently in the context of the workloads it was exposed to during profiling.

PDF optimization with XL C/C++

To use the XL C/C++ implementation of PDF, compile some or all of the source files with -qpdf1. Next, run the program using a data set representing your typical workloads. When the program exits, it will write profiling information to a file in the current working directory or the directory specified by the PDFDIR environment variable. Multiple profiles can be collected, if you run with more than one data set, but they are not aggregated automatically, as with GCC. These profiles are merged using the mergepdf tool, which combines all of the profile files into one. Whether you have one or many profiles collected, the last step is to recompile the program with the -qpdf2 flag and the profiles in the working directory.
Built-in functions

GCC provides you with a selection of built-in functions to help you write more efficient codes. Generally, these functions generate calls to specific machine instructions but allow the compiler to schedule those calls. GCC provides an interface for the PowerPC family of processors, such as the PowerPC 970, to access the AltiVec operations described in the Motorola’s AltiVec Programming Interface Manual. This interface is available to applications by including <altivec.h> and using the `-maltivec` and `-mabi=altivec` compiler options (described above). For more information, consult section 5.45.4 (“PowerPC AltiVec Built-in Functions”) of the GCC manual. (See Related topics.)

XL C/C++ also supports many built-in functions specific to some target architectures. For C++ code, your own functions are automatically mapped to built-in functions, if you include the XL C/C++ header files. For C code, your functions are mapped to built-in functions, if you include math.h and string.h. Built-ins referring to instructions are always available. The entire list of the built-in functions is given in Appendix B of the "Compiler Reference - XL C/C++ Advanced Edition V7.0 for Linux." (See Related topics.)

Using optimized libraries

You can improve the performance of your computationally intensive applications with the inclusion of optimized libraries. Some of these libraries are provided exclusively by IBM and some are available with Linux for POWER distributions, or at large.

Mathematical Acceleration Subsystem (MASS)

The XL C/C++ compiler for Linux ships with the MASS libraries. MASS consists of mathematical libraries optimized for POWER processor architectures. These libraries are thread-safe and support both 32- and 64-bit compilations in C, C++, and Fortran. They also include both scalar and vector functions and are intended for use in applications where slight differences in accuracy or handling of exceptional values can be tolerated.

For more information about MASS for Linux on POWER, please visit the IBM Mathematical Acceleration Subsystem for Linux Web page. (See Related topics.)

An IBM support document, "Using the MASS libraries on Linux," shows how to call MASS library functions from an application and how to compile and link an application with the MASS libraries. (See Related topics.)

Engineering and Scientific Subroutine Library (ESSL)

Another set of IBM optimized mathematical libraries available for Linux on POWER are the ESSL libraries. ESSL is a set of high-performance mathematical subroutines consisting of these components:

- Basic Linear Algebra Subprograms (BLAS)
- Linear Algebraic Equations
- Eigensystem Analysis
- Fourier Transforms
More information about ESSL can be found at the Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL Web page. (See Related topics.)

In addition, you can download the binary and source RPMs of the Open Source HPC software stack (including BLAS, GM, MPICH, FFTW, LAPACK, and more) for Linux running on POWER processor-based servers from the Linux on POWER Open Source Repository. (See Related topics.)

Making the compiler's job easier

Modern compilers have made life easy for developers looking to squeeze performance from their code. However, after applying optimization capabilities offered by the compiler, you may want to take your application a step further by using the techniques described in this section to complement the optimization techniques used by the compiler. In the following, we highlight the techniques discussed in Chapter 7 of the Programming Guide - XL C/C++ Advanced Edition V7.0 for Linux. (See Related topics.)

Reduce function-call overhead

- Functions with constant arguments provide more opportunities for optimization.
- Usually, do not declare virtual functions inline. When declaring functions, specify the const whenever possible.
- In C programs, fully prototype all functions.
- Use the built-in functions, instead of coding your own. In C++ programs, functions are automatically mapped to built-in functions, if you include the XL C/C++ header files. In C program, functions are mapped to built-in functions, if you include math.h and string.h.
- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Avoid breaking your program into too many small functions. If you must use small functions, consider using the -qipa option.
- Use virtual functions and virtual inheritance only when they are necessary. These features are costly in object space and function invocation performance.

Manage memory efficiently

- In a structure, declare the largest members first.
- Place variables in a structure near each other, if they are frequently used together.

Optimize variables

- Use local variables, preferably automatic variables, as much as possible.
- If you must use global variables, use static variables with file scope, rather than external variables, whenever possible.
- If you must use external variables, group them into structures or arrays whenever it makes sense to do so.
- Use constants instead of variables where possible.
- Avoid taking the address of a variable. Taking the address of a variable inhibits optimizations that would otherwise be done on calculations involving that variable.
• Use register-sized integers (long data type) for scalars. For large arrays of integers, consider using one- or two-byte integers or bit fields.

Manipulate strings efficiently
• When storing strings, align the start of the string on an 8-byte boundary.
• By knowing the length of a string, you can use mem* functions instead of str* functions. Using mem* functions, faster code can be generated.
• Make string literals read-only, whenever possible.

Optimize expressions and program logic
• Avoid forcing the compiler to convert numbers between integer and floating-point.
• Avoid go-to statements that jump into the middle of loops.
• Improve the predictability of your code by making the fall-through path more probable.
• In C++ code, use try blocks for exception handling only when necessary because they can inhibit optimization.

Optimize operations in 64-bit mode
• Avoid performing mixed 32- and 64-bit operations.
• Avoid long division whenever possible. Multiplication is usually faster than division.
• Use long types instead of signed, unsigned, and plain int types for variables that will be frequently accessed, such as loop counters and array indexes.

The XL C/C++ advantage: A HOT code example
The XL C/C++ optimization advantage is often conferred by the high order transformations the compiler can process on loops. HOT compilation involves weighing many options against one another to ensure the best mix of optimization tactics (fusion, interchange, and selective unrolling methods, to name a few). In addition to advanced algorithms, XL C/C++ can take advantage of intimate knowledge of the POWER architecture to improve scheduling and cache miss rates.

The following example (Listing 1) illustrates the effect of the -qhot option on a loop intensive computation. The high order transformations made by the XL C/C++ compiler deliver a ten fold performance improvement over GCC with -O3 optimization.

Listing 1. Loop intensive computation with the -qhot option
/* sample.c */
typedef double matrix[4000][4000];
matrix m1, m2;

static void foo (void) {
    int i, j;
    for (i = 0; i < 4000; ++i)
        for (j = 0; j < 4000; ++j)
            m1[j][i] = 8.0/m2[j][i]; /* XLC interchanges i and j loops and vectorizes the float divide */
}

main () {
    int i;
    for (i = 0; i < 10; ++i) foo();
}
Table 6. Compiler flags and execution times

<table>
<thead>
<tr>
<th>Compiler Flags</th>
<th>Execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc -o sample sample.c -mcpu=power5 -mtune=power5 -O3</td>
<td>16.2</td>
</tr>
<tr>
<td>gcc -o sample sample.c -mcpu=power5 -mtune=power5 -funroll-loops -O3</td>
<td>15.3</td>
</tr>
<tr>
<td>xlc -o sample sample.c</td>
<td>14.7</td>
</tr>
<tr>
<td>xlc -o sample sample.c -qarch=pwr5 -qtune=pwr5 -qhot</td>
<td>1.07</td>
</tr>
</tbody>
</table>

GCC, with only high order transformation knowledge being provided by -fweb, compiles code that is executed in 16.2 seconds. With the -funroll-loops option, this improves to an execution time of 15.3 seconds. Meanwhile, the XL C/C++ compiler, with default optimization, produces code that executes in 14.7 seconds, an improvement, but nothing to compare to what happens with the high order transformation feature of XL C/C++. Using -qhot even without -O3 renders an improvement of ~16X when compared to the code generated by GCC with -O3.

Do not expect 1600% increases in your whole program performance just from recompiling with XL C/C++. But as this example illustrates, there are cases where XL C/C++ can confer an incredible benefit to pieces of your code.

Summary

The POWER architecture offers a high performance 32- and 64-bit platform for Linux code. In addition to the most advanced hardware available, Linux on POWER offers two compilers for developers to use when building their applications. The GNU Compiler Collection is available for applications already tailored to the GNU extensions or applications that require a common compiler with other build platforms, while the IBM XL C/C++ compiler offers an alternative that can reap high performance rewards for your code.

GCC and XL C/C++ both provide sophisticated optimization routines for improving your code for Linux on POWER without rewriting a single line. Optimizations may take place at a platform agnostic and local level, such as function inlining. They may also involve object modification to better exploit a given hardware architecture; or optimization may take a collective and integrated approach to improve the runtime performance of your code, such as IPA, PDF, and HOT methods. Be warned: The more optimization performed on code, the more debugging and maintenance challenges you will meet. There is always a trade-off between performance and pragmatism.

We have reviewed the options for GCC and XL C/C++ optimization, side by side, to assist you as a reference, and we also have provided information on types of modern optimization. Where compilers leave off, the programmer must pick up, and so we have reviewed good programming practices set forth in the XL C/C++ manual. Finally, we have concluded with a specific code snippet, which benefits tremendously from XL C/C++ high order transformation. By choosing the correct compiler for your development purposes, you can save months of hand tuning and extract a high performance binary without sacrificing portability to other platforms.

Here’s a quick step-by-step summary of the process to help you get started:
• Plan ahead for your port: Port only code that already compiles and runs bug-free on other architectures.
• Pick your compiler: If you maintain a single code base to be compiled with GCC, then do not migrate to IBM XL C/C++ unless the performance advantage is worth maintaining additional compile environments.
• Optimize step-by-step: Start with no optimization, and make sure the code compiles and runs with verified logic. Then move to default levels of optimization, which are known not to change code logic. If you still need additional performance, use the information provided here to leverage advanced optimization features and squeeze all the performance you can from the architecture.
• Never overlook the importance of thoroughly testing your compiled code, especially if advanced options, such as IPA, PDF, and HOT, have been performed.

Acknowledgements

The authors would like to thank Gary Hook and Roch Archambault for their expertise in the POWER architecture and compiler technology for Linux on POWER. Their help, along with the commentary of Dwayne Moore, Mark Mendell, and the other members of the XL Compiler team, contributed greatly to the content of this document.
Related topics

- The article "How to use IBM XL C/C++ Advanced Edition V7.0 for Linux on POWER: A guide for GCC users" assists Linux developers on IBM POWER processor-based systems to rapidly migrate from the GNU gcc and gcc-c++ compilers (referred to as GCC) to the IBM XL C/C++ Advanced Edition V7.0.
- The GNU Compiler Collection Manual provides information on how to use the GNU compilers.
- Read the GCC mission statement.
- The GCC and XL C/C++ manuals are available online at the GCC home page and the XL C/C++ Library.
- The document "GCC 3.4 Changes, New Features, and Fixes" can be found at the GNU project Web site.
- For more information about MASS for Linux on POWER, please visit the Mathematical Acceleration Subsystem for Linux Web page.
- More information about ESSL can be found at the Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL.
- Download the binary and source RPMs of the Open Source HPC software stack (including BLAS, GM, MPICH, FFTW, LAPACK, and more) for Linux running on POWER processor-based servers from the Linux on POWER Open Source Repository.

© Copyright IBM Corporation 2005
Trademark
(www.ibm.com/developerworks/ibm/trademarks/)