White Paper

Bonnie Aona
Developer Products
Division

Optimizing Applications with Intel® C++ and Fortran Compilers for Windows*, Linux*, and Mac OS* Version 10.x

Introduction

This document describes how developers can use Intel® compilers to optimize applications for IA-32 processors, processors supporting Intel® 64 (Intel® Extended Memory 64 Technology), and IA-64 (Intel® Itanium®) processors. It first shows some of the optimization features common to all of the processors, followed by optimizations for specific processors.

Overview

The Intel® C++ and Fortran Compilers for Windows*, Linux*, and Mac OS* optimize performance and give application developers access to the advanced features of IA-32 processors, processors supporting Intel® 64 (Intel® Extended Memory 64 Technology), and IA-64 (Intel® Itanium®) processors. New and improved compiler features include:

- **Flexibility.** Developers can target specific 32-bit or 64-bit Intel processors for optimization.
- **Visual C++* Compatibility.** Intel C++ Compiler is source- and object- compatible with the Microsoft Visual C++ Compiler. The Intel Visual Fortran Compiler for Windows* is object-compatible with Microsoft Visual C. For more information on specific compatibility of the compilers within the Visual C++ and Visual Studio .NET environments, refer to the Intel® C++ Compiler for Windows* Compatibility with Microsoft Visual C++* 6.0 and .NET white paper, and the compiler release notes and installation document.
- **Integration with Microsoft Visual Studio .NET 2005* and Visual Studio 2003*.** On IA-32 processor-based systems, both Intel compilers are integrated into Microsoft Visual Studio .NET IDE, including the 2005 edition. Microsoft Visual Studio 2005 Express Edition is supported for command line only use of the Intel® Visual Fortran Compiler for Windows*. The 10.0 Intel Visual Fortran Compiler for Windows* introduces a new optional feature, Microsoft Visual Studio 2005 Premier Partner Edition*, which permits installation and use of Intel Visual Fortran for IA-32 and Intel 64 applications without the separate purchase and installation of a Microsoft development product for systems where Microsoft Visual Studio .NET 2003* or Visual Studio 2005* has not already been installed. The Intel C++ Compiler also integrates into the Visual Studio 6.0 IDE. For more information on usage of the compilers within the Visual C++ and Visual Studio .NET environments, refer to the Intel® C++ Compiler for Windows* Compatibility with Microsoft Visual C++* 6.0 and .NET white paper, and the compiler release notes and installation document.
- **Linux* GCC Compatibility.** The Intel C++ Compiler for Linux* is substantially compatible with the GNU C++ Compiler Collection (GCC) and with the Eclipse* IDE version 3.1 with C/C++ Development Tools (CDT) version 3.0.0. The Intel C++ Compiler is binary-compatible with GCC for C-language object files and uses the GNU glibc C-language library. The Intel C++ Compiler supports the C++ Application Binary Interface (ABI), and by default in version 8.1 and later, uses the GNU C++ runtime library, which allows it to be binary-compatible with GCC for C++ language object files. The GNU mudflap library, for more secure programming, is supported with Intel® 10.0 Compiler for Linux*. For details on compatibility, please refer to the Intel® Compilers for Linux* – Compatibility with the GNU Compilers white paper, and the compiler release notes and installation document.
- **Mac OS* Compatibility.** The Intel C++ and Fortran Compilers can be used to compile programs for the Apple Macintosh* operating systems (Mac OS*). The compiler can be used from the command line or with the Xcode* IDE. In most cases, features and options supported for IA-32 and Intel 64 Linux*-based systems are also supported on Intel®-based systems running Mac OS*. C language object files created with the Intel C++ Compiler are binary compatible with the GNU gcc compiler and glibc, the GNU C runtime library. You can use the Intel C++ compiler or the gcc compiler to pass object files to the linker. However, using the Intel compiler will automatically pass the necessary Intel runtime libraries to the linker. The Intel C++ Compiler provides many of the language extensions provided by the GNU C compiler, gcc, and the GNU C++ compiler, g++.
- **Ease of Use.** Automatic optimization features let the compiler do the work necessary to take advantage of the target processor's architecture.
- **Efficiency.** Automatic compiler optimization reduces the need to write different code for different processors. The generated code is highly portable and easy to maintain.
- **Intel® Premier Support.** Intel provides training, product issue support, and more through the secure Intel® Premier Support Web site.
- **Intel Registration Center.** Intel provides product registration, a self-help question and answer feature, software downloads, and more through the secure Intel Registration Center Web site.

Common Intel Compiler Features for All Supported Intel® Processors

The Intel C++ and Fortran Compilers for Windows*, Linux*, and Mac OS* can compile applications for IA-32, Intel 64, and IA-64 processor-based systems, depending on which is the host processor.

On IA-32-based systems running Windows*, Linux* or Mac OS*, developers can also install compiler components to develop 64-bit applications for cross-compilation. However, we do not support cross-compilation from IA-32 to IA-64 on Linux*.

The Intel compilers provide a set of features and benefits common to systems based on all Intel processors, as well as features that are unique to each.

The common features include the following:

- General optimization settings.
- Cache-management features.
- Interprocedural optimization (IPO) methods.
- Profile-guided optimization (PGO) methods.
- Multi-threading support .
- Floating-point arithmetic precision and consistency.
- Compiler optimization and vectorization reports.

These common optimization features are described below.

Windows* Switch Setting	Linux*/Mac OS* Equivalent	Comment
/Od	**-O0**	No optimization. Used during the early stages of application development and debugging; left for a higher setting when the developer knows the application is working correctly.
/O1	**-O1**	Optimize for size. Omits optimizations that tend to increase object size. Creates the smallest code in the majority of cases. This option is useful in many large server/database applications where memory paging due to larger code size is an issue.
/O2	**-O2**	Maximize speed. Default setting. Creates faster code than /O1 (-O1) in most cases.
/O3	**-O3**	Enables /O2 (-O2) optimizations plus more aggressive loop and memory-access optimizations, such as scalar replacement, loop unrolling, code replication to eliminate branches, loop blocking to allow more efficient use of cache and, on IA-64-based systems only, additional data prefetching. The /O3 (-O3) option is particularly recommended for applications that have loops that heavily use floating-point calculations or process large data sets. These aggressive optimizations may occasionally slow down other types of applications compared to /O2 (-O2).
/ZI	**-g**	Generates debug information for use with any of the common platform debuggers. This option turns off /O2 (-O2) and makes /Od (-O0) the default unless /O2 (-O2) (or another O option) is specified.
/debug:full	**-debug full**	Allows easier debugging of optimized code by adding full symbol information, including the local symbol table information, regardless of the optimization level. This may result in minor performance degradation. If this option is specified for an application that makes calls to C library routines that will be debugged, the option /dbglibs must also be specified to link the appropriate C debug library.

Table 1. General Optimization Switch Settings and Their Uses

General Optimization Settings

All Intel compilers automatically optimize applications for the target processor when developers select a switch setting. Table 1 lists the general switch settings that perform automatic optimizations and describes their typical uses.

Use the General Optimization Options (Windows* **/O1**, **/O2**, or **/O3**; Linux* and Mac OS* **-O1**, **-O2** or **-O3**) and determine which one works best for your application by measuring performance with each. Most users should start at **/O2 (-O2)** (default) before trying more advanced optimizations. Next, try **/O3 (-O3)** for loop-intensive applications, especially on IA-64-based systems. Fine-tune performance to target systems based on IA-32 and Intel 64 with processor-specific options such as **/QxT (-xT)** for Intel® Core ™2 processor family. For dual-Core Intel® Itanium® 2 9000 Sequence processors, set **/G2-p9000 (-mtune=itanium2-p9000)**. For a complete list of recommended options for specific processors, see Table 11. "Recommended Optimization Options for Specific Intel® Processors". Before you begin performance tuning, you may want to check correctness of your application by building it without optimization using **/Od (-O0)**. The General Optimization Options should be at the heart of any application tuning for all 32-bit and 64-bit Intel processors.

Interprocedural Optimization

Interprocedural optimization (IPO) is another optimization that works with all Intel compilers. Developers activate IPO through compiler settings (see Table 2). IPO can improve application performance significantly in programs that contain many frequently used small or medium-sized functions. It is especially beneficial for applications that contain calls to these types of functions within loops.

IPO Benefits

IPO enhances application performance through the following optimizations:

- Decreasing the number of branches, jumps, and calls within code; this reduces overhead when executing conditional code.
- Reducing call overhead further through function inlining.
- Providing improved alias analysis, which leads to better code vectorization and loop transformations.
- Enabling limited data layout optimization, resulting in more efficient cache usage.
- Performing interprocedural analysis of memory references, which allows registerization of more memory references and reduces application memory accesses.

How IPO Works

IPO is a two-step, automatic process.

- **Step One: Compilation** – IPO creates an information file that contains an intermediate representation of the source code and summary information used for optimization.
- **Step Two: Linking** – For all modules with a current corresponding information file, IPO allows the compiler to analyze your code to determine where you can benefit from a variety of optimizations:

For IA-32, Intel 64 and IA-64 -based applications:

- Inline function expansion of calls, jumps, branches, and loops.
- Interprocedural constant propagation for arguments, global variables, and return values.
- Monitoring module-level static variables to identify further optimizations and loop invariant code.
- Dead code elimination to reduce code size.
- Propagation of function characteristics to identify call deletion and call movement.
- Identification of loop-invariant code for further optimizations to loop invariant code.

For IA-32 applications only:

- Passing arguments in registers to optimize calls and register usage.

Windows* Switch Setting	Linux*/Mac OS* Equivalent	Comment
/Qip	**-ipo[value]**	Single file optimization. Interprocedural optimizations, including selective inlining, within a single source file. Caution: For large files, this option may sometimes significantly increase compile time and code size.
/Qipo[value]	**-ipo[value]**	Permits inlining and other interprocedural optimizations among multiple source files. The optional **value** argument controls the maximum number of link-time compilations (or number of object files) spawned. Default **value** is 0 (the compiler chooses). Caution: This option can in some cases significantly increase compile time and code size.
/Qipo-jobs[n]	**-ipo-jobs[n]**	Specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO). The default is 1 job.
/Ob2	**-finline-functions** **-finline-level=2**	This option enables function inlining at the compiler's discretion. This option is enabled by default at **/O2** and **/O3** (**-O2** and **-O3**). Caution: For large files, this option may sometimes significantly increase compile time and code size. It can be disabled by **/Ob0** (**-fno-inline-functions** on Linux* and Mac OS*).
/Qinline-factor=n	**-finline-factor=n**	This option scales the total and maximum sizes of functions that can be inlined. The default value of **n** is 100, i.e., 100% or a scale factor of one.
/Qprof-gen	**-prof-gen**	Instruments a program for profiling.
/Qprof-use	**-prof-use**	Enables the use of profiling information during optimization.
/Qprof-dir dir	**-prof-dir dir**	Specifies a directory for the profiling output files, *.dyn and *.dpi.

Table 2. Interprocedural Optimization Compiler Switch Settings

Function Inlining

For frequently executed function calls, inlining copies the body of the function to the calling location. This improves application performance by the following means:

- Removing the need to set up parameters for a function call.
- Eliminating the function call branch.
- Propagating constants.

Profile-Guided Optimization

Profile-guided optimization (PGO) is a three-step compilation process that improves performance when applied to typical application runtime workloads. While IPO looks for performance gains by reviewing application logic, PGO looks for performance gains in the way the application logic is applied to typical uses.

Although PGO is an independent optimization method, it is more effective when developers use it in conjunction with other optimizations, especially IPO.

PGO Benefits

- PGO improves instruction cache usage. It moves frequently accessed code segments adjacent to one another, and moves seldom accessed code to the end of the module. This eliminates some branches and shrinks code size, resulting in more efficient processor instruction fetching.
- PGO increases application performance by improving branch prediction. PGO also generates branch hints for Intel processors during the optimization process.

Applications with the following characteristics are well suited to PGO:

- Applications containing several potential execution paths, some of which the application executes much more frequently than others.
- Large applications with many function calls or branches (especially when used with IPO).

How PGO Works

Please refer to Figure 1.

- **Step One: Instrumented Compilation** – Activate PGO with the compiler switch **/Qprof-gen** (**-prof-gen** on Linux* and Mac OS*). The compiler creates an instrumented program from the source code.
- **Step Two: Instrumented Execution** – Run the instrumented program on one or more typical input data sets. Because different input data sets may result in different optimizations, it is important to choose input data sets that are representative of typical application use. The compiler generates dynamic information files for each run, recording how frequently each code section executes.
- **Step Three: Feedback Compilation** – Recompile the application with the switch **/Qprof-use** (**-prof-use** on Linux* and Mac OS*); PGO feeds the execution data back to the compiler. This action merges the dynamic information files from all the Step Two runs into a profile summary file. The compiler uses the profile summary file to optimize execution of the most heavily traveled paths in the finished application.

  Note: To use IPO together with PGO, apply the IPO switch(es) during the PGO feedback compilation (Step Three).

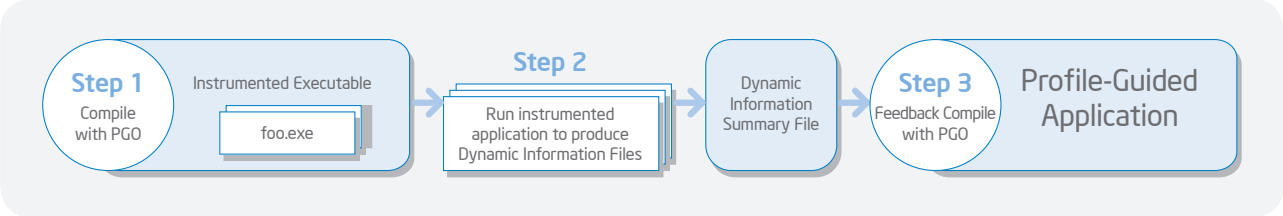


Figure 1. Profile-Guided Optimization (PGO) Steps

Multi-Threading Support

For systems with Hyper-Threading Technology , multi-core and/or multiple processors, Intel compilers support development of multi-threaded applications through two mechanisms:

- **Auto-parallelization.** When the compiler switch **/Qparallel** (**-parallel** on Linux* and Mac OS*) is specified, the compiler detects loops that may benefit from multi-threaded execution, and automatically generates the appropriate threading calls.
- **OpenMP* directives.** When the compiler switch **/Qopenmp** (**-openmp** on Linux* and Mac OS*) is specified, the compiler recognizes industry-standard OpenMP directives (version 2.5). These directives give developers explicit control of the way their application is multi-threaded. Please refer to the following documents for more information on OpenMP directives and their use:

– OpenMP Fortran Application Program Interface, Version 2.5, May 2005 at http://www.openmp.org

Compiler Optimization Reports

The Intel compilers include several optimization reports that provide information on different aspects of the compilation process. Developers can use this information to adjust the program so that the compiler can generate more highly optimized code.

To generate any of the optimization reports, use the switch **/Qopt-report** (**-opt-report** on Linux* and Mac OS*). To select the specific optimization report, use the switch **/Qopt-report-phase <phase>** (**-opt-report-phase <phase>** on Linux* and Mac OS*).

For example: **$ icc -opt-report –opt-report-phase ecg_swp main.c**

Specifying **/Qopt-report-help** (**-opt-report-help** on Linux* and Mac OS*) gives a list of all the possible values for **<phase>**. Table 3 lists key **<phase>** values, with examples of more fine-grained selections.

Phase	Architecture	Description
all	IA-32, Intel 64, IA-64	All possible optimization reports for all phases are enabled (results can be very verbose).
ipo **ipo_inl**	IA-32, Intel 64, IA-64	Optimizations performed as part of the Interprocedural Optimization phase. **ipo_inl** gives only the report on function inlining. This inlining report may be obtained whether or not the **/Qipo (-ipo)** option is selected.
hlo **hlo_prefetch**	IA-32, Intel 64, IA-64	Optimizations performed as part of the High-Level Optimization phase. **hlo_prefetch** gives only the report on compiler-generated prefetching.
hpo	IA-32, Intel 64, IA-64	Optimizations performed as part of the High Performance Optimizer.
ilo	IA-32, Intel 64, IA-64	Optimizations performed as part of the Intermediate-Language Scalar Optimizer.
ecg **ecg_swp**	IA-64	Optimizations performed as part of the Code Generator phase. **ecg_swp** gives only the report on software pipelining. Note: For Mac OS*, this option is not supported.
pgo	IA-32, Intel 64, IA-64	Indicates which parts of the program have profiling information available for use in Profile Guided Optimization.

Table 3. Optimization Report Families Available in Intel® Compilers

Windows* Switch Setting	Linux*/Mac OS* Equivalent	Comment
/fp:name	**-fp-model name**	This method of controlling the consistency of floating point results by restricting certain optimizations is recommended in preference to the **/Op (-mp)** and **/Qprec (-mp1)** switches. The possible values of name are: **precise** – Enables only value-safe optimizations on floating-point code. **double/extended/source** – Implies **precise** and causes intermediates to be computed in double, extended, or source precision. The **double** and **extended** options are not available for Intel® Fortran Compiler. **fast=[1|2]** – Allows aggressive optimizations at a slight cost in accuracy or consistency (**fast=1** is the default). **except** – Enables floating-point exception semantics. **strict** – Strictest mode of operation, enables both the **precise** and **except** options and disables fma contractions. Recommendation: **/fp:source (-fp-model source)** is the recommended form for the majority of situations on IA-64 processors, on processors supporting Intel® 64, and on IA-32 when SSE are enabled with **/QxW (-xW)** or higher where enhanced floating point consistency and reproducibility are needed.
/Qfp-speculation mode	**-fp-speculation mode**	Enables floating-point speculations with one of the following modes: **fast** – Speculate floating-point operations. (default) **off** – Disables speculation of floating-point operations. **safe** – Do not speculate if this could expose a floating-point exception.. **strict** – This is the same as specifying off.
/Qftz[-]	**-ftz[-]**	When the main program or dll main is compiled with this option, denormal results are flushed to zero for the whole program (dll). Setting this option does not *guarantee* that all denormals in a program are flushed to zero. They only cause denormals generated at run time to be flushed to zero. On IA-64-based systems, the default is off except at **/O3 (-O3)**. On IA-32- based systems and Intel® 64-based systems, the default is on except at **/Od (-O0)**, but only denormals resulting from SSE instructions are flushed to zero.

Table 4. Floating-Point Data Options Available in Intel® Compilers

Consistency of Floating-Point Arithmetic Optimizations

The Intel compilers provide options for enhancing the consistency or precision of floating-point results on Intel architecture as described in Table 4. These options allow you to find the best tradeoffs between floating-point consistency and performance for a given application.

Using **/fp:name** (**-fp-model:name** on Linux* and Mac OS*) is preferable to using **/Op (-mp)** or **/Qprec (-mp1)**. More information and finer grained options are available in the Intel Compiler Documentation under "Compiler Options."

Optimizations Specific to IA-32 Processors and Processors Supporting Intel® 64

The Intel compilers employ a number of optimization methods for the latest IA-32 processors and processors supporting Intel 64. Instruction selection and scheduling choose the best instruction sequences for speed and latency; vectorization takes advantage of the single-instruction, multiple data (SIMD) instruction sets; and various other loop optimizations improve memory-access latency. Intel compilers have processor-specific targeting options which utilize these optimizations to automatically target the latest IA-32 processors while also allowing for the flexibility of running the resulting executables on other processors.

Beginning with Intel compilers 9.1, the **-mtune** switch is available on Linux* and replaces the –tpp switches for the various IA-32 processors and processors supporting Intel 64.

Cache Management for Streaming Stores

The Intel C++ and Fortran Compilers optimize applications by reducing memory latency and cache pollution. Intel compilers accomplish this optimization by means of **streaming stores**. By automatically generating streaming stores to bypass the cache and store data directly to memory, Intel compilers reduce cache pollution from data that will not be reused. This leaves the cache free for reuse by other data.

Vectorization and Loop Optimization

Vectorization detects patterns of sequential data accesses by the same instruction and transforms the code for SIMD execution, including use of the SSE, SSE2, SSE3, SSSE3 and SSE4.1 instruction sets.

Intel C++ and Fortran Compilers automatically vectorize code. The vectorizer supports the following features:

- **Multiple data types.** The vectorizer supports the **float/double** and **char/short/int/long** types (both signed and unsigned), as well as the **_Complex float** and **_Complex double** data types.
- **Step-by-step diagnostics.** Through the **/Qvec-reportN** (**-vec-reportN** on Linux* and Mac OS*) setting, the vectorizer can identify, line-by-line and variable-by-variable, what code was vectorized, what code was not vectorized, and more importantly, why it was not vectorized. This feedback gives the developer the information necessary to slightly adjust or restructure code, with dependency directives and *restrict* keywords, to allow vectorization to occur.
- **Advanced, dynamic data-alignment strategies.** Alignment strategies include loop peeling and loop unrolling. Loop peeling can generate aligned loads, enabling faster application performance. Loop unrolling matches the prefetch of a full cache line and allows better scheduling.
- **Portable code.** As Intel introduces new processor technology, developers can use appropriate Intel compiler switches to take advantage of new processor features. This can eliminate extensive rewriting of source code.

Processor-Specific Optimization

The following optimization switches enable the compiler to generate optimized and specialized code for a specific processor and allow the compiler's vectorizer to generate SIMD instructions using SSE, SSE2, SSE3, SSSE3, and/or SSE4, depending on the targeted processor.

- Options of the form **/Qx<id>** (**-x<id>** on Linux* and Mac OS*) generate specialized and optimized code for processor extensions specified by code. The resulting executables from these processor-specific options can be run only on the specified or later processors. For example, an executable targeted for a generic Pentium 4 processor will also run on a Pentium 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support. For Mac OS*, **-xP** is default, and **–xT** is supported on Intel® Core™2 Duo processor.
- Options of the form **/Qax<id>** (**-ax<id>** on Linux* and Mac OS*) generate both specialized code and generic IA-32 or Intel 64 code through the processor-dispatch technology described later. For Mac OS*, **–axT** is supported on Intel® Core™2 Duo processor.
- Table 5 lists possible values for the **<id>** option.

The **O**, **W** and **N** designators all generate executables that run on any Pentium 4 or Pentium M processor. The difference between the three options is in the optimizations performed. The section *Recommended Optimization Settings for Intel Pentium 4 and Pentium M Processors* explains recommendations for their use.

Processor-Specific Runtime Checking

When using the processor-specific targeting options, **/Qx{}** (**-x{}** on Linux* and Mac OS*), developers must be careful to run the resulting executables only on compatible targeted processors. Execution-time errors may occur if such a specialized executable is run on the wrong processor. In some cases, the compiler inserts a runtime check that determines whether the Intel processor that the application is running on is a compatible one.

The **/Qx{N, P, S, T}** (**-x{N, P, S, T}** on Linux* and **-x{P, S, T}** on Mac OS*) options generate an error message if the application is run on an incompatible processor:

> "Fatal Error: This program was not built to run on the processor in your system."

For this check to be effective, ensure that the main program or the main module of a dynamic library is compiled with the options.

<id>	Value
S	May generate a code path using SSE4 Vectorizing Compiler and Media Accelerators for a future Intel processor. This code path may also generate SSSE3, SSE3, SSE2, and SSE instructions. This code path optimizes for a future Intel® processor. The generic code path does not contain such features.
T	May generate a code path using SSSE3, SSE3, SSE2, and SSE instructions for Intel processors. This code path optimizes for the Intel® Core™2 Duo processor family, Quad-Core Intel® Xeon® processors, and Dual-Core Intel® Xeon® 5300, 5100 and 3000 series processors.
P	May generate a code path using SSE3, SSE2, and SSE instructions for Intel processors. This code path optimizes for for Intel® Core™ microarchitecture, Intel® Pentium® 4 processors with SSE3, Intel® Xeon® processors with SSE3, Intel® Pentium® dual-core processor T2060, Intel® Pentium® Extreme Edition processor, and Intel® Pentium® D processor. Performs optimizations not enabled with **/QxO (-xO)**.
O	May Generate SSE3, SSE2, and SSE* instructions. Optimizes for Intel® Core™ microarchitecture, Intel® Pentium® 4 processors with SSE3, Intel® Xeon® processors with SSE3, Intel® Pentium® dual-core processor T2060, Intel® Pentium® Extreme Edition processor, and Intel® Pentium® D processor. Code path may execute on Intel® and Non-Intel Processors which support SSE3. This option is supported for **/Qx (-x)** only.
N	May generate a code path using SSE2 and SSE instructions for Intel® processors. This code path optimizes for the Intel® Pentium® 4 processor, Intel® Xeon® processor with SSE2, and Intel® Pentium® M processor. Performs optimizations not enabled with **/QxW (-xW)**.
W	May generate a code path using SSE2 and SSE* instructions. This code path optimizes for the Intel Pentium® 4 processor and Intel® Xeon® processors with SSE2. This code path may execute on Intel and Non-Intel Processors which support SSE2.
K	May generate a code path using SSE instructions. This code path optimizes for Intel® Pentium® III and Intel Pentium® III Xeon® processors. This code path may execute on Intel and Non-Intel Processors which support SSE*.

Table 5. Possible Values for <id> Option

* The option values **O**, **W**, and **K** produce binaries that should run on processors not made by Intel such as AMD processors that implement the same capabilities as the corresponding Intel processors. **P** and **N** option values perform additional optimizations that are not enabled with option values **O** and **W**.

With **/Qx{K, W}** (**-x{K,W}** on Linux*) or when unable to compile the main program or main module of a dynamically linked library, no such runtime check is made. Thus, execution time failures such as an "illegal instruction" fault may result if the application is run on the wrong processor. For this reason, the **K** and **W** options may be removed in future releases of the Intel compilers.

The **/Qax{}** (**-ax{}** on Linux* and Mac OS*) switches also employ a runtime check, but because of the processor dispatch technology described below, they do not cause such runtime errors, even when an application is run on a processor other than the one targeted.

Processor Dispatch

Processor dispatch allows developers to optimize applications targeting one or more specific IA-32 processors.

For example, you may want to take advantage of the performance features in the Pentium 4 processor, while maintaining compatibility with earlier processors. You may choose automatic or manual modes of processor dispatch. Usually, a developer selects automatic processor dispatch and lets the compiler do the work to tune the application for one target processor.

Both modes produce an optimized generic code version of the application to run on systems using an Intel processor other than one for which the application is specifically optimized. At runtime, the application automatically identifies the Intel processor on which it is running and selects the appropriate implementation, either specialized or generic.

Automatic Processor Dispatch

Automatic processor dispatch **/Qax{}** (**-ax{}** on Linux* and Mac OS*) allows developers to tell the Intel compiler to choose the most efficient optimizations and instructions for the Pentium 4 processor or any other IA-32 or Intel 64 processor.

For example, the **/QaxN** (**-axN** on Linux*) compiler switch generates specialized code for the Pentium 4 processor while also generating generic IA-32 code.

Multiple **<id>** values can be combined to generate an executable that is optimized for multiple target processors.

For example, compiling with **/QaxNP** (**-axNP** on Linux*) generates an executable compatible with all IA-32 processors but that also has optimal code paths for Pentium 4 processors and for Pentium 4 processors with SSE3 instruction support. The correct path is selected at execution time.

One caveat regarding automatic processor dispatch is that the code size of the resulting executable will be larger than that generated by the processor-specific targeting switch **/Qx{}** (**-x{}** on Linux* and Mac OS*). This is because the resulting executable will have multiple versions of functions in which the compiler finds processor-specific optimization.

You can also combine the processor-specific targeting switch with an automatic dispatch switch. The effect of this is to set the base-level targeting for the generic code path that is generated. For example, specifying "**/QaxP /QxW**" (**-axP –xW**) will generate two code paths: the "generic" code path will be set as "**W**" and the maximum optimization code path will be "**P**". This setting is useful for the majority of applications that will run on both IA-32 and Intel 64 capable systems, including those from AMD as described in the recommendations section below.

Manual Processor Dispatch

Manual processor dispatch is useful when the developer wants to write explicit, hand-optimized code for one or more target processors.

Table 6 describes some of the key differences between the manual and automatic dispatch methods.

	Manual Dispatch	Automatic Dispatch
Compatible Intel® compilers	Intel® C++ Compiler only	Intel C++ and Intel® Fortran Compilers
Coding for processor-specific functions	Developer hand-codes processor-specific function versions for each processor the application will support.	Developer codes only one version of each function.
Benefits	▪ Single executable file. ▪ Developer can write explicit code to take advantage of processor-specific features using vector classes, intrinsic functions, and inline assembly.	▪ Single executable file. ▪ No need to hand-code optimizations for the target processor; automatic optimization and vectorization for the specified processor by compiling with the appropriate switch.
Considerations	▪ Must validate on all targeted platforms ▪ Larger code size for multiple targeted processors ▪ Possible slightly larger call overhead ▪ Some inlining disabled	

Table 6. Comparison of Manual and Automatic Processor Dispatch Methods

Recommended Optimization Settings for IA-32 and Processors Supporting Intel® 64

Intel Compilers that support processors with Intel 64 are a separate binary from the IA-32 compilers and generate only 64-bit addressable code. Some of the processor-targeting options function differently for IA-32 processors than for processors supporting Intel 64. This section describes the recommended processor-specific optimization settings for these different processors.

Each application has unique characteristics that affect performance; therefore, it is best to employ a data-driven, experimental approach in trying the various options to see which provides the best performance for your application.

Recommended Optimization Settings for IA-32 Processors

For best performance on IA-32 processors with SSE3 instruction support, the recommended optimization setting is **/QxP** (**-xP** on Linux* and Mac OS*). If your application must also run on older IA-32 processors or non-Intel processors (i.e., AMD processors) that support SSE2, but not SSE3, use **/QxW** (**-xW** on Linux*).

For the best performance on IA-32 processors and compatible Intel processors with SSE2, use **/QxN** (**-xN** on Linux*).

To create applications that are optimized for the latest Pentium 4 processors with and without SSE3, and yet will run on any Intel processor or AMD Athlon or Opteron processors, specify **/QaxNP** (**-axNP** on Linux*). This potentially generates three code paths: one generic for all processors, one for processors with SSE2 instruction support targeted for the Pentium 4 processor, and one targeted for the Pentium 4 processor with SSE3 instruction support. (Note that the application code size could increase due to the multiple processor code paths.) Table 7 summarizes this recommendation.

Recommended Optimization Settings for processors with Intel 64

For best performance on processors supporting Intel 64 utilizing SSE3 when possible, the recommended optimization setting is **/QxP** (**-xP** on Linux*). If your application must run on AMD Athlon 64 or AMD Opteron processors, use **/QaxP /QxW** (**-axP –xW** on Linux*) to generate a binary that will utilize SSE3 and be tuned for non-SSE3 x86-64 processors via CPU dispatch.

One can also use **/QxW** (**-xW** on Linux*) to create an optimized application for processors supporting Intel 64 but without generating SSE3 instructions. As mentioned earlier, **/QxW** (**-xW** on Linux*) does not generate a runtime check for whether an appropriate processor is found. Thus, the resulting binary can run on AMD Athlon 64 and AMD Opteron processors.

For best performance on Intel® Core™2 Duo processors, Intel® Core™2 Extreme processors, and the Dual-Core Intel® Xeon® 5100 series processors, the recommended optimization setting is **/QxT** (**-xT** on Linux* and Mac OS*). If your application must run on AMD processors, use **/QaxT /QxW** (**-axT -xW** on Linux*) to generate a binary that will utilize SSSE3 and be tuned for non-SSSE3 x86-64 processors via CPU dispatch. If your application must run on AMD processors that support SSE3, use **/QxO** (**-xO** on Linux*) to generate a binary that will utilize SSE3 instructions.

The **/QaxT /QxW** (**-axT -xW** on Linux*) combination will produce binaries with two code paths, using the process-dispatch technology described above. One code path will take full advantage of Intel Core 2 Duo processors, Intel Core 2 Extreme processors, and the Dual-Core Intel Xeon 5300, 5100, and 3000 series processors. The other code path also takes advantage of the capabilities provided by the Intel Core Duo processor, Intel Core Solo processor, Intel Pentium 4 processor, Intel Xeon Processor which include SSE2, and other compatible processors that include SSE2 and SSE such as AMD* processors.

Note: The resulting binary will not run on systems which do not include the SSE2 instruction set, which includes older Intel processors through the Intel® Pentium® III processor. SSE2 provides significant floating point optimization capability.

Tables 7 and 8 summarize these recommendations.

Need	Recommendation	Caveat
Best performance on Intel IA-32 processors with SSE3 instruction support	/QxP (-xP on Linux* and Mac OS*)	Single code path; will only run on Intel processors with SSE3 support.
Best performance on non-Intel processors that support SSE3	/QxO (-xO on Linux*)	Single code path; will not run on earlier processors without SSE3 support. Will run on Intel and AMD systems with SSE3 support.
Good performance on processors supporting SSE or SSE2 including non-Intel processor-based systems, without cpu dispatch	/QxW (-xW on Linux*)	Single code path; will not run on earlier processors without SSE2 support. Will run on Intel and AMD systems with SSE2 support.
Best performance on IA-32 processors with SSE3 instruction support for multiple code paths	/QaxP /QxW (-axP –xW on Linux*): Optimized for Pentium 4 processor and Pentium 4 processor with SSE3 instruction support	Generates two code paths: ▪ one for the Intel Pentium 4 processor with SSE3 ▪ one for the Intel Pentium 4 processor or non-Intel processors (i.e., AMD processors) with SSE2 instruction support.

Table 7. Recommended IA-32 Processor Optimization Options (not for processors supporting Intel® 64)

Need	Recommendation	Caveat
Best performance on future Intel processors that support SSE4 Vectorizing Compiler and Media Accelerators	/QxS (-xS on Linux* and Mac OS*)	Single code path; will not run on earlier processors.
Best performance on Intel® Core™2 Duo processors, Intel® Core™2 Extreme processors and Dual-Core Intel® Xeon® 5100 series processors, utilizing SSSE3 and other processor-specific instructions where possible	/QxT (-xT on Linux* and Mac OS*)	Single code path; will not run on earlier processors.
Best performance on Intel® Core™2 Duo processors, Intel® Core™2 Extreme processors and Dual-Core Intel® Xeon® 5100 series processors, utilizing processor-specific instructions where possible, while still running on older Intel as well as non-Intel x86-64 compatible processors supporting SSE3 or SSE2	/QaxT /QxW (-axT -xW on Linux*)	Generates two code paths: ▪ one for the Intel Core 2 Duo processors with SSSE3 ▪ one for Intel and non-Intel processors (i.e., AMD processors) with SSE2 instruction support.
Good performance on processors supporting Intel 64 with SSE or SSE2 including non-Intel x86-64 processor-based systems, without cpu dispatch	/QxW (-xW on Linux*)	Single code path; will not run on earlier processors without SSE2 support. Will run on Intel and AMD systems with SSE2 support.
Best performance on non-Intel processors that support SSE3	/QxO (-xO on Linux*)	Single code path; will not run on earlier processors without SSE3 support. Will run on Intel and AMD systems with SSE3 support.

Table 8. Recommended Intel® 64 Optimization Options

Optimizations Specific to Intel IA-64 Processors

Intel compilers automatically take advantage of the advanced features of IA-64 architecture. Intel compilers enable the following IA-64 processor-specific optimizations:

- Instruction scheduling
- Predication
- Branch prediction
- Speculation
- Software pipelining
- High-performance floating-point optimizations

Instruction Scheduling

On Windows, the, **/G2** and **/G2-p9000** switches are available for IA-64 processors; on Linux* use the"**-mtune**" switches. These switches enable optimal instruction scheduling and cache management for the various Intel processors without making the generated code incompatible with earlier processors, and are described in Table 9.

The **/G2** (**-mtune=itanium2** on Linux*) compiler switch is now on by default. The **/G2** (**mtune=itanium2**) switch enables optimal instruction scheduling and cache management for the Intel Itanium 2 processor without making the generated code incompatible with earlier processors.

The **/G2-p9000** (**-mtune=itanium2-p9000** on Linux*) compiler switch is now available. The **/G2-p9000** switch optimizes for Dual-Core Intel® Itanium® 2 Processor 9000 Sequence processors. This option affects the order of the generated instructions, but the generated instructions are limited to the Intel Itanium 2 processor instructions unless the program uses (executes) intrinsics specific to the Dual-Core Intel Itanium 2 9000 Sequence processors.

Windows* Switch Setting	Linux* Equivalent	Comment
/G2	-mtune=itanium2	Targets optimization for the Itanium 2 processor. Generated code is also compatible with all IA-64 processors (default).
/G2-p9000	-mtune=itanium2-p9000	Targets optimizations for Dual-Core Intel Itanium 2 9000 Sequence processors. Generated code is also compatible with all IA-64 processors, unless the user program calls intrinsic functions specific to the Dual-Core Intel Itanium 2 Processor 9000 Sequence processors.

Table 9. Intel® Itanium® (IA-64) Processor Optimization Options

Predication

Traditional architectures implement conditional execution through branch instructions. The IA-64 processor implements conditional execution with **predicated instructions**.

Removal of branches from program sequences is one of the most important optimizations that predication enables. This results in larger basic blocks and eliminates associated branch misprediction penalties, both of which contribute to improved application performance. Furthermore, since fewer branches exist after predication, dynamic instruction fetching is more efficient, because there are fewer possibilities for control flow changes.

Branch Prediction

Branch prediction attempts to collect all the instructions likely to execute after a branch and places those instructions into an instruction cache. When the branch is predicted correctly, those collected instructions are easily accessible when the processor is ready to execute them, causing the application to run faster.

IA-64 architecture allows the compiler to communicate branch information to the processor, thus reducing the number of branch mispredictions. It also enables the compiled code to manage the processor hardware using runtime information. These two features are complementary to predication and provide the following performance benefits:

- Applications with fewer branch mispredictions run faster.
- The performance cost from any mispredicted branches that may remain is reduced.
- Applications have fewer instruction-cache misses.

Speculation

Speculation is a feature of the IA-64 processor that allows assembly language developers or the compiler, based on conjecture, to improve performance by performing some operations (such as costly load instructions) out of sequence before they are needed.

To ensure that the code is correct in all cases, and not just when the conjecture is correct, the compiler executes recovery code as needed. Recovery code corrects all affected operations if the original conjecture or speculation was false.

There are two kinds of speculation: control speculation and data speculation.

Control Speculation

When the compiler performs control speculation, it moves a load above a conditional branch. It then places a "check operation" at the position of the original load. The check operation identifies whether an exception has occurred on the speculative load, and if so, it branches to recovery code. Figure 2 illustrates this type of speculation.

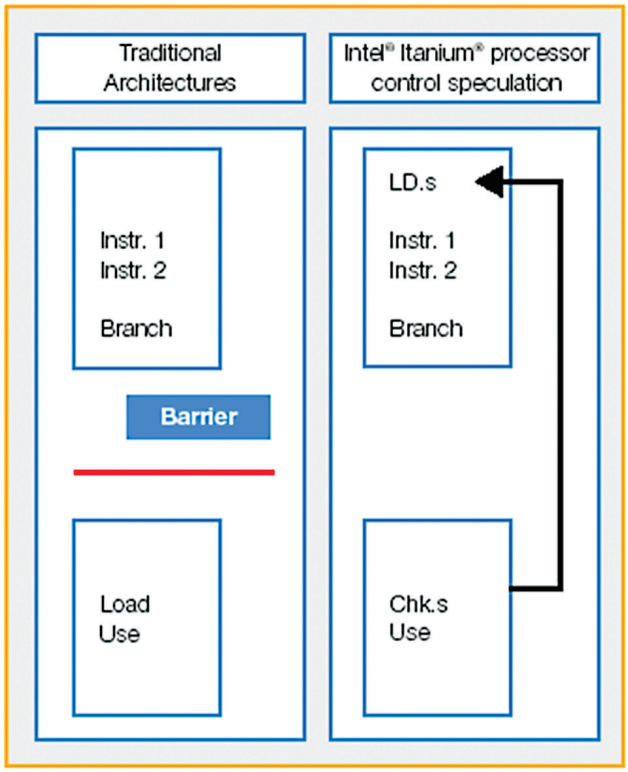


Figure 2. Control Speculation

Control Speculation Benefits

Among the benefits of control speculation are the following:

- Gives developers more control over when and where to use instructions in an application.
- Helps to hide memory latencies by moving loads earlier in the code.
- Is very effective at working around branch barriers in code, leading to improved application performance, because it executes a load operation before evaluating the conditional branch.

Data Speculation

Like control speculation, data speculation is a mechanism to hide memory latencies.

In traditional architectures, if the load operation depends on a store operation, then the load operation cannot be moved ahead of the store. This can seriously limit the ability of the compiler to hide memory latencies.

Data speculation creates dependency checks in the code and provides a recovery mechanism should data dependencies exist. Data speculation can hide memory latencies by allowing the compiler to move the load above the store. This enables a load to execute prior to the store preceding it. This occurs even in the case of an ambiguous memory dependency, where it is unknown at compile time whether the load and the store reference overlapping memory addresses.

In Figure 3, the barrier on the left-hand side represents an ambiguous memory dependency, which prevents the load from executing prior to the store. Data speculation can remove this barrier. The right-hand side illustrates how the compiler avoids the traditional barrier by advancing the load and inserting a "check instruction" to verify that no memory overlaps occurred. If a memory overlap (ambiguous memory dependency) exists, then recovery code executes to validate the code.

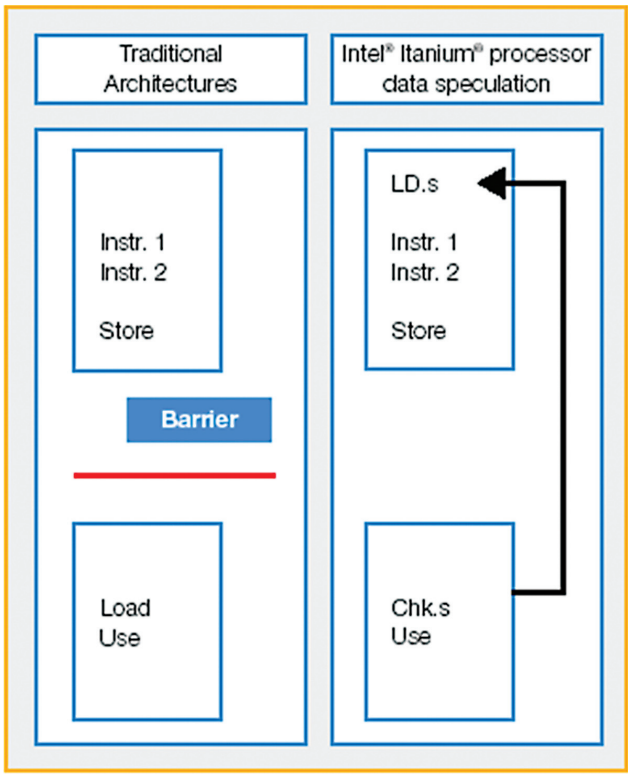


Figure 3. Data Speculation

Data Speculation Benefits

Data speculation can accomplish the following goals:

- Resolve ambiguous memory dependencies, making it highly beneficial for working around data-dependency barriers in code.
- Significantly reduce memory latencies and improve application performance, because it enables load operations to execute ahead of the stores preceding them.

Software Pipelining

Software pipelining reduces the number of clock cycles necessary to process a loop by increasing parallelism at the instruction level. It attempts to overlap loop iterations by dividing each iteration into stages with several instructions in each stage.

Software pipelining is on by default in the Intel C++ and Fortran Compilers (the /O2 setting; -O2 on Linux* and Mac OS*). Not all loops may benefit from software pipelining. For loops that can benefit, however, combining software pipelining with predication and speculation boosts application performance by significantly reducing code expansion, path length, and branch mispredictions.

Data Prefetching

Data prefetching reduces memory latency and improves application performance by intelligently calling up data before the program requires it. Data prefetching inserts prefetch instructions at appropriate points in the application when /O3 (-O3 on Linux* and Mac OS*) is specified. By placing the referenced data into cache memory before the application calls for it, prefetch instructions are able to overlap memory accesses with other computations. This can improve performance significantly in applications that have a regular pattern of memory accesses.

Some benefits of data prefetching include:

- Data prefetching is automatic.
- Data prefetching coordinates with other optimizations (such as software pipelining).
- Compiler-generated prefetching keeps code portable. The developer does not need to manage this aspect of application performance in source code to write processor-specific instructions. The compiler generates data prefetching appropriate for the targeted processor(s).

High-Performance, Floating-Point Optimizations

Sometimes, even if a loop can be software pipelined, the execution latency of the hardware executing the code can increase the loop iteration time.

Intel IA-64 processor provides 128 directly addressable floating-point registers. These enable pipelined floating-point loops and reduce the number of load and store operations, as compared to traditional processor architectures.

#pragma	Architecture	Description
swp **noswp**	IA-64	Place before a loop to override the compiler's heuristics for deciding whether to software pipeline the loop.
loop count(n)	IA-32 Intel® 64 IA-64	Place before a loop to communicate the approximate number of iterations the loop will execute. This affects software pipelining, vectorization, and other loop transformations.
distribute point	IA-64	Place before a loop to cause the compiler to attempt to distribute the loop based on its internal heuristic. Place within a loop to cause the compiler to attempt to distribute the loop at the point of the pragma. All loop-carried dependencies will be ignored.
unroll **unroll(n)** **nounroll**	IA-64	Place before an inner loop (ignored on non-inmost loops). **#pragma** unroll without a count allows the compiler to determine the unroll factor. **#pragma unroll(n)** tells the compiler to unroll the loop n times. **#pragma nounroll** is the same as **#pragma unroll(0)**.
prefetch a,b,... **noprefetch x,y,...**	IA-64	Place before a loop to control data prefetching. This is supported when **/Q3 (-O3)** is on. **#pragma prefetch** a causes the compiler to prefetch for future accesses to array a. The compiler determines the prefetch distance. **#pragma noprefetch x** causes the compiler to not prefetch for accesses to array **x**.
vector always **vector aligned** **vector unaligned** **novector**	IA-32 Intel 64	Place before a loop to control vectorization. **#pragma vector** always overrides compiler heuristics and attempts to vectorize despite non-unit strides or unaligned accesses. **#pragma vector aligned** vectorizes if possible, using aligned memory accesses. **#pragma vector unaligned** vectorizes if possible, but uses unaligned memory accesses. **#pragma novector** disables vectorization for the loop.
vector notemporal	IA-32	Place before a loop to cause the compiler to generate non-temporal (streaming) stores within the loop body.
IVDEP [:option]	IA-32 Intel® 64 IA-64	**IVDEP** is a compiler directive that assists the compiler's dependence analysis of iterative DO loops by asserting to the compiler's optimizer about the order of memory references inside a DO loop. **IVDEP** can have an option value specified, where option can be **LOOP** or **BACK**. **IVDEP:LOOP** implies no loop-carried dependencies; **IVDEP:BACK** implies no backward dependencies.
MEMREF_CONTROL	IA-64	**MEMREF_CONTROL** is a compiler directive that lets you provide cache hints on prefetches, loads, and stores. At least one address argument must be specified and may include optional locality and latency values; an optional second address value with optional locality and latency values may also be specified.

Table 10. Compiler #pragma Directives Available in Intel®Compilers 9.0 and Higher

Compiler Directive Support in Intel Compilers Version 9.0 and Above for Windows*, Linux*, and Mac OS*

Several memory alignment and #pragma (CDIR$ in Fortran) directives for controlling compilation were recently added to Intel compilers. Some are common to both 32-bit and 64-bit architectures and others are specific to one or the other.

Memory Alignment Directives

Intel compilers for IA-32 processors allow programmers to specify a base alignment and an offset from that base for compiler-allocated memory.

For example:

```
__declspec(align(32, 8)) double A[128];
```

The compiler allocates A with a base address that is aligned 32x+8.

Note: Data alignment can have a significant effect on performance. Be careful to specify optimal alignment.

#pragma (CDIR$) Directives

Table 10 describes directives recently added to the Intel Compilers.

For Best Performance on Processor	Windows*	Mac OS*	Linux*
A future Intel processor which includes SSE4 Vectorizing Compiler and Media Accelerators	**/QxS** **/QaxS**	**-xS** **-axS**	**-xS** **-axS**
Intel® Core™2 Extreme processor Intel® Core™2 Duo processor Dual-Core Intel® Xeon® 5300, 5100 and 3000 series processors Quad-Core Intel® Xeon® processors	**/QxT** **/QaxT**	**-xT** **-axT**	**-xT** **-axT**
Intel® Core™ Duo, Intel® Core Solo processor Intel® Pentium® 4 processor with Streaming SIMD Extension 3 (SSE3) instruction support Intel® Pentium® D processor Intel® Xeon® processor (only on processors that support SSE3) Intel® Pentium® dual-core processor T2060 Intel® Pentium® Extreme Edition processor Dual-Core Intel®Xeon®7000, 5000, and 3200 Sequence processors Dual-Core Intel® Xeon® ULV and LV processor Dual-Core Intel® Xeon® 2.8 processor	**/QxP** **/QaxP**	**-xP**	**-xP** **-axP**
Intel processor-based systems supporting SSE2 and SSE* Non-Intel processor-based systems supporting SSE3, SSE2, and SSE* such as AMD processors	**/QxO**		**-xO**
Intel Pentium 4 processor Intel® Pentium® M processor Intel Xeon processors without SSE3 support (IA-32 only)	**/QxN** **/QaxN**		**-xN** **-axN**
Intel processor-based systems supporting SSE* Non-Intel processor-based systems supporting SSE2 and SSE* such as AMD processors	**/QxW** **/QaxW**		**-xW** **-axW**
Intel® Pentium® III processors Intel® Pentium® III Xeon® processors Non-Intel x86 processor-based systems supporting SSE* such as AMD processors	**/QxK** **/QaxK**		**-xK** **-axK**
Intel® Itanium® 2 processor	**/G2**		**-mtune=itanium2**
Dual-Core Intel® Itanium® 2 9000 Sequence processors	**/G2-p9000**		**-mtune=itanium2-p9000**

Table 11. Recommended Optimization Options for Specific Intel® Processors

* The option values O, W, and K produce binaries that should run on processors not made by Intel, such as AMD processors, that implement the same capabilities as the corresponding Intel processors. P and N option values perform additional optimizations that are not enabled with option values O and W.

Summary of Intel Compiler Features and Benefits

Intel Architecture-Specific Optimization

The Intel C++ and Fortran Compilers enable programmers to develop specific architecture optimizations for IA-32 processors, processors supporting Intel® Extended Memory 64 Technology (Intel® 64) and Intel® Itanium® processors (IA-64). The Intel compilers automatically perform many optimizations to maximize application performance, reducing the need for hand-optimizing source code. By changing compiler option settings, developers can still choose the right set of performance-optimization characteristics for their applications. Selections range from no optimization, to general-purpose optimization, to optimization based on how the application is used.

Table 11 identifies the recommended optimization options for each Intel processor. As with each previous step, measure the performance benefit of each option to guide your decisions. Use the Intel compilers' optimization reports to assist in determining whether you can provide more help to the compiler to resolve possible dependencies or aliases and improve memory utilization.

Source Code Portability

Developers who use the automatic optimizations of the Intel C++ and Fortran Compilers can easily tune applications to make the best use of the features of various processors without spending long hours on custom coding.

Intel Premier Support

Intel compiler documentation, tutorials, documented examples, and context-sensitive help files that come with the Intel C++ and Fortran Compilers answer most developer questions. Developers should also register for Intel Premier Support (see the References section for how to register).

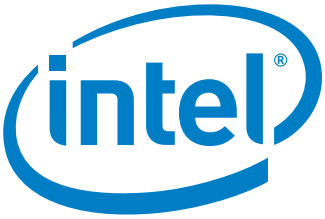
One year of Intel Premier Support through a secure Internet site is included with the purchase of Intel compilers. In addition to support for the Intel C++ and Fortran Compilers, developers can access other useful information:

- FAQs and other proactive notices
- Issue tracking and updates
- Software update and patch downloads
- User forums for interactive discussions with fellow developers

References

Additional information related to the Intel C++ and Fortran Compilers is available:

- General information on Intel C++ and Fortran Compilers: http://www.intel.com/software/products
- Documentation, application notes, and source code for libraries, tools, and code examples: http://developer.intel.com/software/products/opensource
- Information on IA-64 architecture: http://developer.intel.com/products/processor/itanium2/index.htm
- Information on training available for Intel® Software Development Products: http://www.intel.com/software/college
- Intel Premier Support home page: https://premier.intel.com/
- Additional information on OpenMP, including the complete specification and list of directives: http://www.openmp.org/
- Quick Reference Guide to optimization with the Intel Compilers: http://www.intel.com/software/products/compilers/docs/qr_guide.htm
- Apple Development Connection for Mac OS* applications: http://developer.apple.com/



For product and purchase information visit:
www.intel.com/software/products

0505/DXM/OMD/PDF 300348-002

Hyper-Threading Technology requires a computer system with an Intel® Pentium® 4 processor supporting HT Technology and a Hyper-Threading Technology enabled chipset, BIOS, and operating system. Performance will vary depending on the specific hardware and software you use. See www.intel.com/info/hyperthreading for more information including details on which processors support HT Technology