



Optimizing Applications with the Intel[®] C++ and Fortran Compilers

for Windows* and Linux*
Updated for Intel[®] Compilers 9.0

Table of Contents

Executive Summary	3	Optimizations Specific to Intel Itanium Processors	14
Common Intel® Compiler Features for All Supported Intel® Processors	4	Instruction Scheduling.....	14
Automatic Optimization Settings	4	Predication.....	15
Interprocedural Optimization	5	Branch Prediction	15
IPO Benefits.....	5	Speculation.....	15
How IPO Works.....	5	Control Speculation.....	15
Function Inlining.....	5	Control Speculation Benefits	15
Profile-Guided Optimization	6	Data Speculation.....	16
PGO Benefits.....	6	Data Speculation Benefits	16
How PGO Works	6	Software Pipelining.....	16
Creating Linux-Shared Object Files	7	Data Prefetching	17
Multi-Threading Support	8	High-Performance, Floating-Point Optimizations	17
Compiler Optimization Reports	8	Compiler Directive Support in Intel Compilers Version 9.0 for Windows and Linux	17
Floating-Point Arithmetic Optimizations.....	9	Memory Alignment Directives	17
Optimizations Specific to Intel® Pentium® 4 Processors and Processors Supporting Intel® EM64T	9	#pragma (CDIR\$) Directives	17
Pentium 4 Processor Scheduling	9	Summary of Intel Compiler Features and Benefits	19
Cache Management for Streaming Stores.....	9	Intel® Architecture-Specific Optimization	19
Vectorization and Loop Optimization.....	10	Source Code Portability	19
Processor-Specific Optimization.....	10	Compatibility with Other Compilers	19
Processor-Specific Runtime Checking	11	Intel® Premier Support.....	19
Processor Dispatch.....	11	References	19
Automatic Processor Dispatch.....	12		
Manual Processor Dispatch	12		
Recommended Optimization Settings for IA-32 and Processors supporting Intel EM64T	12		
Recommended Optimization Settings for Intel Pentium 4 and Pentium M Processors	13		
Recommended Intel EM64T Optimization Settings.....	14		

Executive Summary

The Intel® C++ and Fortran Compilers for Windows* and Linux* optimize performance and give application developers access to the advanced architectures of processors supporting Intel® Extended Memory 64 Technology (Intel® EM64T), the Intel® Pentium® 4 processor, Intel Pentium M processor, and Intel® Itanium® 2 processors. The compilers feature several improvements to maximize application performance:

- **Flexibility:** Developers can target specific 32-bit or 64-bit Intel® processors for optimization, including Pentium 4, Pentium M, and Itanium processors, as well as processors supporting Intel EM64T
- **Visual C++* Compatibility:** The Intel C++ Compiler for Windows is source- and object-compatible with the Visual C++ compiler. It also integrates into the Visual Studio* .NET IDE. The Intel® Visual Fortran Compiler for Windows provides integration into the Microsoft Visual C++ .NET environment on IA-32 systems only. For more information on specific compatibility and 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.

- **Linux GCC Compatibility:** The Intel C++ Compilers for Linux have substantial compatibility with the GNU Compiler Collection (GCC). For details on compatibility, please refer to the [Intel® Compilers for Linux* – Compatibility with the GNU Compilers](#) white paper.
- **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. Code is highly portable and easy to maintain.
- **Intel® Premier Support:** Intel provides training, answers to specific questions, software patches, and more through the secure [Intel® Premier Support](#) site.

This document focuses on how developers can use the Intel® Compilers to optimize applications for Itanium architecture, Pentium M and Pentium 4 processors, and processors supporting Intel EM64T. It first shows some of the optimization features common to all of the processors, followed by optimizations for specific processors.

Common Intel® Compiler Features for All Supported Intel® Processors

Intel C++ and Fortran Compilers for Windows and Linux can compile applications for IA-32 processors (32-bit applications), for processors supporting Intel EM64T (32-bit or 64-bit applications), or for Itanium processors (64-bit applications), depending on which is the host processor.

On IA-32-based systems running Windows, developers can also install compiler components to develop for 64-bit applications (cross-compilation).

The Intel Compilers present a set of features and benefits that are common to systems based on all Intel processors, as well as features that are unique to each. The common features include the following:

- Automatic optimization settings
- Cache-management features
- Interprocedural optimization (IPO) methods
- Profile-guided optimization (PGO) methods
- Multi-threading support
- Compiler optimization and vectorization reports

These common optimization features are described below.

Automatic Optimization Settings

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

Table 1. Automatic Optimization Switch Settings and Their Uses

Windows* Switch Setting	Linux* Equivalent	Comment
/O_d (No Optimizations)	-O₀	Used during the early stages of application development; left for a higher setting when the developer knows the application is working correctly.
/O₁	-O₁	Omits optimizations that tend to increase object size. Creates the smallest code in the majority of cases.
/O₂ (Maximum Speed)	-O₂	Default setting. Creates the fastest code in most cases, but may increase code size significantly over /O₁ . On IA-32 Linux*-based systems, -O₁ and -O₂ are equivalent.
/O_x (Maximum Optimizations)	n/a	Equivalent to /O₂ except that /O_x does not imply /G_y (function packaging) or /G_f (string pooling).
/O₃ (High-Level Optimizations)	-O₃	Same as /O₂ , plus loop transformations and cache optimizations. To get the full benefit of the /O₃ switch with IA-32 processors, developers must also use the /Q_x or /Q_{ax} switches for Intel® Pentium® M and Pentium 4 processors, as well as for processors supporting Intel® EM64T.

Table 2. Interprocedural Optimization Compiler Switch Settings

Windows* Switch Setting	Linux* Equivalent	Comment
/Qip	-ip	Single-file optimization. Allows inlining and other optimizations within a single source file.
/Qipo[<i>value</i>]	-ipo[<i>value</i>]	Multi-file optimization. Permits inlining and other optimizations across multiple source files; <i>value</i> specifies the maximum number of object files to be produced. For example, <code>/Qipo4</code> specifies a maximum of four object files the compiler may choose to create. The default, if <i>value</i> is unspecified, is one file.

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 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 better 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 invokes the compiler again and performs function inlining.

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
- Constant propagation

Two compiler settings govern the mode of interprocedural optimizations, including automatic function inlining (see Table 2).

Profile-Guided Optimization

Profile-guided optimization (PGO) is a three-step compilation process that improves performance when applied to typical application runtime loads. 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 the Pentium 4 and Itanium 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

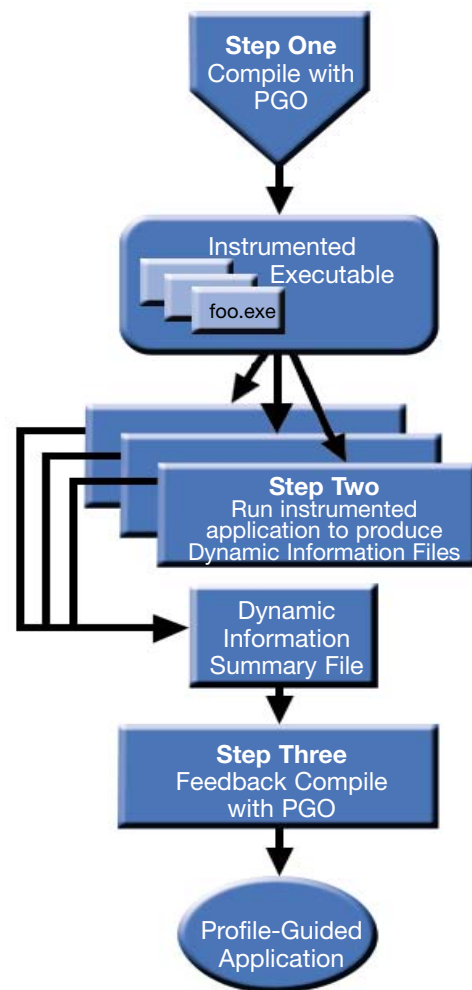
Step One: Instrumented Compilation

A developer activates PGO with the compiler switch `/Qprof_gen` (`-prof_gen` on Linux). The compiler creates an instrumented program from the source code.

Step Two: Instrumented Execution

The developer runs 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.

Figure 1. Profile-Guided Optimization (PGO) Steps



Step Three: Feedback Compilation

When the developer recompiles the application with the switch `/Qprof_use` (`-prof_use` on Linux), PGO feeds the execution data back to the compiler. This 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 illustrates the steps involved in compiling with PGO.

Creating Linux*-Shared Object Files

Creating Linux-shared object files (.so files) when using IPO and PGO also requires using shared objects. The following examples are based on 32-bit compilation; the principle is the same on 64-bit systems. Optimizations used in this build process are described in other sections of this document.

Step 1a: Compile the Objects to Create .o Files

Example :

```
$ gcc -c -O3 -axW -prof_dir /tmp -prof_gen ini.c lib1.c lib2.c
```

The **-prof_dir** switch specifies the directory for profiling output files. The compiler may warn of disabling several of the optimizations in the process of instrumenting the binary; this is normal.

Step 1b: Link the Objects to Create a Shared Object

Example :

```
$ xild -shared -soname libpi.so.1 -o libpi.so.1.0 lib1.o lib2.o ini.o
```

xild is the driver to the **'ld'** command and is provided with the Intel Compiler; **xild** enables IPO by default. For builds that do not involve IPO, use the **'xild -qnoipo'** command or the **'ld'** command directly.

Step 1c: (Optional) Install the Shared Library in Your Standard System Location

Example :

```
$ cp libpi.so.1.0 /usr/local/lib
$ cd /usr/local/lib
$ ldconfig -v -n .
$ ln -s libpi.so.1 libpi.so
```

The example installs the shared library in /usr/local/lib. The standard **'ldconfig'** tool can be used for this purpose.

Step 1d: Build Example Application(s) to Use the Shared Library

Example:

```
$ cd app_directory
$ gcc main.c -o main -lpi
```

NOTE: The optimization option for **main.c** need not correspond to the optimization options of the shared library. However, Intel recommends that you invoke the advanced optimizations on the main program as well, for maximum performance (not shown in the example).

Step 2: Verify 'main' Dependency and Run the Application

Example:

```
$ ldd ./main
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
$ ./main
```

NOTE: You can use the standard **'ldd'** tool to determine dependency information. In addition to running the application, profile information (.dyn file) is now created in the /tmp directory (as specified by the **-prof_dir** switch).

Step 3: Rebuild Shared Library with Profile Information

Example:

```
$ gcc -c -O3 -axW -ipo -prof_dir /tmp -prof_use ini.c lib1.c lib2.c
```

This is the same command as in Step 1a with **-prof_gen** replaced with **-prof_use** and **-ipo** added.

The **-prof_dir** option now points to the location of the intermediate files.

Repeat steps 1b through 1d to obtain the optimized application.

NOTE: Step 3 is required only in the presence of **-prof_gen** and **-prof_use**. If PGO is not used, steps 1a-1d result in the optimized application.

Multi-Threading Support

For systems with Hyper-Threading Technology¹, multi-core and/or multiple processors, the Intel Compilers support development of multi-threaded applications through two mechanisms:

- **Auto-parallelization:** when the compiler switch **/Qparallel** (**-parallel** on Linux) is specified, the compiler detects loops that may benefit from multi-threaded execution, and it automatically generates the appropriate threading calls.
- **OpenMP* directives:** When the compiler switch **/Qopenmp** (**-openmp** on Linux) is specified, the compiler recognizes industry-standard OpenMP directives (version 2.0). These directives give developers explicit control of how their application is multi-threaded.

Compiler Optimization Reports

The Intel Compilers 9.0 include several optimization reports that give 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). To select the specific optimization report, use the switch **/Qopt_report_phase <phase>** (**-opt_report_phase <phase>** on Linux).

For example:

```
$ gcc -opt_report -opt_report_  
phase ecg_swp main.c
```

Specifying **/Qopt_report_help** (**-opt_report_help** on Linux) gives a list of all the possible values for **<phase>**. Table 3 lists key **<phase>** values, with examples of more fine-grained selections.

Table 3. Optimization Report Families Available in Intel® Compilers 9.0

Phase	Architecture	Description
all	IA-32 Intel® Itanium® architecture	All possible optimization reports are enabled (results can be very verbose).
ipo ipo_inl	IA-32 Itanium architecture	Optimizations performed as part of the Interprocedural Optimization phase. For example, ipo_inl reports on function inlining.
hlo hlo_prefetch	Itanium architecture	Optimizations performed as part of the High-Level Optimization phase. For example, hlo_prefetch reports on compiler-generated prefetching.
ilo	Itanium architecture	Optimizations performed as part of the Intermediate-Language phase.
Ecg Ecg_swp	Itanium architecture	Optimizations performed as part of the Code Generator phase. For example, ecg_swp reports on software pipelining.
Pgo	IA-32, Itanium architecture	Optimizations performed as part of the Profile-Guided Optimization phase.

Table 4. Floating-Point Data Options Available in Intel® Compilers 9.0

Windows* Switch Setting	Linux* Equivalent	Comment
/Op	-mp	Restricts optimization to maintain declared precision and to ensure that floating-point arithmetic conforms more closely to the ANSI and IEEE standards. This has the greatest performance impact but is the strictest mode of operation.
/Qprec	-mp1	Improves floating-point precision. This option disables fewer optimizations and has less impact on performance than the options mp or Op . It enables IEEE NaN compare semantics, implies -Qprec_div and -Qprec_sqrt , disables inline expansion of some math intrinsics, and rounds to source precision prior to floating-point compares.
/fp:name	-fp-model name	New option in the Intel C++ Compiler 9.0, except the Intel Fortran Compiler 9.0 for Windows. The possible values of name are as follows: <ul style="list-style-type: none"> • precise – enables only value-safe optimizations on floating-point code • double/extended/source – enables intermediates to be computed in double, extended, or source precision • fast – allows aggressive optimizations at the expense of accuracy (this is the default) • except – enables floating-point exception semantics • strict – strictest mode of operation; enables both the precise and except options and disables contractions.

Floating-Point Arithmetic Optimizations

The Intel Compilers provide options for floating-point data and precision on IA-32, processors supporting Intel EM64T, and Itanium architectures as described in Table 4. More information and finer grained options are available in the Users Guide under “Floating-Point Arithmetic Optimizations.”

Optimizations Specific to Intel® Pentium® 4 Processors and Processors Supporting Intel® EM64T

Pentium 4 Processor Scheduling

The Intel Compilers employ a number of optimization methods for Pentium 4 and Pentium M processors and processors supporting Intel EM64T. 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; various other loop optimizations improve memory-access latency.

The Intel Compilers’ processor-specific targeting options utilize these optimizations to automatically target Pentium 4 processors while also allowing for the flexibility of running the resulting executables on other processors

Beginning with version 7.0 of the Intel Compilers, the **/G7 (-tpp7** on Linux) compiler switch is on by default. This switch enables optimal instruction scheduling and cache management for the Pentium 4 and Pentium M processors and processors supporting Intel EM64T, without making the generated code incompatible with earlier processors.

Cache Management for Streaming Stores

The Intel C++ and Fortran Compilers for Windows and Linux optimize applications by reducing memory latency and cache pollution. The 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, the Intel C++ and Fortran Compilers reduce cache pollution from data that will not be reused. This leaves the cache free for other data that may be reused.

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, and SSE3 instruction sets.

The 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) 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, and/or SSE3, depending on the targeted processor.

- Options of the form **/Qx<code> (-x<code>** on Linux) 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.
- Options of the form **/Qax<code> (-ax<code>** on Linux) generate both specialized code and generic IA-32 code, through the processor-dispatch technology described later. Table 5 lists possible values for the **<code>** option.

Table 5. Possible Values for <code> Option

<code>	Value
K	Generate instructions and optimize for Intel® Pentium® III processors and AMD Athlon* XP processors.
W	Generate instructions and optimize for Pentium 4, AMD Athlon* 64 and Opteron* processors, and processors supporting Intel® EM64T.
N	Generate instructions and optimize for Pentium 4 and compatible processors including SSE and SSE2. This option also performs some new Pentium 4 processor-specific optimizations not enabled with W .
B	Generate instructions and optimize for the Pentium® M and compatible processors based on Intel® Centrino™ mobile technology ² , which includes SSE and SSE2.
P	Generate instructions and optimize for the Pentium 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support, including processors supporting Intel EM64T. This option supports SSE3 as well as SSE and SSE2.

NOTE: The Intel Fortran and C++ Compilers 8.x and later no longer support the **i** (Pentium Pro processor) and **M** (Intel® MMX™ technology) codes. Warnings appear if **/Qxi**, **/Qaxi** (**-xi**, **-axi** on Linux) or **/QxM**, **/QaxM** (**-xM**, **-axM** on Linux) are specified on the command line. In versions of the Intel Compilers beyond the Version 8.1 release, the **K** and **W** options are deprecated and will eventually be removed.

The **W**, **B**, 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 Intel Pentium 4 Processor Optimization Settings* explains recommendations for their use.

Processor-Specific Runtime Checking

When using the processor-specific targeting options, **/Qx{}** (**-x{}** on Linux), 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.

In the Intel Compilers version 8.x and above, the **/Qx{N, B, P}** (**-x{N, B, P}** on Linux) 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.

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. This is why the **K** and **W** options may eventually be removed in later releases of the Intel Compilers.

The **-Qax{}** (**-ax{}** on Linux) 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)` 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 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 `<code>` 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)`. This is because the resulting executable will have multiple versions of functions in which the compiler finds processor-specific optimization.

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.

Recommended Optimization Settings for IA-32 and Processors Supporting Intel EM64T

The Intel Compilers that support processors with Intel EM64T are a separate binary from the IA-32 compilers and generate only 64-bit addressable code. Thus, some of the processor-targeting options function differently for IA-32 processors than they do for processors supporting Intel EM64T. This section describes the recommended processor-specific optimization settings for these different processors.

Of course, all applications are different and have different 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.

Table 6. Comparison of Manual and Automatic Processor Dispatch Methods

Feature	Manual Dispatch	Automatic Dispatch
Compatible Intel® Compilers	Intel® C++ Compiler only	Intel C++ and 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	<ul style="list-style-type: none">• Single executable file• Developer can write explicit code to take advantage of processor-specific features using vector classes, intrinsic functions, and inline assembly	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• Must validate on all targeted platforms• Larger code size for multiple targeted processors• Possible slightly larger call overhead• Some inlining disabled	

Recommended Optimization Settings for Intel Pentium 4 and Pentium M Processors

For best performance on Pentium 4 processors with SSE3 instruction support, the recommended optimization setting is **/QxP (-xP** on Linux). If your application needs to run on older Pentium 4 processors or non-Intel processors that are Pentium 4 processor-compatible, use **/QaxP**.

For the best performance on any other Pentium 4 or Pentium M processor, use **/QxN (-xN** on Linux). First, verify whether there is a performance difference between using **N** versus **B** on the Pentium M processor, as there are slight differences in the optimizations performed. If the application must also run on other Pentium processors and AMD Athlon* or AMD Opteron* processors, use **/QaxN (-axN** 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 specifically 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.

Table 7. Recommended Intel® Pentium® 4 and Pentium M Processor Optimization Options (not for processors supporting Intel® EM64T)

Need	Recommendation	Caveat
Best performance on Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support	/QxP (-xP on Linux*)	Single code path; will not run on earlier processors
Best performance on any Intel processor, particularly on Pentium 4 processors	/QaxN (-axN on Linux): Optimized for the Pentium 4 and Pentium M processors, and an optimized, generic code-path to run on other processors	Multiple code paths are generated. Use /QxN (-xN for Linux) if you know your application will not be run on processors other than the Pentium 4 or Pentium M processors. Note that there are different optimizations performed with /QxB or /QaxB , specifically for the Pentium M processor that still run well on all Pentium 4 processors. If you expect to deploy your application on Pentium M processors, you should verify whether you get a performance benefit from B versus N targeting.
Best performance on Pentium 4 processors, including Pentium 4 processors with SSE3 instruction support	/QaxNP (-axNP on Linux): Optimized for the Pentium 4 processor and Intel Pentium 4 processor with SSE3 instruction support	This generates three code paths: <ul style="list-style-type: none"> • one generic • one for the Pentium 4 processor • one for the Pentium 4 processor with SSE3 instruction support If you know that your application will never run on processors older than the Pentium 4 processor, specify: /QxN /QaxP (-xN -axP for Linux). This will set the base target to a Pentium 4 processor and generate one additional code path where possible for the Pentium 4 processor with SSE3 instruction support.

Recommended Intel EM64T Optimization Settings

For best performance processors supporting Intel EM64T utilizing SSE3 when possible, the recommended optimization setting is **/QxP (-xP** on Linux). If your application needs to run on AMD Athlon 64 or AMD Opteron processors, use **/QaxP (-axP** 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 EM64T 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.

Table 8 summarizes these recommendations.

Optimizations Specific to Intel® Itanium® Processors

The Intel Compilers automatically take advantage of the advanced features of the Itanium architecture. The following Itanium processor-specific optimizations are enabled by the Intel Compilers:

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

Instruction Scheduling

The **/G2 (-tpp2** on Linux) compiler switch is now on by default. The **/G2** switch enables optimal instruction scheduling and cache management for the Itanium 2 processor without making the generated code incompatible with earlier processors.

Table 8. Recommended Intel® EM64T Optimization Options

Need	Recommendation	Caveats
Best performance on processors supporting Intel® EM64T, utilizing SSE3 where possible	/QxP (-xP on Linux)	Single code path; will not run on processors that do not support Intel EM64T
Best performance on processors supporting Intel EM64T, utilizing SSE3 where possible, and on non-Intel x86-64 compatible processors	/QaxP (-axP on Linux)	Multiple-code paths are used; be sure to validate your application on all systems where it may be deployed.
Good performance on processors supporting Intel EM64T without utilizing SSE3, and still will run on non-Intel x86-64 processor-based systems	/QxW (-xW on Linux)	Does not utilize runtime checks, so you need to validate that the system is x86-64 compatible and has SSE and SSE2 support. The binary will not be as optimal as the “P” switches for processors supporting Intel EM64T.

Predication

Traditional architectures implement conditional execution through branch instructions. The Itanium 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.

Itanium 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 Itanium 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.

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

Figure 2. Control Speculation

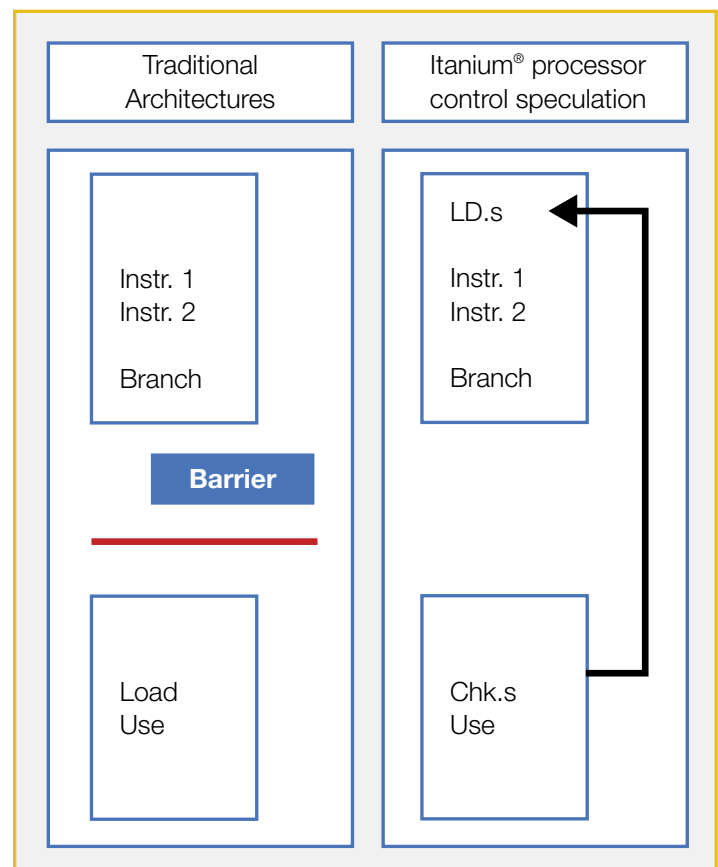
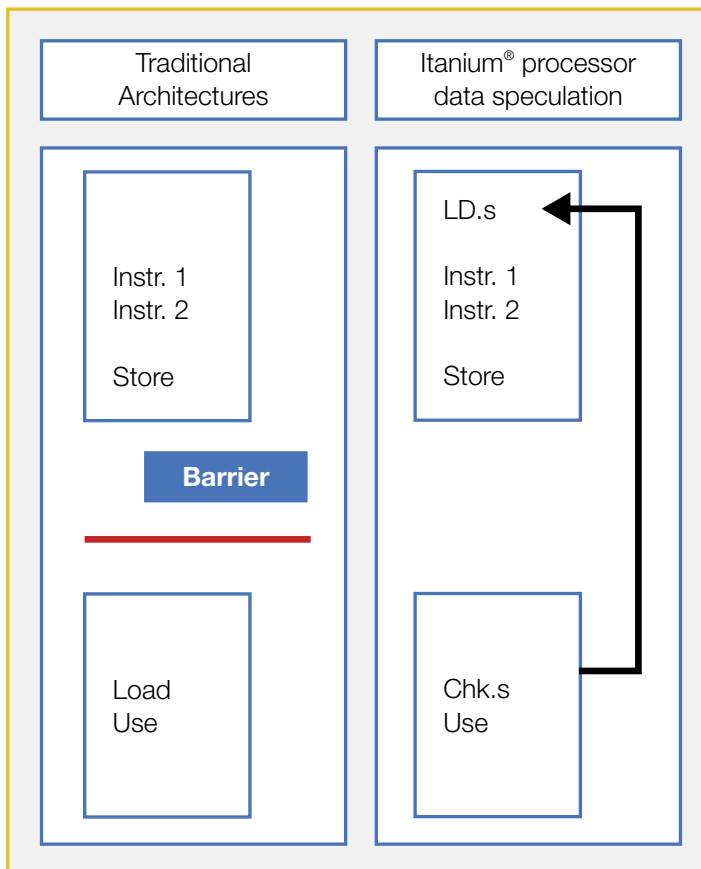


Figure 3. Data Speculation



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.

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). Not all loops may benefit from software pipelining. For loops that can benefit, however, combining software pipelining with predication and speculation helps to boost 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)** 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 include the following:

- 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.

The Itanium 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.

Compiler Directive Support in Intel® Compilers Version 9.0 for Windows* and Linux

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

Memory Alignment Directive

The 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 9 describes directives recently added to the Intel Compilers.

Table 9. Compiler #pragma Directives Available in Intel® Compilers 9.0

#pragma	Architecture	Description
swp noswp	Itanium® architecture	Place before a loop to override the compiler's heuristics for deciding whether to software pipeline the loop.
loop count(n)	IA-32 Itanium architecture	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	Itanium architecture	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	Itanium architecture	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,...	Itanium architecture	Place before a loop to control data prefetching. This is supported when -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	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.

Summary of Intel Compiler Features and Benefits

Intel® Architecture-Specific Optimization

The Intel C++ and Fortran Compilers enable programmers to develop specific IA-32 architecture optimizations for any processor in the Pentium processor family. Intel also offers 64-bit compilers for the Itanium processor and processors with Intel EM6T.

In most cases, hand optimizing source code is unnecessary; the Intel Compilers automatically perform many optimizations to maximize application performance. 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.

Source Code Portability

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

Compatibility with Other Compilers

- **Microsoft Visual C++ Compatibility:** The Intel C++ Compiler is source and object compatible with the Microsoft Visual C++ compiler. On IA-32 processor-based systems, both the Intel Compilers are integrated into the Microsoft Visual Studio .NET Integrated Development Environment (IDE). The Intel C++ Compiler also plugs into the Visual Studio 6.0 IDE.
- **Linux GCC Compatibility:** The Intel C++ Compilers for Linux have substantial compatibility with the GNU C++ Compiler. The Intel C++ Compilers are binary-compatible with GCC for C language object files and use 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.

Intel® Premier Support

The user manuals, 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 references below for how to register).

One year of Intel Premier Support is available with the purchase of the Intel Compilers through a secure Internet site. 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
- Users forums for interactive discussions with fellow developers

References

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

- [General information on the Intel C++ and Fortran Compilers](#)
- [Documentation, application notes, and source code for libraries, tools, and code examples](#)
- [Information on Itanium architecture](#)
- [Information on training available for Intel® Software Development Products](#)
- [Intel Premier Support home page](#)
- [Additional information on OpenMP, including the complete specification and list of directives](#)
- [Quick Reference Guide to optimization with the Intel Compilers](#)

¹ Hyper-Threading Technology requires a computer system with a 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.

² Wireless connectivity requires additional software, services or external hardware that may need to be purchased separately. Availability of public wireless access points is limited. System performance, battery life and functionality will vary depending on your specific hardware and software.



Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052-8119
USA

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

Intel, the Intel logo, Itanium, Pentium, Intel Xeon, Intel Centrino, Intel XScale, MMX, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © 2004-2005, Intel Corporation. All Rights Reserved. 0505/DMX/ITF/PDF
