Optimizing Applications with the Intel® C++ and Fortran Compilers for Windows* and Linux*

Updated for Version 7.1 Compilers

Copyright © 2003, Intel Corporation. All rights reserved.

Intel Corporation
2111 NE 25th
Hillsboro, Oregon 97124-6497

Intel Corporation assumes no responsibility for errors or omissions in this document. Nor does Intel make any commitment to update the information contained herein.

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.

Intel, Pentium, Itanium, 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.

Table of Contents

Introduction ... 5

Intel® Compiler Features and Benefits Common to IA-32 and Intel® Itanium® Processor-based Systems ... 5

- Automatic Optimization Settings ... 6
- Cache Management Features ... 6
  - Data Prefetching ... 6
    - Data Prefetching Benefits ... 6
  - Streaming Stores (IA-32 only) ... 6
- Inter-Procedural Optimization (IPO) ... 6
  - IPO Benefits ... 6
  - How IPO Works ... 6
    - Step One: Compilation ... 6
    - Step Two: Linking ... 6
    - Function Inlining ... 6
- Profile-Guided Optimization (PGO) ... 6
  - PGO Benefits ... 6
  - How PGO Works ... 6
    - Step One: Instrumented Compilation ... 6
    - Step Two: Instrumented Execution ... 6
    - Step Three: Feedback Compilation ... 6
- Multi-Threading Support ... 6
- Compiler optimization reports ... 6

Intel® Pentium® 4 Processor-Specific Optimizations ... 6

- Pentium-4 Specific Tuning ... 6
- Vectorization ... 6
- Processor Dispatch ... 6
  - Automatic Processor Dispatch ... 6
  - Manual Processor Dispatch ... 6

Intel Itanium Processor-Specific Optimizations ... 6

- Instruction Scheduling ... 6
- Predication ... 6
- Improved Branch Prediction ... 6
- Speculation ... 6
  - Control Speculation ... 6
    - Control Speculation Benefits ... 6
  - Data Speculation ... 6
    - Data Speculation Benefits ... 6
- Software Pipelining ... 6
- High-Performance Floating-Point Optimizations ... 6

Compiler Directive Support in 7.1 Compilers ... 6

- Memory Alignment Directive ... 6
- #pragma (CDIR$) Directives ... 6

Creating Linux Shared Object Files 6
Step 1a: Compile the objects to create .o files 6
Step 1b: Link the objects to create a shared object 6
Step 1c: (Optional) Install the shared library in your system standard location........... 6
Step 1d: Build example application(s) to use the shared library 6
Step 2: Verify 'main' dependency and run the application.. 6
Step 3: Rebuild shared library with profile information 6

Summary of Intel Compiler Features and Benefits 6
Intel® Architecture-Specific Optimization 6
Source Code Portability 6
Compatibility with Other Compilers 6
Intel Premier Support 6

References 6

Figures

Figure 1: Profile-Guided Optimization Steps 6
Figure 2: Additional vectorization capabilities of the Intel Pentium 4 processor.......... 6
Figure 3: Control Speculation.. 6
Figure 4: Data Speculation.. 6

Tables

Table 1: Automatic optimization switch settings and their uses 6
Table 2: Interprocedural optimization compiler switch settings.................................. 6
Table 3: Optimization reports available in the Intel 7.1 compilers 6
Table 4: Comparison of manual and automatic processor dispatch methods............. 6
Table 5: Compiler #pragma directives available in the Intel 7.1 compilers 6

Introduction

Intel® C++ and Fortran compilers for Windows* and Linux* give application developers access to the advanced architecture of the Intel® Pentium® 4 processor, the new Intel® Pentium® M processor based on Intel® Centrino™ mobile technology, and include compilers that support the Intel® Itanium® processor family (hereafter referred to as simply Itanium). The compilers feature several improvements to maximize application performance:

- *Flexibility.* Developers can target specific 32-bit or 64-bit Intel® processors for optimization, including the Pentium 4 processors, Pentium M processors based on Centrino mobil technology, and Itanium processors.
- *Microsoft Visual C++* Compatibility.* The Intel C++ Compiler for Windows is source and object compatible with the Microsoft Visual C++ compiler. It also integrates into the Microsoft Visual Studio .Net Integrated Development Environment (IDE). The Intel Fortran Compiler for Windows provides integration into Microsoft's Visual C++ .NET environment on IA-32 systems only. For more information on specific compatibility and usage of the compilers within the Microsoft Visual C++ and Visual Studio .Net environments, please refer to the *Intel C++ Compiler 7.1 for Windows Compatibility with Microsoft* Visual C++* 6.0 and .NET** white paper at http://www.intel.com/software/products/compilers/cwin.
- *Linux gcc* Compatibility.* The Intel C++ Compilers for Linux have substantial compatibility with the GNU gcc* compiler. New features for the 7.1 release include additional support for GNU C and C++ language extensions. For details on compatibility, please refer to the *Intel® Compilers for Linux* - Compatibility with the GNU Compilers* white paper available at http://www.intel.com/software/products/compilers/clin.
- *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 remains much more portable and easy to maintain.
- *Intel® Premier Support.* Intel provides training, answers to specific questions, software patches and more through its secure Premier Support Internet site.

This paper focuses on how developers can use the Intel compilers to optimize for the IA-32 architecture, including the Pentium 4 processor and the Pentium M processor based on Centrino mobile technology, and optimize for the Itanium architecture.

Intel® Compiler Features and Benefits Common to IA-32 and Intel® Itanium® Processor-based Systems

Intel C++ and Fortran compilers have the ability to compile applications for Pentium processors (32-bit applications) or for Itanium processors (64-bit applications), depending on which is the host processor.

On 32-bit Windows-based systems, the developer can also install compiler components to enable development of 64-bit applications (cross-compilation).

Intel compilers present a set of features and benefits that are common to systems based on Pentium and Itanium processors, as well as features that are unique to each. The common features include:

- Automatic optimization settings;
- Cache management features;
- Interprocedural optimization methods;
- Profile-guided optimization methods;
- Multi-threading support; and
- Compiler optimization reports.

Automatic Optimization Settings

Both the 32-bit and 64-bit Intel compilers automatically optimize applications for the target processor when the developer selects a switch setting. Table 1 below 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
/Od (No Optimizations)	**-O0**	Used during the early stages of application development, until the developer knows the application is working correctly; then switch to a higher setting (usually **/O2**).
/O1	**n/a**	Omits optimizations that tend to increase object size. Creates the smallest code in the majority of cases.
/O2 (Maximize Speed)	**-O1** or **-O2**	Default setting. Creates the fastest code in most cases, but may increase code size significantly over **/O1**. On Linux, **-O1** and **-O2** are equivalent.
/Ox (Maximum Optimizations)	**n/a**	Equivalent to **/O2** except that **/Ox** does not imply **/Gy** (function packaging) or **/Gf** (string pooling).
/O3 (High Level Optimizations)	**-O3**	Same as **/O2**, plus loop transformations and data prefetching. For 32-bit Intel architecture, the Pentium III processor and subsequent processors support data prefetching. To get the full benefit of the **/O3** switch with these processors, the developer must also use the **/QxK** or **/QaxK** switches for Pentium III processors, or the **/QxW** or **/QaxW** switches for the Pentium 4 processors and subsequent 32-bit processors.

Cache Management Features

The Intel C++ and Fortran compilers optimize applications by reducing memory latency and cache pollution on both 32-bit and 64-bit applications. The Intel compilers accomplish this through two mechanisms: *data prefetching* and *streaming stores*.

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, the prefetch instructions are able to overlap memory accesses with other computations. This can significantly improve performance in applications that have a regular pattern of memory accesses.

Data Prefetching Benefits

- Data prefetching is automatic.
- Data prefetching coordinates with other optimizations (e.g., software pipelining).
- By using compiler-generated prefetching, code remains 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).

Streaming Stores (IA-32 only)

By automatically generating streaming stores to bypass the cache and store data direct to memory, 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.

Inter-Procedural Optimization (IPO)

Inter-Procedural Optimization is another optimization that works with 32-bit and 64-bit Intel compilers. Developers activate IPO through compiler settings; see Table 2 below. IPO can significantly improve application performance in programs that contain many frequently used small or medium-sized functions. It is especially beneficial for applications containing calls to these functions within loops.

IPO produces best results when the developer uses it in conjunction with Profile-Guided Optimization (see below). To use IPO with PGO together, apply the IPO switch(es) during PGO Step Three: Feedback Compilation.

IPO Benefits

IPO enhances application performance through the following optimizations:

- Decreasing the number of branches, jumps and calls within code, which reduces overhead when executing conditional code;
- Reducing call overhead still 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; and
- 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

During the first step, IPO creates an information file. This file contains an intermediate representation of the source code, and summary information used for optimization.

Step Two: Linking

During the second step, for all modules having 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:

- Removing the need to set up parameters for a function call, and
- Eliminating the function call branch.

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

Table 2: Interprocedural optimization compiler switch settings

Windows Switch Setting	Linux Equivalent	Comment
/Qip	-ip	Single file optimization. Allows selective inlining optimization within a single source file.
/Qipo	-ipo	Multi-file optimization. Permits inlining and other optimizations among multiple source files.

Profile-Guided Optimization (PGO)

Profile-Guided Optimization is a three-step compilation process that improves performance when applied to typical application runtime loads. Where IPO looks for performance gains by reviewing application logic, PGO looks for performance gains in the way the application logic is applied in typical uses.

Although PGO is an independent optimization method, it is most effective when the developer uses it in conjunction with other optimizations, especially IPO.

PGO Benefits

- PGO allows improved 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 improves application performance by enabling better branch prediction. PGO also generates branch hints for the Pentium 4 and Itanium processors during the optimization process.

While PGO benefits vary, applications with these characteristics are well suited for it:

- Applications containing several possible execution paths, a few of which the application executes much more frequently than others; and
- 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 representative of typical application use.

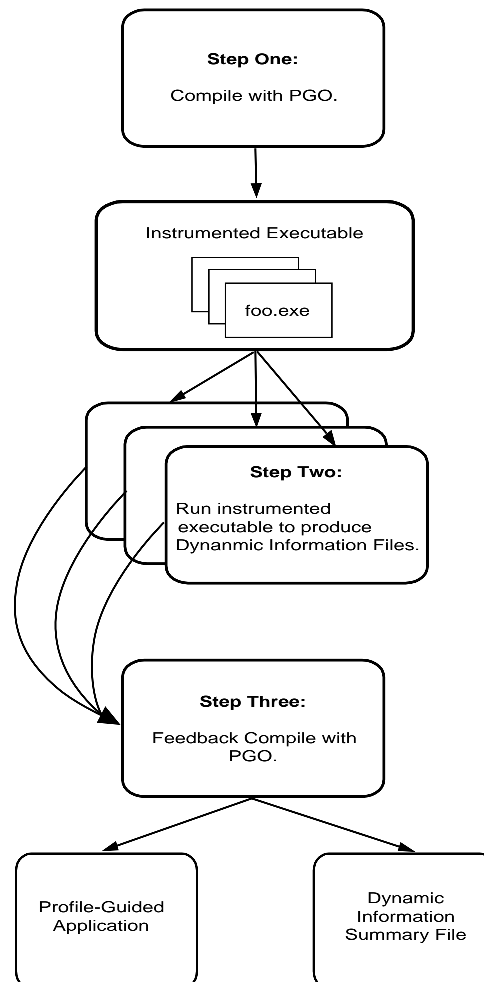
The compiler generates dynamic information files for each run, recording the frequency with which each code section is executed.

Step Three: Feedback Compilation

When the developer re-compiles the application, PGO feeds the execution data back to the compiler with the switch **/Qprof_use** (**-prof_use** on Linux[1]). 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.

Figure 1 below illustrates the steps involved in compiling with PGO:

Figure 1: Profile-Guided Optimization Steps



[1] Use **-prof_use** at compile and link time on IA-64/Linux

Multi-Threading Support

The Intel compilers support development of multi-threaded applications, for systems with Intel® Hyper-Threading Technology and/or multiple processors, through two different mechanisms.

- *Auto-parallelization*. When the compiler switch **/Qparallel** (**-parallel** on Linux) is specified, the compiler can detect loops which may benefit from multi-threaded execution, and generate the appropriate threading calls automatically.
- *OpenMP* directives*. The compilers recognize industry-standard OpenMP directives (version 2.0[2]). These directives give the user complete control of how their application is multi-threaded.

Compiler optimization reports

The 7.1 compiler includes several optimization reports which give information on different aspects of the compilation process. The programmer can use this information to guide adjustments to the program so that the compiler can generate more highly optimized code.

To generate any of the optimization reports, the switch **/Qopt_report** (**-opt_report** on Linux) must be used. Then, in addition, the specific optimization report may be selected by using **/Qopt_report_phase** *<phase>* (**-opt_report_phase** *<phase>* on Linux). For example:

```
$ ecc -opt_report
-opt_report_phase ecg_swp main.c
```

Specifying **/Qopt_report_help** (**-opt_report_help** on Linux) will give a list of all the possible values for *<phase>*. Table 3 below lists main *<phase>* values, with examples of more fine-grain selections.

Table 3: Optimization reports available in the Intel 7.1 compilers

Phase	Arch.	Description
all	IA-32 IA-64	Enable all possible optimization reports.
ipo **ipo_inl**	IA-32 IA-64	Optimizations performed as part of the InterProcedural Optimization phase. For example, **ipo_inl** reports on function inlining.
hlo **hlo_prefetch**	IA-64	Optimizations performed as part of the High-Level Optimization phase. For example, **hlo_prefetch** reports on compiler-generated prefetching.
ilo	IA-64	Optimizations performed as part of the Intermediate-Language phase.
ecg **ecg_swp**	IA-64	Optimizations performed as part of the Code Generator phase. For example, **ecg_swp** reports on software pipelining.
pgo	IA-32 IA-64	Optimizations performed as part of the Profile-Guided Optimization phase.

[2] There are two exceptions to the OpenMP 2.0 support: The WORKSHARE directive is not supported, and the support for array reductions is incomplete

Intel® Pentium® 4 Processor-Specific Optimizations

Pentium 4 Processor Specific Tuning

The **/G7** (**-tpp7** on Linux) compiler switch is now on by default. The **/G7** switch enables optimal instruction selection, scheduling and cache management for the Pentium 4 processor without making the generated code incompatible with earlier processors.

Vectorization

Vectorization detects patterns of sequential data accesses by the same instruction, and transforms the code for Single Instruction Multiple Data (SIMD) execution.

The Intel C++ and Fortran compilers automatically vectorize code. For the Pentium 4 processor, the vectorizer that comes with the Intel C++ and Fortran compilers provides these features:

- *Supported data types.* The vectorizer supports the **float/double** and **char/short/int/long long** data types (both signed and unsigned).
- *Diagnostics*. Through the **/Qvec_report*N*** (**-vec_report*N*** 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 necessary information to slightly adjust or restructure code with dependency and restrict directives, to allow vectorization to occur.
- *Support for 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.
- *Code portability.* As Intel develops new processor technology, developers can use appropriate Intel compiler switches to take advantage of new processor features. This can avoid the need to extensively rewrite the source code.

The following optimization switches will enable the compiler to generate code specialized for a specific Pentium processor and allows the compiler's vectorizer to generate SIMD instructions.

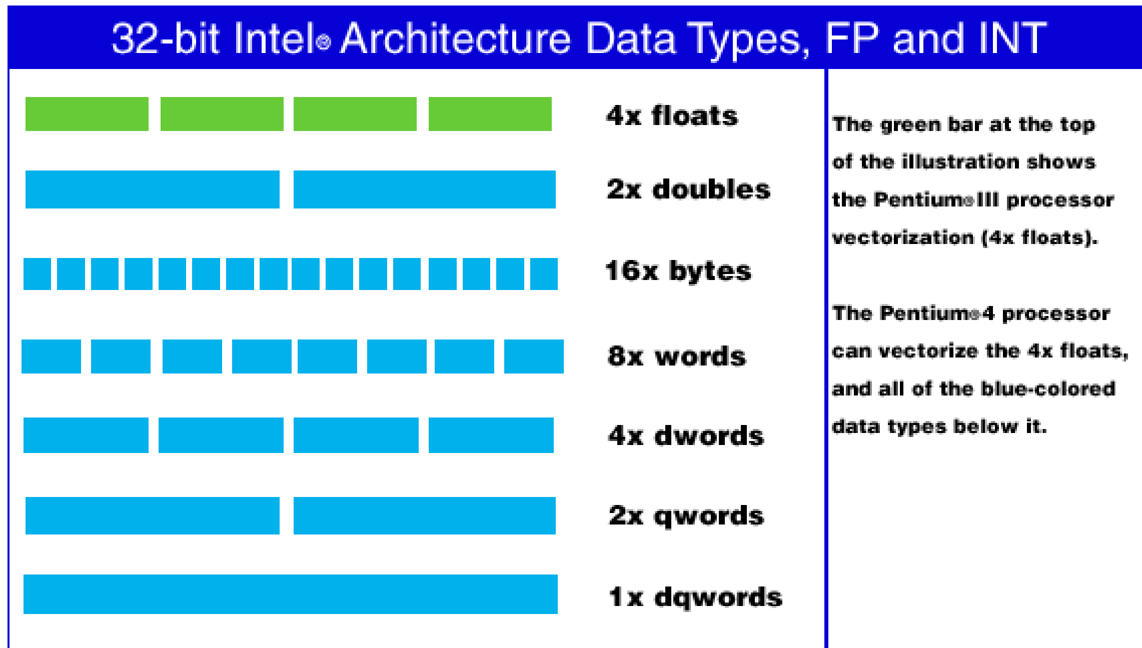
Options of the form **/Qx**<*code*> (**-x**<*code*> on Linux) generate specialized code for processor extensions specified by *code*; options of the form **/Qax**<*code*> (**-ax**<*code*> on Linux) generate both specialized code and also generic IA-32 code through the Processor Dispatch technology, described below. The value for <*code*> can be:

M	Generate instructions for Pentium processors with Intel® MMX™ technology.
K	Generate instructions for Pentium III processors (including SSE instructions).
W	Generate instructions for Pentium 4 processors and Pentium M processors based on Centrino mobil technology (including SSE2 instructions).

Multiple <*code*> values can be combined to generate an executable which is optimized for multiple target processors. For example, compiling with **/QaxKW** (**-axKW** on Linux) will generate an executable which is compatible with all Intel 32-bit processors, but also has optimal code paths for Pentium III and Pentium 4 processors. The correct path is selected at execution time. See Processor Dispatch, below, for details.

Figure 2 below demonstrates the added vectorization capabilities of the Pentium 4 processor compared to the Pentium III processor.

Figure 2: Additional vectorization capabilities of the Intel Pentium 4 processor



Processor Dispatch

Processor dispatch allows developers to optimize applications targeting one or more specific Intel 32-bit processors. For example, developers may want to take advantage of the performance features in the Pentium 4 processor, while at the same time remain compatible with earlier Pentium processors.

Developers may choose automatic or manual modes of processor dispatch. Usually, a developer can select automatic processor dispatch and let the compiler do the work to tune the application for one target processor.

Both modes produce a generic code version that allows the application to run on systems using an Intel processor other than one for which the application is optimized.

At runtime, the application automatically identifies the Intel processor it is running on and selects the appropriate optimized or generic implementation.

Automatic Processor Dispatch

Automatic processor dispatch allows developers to let the Intel compiler choose the most efficient optimizations for the Pentium 4 processor or any other Intel 32-bit processor (for the Pentium 4 processor, vectorization must be turned on to take advantage of automatic processor dispatch).

For example, the **/QaxW** (**-axW** on Linux) compiler switch generates specialized code for the Pentium 4 processor while also generating generic IA-32 code. The **/QxW** (**-xW** on Linux) compiler switch generates code to run exclusively on the Pentium 4 processor.

Manual Processor Dispatch

Manual processor dispatch is useful in the following cases:

- The developer needs to explicitly optimize one application to target two or more Intel 32-bit processors; or
- The developer wishes to write explicit, hand optimized code for the target processor.

Table 4 below describes some of the key differences between the manual and automatic dispatch methods.

Table 4: Comparison of manual and automatic processor dispatch methods

Feature	Manual Dispatch	Automatic Dispatch
Compatible Intel Compilers	C++ compiler only.	C++ and Fortran compilers.
Number of processor-specific functions	Developer may manually optimize for multiple processor-specific implementations.	One automatically generated processor-specific implementation.
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. • The application can support as many different target processors as the developer chooses. • 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; compiler does this automatically. • Automatic Vectorization and prefetch features 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.	

Intel Itanium Processor-Specific Optimizations

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

- Instruction Scheduling;
- Predication;
- Improved branch prediction;
- Speculation;
- Software pipelining; and
- 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.

Predication

Traditional architectures implement conditional execution through branch instructions. The Itanium processor implements conditional execution with *predicated instruction*s.

One of the most important optimizations that predication enables, is the removal of some branches from program sequences. This results in larger basic blocks and the elimination of associated branch misprediction penalties, both of which contribute to improved application performance.

Furthermore, as fewer branches exist after predication, it makes dynamic instruction fetching more efficient because there are fewer possibilities for control flow changes.

Improved Branch Prediction

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

The 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; and
- 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 doing some operations (e.g., costly load instructions) out of sequence before they are needed.

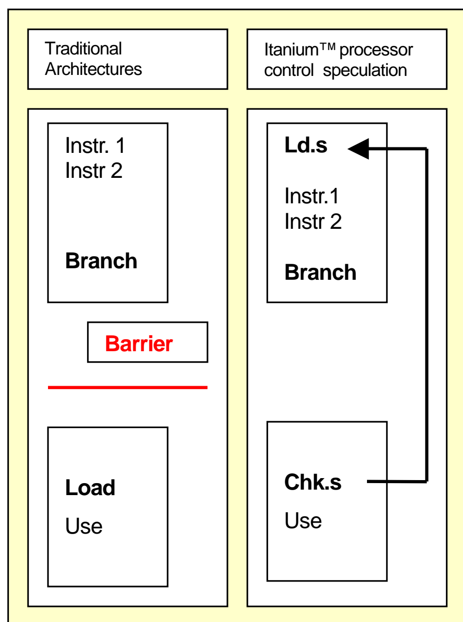
To ensure that the code is correct in all cases, and not just when the conjecture is correct, the compiler executes *recovery code* as needed. Recovery code corrects all affected operations if the original conjecture or speculation was false.

There are two kinds of speculation: *control speculation* and *data speculatio*n.

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 3: Control Speculation



Control Speculation Benefits

- Control speculation gives the developer more control over when and where to use instructions in an application;
- Control speculation helps to hide memory latencies by moving loads earlier in the code; and
- Because control speculation executes a load operation before the evaluation of the conditional branch, it is very effective at working around branch barriers in code, leading to improved application performance.

Data Speculation

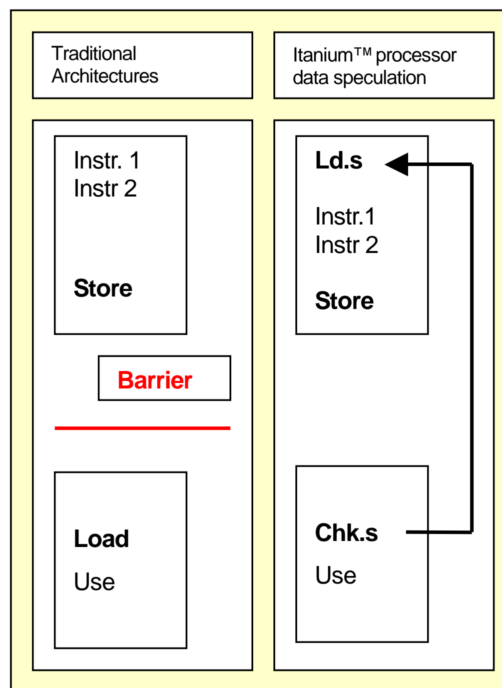
Like control speculation, data speculation is a mechanism to hide memory latencies.

In traditional architectures, if the load operation actually or potentially 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 is the answer to this limitation. By creating dependency checks in the code and providing a mechanism to recover if data dependencies exist, it 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, 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 4 below, 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) did exist, then recovery code executes to validate the code.

Figure 4: Data Speculation



Data Speculation Benefits

- Data speculation can resolve ambiguous memory dependencies, making it highly beneficial for working around data dependency barriers in code; and
- Because it enables load operations to execute ahead of the stores preceding them, data speculation can significantly reduce memory latencies and improve application performance.

Software Pipelining

Software pipelining works to reduce the number of clock cycles necessary to process a loop. 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; but for those that can, when software pipelining combines with predication and speculation it helps to boost application performance by significantly reducing code expansion, path length and branch mispredictions.

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 when compared to traditional architecture processors.

Compiler Directive Support in 7.1 Compilers

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- and 64-bit architectures, and others are specific to one or the other.

Memory Alignment Directive

The Intel compilers for 32-bit processors allow the programmer 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 will allocate A with a base address which has alignment of 32x+8.

#pragma (CDIR$) Directives

Table 5 below describes directives recently added to the Intel compilers.

Table 5: Compiler #pragma directives available in the Intel 7.1 compilers

#pragma	Architecture	Description
swp **noswp**	IA-64	Place before a loop to override the compiler's heuristics for deciding to software pipeline the loop or not.
loop count(n)	IA-32, IA-64	Place before a loop to communicate the approximate number of iterations the loop will execute. Affects software pipelining, vectorization and other loop transformations.
distribute point	IA-64	Placed before a loop, the compiler will attempt to distribute the loop based on its internal heuristic. Placed within a loop, the compiler will 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)** tell 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. Supported when -O3 is on. **#pragma prefetch a** will cause the compiler to prefetch for future accesses to array a. The compiler determines the prefetch distance. **#pragma noprefetch x** will cause 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.

Creating Linux Shared Object Files

Creating Linux shared object files (.so files) when using IPO and PGO requires special techniques. This section describes a method for using IPO and PGO with shared objects. The 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 :

```
$ icc -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 the IPO, use the command 'xild -qnoipo' command or the 'ld' command directly.

Step 1c: (Optional) Install the shared library in your system standard 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
$ icc main.c -o main -lpi
```

Note that the optimization option for main.c need not correspond to the optimization options of the shared library. It is however recommended that you invoke the advanced optimizations on the main program as well, for maximum performance. (Not shown in 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 that the standard 'ldd' tool can be used to determine dependency information.

In addition to running the application, profile information (.dyn file) will now have been created in the /tmp directory (as specified by the **-prof_dir** switch).

Step 3: Rebuild shared library with profile information

Example:

```
$ icc -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 will now point to the location of the intermediate files.

Repeat *Step 1b* through *Step 1d* to get the optimized application.

Note that *Step 3* is required only in the presence of **-prof_gen** and **-prof_use**. If PGO is not used, *Steps 1a-1d* will result in the optimized application.

Summary of Intel Compiler Features and Benefits

Intel® Architecture-Specific Optimization

The Intel C++ and Fortran compilers enable programmers to develop specific 32-bit architecture optimizations for any processor in the Pentium processor series, and include the 64-bit compilers, for the Itanium processor. In most cases, hand optimizing the source code is unnecessary; the Intel compilers will automatically perform several optimizations to maximize application performance. By changing the compiler option settings, developers can still choose the right set of performance optimization characteristics for their applications: from none at all, to logic optimization, to optimization based on how the application is used.

Source Code Portability

Developers who use the automatic optimizations of Intel C++ and Fortran compilers can easily tune applications to make best use of the Pentium 4 and Itanium processors, without spending long hours of 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. The Intel Fortran Integrated Development Environment (IDE) provides integration into Microsoft's Visual C++ .NET environment on IA-32 systems. Both Intel compilers plug into the Microsoft Visual Studio* development environment.

Linux gcc Compatibility.* The Intel C++ compilers for Linux have substantial compatibility with the GNU gcc* 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++ ABI, which when fully implemented by all compiler vendors, will allow C++ objects files and libraries to be compatible with different compilers.

Intel® Premier Support

The user manuals, tutorials, documented examples and context-sensitive help files that come with Intel C++ and Fortran compilers will answer most developer questions. Developers should also register for Intel Premier Support; see the product documentation for instructions on how to register.

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

- FAQs and other proactive notices;
- Issue tracking and updates;
- Software beta and patch downloads; and
- Users Forums for interactive discussions with fellow developers.

References

Additional information on Intel C++ and Fortran compilers is available at:

- http://www.intel.com/software/products (general information on Intel C++ and Fortran compilers);
- http://developer.intel.com/software/products/opensource/ (documentation, application notes and source code for libraries, tools, and code examples);
- http://developer.intel.com/vtune/cbts/appnotes.htm (Application notes and code examples);
- http://developer.intel.com/design/itanium/index.htm (information on Intel Itanium architecture).
- http://www.intel.com/software/college (information on training available for Intel software products).
- https://premier.intel.com (Intel Premier Support home page).

Additional information on OpenMP, including the complete specification and list of directives, is available at:

- http://www.openmp.org