

SEARCH
MAP

RS/6000

© 1997 IBM Corporation

RS/6000

Using the XL Compiler Options to Improve Application Performance

Katherine E. Stewart

Contents

- Introduction
 - Methodology
 - Using Basic Optimization
 - Using Architecture Information
 - Using Application Information
 - Inlining and Interprocedural Analysis
 - Aliasing
 - Floating-Point Precision
 - Using Source Preprocessors
 - Conclusion
 - Acknowledgments
 - References
-

Introduction

The effect of compiler and preprocessor options on an application's overall performance is generally not well understood. This paper examines the problem in the context of the new POWER2 and PowerPC 601 implementations in the RS/6000 family of machines, using the new XL compilers that exploit them. Experimentation shows that most applications can achieve a significant portion of their potential performance improvement using just basic optimization options. With additional architecture and application specific options, the compilers and preprocessors can provide performance improvements. This discussion illustrates the effect that tuning can have, and it also provides a framework for the user to judge the amount of tuning effort likely to be worthwhile.

Methodology

This paper uses the CINT92 and CFP92 benchmark suites from SPEC [1] as a set of sample applications for illustrating the improvements possible by the addition of various compiler and preprocessor options. The results are SPECratios that are calculated by dividing a benchmark's standard reference time by its actual execution time. The SPEC benchmark aggregates are the geometric mean of the SPECratios of the integer (SPECint92) and floating-point (SPECfp92) application sets. These aggregates should only be used as a general guide. If one of the benchmarks is similar to your own application, it should be used in preference to the aggregates.

All measurements were made with early versions of the compilers (IBM C Set ++ for AIX/6000 version 2.1 and XL Fortran version

3.1) on a RS/6000 Model 250 (PowerPC 601 processor), a RISC System/6000 Model 590 (POWER2 processor), and, for comparison purposes, a RS/6000 Model 580 (POWER processor). The KAP for IBM C 1.3 preprocessor was used with the C applications. The preprocessors used for Fortran applications were early versions of the new release of KAP for IBM Fortran from Kuck and Associates, Inc. and VAST-2 for XL Fortran from Pacific-Sierra Research. Although the experiments were done with early versions of the compilers, the general conclusions will hold when the production-level versions of the compilers become generally available. Figures 1 through 3 show the SPECratios of the benchmarks compiled for the POWER processor. Figures 4 through 6 show the SPECratios of the benchmarks compiled for the POWER2 processor. Figures 7 through 9 show the SPECratios of the benchmarks compiled for the PowerPC 601 processor. Figures 1 through 9 illustrate performance results in SPECratios to provide a normalized basis for performance comparison. These figures can be found at the end of the paper.

Using Basic Optimization

Adding the basic optimization flags substantially improves the performance of most applications. The primary optimization flag **-O** is equivalent to **-O2**. It performs only safe transformations on the code with very little context knowledge of the code. As a rough guide, applications compiled with **-O** will run between two and three times as fast as those with no optimization. Table 1 gives the increase in performance for the SPEC aggregates when the **-O** option is used. Figures 1, 4, and 7 graphically depict the data.

The amount of improvement will vary, depending on the implementation and the specific application. Overall, the compiler produces more dramatic increases on POWER and POWER2 systems than on PowerPC 601 systems when the **-O** option is added. On a benchmark-by-benchmark basis, adding **-O** yields the range of improvements shown in Table 2 for the six CINT92 and 14 CFP92 benchmarks.

Benchmark reports often use the best known set of options for the compiler and preprocessors. Controversy over which flags are acceptable when measuring benchmarks has developed as vendors become more aggressive in their search for higher SPEC numbers [2]. Table 3, as well as Figures 1 through 9 at the end of the paper, demonstrate that the **-O** option on its own can achieve a significant portion of this final performance.

In this release, the **-O3** level of optimization has been significantly enhanced. This option allows the compiler to use more time, more memory intensive transformations, and more aggressive optimization assumptions. Often, an improvement in the performance of an application results. Suitable floating-point applications should benefit the most from this more aggressive optimization level. An application is suitable if it does not raise exceptions, care about the sign of zero, and is not sensitive to side effects in precision from reassociating floating-point expressions. Applications dominated by memory accesses may also see some improvement from the more aggressive optimization assumptions.

The **-O3** optimization level does have the potential to change the semantics of a program when the **-qstrict** option is not specified. When **-O3** is used on its own, the compiler may move code that might trigger an exception. Specifically, the compiler can move loads and floating-point calculations into areas where they are determined to be profitable, even though the operations may not have been executed in a stricter interpretation of the original program. The compiler also ignores the sign of (floating-point) zero when this option is enabled. The compiler may reassociate floating-point expressions, providing more opportunities to detect common subexpressions in the calculations, and may also enhance floating multiply-add instruction generation. This may affect the rounding of results; equivalent mathematical results are delivered, but the answers may not be identical because the calculations are performed with finite precision. If an application is sensitive in any of these areas, some of the benefit can still be obtained by using **-O3** with the **-qstrict** option. The **-qstrict** option directs the compiler not to change the semantics of the program in the manner outlined previously. The **-O** option always has the **-qstrict** option implied.

When **-O3** is specified, the default settings of some related options change as well; specifically, **-qfloat=fltint:rsqrt** is selected and **-qmaxmem** is set to use unlimited memory during compilation. The **-qfloat=fltint** option removes range checking on conversion of floating-point values to integer values. The **-qfloat=rsqrt** option means that when a square root operation appears as the divisor in a calculation, the optimizer can substitute a multiply that uses the result from a reciprocal square-root library call. This eliminates the need for an expensive divide operation. The **-qmaxmem** option allows the user to specify the amount of memory to be used during compilation. When **-O3** is specified, this option's default settings allow the compiler to use as much memory as it needs during the optimization of an application. The use of this option generally increases compile time for complex applications. The **-qmaxmem** option defaults to a lower setting when the **-O** option is used.

Table 4 quantifies the improvements in the SPEC aggregates' performance realized by increasing the optimization from **-O** to **-O3**. The aggregate numbers are slightly misleading for the PowerPC 601 processor; certain individual benchmarks are improved by adding **-O3** while others became worse (with the early compilers). A user should always verify that **-O3** **does** improve the performance of an application before using it by default. By providing more information about the application and the target machine on which the application will run, further performance improvements may result.

Using Architecture Information

The new POWER2 floating-point instructions present opportunities, recognized by the compiler, that are not present on the POWER machines [3]. Similarly, the PowerPC 601 has single-precision floating-point instructions which are not present in the POWER Architecture [4]. The XL compilers introduce two new options: **-qarch** and **-qtune**. These options permit an application to exploit the new architectures. A binary created using the **-qarch** option may have instructions that will not run on architectures other than the one to which the binary is explicitly targeted.

Using the architecture-specific option (**-qarch**) and the **-O3** option together, a larger portion of the possible performance is obtained than with just **-O**. Table 5 illustrates this by comparing the aggregates for the SPEC benchmarks obtained with just the **-O3** and **-qarch** options applied compared to the aggregates obtained with the best known set of options. A comparison of Table 5 with Tables 3 and 4 illustrates the potential improvement for floating-point

applications on POWER2 and PowerPC 601 systems when the **-qarch** option is used.

The **-qarch** option tells the compiler which instruction set to use to generate code. Each machine architecture has its own set of valid instructions. Valid **-qarch** values are **pwr**, **pwr2**, **ppc**, and **com**. Using **-qarch=pwr** instructs the compiler to generate code for the POWER processors. The POWER, POWER2, and PowerPC 601 implementations all support the set of POWER user instructions generated by the compiler.

The **-qarch=pwr2** option instructs the compiler to use the instruction set of the POWER2 processor and to tune the application to this platform. If floating-point loads, stores, square roots [5], or double-precision calculations dominate an application, it may be beneficial to specify **-qarch=pwr2** when compiling the application to run on the POWER2 machines.

The **-qarch=ppc** option tells the compiler to generate code for the PowerPC Architecture and tune it for the 601 implementation of PowerPC Architecture. Applications containing single-precision floating-point code should benefit from this.

If binary compatibility across current and future machines is desired, the **-qarch=com** option should be used. Although this option currently does not generate a true intersection of the POWER and PowerPC Architectures, it instructs the compiler to generate primarily instructions which are common to both architectures. It may generate a couple of seldom-used instructions that exist on current machines and that will need to be emulated on future PowerPC implementations. However, it greatly minimizes the amount of emulation that will be required on these future systems. This setting is the default assumed by the compiler if the **-qarch** option is not explicitly specified.

If the user requires binary compatibility but expects to run the application predominantly on a specific set of systems, the **-qtune** option may improve performance. Valid settings are **pwr**, **pwr2**, and **601**. When **-qarch=com** is active, the default **-qtune** option is set to **pwr**. If the **-qarch** option is set to **pwr**, **pwr2**, or **ppc**, the default **-qtune** option is set to **pwr**, **pwr2**, or **601**, respectively. If an application is expected to run primarily on a POWER2 processor, it may be desirable to tune the generated code to favor that processor. This can be accomplished by specifying **-qtune=pwr2**, which will schedule for the multiple execution units of the POWER2 and perform some optimizations specific for that processor [6]. Similarly, if an application is expected to run predominantly on a PowerPC 601 machine, experimentation with the option **-qtune=601**

should be performed. Since the PowerPC 601 processor is a bridge between the POWER and PowerPC Architectures, some applications may perform better if **-qtune=pwr** is specified. If **-qtune** is specified without **-qarch**, the default is to assume **-qarch=com**. Table 6 indicates the valid combinations of the **-qarch** and **-qtune** options.

The **-qcache** option in the new Fortran compiler (XLF 3.1) provides control over the memory model used during the compilation. Customizing the assumptions about the sizes and accessing penalties for the various levels of the memory hierarchy allows the compiler to more accurately transform storage references. This is only effective when used with the **-qhot** option, which allows the compiler to manipulate the order of references to storage, minimizing the total penalty for accessing memory. Default compiler settings for the memory hierarchy configuration correspond to the **-qtune** options; if your machine does not match these configurations, adjusting the **-qcache** option can improve your performance [7]. This option is particularly important for Fortran applications that are dominated by patterned array accesses (such as stride-1, stride-n, and triangular).

C applications compiled for the POWER2 may benefit from the **-qunroll** option introduced in the C Set ++ 2.1 compiler. Some unrolling of inner loops will occur by default when **-O3** is specified. This option allows the user further control of the number of iterations that will be unrolled. On POWER2 machines, this may benefit applications whose inner loops are dominated by memory references.

Using Application Information

A compiler must always make pessimistic assumptions when there is no information to the contrary. Compiler options are one mechanism to communicate information that allows the compiler to relax assumptions and to produce faster correct code. Options aimed at minimizing call penalties, minimizing storage reference penalties, and enabling exploitation of native floating-point capabilities can enhance the performance of an application significantly beyond the basic optimization levels. These options do not affect the binary compatibility of applications across the POWER, POWER2, and PowerPC 601 implementations.

Table 7 shows the improvement of the SPEC aggregates from applying the additional compiler options compared to runs using only the **-O3** and **-qarch** options. Figures 1 through 9 show the result of applying these additional options to the SPEC benchmarks.

In these figures, the shading that indicates the best compiler options refers to the best set that was known at the time the runs were made. Better combinations of options may exist.

Some of these additional options focus on reducing the cost of calling and returning from routines in well-structured modular programs. Using the inlining and interprocedural analysis options while minimizing aliases can aid the compiler in improving the overall application performance.

Inlining and Interprocedural Analysis

In all of the XL compilers, the **-Q** option tells the compiler to automatically inline (copy the body of a called routine to the call site) the routines it believes are suitable, based on size [8,9,10]. This is only valid when one of the optimization options (**-O/-O3**) is applied as well. Manual control over which routines are inlined is also possible through **-Q+name1:name2** (inlines named routines) and **-Q-name1:name2** (excludes named routines from being inlined). If a routine is called from within a loop, inlining may be particularly beneficial, as certain optimizations can be enabled that were blocked by the presence of the call. Excluding specific calls from being inlined when they are not in the main path of execution (for example, debug code and error code) may also improve the performance if automatic inlining is enabled. One side effect of inlining code is that the overall code size usually increases. The overall performance of an application may decrease when inlining is specified as a result of a change in the size of the application or the order in which instructions are accessed.

The new XLF and C Set ++ compilers introduce a new option: **-qipa**. This option enables interprocedural analysis of the application, and, based on this information, the compiler may optimize the code further to improve the overall application performance [11]. It can optionally make use of the profile data provided by the **prof** and **gprof** commands, if they have been run for the application. This allows the optimizer to concentrate its efforts on the routines that are believed to dominate the application's running time. A significant increase in the compile time may result when this option is used, as the application must be compiled twice to allow the information to be collected and then applied to the entire application. The **-qipa** option should be particularly beneficial for large applications with minimal aliasing side effects in the calls between routines.

Aliasing

Aliasing is a compiler term for a storage location having multiple

variables that reference it. When potential aliases occur, they inhibit the assumptions a compiler can make when optimizing a program. One example of an aliased reference in Fortran is a variable in a COMMON block associated with other variables through an EQUIVALENCE statement. Although not legal by the ANSI standard for Fortran, another typical example is the use of the same variable for multiple parameters in a procedure call whose variables are passed by reference.

By default, the Fortran compiler does not check for aliasing side effects. In cases where aliasing in an application may violate the ANSI standard, it may be necessary to specify **-qxflag=xalias**. This option may slow down the application.

In C and C++ applications, pointers may result in pessimistic compiler aliasing assumptions. The **-qansialias** flag in the C compilers allows the compiler to assume ANSI compliant pointers (that is, the pointers in an application are known to point only to objects of the same type).

In the C Set ++ compiler, the options **-qdisjoint** and **-qassert** allow the user to provide further aliasing information to the compiler [7,10]. The **-qdisjoint** option allows the user to specify pointer variables in a function that are explicitly not aliased to each other. The **-qassert** option can be set to the following assertions:

- **type ptr** indicates that no two pointers to different types will point to the same storage location.
- **allptrs** indicates that no two pointers will point to the same storage location.
- **addrtaken** indicates that pointers never interact with variables unless one variable of a class had its address explicitly taken.

In code dominated by well-behaved pointers, the appropriate assertions allow the compiler more opportunities to improve the performance of an application. Aliasing should be avoided when possible.

Floating-Point Precision

Since POWER and POWER2 floating-point hardware runs in double-precision mode, true single-precision applications require "round to single-precision" instructions. Although transforming a single-precision application into a double-precision application sometimes has negative effects on cache performance, this

transformation often boosts performance significantly by eliminating the need for the explicit rounding operations. On POWER2, double-precision variables also allow the use of the Load Quad and Store Quad instructions.

This transformation can be automated by using the **-qautodbl=dblpad** option for Fortran applications. This option promotes REAL*4 and REAL*8 variables to REAL*8 and REAL*16 respectively and pads appropriately. XLF 3.1 introduces a new subset of values for this option, giving the user a greater degree of control over the data types promoted. Specifically, **-qautodbl=dblpad4** will pad and promote REAL*4 to REAL*8, but will not promote REAL*8 variables. In C and C++, there is no equivalent of this, but using the substitution **-Dfloat=double** may achieve the same effect.

If promoting an application to use double-precision values is not possible, the XL compilers provide options which minimize the single-precision rounding. The **-qfloat=norndsngl** tells the compiler to round only after the full expression has been evaluated. This is the default for the compiler. (If **-qfloat=rndsngl** is in effect, rounding occurs after every single-precision floating-point instruction.) The **-qfloat=hssngl** tells the compiler to keep all intermediate results in double-precision format and only convert to single-precision format when a store occurs. The **-qfloat=hsflt** option is even more aggressive; no explicit conversion to single-precision values occurs before storing the results and no checking occurs for overflows on floating-point to integer conversion. This option is not appropriate unless rounding errors are not an issue and the numbers can not be out of range.

The PowerPC 601 processor differs from the POWER and POWER2 processors in that it performs single-precision computations more efficiently than double-precision ones. When the compiler is allowed to generate single-precision floating-point instructions (through use of the **-qarch=ppc** option), the performance of applications dominated by single-precision computations usually improves.

If a floating-point application can accept a rounding error and can tolerate a looser implementation of the fringes of the IEEE 754 standard, then some of the **-qfloat** suboptions may be of benefit. The compiler assumes some of these suboptions by default for **-O3**. In addition to previously described **-O3** default suboptions, the **-qfloat=fold** option allows the compiler to evaluate constant floating-point expressions at compile time rather than at run time. The **-qfloat=nonans** option instructs the compiler not to worry about generating code to detect signaling NaNs (Not a Number) on

conversion from single-precision format to double-precision format at run time. Default settings for these options and others can be found under the description of **-O** and **-O3** options [8,9,10].

Using Source Preprocessors

Some applications may see further improvements in performance from the use of source preprocessors. These preprocessors read in a source file (currently only C and Fortran preprocessors are available), parse it, apply optimizations, and produce a modified source file for the compiler to optimize. There are currently three preprocessors available that work with the XL compilers. KAP for IBM C will work with the XLC and C Set ++ compilers. KAP for IBM Fortran from Kuck and Associates, Inc. and VAST-2 for XL Fortran from Pacific-Sierra Research will work both with the XLF compiler. Guidelines on how and when to use them can be found in AIX Version 3 for RISC System/6000: Optimization and Tuning Guide for Fortran, C and C++ [7].

These preprocessors contribute significantly to the final performance of some applications. Figures 1 through 9 illustrate this at the benchmark level. Table 8 shows the improvement in the SPEC aggregates that has been obtained through use of the preprocessors.

One set of floating-point applications which benefit from the use of the preprocessors are those containing algebraic algorithms that are also present in specially-tuned libraries. In code that contains algebraic algorithms that are equivalent to BLAS (Basic Linear Algebra Subroutines) or ESSL (Engineering and Scientific Subroutine Library) routines, the preprocessors will attempt to recognize the algorithm and transform it into an appropriate library call. In addition to the current version tuned for POWER processors, a version of the ESSL library tuned to the POWER2 processors exists. The BLAS routines are tuned to the POWER processors. Of the two Fortran preprocessors, our experiments show that VAST-2 for XL Fortran recognizes more of these special library idioms.

Another group of floating-point applications that benefits from the preprocessors are those dominated by loop-based accesses to large memory structures. The preprocessors allow the user a greater degree of control over the source transformations to be applied to exploit the memory organization of the machine. Of the two Fortran preprocessors, our experiments show that KAP for IBM Fortran performs more sophisticated memory management.

Applications written in C and dominated by I/O may benefit from the KAP for IBM C preprocessor's ability to reduce the overhead of

reading and writing variables using the **-stdio** option. In particular, those applications dominated by **printf** statements should benefit from this optimization [12].

Historically, integer-based applications have benefited from a preprocessor's ability to do inter-file inlining. This ability extended the XL compilers' native ability to inline code and allowed for the penalties of calls to be minimized for applications spread across several source files. This function may now be achieved through the **-qipa** and **-Q** options in the XL compilers as well. Experimenting with the preprocessor's inter-file inlining and the compiler's native ability to find the best one for an application is recommended.

Both integer and floating-point applications can also benefit from a preprocessor's ability to transform the source code to minimize the delays in a loop and maximize the parallelism available to the compiler. Transformations, such as unrolling, are under explicit user control in the preprocessor. This extends the capability provided by the XL compilers natively. For a further description of how unrolling benefits an application on the POWER2 environment, refer to "POWER2 CPU-Intensive Workload Performance" [5].

PowerPC 601 Performance

RISC System/6000 Model 250

Figure 7

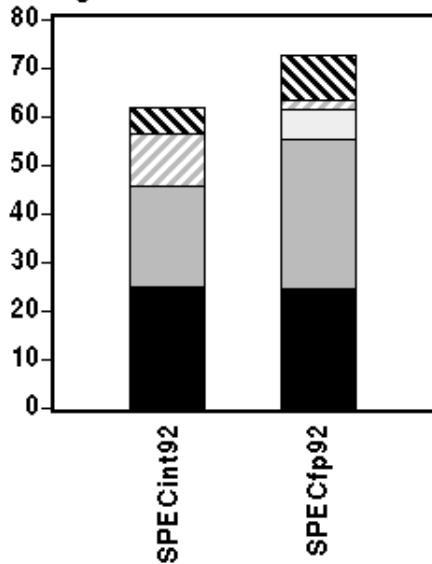


Figure 8: CINT92

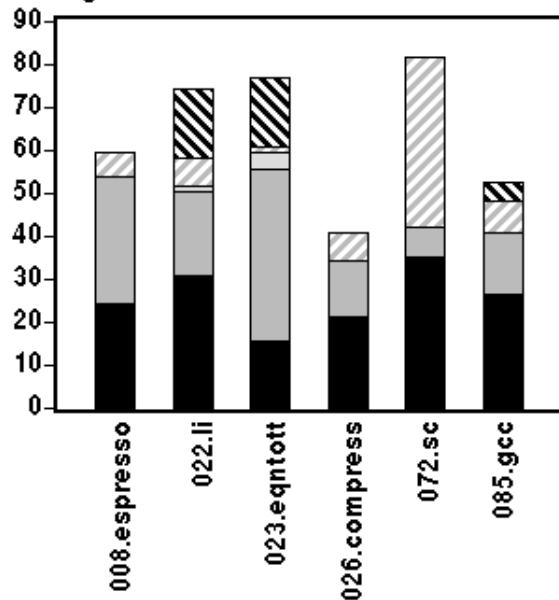
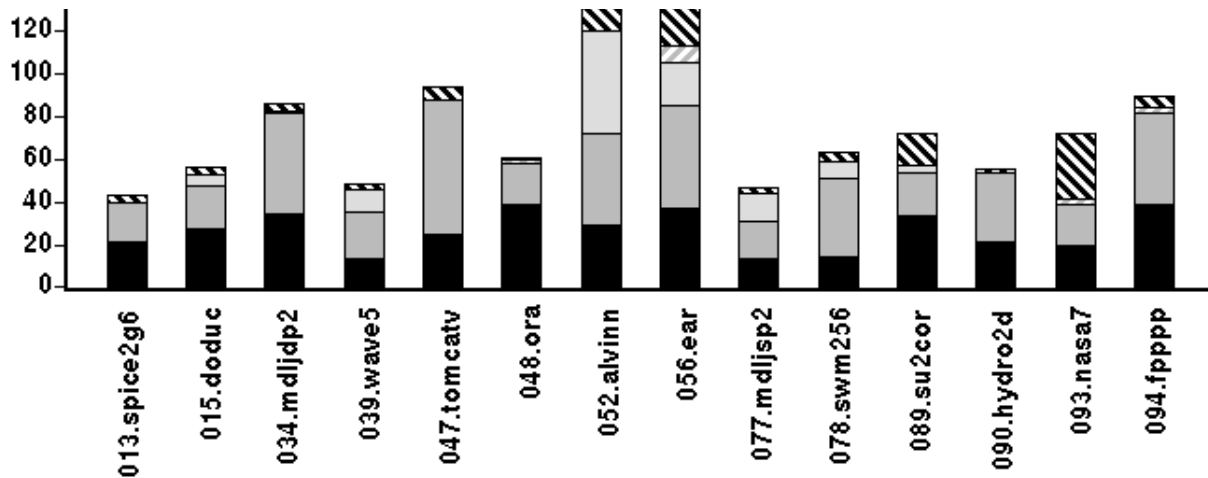


Figure 9: CFP92





- Best using preprocessors
- Best using compiler options
- O3 -qarch=ppc
- O
- No optimization used

ENVIRONMENT:

compilers: C Set ++ 2.1 alpha, XLF 3.1 beta

preprocessors: KAP/C 1.3, KAP/FORTRAN beta, PSR/VAST beta

operating system: AIX 3.2.0 beta

Further details on the SPEC options are available from the author.

[RS/6000](#) | [Solutions](#) | [Hardware](#) | [Software](#) | [Support](#) | [ReSource](#) | [sitemap](#)
[IBM](#) | [Order](#) | [Search](#) | [Contact IBM](#) | [Help](#) | © | ®

© Copyright IBM Corp. 1994, 1995, 1996. All rights reserved.
Last modified: Fri Sep 20 9:03:03 US/Eastern 1996