

# **Intel® C++ Compiler Optimizing Applications**

---

Document Number: 307776-003US

# Disclaimer and Legal Information

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, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

This document as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Chips, Core Inside, Dialogic, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, MMX logo, Optimizer logo, OverDrive, Paragon, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, Pentium Inside, skool, Sound Mark, The Computer Inside., The Journey Inside, VTune, Xeon, Xeon Inside and Xircom 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.

Copyright (C) 1996-2006, Intel Corporation.

Portions Copyright (C) 2001, Hewlett-Packard Development Company, L.P.

# Table Of Contents

Disclaimer and Legal Information.....	ii
Table Of Contents .....	iii
Introduction to Optimizing Applications .....	1
How to Use This Document .....	2
Programming for High Performance Overview .....	2
Optimizing Performance Overview.....	2
Using Intel® Performance Analysis Tools.....	3
Using Tuning Tools and Strategies .....	4
Identifying and Analyzing Hotspots .....	4
Using the Intel® Compilers for Tuning.....	5
Using a Performance Enhancement Methodology.....	6
Gather performance data .....	7
Analyze the data.....	7
Generate alternatives .....	7
Implement enhancements .....	8
Test the results .....	8
Applying Performance Enhancement Strategies .....	9
Understanding Run-time Performance.....	15
Helping the Compiler .....	15
Memory Aliasing on Itanium®-based Systems.....	15
Applying Optimization Strategies .....	18
Loop Interchange.....	18
Unrolling .....	19

Cache Blocking.....	20
Loop Distribution.....	21
Loop Fusion.....	23
Load Pair (Itanium® Compiler).....	23
Manual Loop Transformations.....	23
Understanding Data Alignment .....	24
Timing Your Application .....	25
Considerations on Timing Your Application.....	25
Compiler Optimizations Overview .....	27
Optimization Options Summary .....	27
Setting Optimization Levels .....	28
Restricting Optimization.....	31
Diagnostic Options .....	32
Optimizing for Specific Processors Overview .....	33
Targeting a Processor.....	33
Options for IA-32 and Intel® EM64T Processors .....	34
Options for Itanium® Processors .....	35
Processor-specific Optimization (IA-32 only) .....	36
Automatic Processor-specific Optimization (IA-32 Only) .....	37
Processor-specific Run-time Checks for IA-32 Systems.....	38
Check for Supported Processor .....	38
Setting FTZ and DAZ Flags.....	39
Manual CPU Dispatch (IA-32 only) .....	40
Symbol Visibility Attribute Options (Linux* and Mac OS*).....	42

Global Symbols and Visibility Attributes .....	42
Symbol Preemption and Optimization .....	43
Specifying Symbol Visibility Explicitly .....	43
Interprocedural Optimizations Overview .....	45
IA-32 and Itanium®-based applications.....	45
IA-32 applications only .....	45
-auto-ilp32 (Linux*) or /Qauto-ilp32 (Windows*) for Intel® EM64T and Itanium®-based Systems.....	46
IPO Compilation Model .....	46
Understanding IPO-Related Performance Issues .....	47
Improving IPO Performance .....	48
Command Line for Creating an IPO Executable .....	48
Generating Multiple IPO Object Files.....	49
Capturing Intermediate Outputs of IPO.....	50
Creating a Multifile IPO Executable .....	51
Usage Rules .....	52
Understanding Code Layout and Multi-Object IPO .....	54
Implementing IL Files with Version Numbers.....	55
IL in Objects and Libraries: More Optimizations.....	55
Criteria for Inline Function Expansion .....	55
Selecting Routines for Inlining .....	56
Using Qoption Specifiers.....	57
Compiler Directed Inline Expansion of User Functions.....	58
Developer Directed Inline Expansion of User Functions.....	59
Inline Expansion of Library Functions .....	61

Profile-Guided Optimizations Overview .....	62
Instrumented Program.....	62
Added Performance with PGO .....	62
Understanding Profile-Guided Optimization.....	63
PGO Phases.....	63
PGO Usage Model .....	65
Example of Profile-Guided Optimization .....	65
Profile-guided Optimization (PGO) Phases .....	66
Basic PGO Options .....	68
Advanced PGO Options.....	70
Generating Function Order Lists .....	71
Function Order List Usage Guidelines (Windows*) .....	71
Comparison of Function Order Lists and IPO Code Layout .....	71
Example of Generating a Function Order List .....	72
PGO API Support Overview.....	73
The Profile IGS Environment Variable.....	74
PGO Environment Variables .....	74
Dumping Profile Information.....	75
Recommended usage .....	75
Resetting the Dynamic Profile Counters .....	76
Recommended usage .....	76
Dumping and Resetting Profile Information .....	76
Recommended usage .....	76
Interval Profile Dumping.....	76

Recommended usage .....	78
PGO Tools Overview .....	78
Code-Coverage Tool.....	78
Code-coverage tool Requirements.....	79
Visually Presenting Code Coverage for an Application.....	82
Excluding Code from Coverage Analysis .....	88
Exporting Coverage Data .....	91
Test-Prioritization Tool .....	94
Features and Benefits .....	94
Test-prioritization tool Requirements.....	94
Tool Usage Examples .....	99
Using Other Options.....	100
Profmerge and Proforder Utilities.....	101
profmerge Utility .....	101
proforder Utility .....	103
Profrun Utility.....	103
Profrun Utility Requirements and Behavior .....	103
Using the profrun Utility .....	104
Profrun Utility Options.....	105
Software-based Speculative Precomputation (IA-32) .....	108
SSP Behavior .....	108
Using SSP Optimization .....	109
HLO Overview.....	112
IA-32 and Itanium®-based Applications .....	112

IA-32 Applications.....	112
Tuning Itanium-based Applications .....	112
Loop Transformations .....	113
Scalar Replacement.....	114
Absence of Loop-carried Memory Dependency with IVDEP Directive.....	114
prefetch Directive.....	115
Loop Unrolling .....	115
Loop Independence .....	116
Flow Dependency - Read After Write .....	117
Anti Dependency - Write After Read .....	118
Output Dependency - Write After Write .....	118
Reductions.....	119
Prefetching with Options .....	119
Floating-point Arithmetic Optimizations Overview.....	120
Floating-point Options for Multiple Architectures .....	120
Floating-point Options for IA-32 and Intel® EM64T .....	121
Floating-point Options for Itanium®-based Systems.....	124
Improving or Restricting FP Arithmetic Precision.....	126
Understanding Floating-point Performance .....	126
Denormal Computations.....	126
Inexact Floating Point Comparisons.....	128
Compiler Reports Overview .....	129
Optimizer Report Generation .....	129
Specifying Optimizations to Generate Reports.....	131



High-Level Optimization (HLO) Report .....	133
Interprocedural Optimizations (IPO) Report.....	134
Software Pipelining (SWP) Report (Linux* and Windows*).....	135
Reading the Reports.....	136
Vectorization Report.....	138
Usage with Other Options .....	140
Changing Code Based on Report Results.....	140
Parallelism Overview.....	143
Parallel Program Development.....	144
Parallelization with OpenMP* Overview.....	148
Parallel Processing with OpenMP .....	149
Performance Analysis.....	149
Targeting a Processor Run-time Check .....	149
Parallel Processing Thread Model .....	150
The Execution Flow .....	150
Using Orphaned Directives.....	151
OpenMP* and Hyper-Threading Technology .....	153
Compiling with OpenMP, Directive Format, and Diagnostics.....	156
OpenMP Option.....	156
OpenMP Pragma Format and Syntax .....	156
OpenMP Diagnostic Reports .....	157
OpenMP* Directives and Clauses Summary .....	158
OpenMP Directives.....	158
OpenMP Clauses .....	159

OpenMP* Support Libraries .....	159
Execution modes .....	160
OpenMP* Environment Variables .....	161
Standard Environment Variables.....	161
Intel Extension Environment Variables.....	161
OpenMP* Run-time Library Routines.....	162
Execution Environment Routines .....	162
Lock Routines.....	163
Timing Routines.....	164
Intel Extension Routines/Functions.....	164
Stack Size.....	164
Memory Allocation .....	165
Examples of OpenMP* Usage .....	166
A Simple Difference Operator.....	166
Two Difference Operators .....	166
OpenMP* Advanced Issues .....	167
Performance .....	169
THREADPRIVATE Directive .....	170
Auto-parallelization Overview.....	171
Programming with Auto-parallelization.....	172
Guidelines for Effective Auto-parallelization Usage.....	172
Auto-parallelization Data Flow.....	173
Programming for Multithread Platform Consistency.....	173
Auto-parallelization: Enabling, Options, Directives, and Environment Variables .....	178

Auto-parallelization Options.....	178
Auto-parallelization Environment Variables.....	178
Auto-parallelization: Threshold Control and Diagnostics .....	179
Threshold Control .....	179
Diagnostics .....	179
Vectorization Overview (IA-32 and Intel® EM64T) .....	181
Vectorizer Options.....	181
Key Programming Guidelines for Vectorization .....	182
Guidelines.....	182
Restrictions.....	182
Loop Parallelization and Vectorization .....	183
Types of Vectorized Loops.....	183
Statements in the Loop Body .....	184
Floating-point Array Operations.....	184
Integer Array Operations .....	184
Other Operations .....	184
Data Dependency .....	185
Data dependency Analysis.....	185
Loop Constructs .....	186
Loop Exit Conditions .....	187
Strip-mining and Cleanup.....	188
Loop Blocking .....	189
Vectorization Examples.....	191
Argument Aliasing: A Vector Copy .....	191

Data Alignment .....	191
Loop Interchange and Subscripts: Matrix Multiply .....	193
Language Support and Directives .....	193
Language Support .....	194
Multi-version Code .....	194
Pragma Scope .....	194
Optimization Support Features Overview .....	197
Compiler Directives Overview .....	198
Pipelining for Itanium®-based Applications .....	198
Loop Count and Loop Distribution .....	199
Loop Count .....	199
Loop Distribution .....	199
Loop Unrolling Support .....	200
Vectorization Support .....	201
vector always Directive .....	201
ivdep Directive .....	201
vector aligned Directive .....	201
novector Directive .....	203
vector nontemporal Directive (Windows*) .....	203
Prefetching Support .....	204
Pragmas .....	204
Intrinsics .....	208
Optimization Restriction Support .....	209
optimize pragma .....	209

## Table Of Contents

optimization_level pragma.....	210
Optimizing Applications Glossary.....	211
Index .....	215



# Introduction to Optimizing Applications

This document explains how to use the Intel® C++ Compiler to enhance application performance. How you use the information presented in this document depends on what you are trying to accomplish.

Where applicable, this document explains how compiler options and optimization methods apply on IA-32, Itanium®, and Intel® Extended Memory 64 Technology (Intel® EM64T) architectures on Linux\*, Windows\*, and Mac OS\* systems. In general, the compiler features and options supported for IA-32 Linux are supported on Intel®-based systems running Mac OS. For more detailed information about compiler support on a specific operating system see the *Intel® Compiler Release Notes*.

In addition to compiler options that can affect application performance, the compiler includes features that enhance your application performance such as directives, intrinsics, run-time library routines, and various utilities.

The following table lists some possible starting points for your optimization efforts.

If you are trying to...	Then start with...
Improve performance for a specific type of application	Applying Performance Enhancement Strategies Using a Performance Enhancement Methodology Using Intel® Performance Analysis Tools
Optimize an application for speed or a specific architecture	Compiler Optimization Overview Optimization Options Summary Optimizing for Specific Processors Overview
Create application profiles to help optimization	Profile-guided Optimizations (PGO) PGO Tools Overview
Enable optimization for calls and jumps	Interprocedural Optimizations (IPO) IPO Compilation Mode
Optimize loops, arrays, and data layout	High-level Optimizations (HLO) Loop Unrolling Loop Independence
Generate reports on compiler optimizations	Compiler Reports Overview
Use language intrinsics, pragmas and directives, run-time libraries to enhance application performance	Optimization Support Features Overview Compiler Directives Overview

Create parallel programs or parallelize existing programs	Parallelism Overview
	Auto-vectorization (IA-32 Only)
	Auto-parallelization
	Parallelization with OpenMP*

## How to Use This Document

This document assumes you are familiar with general compiler usage, programming methods, and the appropriate Intel® processor architectures. You should also be familiar with the host operating system on your computer. Some of the content is labeled according to operating system; the following table describes the operating system notations.

Notation Conventions	
Linux*	This term refers to information that is valid on all supported Linux operating systems.
Mac OS*	This term refers to information that is valid on Intel®-based systems running Mac OS.
Windows*	This term refers to information that is valid on all supported Microsoft* Windows operating systems.

To understand other conventions used in this and other documents in this set, read *How to Use This Document*, which is located with the introductory topics in the Intel® C++ Compiler Documentation.

## Programming for High Performance Overview

This section provides information on the following:

- **Optimizing Performance:** This section discusses optimization-related strategies and methods.

## Optimizing Performance Overview

While many optimization options can improve performance, most applications will benefit from additional tuning. The high-level information presented in this section discusses methods, tools, compiler options, and pragmas used for analyzing runtime performance-related problems and increasing application performance.

The topics in this section discuss performance issues and methods from a general point of view, focusing primarily on general performance enhancements. The information in this section is separated in the following topics:



- Using Intel Performance Analysis Tools
- Using Tuning Tools and Strategies
- Using a Performance Methodology
- Applying Performance Enhancement Strategies
- Understanding Run-time Performance
- Applying Optimization Strategies
- Understanding Data Alignment
- Timing Your Application

In most cases, other sections and topics in this document contain detailed information about the options, concepts, and strategies mentioned in this section.

## Using Intel® Performance Analysis Tools

Intel® offers an array of performance tools to take advantage of the Intel processors.

These performance tools can help you analyze your application, find problem areas, and develop efficient programs. In some cases, these tools are critical to the optimization process. See Using Tuning Tools and Strategies for suggested uses.

The following table summarizes each of the recommended Intel® performance tools.

Tool	Description
Intel® VTune™ Performance Analyzer	<p>This is a recommended tool for many optimizations. The VTune™ Performance Analyzer collects, analyzes, and provides Intel architecture-specific software performance data from the system-wide view down to a specific module, function, and instruction in your code.</p> <p>The compiler, in conjunction with this tool, can use samples of events monitored by the Performance Monitoring Unit (PMU) and create a hardware profiling data to further enhance optimizations for some programs.</p> <p>For more information, see <a href="http://www.intel.com/software/products/vtune/">http://www.intel.com/software/products/vtune/</a>.</p> <p>Available on Linux* and Windows*.</p>
Intel® Debugger (IDB)	<p>The Intel® Debugger is a full-featured symbolic source code debugger with a command-line interface and a basic graphical user interface.</p> <p>Use this tool to make your applications run more efficiently and to locate programming errors in your code.</p> <p>Available on Linux*, Mac OS*, and Windows*.</p>
Intel® Threading Tools	<p>Intel® Threading Tools consist of the Intel® Thread Checker and the Intel® Thread Profiler.</p> <p>The Intel® Thread Checker can help identify shared and private</p>

	<p>variable conflicts, and can isolate threading bugs to the source code line where the bug occurs.</p> <p>The Intel® Thread Profiler can show the critical path of an application as it moves from thread to thread, and identify synchronization issues and excessive blocking time that cause delays for Win32*, POSIX* threaded, and OpenMP* code.</p> <p>For general information, see <a href="http://www.intel.com/software/products/threading/tcwin/">http://www.intel.com/software/products/threading/tcwin/</a>.</p> <p>Available on Linux* and Windows*.</p>
--	--

## Using Tuning Tools and Strategies

### Identifying and Analyzing Hotspots

To identify opportunities for optimization, you should start with the time-based and event-based sampling functionality of the VTune™ Performance Analyzer.

Time-based sampling identifies the sections of code that use the most processor time. Event-based sampling identifies microarchitecture bottlenecks such as cache misses and mispredicted branches. Clock ticks and instruction count are good counters to use initially to identify specific functions of interest for tuning.

Once you have identified specific functions through sampling, use call-graph analysis to provide thread-specific reports. Call-graph analysis returns the following information about the functions:

- Number of times each function was called
- Location from which each function was called
- Total processor time spent executing each function

You can also use the Counter monitor (which is equivalent to Microsoft\* Perfmon\*) to provide real-time performance data based on more than 200 possible operating system counters, or you can create custom counters created for specific environments and tasks.

You can use Intel® Tuning Assistant and Intel® Thread Checker, which ship as part of the VTune™ Performance Tools. The Intel® Tuning Assistant interprets data generated by the VTune™ Performance Tools and generates application-specific tuning advice based on that information. Intel® Thread Checker provides insight into the accuracy of the threading methodology applied to an application by identifying specific threading issues that should be addressed to improve performance.

See Using Intel Performance Analysis Tools for more information about these tools.

## Using the Intel® Compilers for Tuning

The compilers provide advanced optimization features for Intel processors, which make them an efficient, cost-effective way to improve performance for Intel® architectures. IA-32 and Intel® EM64T compilers support processor dispatch, which allows a single executable to run on current IA-32 Intel microarchitectures and on legacy processors.

You might find the following options useful when tuning:

- Linux\*: `-mtune`, `-x`, `-ax`, `-prof-gen` and `-prof-use`
- Windows\*: `/G{n}`, `/Qx`, `/Qax`, `/Qprof-gen` and `/Qprof-use`

### Note

Mac OS\*: The `-mtune` option is not supported, and P is the only supported value for the `-x` and `-ax` options.

See Optimizations Option Summary and Optimizing for Specific Processors Overview to get more information about the options listed above.

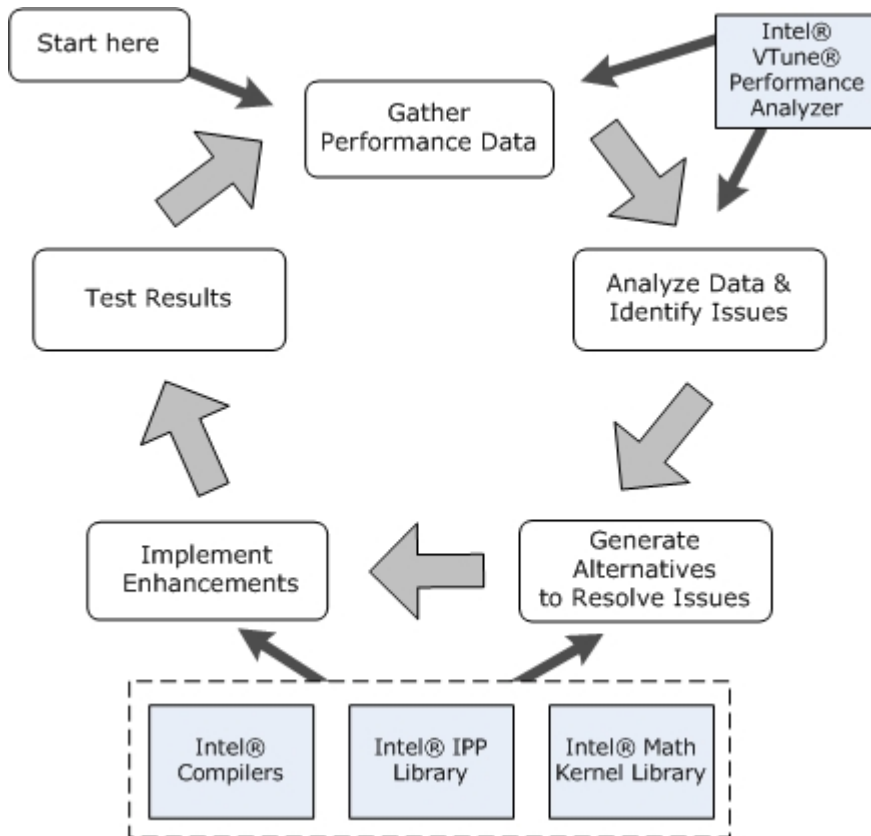
Intel compilers provide support for auto-parallelization and substantial support of OpenMP\* as described in the OpenMP C/C++ version 2.5 specification.

## Using a Performance Enhancement Methodology

The recommended performance enhancement method for optimizing applications consists of several phases. When attempting to identify performance issues, move through the following general phases in the order presented:

- Gather performance data
- Analyze the data
- Generate alternatives
- Implement enhancements
- Test the results

The following figure shows the methodology phases and their relationships, along with some recommended tools to use in each appropriate phase.



In general, the methodology can be summarized by the following two statements:

- Make small changes and measure often.
- If you approach a point of diminishing return and can find no other performance issues, stop optimizing.

## Gather performance data

Use tools to measure where performance bottlenecks occur; do not waste time guessing. Using the right tools for analysis provides an objective data set and baseline criteria to measure implementation changes and improvements introduced in the other stages. The VTune™ Performance Analyzer is one tool you can use to gather performance data and quickly identify areas where the code runs slowly, executes infrequently, or executes too frequently (hotspots) when measured as a percentage of time taken against the total code execution.

See *Using Intel Performance Analysis Tools* and *Using Tuning Tools and Strategies* for more information about some tools and strategies you can use to gather performance data.

## Analyze the data

Determine if the data meet your expectations about the application performance. If not, choose one performance problem at a time for special interest. Limiting the scope of the corrections is critical in effective optimization.

In most cases, you will get the best results by resolving hotspots first. Since hotspots are often responsible for excessive activity or delay, concentrating on these areas tends to resolve or uncover other performance problems that would otherwise be undetectable.

Use the VTune™ Performance Analyzer, or some other performance tool, to discover where to concentrate your efforts to improve performance.

## Generate alternatives

As in the analysis phase, limit the focus of the work. Concentrate on generating alternatives for the one problem area you are addressing. Identify and use tools and strategies to help resolve the issues. For example, you can use compiler optimizations, use Intel® Performance Library routines, or use some other optimization (like improved memory access patterns, reducing or eliminating division or other floating-point operations, rewriting the code to include intrinsics or assembly code, or other strategies).

See *Applying Performance Enhancement Strategies* for suggestions.

While optimizing for the compiler and source levels, consider using the following strategies in the order presented:

1. Use available supported compiler options. This is the most portable, least intrusive optimization strategy.
2. Use compiler directives embedded in the source. This strategy is not overly intrusive since the method involves including a single line in code, which can be ignored (optionally) by the compiler.
3. Attempt manual optimizations.

The preferred strategy within optimization is to use available compiler intrinsics. Intrinsics are usually small single-purpose built-in library routines whose function names usually start with an underscore (`_`), such as the `__mm_prefetch` intrinsic.

If intrinsics are not available, try to manually apply the optimization. Manual optimizations, both high-level language and assembly, are more labor intensive, more prone to error, less portable, and more likely to interfere with future compiler optimizations that become available.

See C++ Intrinsics Reference for more information about the available intrinsics.

## Implement enhancements

As with the previous phases, limit the focus of the implementation. Make small, incremental changes. Trying to address too many issues at once can defeat the purpose and reduce your ability to test the effectiveness of your enhancements.

The easiest enhancements will probably involve enabling common compiler optimizations for easy gains. For applications that can benefit from the libraries, consider implementing Intel® Performance Library routines that may require some interface coding.

## Test the results

If you have limited the scope of the analysis and implementation, you should see measurable differences in performance in this phase. Have a target performance level in mind so you know when you have reached an acceptable gain in performance.

Use a consistent, reliable test that reports a quantifiable item such as seconds elapsed, frames per second, etc., to determine if the implementation changes have actually helped performance.

If you think you can make significant improvement gains or you still have other performance issues to address, repeat the phases beginning with the first one: gather performance data.

## Applying Performance Enhancement Strategies

Improving performance starts with identifying the characteristics of the application you are attempting to optimize. The following table lists some common application characteristics, indicates the overall potential performance impact you can expect, and provides suggested solutions to try. These strategies have been found to be helpful in many cases; experimentation is key with these strategies.

In the context of this discussion, view the potential impact categories as an indication of the possible performance increases that might be achieved when using the suggested strategy. It is possible that application or code design issues will prohibit achieving the indicated increases; however, the listed impacts are generally true. The impact categories are defined in terms of the following performance increases, when compared to the initially tested performance:

- Significant: more than 50%
- High: up to 50%
- Medium: up to 25%
- Low: up to 10%

The following table is ordered by application characteristics and then by strategy with the most significant potential impact.

Application Characteristics	Impact	Suggested Strategies
<b>Technical Applications</b>		
Technical applications with loopy code	High	<p>Technical applications are those programs that have some subset of functions that consume a majority of total CPU cycles in loop nests.</p> <p>Target loop nests using <code>-O3</code> (Linux* and Mac OS*) or <code>/O3</code> (Windows*) to enable more aggressive loop transformations and prefetching.</p> <p>Use High-Level Optimization (HLO) reporting to determine which HLO optimizations the compiler elected to apply.</p> <p>See High-Level Optimization Report.</p>
(same as above) Itanium® Only	High	<p>For <code>-O2</code> and <code>-O3</code> (Linux) or <code>/O2</code> and <code>/O3</code> (Windows), use the SWP report to determine if Software Pipelining occurred on key loops, and if not, why not.</p> <p>You might be able to change the code to allow</p>

		<p>software pipelining under the following conditions:</p> <ul style="list-style-type: none"> <li>• If recurrences are listed in the report that you suspect do not exist, eliminate aliasing problems (for example, using the <code>restrict</code> keyword), or use <code>ivdep</code> pragma on the loop.</li> <li>• If the loop is too large or runs out of registers, you might be able to distribute the loop into smaller segments; distribute the loop manually or by using the <code>distribute</code> pragma.</li> <li>• If the compiler determines the Global Acyclic Scheduler can produce better results but you think the loop should still be pipelined, use the <code>SWP</code> pragma on the loop.</li> </ul>
(same as above) IA-32 and Intel® EM64T Only	High	<p>See Vectorization Overview and the remaining topics in the Auto-Vectorization section for applicable options.</p> <p>See Vectorization Report for specific details about when you can change code.</p>
(same as above)	Medium	<p>Use PGO profile to guide other optimizations.</p> <p>See Profile-guided Optimizations Overview.</p>
Applications with many denormalized floating-point value operations	Significant	<p>Attempt to use flush-to-zero where appropriate.</p> <p>Decide if a high degree of precision is necessary; if it's not then using flush-to-zero might help. Denormal values require hardware or operating system interventions to handle the computation. Denormalized floating point values are those which are too small to be represented in the normal manner, that is, the mantissa cannot be left-justified. Using flush-to-zero causes denormal numbers to be treated as zero by the hardware.</p> <p>Remove floating-point denormals using some common strategies:</p> <ul style="list-style-type: none"> <li>• Change the data type to a larger data type.</li> <li>• Depending on the target architecture, use flush-to-zero or vectorization options.</li> </ul> <p>IA-32 and Intel® EM64T:</p> <ul style="list-style-type: none"> <li>• Flush-to-zero mode is enabled by default for SSE2 instructions. The Intel® EM64T compiler generates SSE2 instructions by</li> </ul>



		<p>default. Enable SSE2 instructions in the IA-32 compiler by using <code>-xW</code>, <code>-xN</code>, <code>-xB</code> or <code>-xP</code> (Linux) or <code>/QxW</code>, <code>/QxN</code>, <code>/QxB</code> or <code>/QxP</code> (Windows).</p> <ul style="list-style-type: none"> <li>See Vectorization Support for more information.</li> </ul> <p>Itanium®:</p> <ul style="list-style-type: none"> <li>The most common, easiest flush-to-zero strategy is to use the <code>-ftz</code> (Linux) or <code>/Qftz</code> (Windows) option on the source file containing <code>main()</code>.</li> <li>Selecting <code>-O3</code> (Linux) or <code>/O3</code> (Windows) automatically enables <code>-ftz</code> (Linux) or <code>/Qftz</code> (Windows).</li> </ul> <p>After using flush-to-zero, ensure that your program still gives correct results when treating denormalized values as zero.</p>
Sparse matrix applications	Medium	<p>See the suggested strategy for memory pointer disambiguation (below).</p> <p>Use <code>prefetch</code> pragma or prefetch intrinsics. Experiment with different prefetching schemes on indirect arrays.</p> <p>See HLO Overview or Data Prefetching starting places for using prefetching.</p>
<b>Server Applications</b>		
Server application with branch-centric code and a fairly flat profile	Medium	<p>Flat profile applications are those applications where no single module seems to consume CPU cycles inordinately.</p> <p>Use PGO to communicate typical hot paths and functions to the compiler, so the Intel® compiler can arrange code in the optimal manner.</p> <p>Use PGO on as much of the application as is feasible.</p> <p>See Profile-guided Optimizations Overview.</p>
Very large server applications (similar to above)	Low	<p>Use <code>-O1</code> (Linux and Mac OS) or <code>/O1</code> (Windows) to optimize for this type of application. Streamlines code in the most generic manner available.</p> <p>This strategy reduces the amount of code being generated, disables inlining, disables speculation, and enables caching of as much of the instruction</p>

		code as possible.
Database engines	Medium	Use <code>-O1</code> (Linux and Mac OS) or <code>/O1</code> (Windows) and PGO to optimize the application code.
(same as above)	Medium	Use <code>-ipo</code> (Linux and Mac OS) or <code>/Qipo</code> (Windows) on entire application.  See Interprocedural Optimizations Overview.
<b>Other Application Types</b>		
Applications with many small functions that are called from multiple locations	Low	Use <code>-ip</code> (Linux and Mac OS) or <code>/Qip</code> (Windows) to enable inter-procedural inlining within a single source module.  Streamlines code execution for simple functions by duplicating the code within the code block that originally called the function. This will increase application size.  As a general rule, do not inline large, complicated functions.  See Interprocedural Optimizations Overview.
(same as above)	Low	Use <code>-ipo</code> (Linux and Mac OS) or <code>/Qipo</code> (Windows) to enable inter-procedural inlining both within and between multiple source modules. You might experience an additional increase over using <code>-ip</code> (Linux and Mac OS) or <code>/Qip</code> (Windows).  Using this option will increase link time due to the extended program flow analysis that occurs.  Interprocedural Optimization (IPO) can perform whole program analysis to help memory pointer disambiguation.

Apart from application-specific suggestions listed above, there are many application-, OS/Library-, and hardware-specific recommendations that can improve performance as suggested in the following tables:

### Application-specific Recommendations

Application Area	Impact	Suggested Strategies
Cache Blocking	High	Use <code>-O3</code> (Linux and Mac OS) or <code>/O3</code> (Windows) to enable automatic cache blocking; use the HLO report to determine if the compiler enabled cache blocking automatically. If not consider manual cache blocking.  See Cache Blocking.

Compiler pragmas for better alias analysis	Medium	Ignore vector dependencies. Use <code>ivdep</code> and other pragmas to increase application speed.  See Vectorization Support.
Memory pointer disambiguation compiler keywords and options	Medium	Use <code>restrict</code> keyword and the <code>-restrict</code> (Linux) or <code>/Qrestrict</code> (Windows) option to disambiguate memory pointers.  If you use the <code>restrict</code> keyword in your source code, you must use <code>-restrict</code> (Linux) or <code>/Qrestrict</code> (Windows) option during compilation.  Instead of using the <code>restrict</code> keyword and option, you can use the following compiler options: <ul style="list-style-type: none"> <li>• <code>-fno-fnalias</code> (Linux)</li> <li>• <code>-ansi-alias</code> (Linux) or <code>/Qansi-alias</code> (Windows)</li> <li>• <code>/Oa</code> (Windows)</li> <li>• <code>/Ow</code> (Windows)</li> <li>• <code>-alias-args</code> (Linux) or <code>/Qalias-args</code> (Windows)</li> </ul>
Light-weight volatile	Low	Some application use <code>volatile</code> to ensure memory operations occur, the application does not need strong memory ordering in the hardware.  Itanium: <ul style="list-style-type: none"> <li>• Use <code>-mno-serialize-volatile</code>.</li> </ul>
Math functions	Low	Use float intrinsics for single precision data type, for example, <code>sqrtof()</code> not <code>sqrt()</code> .  Call Math Kernel Library (MKL) instead of user code.
Use intrinsics instead of calling a function in assembly code	Low	The Intel® compiler includes intrinsics; use these intrinsics to optimize your code. Using compiler intrinsics can help to increase application performance while helping to your code to be more portable.

## Library/OS Recommendations

Area	Impact	Description
Library	Low	Itanium®-based systems only. If you have been using the <code>setjump</code> function, you might consider using the light ( <code>_setjump</code> ) version instead of the heavy ( <code>setjump</code> ) version of the function to reduce the

		amount of floating-point state saved in the setjmp buffer.
Symbol preemption	Low	Linux has a less performance-friendly symbol preemption model than Windows. Linux uses full preemption, and Windows uses no preemption. Use <code>-fminshared -fvisibility=protected</code> .  See Symbol Visibility Attribute Options.
Memory allocation	Low	Using third-party memory management libraries can help improve performance for applications that require extensive memory allocation.

## Hardware/System Recommendations

Component	Impact	Description
Disk	Medium	Consider using more advanced hard drive storage strategies. For example, consider using SCSI instead of IDE.  Consider using the appropriate RAID level.  Consider increasing the number hard drives in your system.
Memory	Low	You can experience performance gains by distributing memory in a system. For example, if you have four open memory slots and only two slots are populated, populating the other two slots with memory will increase performance.
Processor		For many applications, performance scales is directly affected by processor speed, the number of processors, processor core type, and cache size.

## Other Optimization Strategy Information

For more information on advanced or specialized optimization strategies, refer to the Intel® Developer Services: Developer Centers (<http://www.intel.com/cd/ids/developer/asmo-na/eng/19284.htm>) web site.

Refer to the articles and links to additional resources in the listed topic areas of the following Developer Centers:

### Tools and Technologies:

- *Intel® EM64T*
- *Threading*
- *Intel® Software Tools*

### Intel® Processors:

- All areas

## Understanding Run-time Performance

The information in this topic assumes that you are using a performance optimization methodology and have analyzed the application type you are optimizing.

After profiling your application to determine where best to spend your time, attempt to discover what optimizations and what limitations have been imposed by the compiler. Use the compiler reports to determine what to try next.

Depending on what you discover from the reports you may be able to help the compiler through options, pragmas, and slight code modifications to take advantage of key architectural features to achieve the best performance.

The compiler reports can describe what actions have been taken and what actions cannot be taken based on the assumptions made by the compiler. Experimenting with options and pragmas allows you to use an understanding of the assumptions and suggest a new optimization strategy or technique.

## Helping the Compiler

You can help the compiler in some important ways:

- Read the appropriate reports to gain an understanding of what the compiler is doing for you and the assumptions the compiler has made with respect to your code.
- Use specific options, intrinsics, libraries, and pragmas to get the best performance from your application.

For example, if your code is attempting to constantly compute square roots of single precision values, you can gain performance by using the appropriate intrinsic for single precision data type; for example, `sqrtf()` instead of `sqrt()` in C.

See Applying Optimization Strategies for other suggestions.

## Memory Aliasing on Itanium®-based Systems

Memory aliasing is the single largest issue affecting the optimizations in the Intel® compiler for Itanium®-based systems. Memory aliasing is writing to a given memory location with more than one pointer. The compiler is cautious to not optimize too aggressively in these cases; if the compiler optimizes too aggressively, unpredictable behavior can result (for example, incorrect results, abnormal termination, etc.).

Since the compiler usually optimizes on a module-by-module, function-by-function basis, the compiler does not have an overall perspective with respect to variable use for global variables or variables that are passed into a function; therefore, the compiler usually assumes that any pointers passed into a function are likely to be aliased. The compiler

makes this assumption even for pointers you know are not aliased. This behavior means that perfectly safe loops do not get pipelined or vectorized, and performance suffers.

There are several ways to instruct the compiler that pointers are not aliased:

- Use a comprehensive compiler option, such as `-fno-alias` (Linux\*) or `/Oa` (Windows\*). These options instruct the compiler that no pointers in any module are aliased, placing the responsibility of program correctness directly with the developer.
- Use a less comprehensive option, like `-fno-fnalias` (Linux) or `/Ow` (Windows). These options instruct the compiler that no pointers passed through function arguments are aliased.  
Function arguments are a common example of potential aliasing that you can clarify for the compiler. You may know that the arguments passed to a function do not alias, but the compiler is forced to assume so. Using these options tells the compiler it is now safe to assume that these function arguments are not aliased. This option is still a somewhat bold statement to make, as it affects all functions in the module(s) compiled with the `-fno-nalias` (Linux) or `/Ow` (Windows) option.
- Use the `ivdep` pragma. Alternatively, you might use a pragma that applies to a specified loop in a function. This is more precise than specifying an entire function. The pragma asserts that, for a given loop, there are no vector dependencies. Essentially, this is the same as saying that no pointers are aliasing in a given loop.
- Use of keyword `restrict`. An even more precise method of disambiguating pointers is the `restrict` keyword. The `restrict` keyword is used to identify individual pointers as not being aliased. You would use the `restrict` keyword to tell the compiler that a given memory location is not written to by any other pointer.

The following example demonstrates using the `restrict` keyword to tell the compiler that the memory address pointed to by `z` is not written to by any other pointer. With this new information the compiler can then vectorize or software pipeline the loop as follows:

### Example

```
// One-dimension array.
void restrict1(int *x, int *y, int * restrict z)
{
    int A = 42;
    int i;
    double temp;
    for(i=0;i<100;i++) {
        z[i] = A * x[i] + y[i];
    }
}

// Two-dimension array.
void restrict2(int a[][100], int b[restrict][100]) {
    /* ... */
}
```

### Caution

To use the `restrict` keyword as in the example above, you must also use the `-restrict` (Linux\*) or `/Qrestrict` (Windows\*) option on the compile line.

## Non-Unit Stride Memory Access

Another issue that can have considerable impact on performance is accessing memory in a non-Unit Stride fashion. This means that as your inner loop increments consecutively, you access memory from non adjacent locations. For example, consider the following matrix multiplication code:

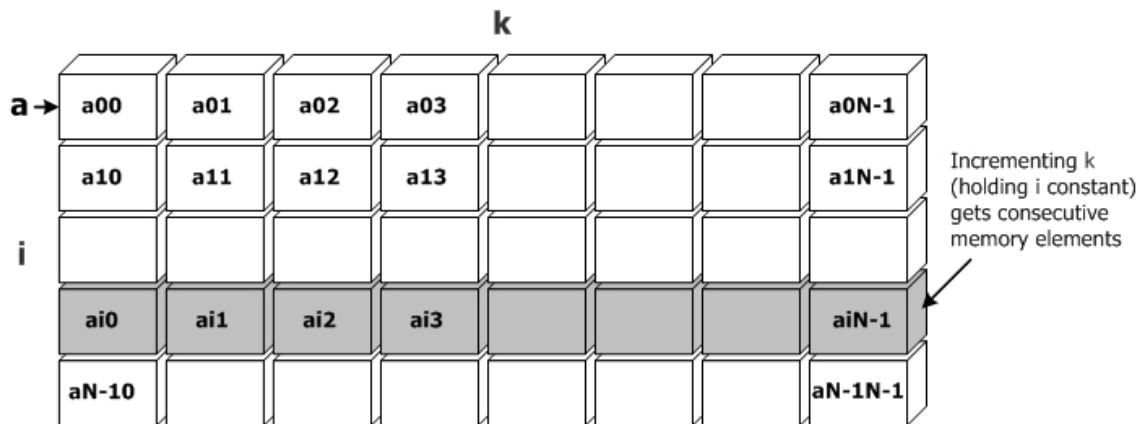
### Example

```
// Non-Unit Stride access problem with b[k][j]
void non_unit_stride(int **a, int **b, int **c)
{
    int A = 42;
    for(int i=0; i<A; i++)
        for(int j=0; j<A; j++)
            for(int k=0; k<A; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

Notice that  $c[i][j]$ , and  $a[i][k]$  both access consecutive memory locations when the inner-most loops associated with the array are incremented. The  $b$  array however, with its loops with indexes  $k$  and  $j$ , does not access Memory Unit Stride. When the loop reads  $b[k=0][j=0]$  and then the  $k$  loop increments by one to  $b[k=1][j=0]$ .

Loop transformation (sometimes called loop interchange) helps to address this problem. While the compiler is capable of doing loop interchange automatically, it does not always recognize the opportunity.

The memory access pattern for the example code listed above is illustrated in the following figure:



Assume you modify the example code listed above by making the following changes to introduce loop interchange:

### Example

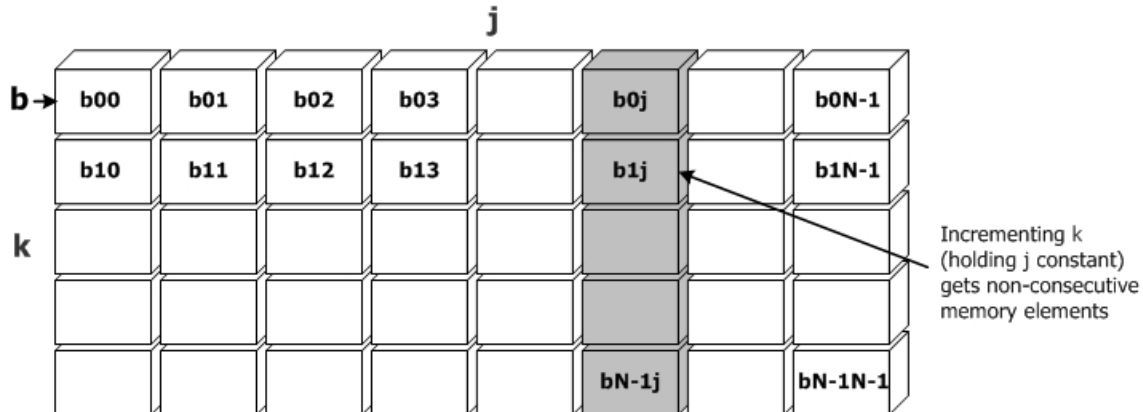
```
// After loop interchange of k and j loops.
void unit_stride(int **a, int **b, int **c)
{
```

```

int A = 42;
for(int i=0; i<A; i++)
  for(int k=0; k<A; k++)
    for(int j=0; j<A; j++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
}

```

After the loop interchange the memory access pattern might look the following figure:



## Applying Optimization Strategies

The compiler may or may not apply the following optimizations to your loop: Interchange, Unrolling, Cache Blocking, Loop Distribution, Loop Fusion, and LoadPair. These transformations are discussed in the following sections, including how to transform loops manually and how to control them with pragmas or internal options.

### Loop Interchange

Loop Interchange is a nested loop transformation applied by High-level Optimization (HLO) that swaps the order of execution of two nested loops. Typically, the transformation is performed to provide sequential Unit Stride access to array elements used inside the loop to improve cache locality. The compiler `-O3` (Linux\*) or `/O3` (Windows\*) optimization looks for opportunities to apply loop interchange for you.

The following is an example of a loop interchange:

#### Example

```

#define NUM 1024
void loop_interchange(
    double a[][NUM], double b[][NUM],
    double c[][NUM] )
{
    int i,j,k;
    // Loop before Loop Interchange
    for(i=0; i<NUM; i++)
        for(j=0; j<NUM; j++)

```



```

    for(k=0;k<NUM;k++)
        c[i][j] =c[i][j] + a[i][k] * b[k][j];
// Loop after Loop Interchange of k & j loops
for(i=0;i<NUM;i++)
    for(k=0;k<NUM;k++)
        for(j=0;j<NUM;j++)
            c[i][j] =c[i][j] + a[i][k] * b[k][j];
}

```

See discussion under Non-Unit Stride Memory Access for more detail.

## Unrolling

Loop unrolling is a loop transformation generally used by HLO that can take better advantage of Instruction-Level Parallelism (ILP), keeping as many functional units busy doing useful work as possible during single loop iteration. In loop unrolling, you add more work to the inside of the loop while doing fewer loop iterations in exchange.

There are pragmas and internal options to control unroll behavior.

Pragma	Description
#pragma unroll	Specifying unroll by itself allows the compiler to determine the unroll factor.
#pragma unroll(n)	Specifying <code>-unroll n</code> (Linux) or <code>/Qunroll:n</code> (Windows) instructs the compiler to unroll the loop <i>n</i> times.
#pragma nounroll	Specifies nounroll instructs the compiler not to unroll a specified loop.

Generally, you should unroll loops by factors of cache line sizes; experiment with the number. Consider the following loop:

Example
<pre> #define NUM 1025 void loop_unroll_before(     double a[][NUM], double b[][NUM],     double c[][NUM]) {     int i,j;     int N,M;     N=NUM;     M=5;     for(i=0;i&lt;N; i++)         for (j=0;j&lt;M; j++)             a[i][j] = b[i][j] + c[i][j]; } </pre>

Assume you want to unroll the “i” or outer loop by a factor 4, but you notice that 4 does not evenly divide N of 1025. Unrolling in this case is difficult; however, you might use a “post conditioning loop” to take care of the unusual case as follows:

**Example**

```
#define NUM 1025
void loop_unroll_after(
    double a[][NUM], double b[][NUM],
    double c[][NUM])
{
    int i,j,K;
    int N,M;
    N=NUM;
    M=5;
    K = N % 4;
    // Main part of loop.
    for(i=0;i<N-K; i+=4)
        for (j=0;j<M; j++) {
            a[i][j] = b[i][j] + c[i][j];
            a[i+1][j] = b[i+1][j] + c[i+1][j];
            a[i+2][j] = b[i+2][j] + c[i+2][j];
            a[i+3][j] = b[i+3][j] + c[i+3][j];
        }
    // Post conditioning part of loop.
    for(i= N-K+1;i<N; i+=4)
        for (j=0;j<M; j++)
            a[i][j] = b[i][j] + c[i][j];
}
```

Post conditioning is preferred over pre-conditioning because post conditioning will preserve the data alignment and avoid the cost of memory alignment access penalties.

## Cache Blocking

Cache blocking involves structuring data blocks so that they conveniently fit into a portion of the L1 or L2 cache. By controlling data cache locality, an application can minimize performance delays due to memory bus access. The application controls the behavior by dividing a large array into smaller blocks of memory so a thread can make repeated accesses to the data while the data is still in cache.

For example, image processing and video applications are well suited to cache blocking techniques because an image can be processed on smaller portions of the total image or video frame. Compilers often use the same technique, by grouping related blocks of instructions close together so they execute from the L2 cache.

The effectiveness of the cache blocking technique depends on data block size, processor cache size, and the number of times the data is reused. Cache sizes vary based on processor. An application can detect the data cache size using the `CPUID` instruction and dynamically adjust cache blocking tile sizes to maximize performance. As a general rule, cache block sizes should target approximately one-half to three-quarters the size of the physical cache. For systems that are Hyper-Threading Technology (HT Technology) enabled target one-quarter to one-half the physical cache size. (See Designing for Hyper-Threading Technology for more other design considerations.)

Cache blocking is applied in HLO and is used on large arrays where the arrays cannot all fit into cache simultaneously. This method is one way of pulling a subset of data into cache (in a small region), and using this cached data as effectively as possible before the data is replaced by new data from memory.

**Example**

```
#define NUM 1024
void cache blocking before(
    double a[][NUM][NUM], double b[][NUM][NUM],
    double c[][NUM][NUM], int N )
{
    int i,j,k;
    N=1000;
    for (i=0;i < N; i++)
        for (j=0;j < N; j++)
            for (k=0;k < N; k++)
                a[i][j][k] = a[i][j][k] + b[i][j][k];
}

#define NUM 1024
void cache blocking after(
    double a[][NUM][NUM], double b[][NUM][NUM],
    double c[][NUM][NUM], int N )
{
    int i,j,k,u,v;
    N=1000;
    for (v=0; v<N; v+=20)
        for (u=0; u<N; u+=20)
            for (k=v; k<v+20; k++)
                for (j=u; j<u+20; j++)
                    for (i=0; i < N; i++)
                        a[i][j][k] = a[i][j][k] + b[i][j][k];
}
```

The cache block size is set to 20. The goal is to read in a block of cache, do every bit of computing we can with the data in this cache, then load a new block of data into cache. There are 20 elements of **A** and 20 elements of **B** in cache at the same time and you should do as much work with this data as you can before you increment to the next cache block.

Blocking factors will be different for different architectures. Determine the blocking factors experimentally. For example, different blocking factors would be required for single precision versus double precision. Typically, the overall impact to performance can be significant.

## Loop Distribution

Loop distribution is a high-level loop transformation that splits a large loop into two smaller loops. It can be useful in cases where optimizations like software-pipelining (SWP) or vectorization cannot take place due to excessive register usage. By splitting a loop into smaller segments, it may be possible to get each smaller loop or at least one of the smaller loops to SWP or vectorize. An example is as follows:

**Example**

```
#define NUM 1024
void loop distribution before(
    double a[NUM], double b[NUM], double c[NUM],
    double x[NUM], double y[NUM], double z[NUM] )
{
    int i;
```

```

// Before distribution or splitting the loop.
for (i=0; i< NUM; i++) {
    a[i] = a[i] + i;
    b[i] = b[i] + i;
    c[i] = c[i] + i;
    x[i] = x[i] + i;
    y[i] = y[i] + i;
    z[i] = z[i] + i;
}
}

#define NUM 1024
void loop_distribution_after(
    double a[NUM], double b[NUM], double c[NUM],
    double x[NUM], double y[NUM], double z[NUM] )
{
    int i;
    // After distribution or splitting the loop.
    for (i=0; i< NUM; i++) {
        a[i] = a[i] + i;
        b[i] = b[i] + i;
        c[i] = c[i] + i;
    }
    for (i=0; i< NUM; i++) {
        x[i] = x[i] + i;
        y[i] = y[i] + i;
        z[i] = z[i] + i;
    }
}

```

There are pragmas to suggest distributing loops to the compiler as follows:

### Example

```
#pragma distribute point
```

Placed outside a loop, the compiler will attempt to distribute the loop based on an internal heuristic. The following is an example of using the pragma outside the loop:

### Example

```

#define NUM 1024
void loop_distribution_pragma1(
    double a[NUM], double b[NUM], double c[NUM],
    double x[NUM], double y[NUM], double z[NUM] )
{
    int i;
    // Before distribution or splitting the loop
    #pragma distribute point
    for (i=0; i< NUM; i++) {
        a[i] = a[i] + i;
        b[i] = b[i] + i;
        c[i] = c[i] + i;
        x[i] = x[i] + i;
        y[i] = y[i] + i;
        z[i] = z[i] + i;
    }
}

```

Placed within a loop, the compiler will attempt to distribute the loop at that point. All loop-carried dependencies will be ignored. The following example uses the pragma within a loop to precisely indicate where the split should take place:

### Example

```
#define NUM 1024
void loop_distribution_pragma2(
    double a[NUM], double b[NUM], double c[NUM],
    double x[NUM], double y[NUM], double z[NUM] )
{
    int i;
    // After distribution or splitting the loop.
    for (i=0; i< NUM; i++) {
        a[i] = a[i] +i;
        b[i] = b[i] +i;
        c[i] = c[i] +i;
        #pragma distribute point
        x[i] = x[i] +i;
        y[i] = y[i] +i;
        z[i] = z[i] +i;
    }
}
```

## Loop Fusion

Loop Fusion is the inverse of Loop Distribution. The idea in loop fusion is to join two loops that have the same trip count in order to reduce loop overhead. The `-O3` (Linux) or `/O3` (Windows) option will attempt loop fusion if the opportunity is present.

## Load Pair (Itanium® Compiler)

Load pairs (ldfp) are instructions that load two contiguous single or double precision values from memory in one move. Load pairs can significantly improve performance.

## Manual Loop Transformations

There might be cases where these manual transformations are called acceptable or even preferred. As a general rule, you should let the compiler transform loops for you. Manually transform loops as a last resort; use this strategy only in cases where you are attempting to gain performance increases.

Manual loop transformations have many disadvantages, which include the following:

- Application code becomes harder to maintain over time.
- New compiler features can cause you to lose any optimization you gain by manually transforming the loop.
- Architectural requirements might restrict your code to a specific architecture unintentionally.

The HLO report can give you an idea of what loop transformations have been applied by the compiler.

Experimentation is a critical component in manually transforming loops. You might try to apply a loop transformation that the compiler ignored. Sometimes, it is beneficial to apply a manual loop transformation that the compiler has already applied with `-O3` (Linux) or `/O3` (Windows).

## Understanding Data Alignment

Aligning data on boundaries can help performance. The Intel® compiler attempts to align data on boundaries for you. However, as in all areas of optimization, coding practices can either help or hinder the compiler and can lead to performance problems. Always attempt to optimize using compiler options first. See Optimization Options Summary for more information.

To avoid performance problems you should keep the following guidelines in mind, which are separated by architecture:

### IA-32, Intel® EM64T, Intel® Itanium® architectures:

- Do not access or create data at large intervals that are separated by exactly  $2^n$  (for example, 1 KB, 2 KB, 4 KB, 16 KB, 32 KB, 64 KB, 128 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, etc.).
- Align data so that memory accesses does not cross cache lines (for example, 32 bytes, 64 bytes, 128 bytes).
- Use `__mm_malloc(size, alignment, [offset])` to force allocated structures to be enforce the rules above.
- Use Application Binary Interface (ABI) for the Itanium® compiler to insure that ITP pointers are 16-byte aligned.

### IA-32 and Intel® EM64T architectures:

- Align data to correspond to the SIMD or Streaming SIMD Extension registers sizes.
- Use either `__assume_aligned()` or `#pragma vector aligned` to instruct the compiler that the data is aligned.

### Itanium® architecture:

- Avoid using packed structures.
- Avoid casting pointers of small data elements to pointers of large data elements.
- Do computations on unpacked data, then repack data if necessary, to correctly output the data.
- Use `__unaligned` keyword on pointers to unaligned data to cause the structure to be accessed one byte at a time. This is a slow alternative.

In general, keeping data in cache has a better performance impact than keeping the data aligned. Try to use techniques that conform to the rules listed above.

## Pack

When structures are packed with the `pack` pragma, pointers to interior members of the structure can cause unaligned access. Unaligned access will cause an application on Itanium®-based systems to terminate prematurely by default. You can get around this limitation by calling WINAPI function `seterrormode`. The condition is not fatal, only less efficient.

For the Itanium compiler packed structures are smaller in size but much slower. You will get software exceptions almost every time you access unaligned data.

## Timing Your Application

You can start collecting information about your application performance with simply timing your application. More sophisticated and helpful data can be collected by using performance analyzing tools.

## Considerations on Timing Your Application

One of the performance indicators is your application timing. The following considerations apply to timing your application:

- Run program timings when other users are not active. Your timing results can be affected by one or more CPU-intensive processes also running while doing your timings.
- Try to run the program under the same conditions each time to provide the most accurate results, especially when comparing execution times of a previous version of the same program. Use the same system (processor model, amount of memory, version of the operating system, and so on) if possible.
- If you do need to change systems, you should measure the time using the same version of the program on both systems, so you know each system's effect on your timings.
- For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Certain overhead functions like loading libraries might influence short timings considerably.
- If your program displays a lot of text, consider redirecting the output from the program. Redirecting output from the program will change the times reported because of reduced screen I/O.

Timings that show a large amount of system time may indicate a lot of time spent doing I/O, which might be worth investigating.

- For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Overhead functions like loading shared libraries might influence short timings considerably.

Use the `time` command and specify the name of the executable program to provide the following:

- The elapsed, real, or "wall clock" time, which will be greater than the total charged actual CPU time.

- Charged actual CPU time, shown for both system and user execution. The total actual CPU time is the sum of the actual user CPU time and actual system CPU time.

The following program illustrates a model for program timing:

### Example

```
/* Sample Timing */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    clock_t start, finish;
    long loop;
    double duration, loop_calc;
    start = clock();
    for(loop=0; loop <= 2000; loop++)
    {
        loop_calc = 123.456 * 789;
        //printf() included to facilitate example
        printf("\nThe value of loop is: %d", loop);
    }
    finish = clock();
    duration = (double)(finish - start)/CLOCKS_PER_SEC;
    printf("\n%.3f seconds\n", duration);
}
```



## Compiler Optimizations Overview

The variety of optimizations used by the Intel® compiler enable you to quickly enhance the application performance. Each optimization is performed by a set of options and, in some cases, tools. The options and tools are discussed in the following optimization-related sections:

- Optimizing for Specific Processors
- Interprocedural Optimizations (IPO)
- Profile-guided Optimizations (PGO)
- High-level Optimizations (HLO)
- Floating-Point Arithmetic Optimizations
- Compiler Reports

In addition to optimizations invoked by the compiler command-line options, the compiler includes features which enhance your application performance such as directives, pragmas, intrinsics, run-time library routines and various utilities. These features are discussed in the Optimization Support Features section.

## Optimization Options Summary

This topic discusses the compiler options affecting code size, locality and code speed. The following table summarizes the most common code optimization options you can use for quick results.

Windows*	Linux*	Description
/O3	-O3	Enables aggressive optimization for code speed. Recommended for code with loops that perform calculations or process large data sets.
/O2	-O2 (or -O)	Affects code speed. This is the default option, so the compiler uses this optimization level if you do not specify anything.
/O1	-O1	Affects code size and locality. Disables specific optimizations.
/fast	-fast	Enables a collection of common, recommended optimizations for run-time performance.
/Od	-O0	Disables optimization. Use this for debugging and rapid compilation.

The following sections summarize the common optimizations:

- Setting optimization levels
- Restricting optimizations
- Diagnostics

## Setting Optimization Levels

The following table lists the relevant code optimization options, describes the characteristics shared by IA-32, Itanium®, and Intel® EM64T architectures, and describes the general behavior on each architecture.

The architectural differences and compiler options these code optimizations either enable or disable are also listed in more detail in the associated Compiler Options topics; each option discussion includes a link to the appropriate Compiler Options topic.

Windows	Linux	Effect
/O1	-O1	<p>Optimizes to favor code size and code locality. This optimization level might improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops. In general, this optimization level does the following:</p> <ul style="list-style-type: none"> <li>• Enables global optimization.</li> <li>• Disables intrinsic recognition and intrinsics inlining.</li> <li>• On Itanium®-based systems, it disables software pipelining, loop unrolling, and global code scheduling.</li> </ul> <p>In most cases, -O2 (Linux) or /O2 (Windows) is recommended over this option.</p> <p>To see which options this option sets or to get detailed information about the architecture- and operating system-specific behaviors, see the following topic:</p> <ul style="list-style-type: none"> <li>• -O1 compiler option</li> </ul>
/O2	-O2, -O	<p>Optimizes for code speed. Since this is the default optimization, if you do not specify an optimization level the compiler will use this code optimization level automatically. This is the generally recommended optimization level; however, specifying other compiler options can affect the optimization normally gained using this level.</p> <p>This option enables intrinsics inlining and the following capabilities for performance gain: constant propagation, copy propagation, dead-code elimination, global register allocation, global instruction scheduling and control speculation, loop unrolling, optimized code selection, partial redundancy elimination, strength reduction/induction variable simplification, variable renaming, exception handling optimizations, tail recursions, peephole optimizations, structure assignment lowering optimizations, and dead store</p>

		<p>elimination.</p> <p>This option behaves differently depending on architecture.</p> <p>Itanium®-based systems:</p> <ul style="list-style-type: none"> <li>Enables optimizations for speed, including global code scheduling, software pipelining, predication, speculation, and data prefetch.</li> </ul> <p>To see what options this option sets, or to get detailed information about the architecture- and operating system-specific behaviors, see the following topic:</p> <ul style="list-style-type: none"> <li>-O2 compiler option</li> </ul>
/O3	-O3	<p>Enables -O2 (Linux) or /O2 (Windows) optimizations, and enables more aggressive optimizations such as prefetching, scalar replacement, cache blocking, and loop and memory access transformations. Enables optimizations for maximum speed, but does not guarantee higher performance unless loop and memory access transformation take place. The optimizations enabled by this option can slow down code in some cases compared to -O2 (Linux) or /O2 (Windows) optimizations.</p> <p>Recommended for applications that have loops that heavily use floating-point calculations and process large data sets.</p> <p>Like the other code optimization options, this option behaves differently depending on architecture and operating system.</p> <p>IA-32 systems:</p> <ul style="list-style-type: none"> <li>When used with the -ax or -x (Linux) or /Qax or /Qx (Windows), this option causes the compiler to perform more aggressive data dependency analysis than for -O2 (Linux) or /O2 (Windows); however, this scenario might result in longer compilation times.</li> <li>-xP and -axP are the only valid values on Mac OS* systems.</li> </ul> <p>Itanium®-based systems:</p> <ul style="list-style-type: none"> <li>Enables optimizations for technical computing applications (loop-intensive code): loop optimizations and data prefetch.</li> </ul>

		<p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-O3</code> compiler option</li> </ul>
<code>/Ox</code>	No equivalent	<p>Enables maximum optimization by combining compiler options to perform the following: let the compiler determine which functions to inline, enable global optimizations, specify that EBP be used as a general-purpose register for optimizations, enable all speed optimizations, and enable inline expansion of intrinsic functions.</p> <p>This options is supported on all architectures.</p> <p><b>Note</b></p> <p>Mac OS*: The option is not supported.</p>
<code>/fast</code>	<code>-fast</code>	<p>Provides a simple, single optimization that allows you to enable a collection of optimizations for run-time performance. This option sets specific options, depending on architecture and operating system:</p> <p>Itanium®-based systems:</p> <ul style="list-style-type: none"> <li>• Linux: <code>-ipo</code>, <code>-O3</code>, and <code>-static</code></li> <li>• Windows: <code>/O3</code> and <code>/Qipo</code></li> </ul> <p>IA-32 and Intel® EM64T systems:</p> <ul style="list-style-type: none"> <li>• Linux: <code>-ipo</code>, <code>-O3</code>, <code>-no-prec-div</code>, <code>-static</code>, and <code>-xP</code></li> <li>• Windows: <code>/O3</code>, <code>/Qipo</code>, <code>/Qprec-div-</code>, and <code>/QxP</code></li> </ul> <p><b>Note</b></p> <p>Mac OS*: The <code>-xP</code> and <code>-static</code> options are not supported.</p> <p>Programs compiled with the <code>-xP</code> (Linux) or <code>/QxP</code> (Windows) option will detect non-compatible processors and generate an error message during execution.</p> <p>Additionally, for IA-32 and Intel® EM64T systems, the <code>-xP</code> (Linux) or <code>/QxP</code> (Windows) option that is set by this option cannot be overridden by other command line options. If you specify this option along with a different processor-specific option, such as <code>-xN</code> (Linux) or <code>/QxN</code> (Windows), the compiler will issue a warning stating the <code>-xP</code> or <code>/QxP</code> option cannot be overridden; the best strategy for dealing with this restriction is to explicitly specify the options you want to set from the</p>

		<p>command line.</p> <p>While this option enables other options quickly, the specific options enabled by this option might change from one compiler release to the next. Be aware of this possible behavior change in the case where you use makefiles.</p> <p>For more information on restrictions and usage, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-fast</code> compiler option</li> </ul>
--	--	--

## Restricting Optimization

The following table lists options that restrict the ability of the Intel® compiler to optimize programs.

Windows	Linux	Effect
<code>/Od</code>	<code>-O0</code>	<p>Disables all optimizations. Use this during development stages where fast compile times are desired.</p> <p>Linux:</p> <ul style="list-style-type: none"> <li>• Sets option <code>-fp</code> and option <code>-fmath-errno</code>.</li> </ul> <p>Windows:</p> <ul style="list-style-type: none"> <li>• Use <code>/Od</code> to disable all optimizations while specifying particular optimizations, such as: <code>/Od /Ob1</code> (disables all optimizations, but only enables inlining)</li> </ul> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-O0</code> compiler option</li> </ul>
<code>/Zi, /Z7</code>	<code>-g</code>	<p>Generates symbolic debugging information in object files for use by debuggers.</p> <p>This option enables or disables other compiler options depending on architecture and operating system; for more information about the behavior, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-g</code> compiler option</li> </ul>
<code>/Op</code>	<code>-mp</code>	<p>Enables conformance to the ANSI C and IEEE 754 standards for floating-point arithmetic. Restricts optimizations that cause</p>

		<p>some minor loss or gain of precision in floating-point arithmetic to maintain a declared level of precision and to ensure that floating-point arithmetic more nearly conforms to the ANSI and IEEE* standards. See <a href="#">Improving or Restricting FP Arithmetic Precision</a> for more details.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-mp</code> compiler option</li> </ul>
/Oi-	-fno-builtin	<p>Disables inline expansion of intrinsic functions.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-fbuiltin</code> compiler option</li> </ul>
No equivalent	-fmath-errno, -fno-math-errno	<p>Instructs the compiler to assume that the program tests <code>errno</code> after calls to math library functions.</p> <ul style="list-style-type: none"> <li>• <code>-fmath-errno</code> compiler option</li> </ul>

For more information on ways to restrict optimization, see [Using Option Specifiers and Optimization Restriction Support](#).

## Diagnostic Options

Windows	Linux	Effect
/Qsox	-sox	<p>Instructs the compiler to save the compiler options and version number in the executable. During the linking process, the linker places information strings into the resulting executable. Slightly increases file size, but using this option can make identifying versions for regression issues much easier.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-sox</code> compiler option</li> </ul>

## Optimizing for Specific Processors Overview

This section summarizes the options used to target specific processors. This section also provides some compiler command examples that demonstrate how to use these options. See Targeting a Processor.

The following table lists the most common options used to target compiles for specific processor families.

Windows*	Linux*	Target Processors
/G{5 6 7}	-mtune=<target cpu>  -mcpu=<target cpu>	IA-32 and Intel® EM64T processors.  See Options for IA-32 Processors in Targeting a Processor.  <b>Note</b>  Mac OS*: This option is not supported.
/G{1 2}	-mtune=<target cpu>  -mcpu=<target cpu>	Itanium® processors.  See Options for Itanium® processors in Targeting a Processor.
/Qx{K W N B P T}  /Qax{K W N B P T}	-x{K W N B P T}  -ax{K W N B P T}	Target applications to run on the specific processor-based systems and still gear your code to any processors that took advantage of the following options:  <ul style="list-style-type: none"> <li>Linux: -x{K W N} or -ax{K W N}</li> <li>Windows: /Qx{N} or /Qax{N}</li> </ul> See Processor-specific Optimization and Automatic Processor-specific Optimization.

## Targeting a Processor

The Intel compiler supports specific options to help optimize performance for specific Intel processors.

The -mtune (Linux\*) or /G{n} (Windows\*) option generates code that is backwards compatible with Intel® processors of the same family. This means that code generated with -mtune=pentium (Linux) or /G7 (Windows) will run correctly on Pentium Pro or Pentium III processors, possibly just not quite as fast as if the code had been compiled with -mtune=pentiumpro (Linux) or /G6 (Windows). Similarly, code generated with -mtune=itanium2 or -tpp2 (Linux) or /G2 (Windows) will run correctly on Itanium

processor, but possibly not quite as fast as if it had been generated using `-mtune=itanium` or `-tpp1` (Linux) or `/G1` (Windows).

### Note

Mac OS\*: The options listed in this topic are not supported.

## Options for IA-32 and Intel® EM64T Processors

The following options optimize application performance for a specific IA-32 or Intel® EM64T processor. The resulting binaries will also run correctly on any of the processors mentioned in the table.

Windows*	Linux*	Optimizes applications for...
/G5	<code>-mtune=pentium</code>  <code>-mtune=pentium-mmx</code>  <code>-mcpu=pentium</code>	Intel® Pentium® and Pentium® with MMX™ technology processor
/G6	<code>-mtune=pentiumpro</code>  <code>-mcpu=pentiumpro</code>	Intel® Pentium® Pro, Pentium® II and Pentium® III processors
/G7	<code>-mtune=pentium4</code>  <code>-mcpu=pentium4</code>	Default. Intel® Pentium® 4 processors, Intel® Core™ Duo processors, Intel® Core™ Solo processors, Intel® Xeon® processors, Intel® Pentium® M processors, and Intel® Pentium® 4 processors with Streaming SIMD Extensions 3 (SSE3) instruction support  Intel® EM64T:  <ul style="list-style-type: none"> <li><code>-mcpu=pentium4</code> is the only supported value.</li> </ul>

For more information about the options listed in the table above, see the following topic:

- `mtune` compiler option

The example commands shown below each result in a compiled binary of the source program `prog.cpp` optimized for Pentium 4 and Intel® Xeon® processors by default. The same binary will also run on Pentium, Pentium Pro, Pentium II, Pentium III, and more advanced processors.



The following examples demonstrate using the default options to target a processor:

Platform	Example
Linux	<code>icc -mtune=pentium4 prog.cpp</code>
Windows	<code>icl /G7 prog.cpp</code>

## Options for Itanium® Processors

The following options optimize application performance for an Itanium® processors.

Windows*	Linux*	Optimizes applications for...
/G1	-tpp1  -mtune=itanium  -mcpu=itanium	Intel® Itanium® processors
/G2	-tpp2  -mtune=itanium2  -mcpu=itanium2	Default. Intel® Itanium® 2 processors
/G2-p9000	-mtune=itanium2-p9000  -mcpu=itanium2-p9000	Dual-Core Intel® Itanium® 2 Processor 9000 Sequence processors

For more information about the options listed in the table above, see the following topic:

- `mtune`, `tpp1`, `tpp2`, `G1`, `G2`, `mcpu` compiler options

The following examples demonstrate using the default options to target an Itanium® 2 processor. The same binary will also run on Itanium processors.

Platform	Example
Linux	<code>icpc -mtune=itanium2 prog.cpp</code>
Windows	<code>icl /G2 prog.cpp</code>

## Processor-specific Optimization (IA-32 only)

The `-x` (Linux\*) or `/Qx` (Windows\*) options target your program to run on a specific Intel® processor. The resulting code might contain unconditional use of features that are not supported on other processors.

Windows*	Linux*	Optimizes code for...
<code>/QxK</code>	<code>-xK</code>	Intel® Pentium® III and compatible Intel processors
<code>/QxW</code>	<code>-xW</code>	Intel® Pentium® 4 and compatible Intel processors; this is the default for Intel® EM64T systems
<code>/QxN</code>	<code>-xN</code>	Intel Pentium 4 and compatible Intel processors with Streaming SIMD Extensions 2 (SSE2)
<code>/QxB</code>	<code>-xB</code>	Intel Pentium M and compatible Intel processors
<code>/QxP</code>	<code>-xP</code>	Intel® Core™ Duo processors and Intel® Core™ Solo processors, Intel Pentium 4 processors with Streaming SIMD Extensions 3, and compatible Intel processors with Streaming SIMD Extensions 3 (SSE3)
<code>/QxT</code>	<code>-xT</code>	Intel® Core™2 Duo processors, Intel® Core™2 Extreme processors, and the Dual-Core Intel® Xeon® processor 5100 series

For the `N`, `P`, and `T` processor values, the compiler enables processor-specific optimizations that include advanced data layout and code restructuring to improve memory accesses for Intel processors.

On Intel® EM64T systems, `W`, `P`, and `T` are the only valid processor values.

On Intel-based systems running Mac OS\*, `P` is the only valid processor value. For processors that support SSE2 or SSE3, specifying the `N` or `P` values may cause the resulting generated code to contain unconditional use of features that are not supported on other processors.

The default behavior varies depending on the operation system. For more information about the options listed in the table above, see the following topic:

- `-x` compiler option

The following examples compile an application for Intel Pentium 4 and compatible processors. The resulting binary might not execute correctly on Pentium, Pentium Pro, Pentium II, Pentium III, or Pentium with MMX™ technology processors, or on x86 processors not provided by Intel Corporation.

Platform	Example
Linux	<code>icc -xW prog.cpp</code>
Windows	<code>icl /QxW prog.cpp</code>

**Caution**

If a program compiled with `-x` (Linux) or `/Qx` (Windows) is executed on a non-compatible processor, it might fail with an illegal instruction exception or display other unexpected behavior. Executing programs compiled with `-xN`, `-xB` or `-xP` (Linux) or `/QxN`, `/QxB` or `/QxP` (Windows) on unsupported processors will display a run-time error similar to the following:

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

## Automatic Processor-specific Optimization (IA-32 Only)

The `-ax` (Linux\*) or `/Qax` (Windows\*) options direct the compiler to find opportunities to generate separate versions of functions that take advantage of features in a specific Intel® processor.

If the compiler finds such an opportunity, the compiler first checks whether generating a processor-specific version of a function is likely to result in a performance gain. If the compiler determines there is a likely performance gain, it generates both a processor-specific version of a function and a generic version of the function. The generic version will run on any IA-32 processor.

At run time, one of the generated versions executes depending on the Intel processor being used. Using this strategy, the program can benefit from performance gains on more advanced Intel processors, while still working properly on older IA-32 processors.

Windows*	Linux*	Optimizes code for...
<code>/QaxK</code>	<code>-axK</code>	Pentium® III and compatible Intel processors
<code>/QaxW</code>	<code>-axW</code>	Pentium 4 and compatible Intel processors
<code>/QaxN</code>	<code>-axN</code>	Pentium 4 and compatible Intel processors
<code>/QaxB</code>	<code>-axB</code>	Pentium M and compatible Intel processors
<code>/QaxP</code>	<code>-axP</code>	Intel® Core™ Duo processors and Intel® Core™ Solo processors, Intel Pentium 4 processors with Streaming SIMD Extensions 3, and compatible Intel processors with Streaming SIMD Extensions 3 (SSE3)
<code>/QaxT</code>	<code>-axT</code>	Intel® Core™2 Duo processors, Intel® Core™2 Extreme processors, and the Dual-Core Intel® Xeon® processor 5100 series

On Intel® EM64T systems, `W`, `P`, and `T` are the only valid processor values. On Intel-based systems running Mac OS\*, `P` is the only valid processor value.

The following compilation examples demonstrate how to generate an executable that includes:

- An optimized version for Pentium 4 processors, as long as there is a performance gain
- An optimized version for Pentium M processors, as long as there is a performance gain
- A generic version that runs on any IA-32 processor

Platform	Example
Linux	<code>icc -axNB prog.cpp</code>
Windows	<code>icl /QaxNB prog.cpp</code>

For more information about the options listed in the table above, see the following topic:

- `-ax` compiler option

The disadvantages of using `-ax` (Linux) or `/Qax` (Windows) are:

- The size of the compiled binary increases because it contains processor-specific versions of some of the code, as well as a generic version of the code.
- Performance is affected slightly by the run-time checks needed to determine which code to use.

Applications that you compile with this option will execute on any IA-32 processor. If you specify both the `-x` (Linux) or `/Qx` (Windows) and `-ax` (Linux) or `/Qax` (Windows) options, the `-x` (Linux) or `/Qx` (Windows) option forces the generic code to execute only on processors compatible with the processor type specified by the `-x` (Linux) or `/Qx` (Windows) option.

## Processor-specific Run-time Checks for IA-32 Systems

Specific optimizations can take effect at run-time. On IA-32 systems the compiler enhances processor-specific optimizations by inserting in the main routine a code segment that performs run-time checks, as described below.

### Check for Supported Processor

To prevent from execution errors, the compiler inserts code in the main routine of the program to check for proper processor usage.

Programs compiled with the options `-xN`, `-xB`, or `-xP` (Linux\*) or `/QxK`, `/QxW`, `/QxN`, `/QxB` or `/QxP` (Windows\*) check at run-time whether they are being executed on the associated processor or a compatible Intel processor. If the program is not executed on one of these processors, the program terminates with an error.

On Mac OS\* systems, P is the only valid value for the `-x` and `-ax` options.

The following example demonstrates how to optimize a program for run-time checking on an Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support.

Platform	Example
Linux	<code>icc -xP prog.cpp</code>
Windows	<code>icl /QxP prog.cpp</code>

The resulting program will abort if it is executed on a processor that is not validated to support the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support.

If you intend to run your programs on multiple IA-32 processors, do not use the `-x` (Linux) or `/Qx` (Windows) options; consider using the `-ax` (Linux) or `/Qax` (Windows) option to attain processor-specific performance and portability among different processors.

## Setting FTZ and DAZ Flags

In earlier versions of the compiler, the values of the flags flush-to-zero (FTZ) and denormals-as-zero (DAZ) for IA-32 processors were set to off by default; however, even at the cost of losing IEEE compliance, turning these flags on can significantly increase the performance of programs with denormal floating-point values in the gradual underflow mode run on the most recent IA-32 processors.

### Note

Mac OS\*: The `-ftz` option is not supported.

By default, the `-O3` (Linux) or `/O3` (Windows) option enables FTZ mode; in contrast, the `-O2` (Linux) or `/O2` (Windows) option disables it. Alternately, you can use `-no-ftz` (Linux) or `/Qftz-` (Windows) to disable flushing denormal results to zero.

When SSE instructions are used, options `-no-ftz` (Linux) and `/Qftz-` (Windows) are ignored. However, you can enable gradual underflow with code in the main program that clears the FTZ and DAZ bits in the MXCSR.

The compiler inserts code in the program to perform a run-time check for the processor on which the program runs to verify it is one of the previously mentioned Intel processors.

- Executing a program on a Pentium III processor enables the FTZ flag, but not DAZ.
- Executing a program on an Intel® Pentium® M processor or Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support enables both the FTZ and DAZ flags.

These flags are enabled only on by Intel processors that have been validated to support them.

For non-Intel processors, you can set the flags manually with the following macros:

Feature	Examples
Enable FTZ	<code>_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON)</code>
Enable DAZ	<code>_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON)</code>

The prototypes for these macros are in `xmmintrin.h` (FTZ) and `pmmmintrin.h` (DAZ).

## Manual CPU Dispatch (IA-32 only)

Use manual CPU dispatch to write code that will execute only on a specific processor. Your code can detect the processor type by using the dispatch functionality supported in the compiler by using the `cpu_specific` and `cpu_dispatch` keywords.

Use the `__declspec(cpu_specific)` and `__declspec(cpu_dispatch)` syntax in your code to write code specific to a targeted Intel processor and allow the application to execute correctly on other IA-32 processors.

### Note

Manual CPU dispatch will not recognize Intel® Itanium® processors.

The general syntax for these keywords change a function declaration by using the following arguments:

- `cpu_specific(cpu_id)`
- `cpu_dispatch(cpu_id-list)`

The following table lists the values for `cpu_id`:

Processor	Argument for <code>cpu_id</code>
Intel® Pentium® processors	<code>pentium</code>
Intel Pentium processors with MMX™ Technology	<code>pentium_mmx</code>
Intel Pentium Pro processors	<code>pentium_pro</code>
Intel Pentium II processors	<code>pentium_ii</code>
Intel Pentium III processors	<code>pentium_iii</code>
Intel Pentium III (exclude xmm registers)	<code>pentium_iii_no_xmm_regs</code>
Intel Pentium 4 processors	<code>pentium_4</code>
Intel Pentium M processors	<code>pentium_m</code>
Intel Pentium 4 processor with Streaming SIMD Extensions 3 (SSE3), Intel® Core™ Duo processors, Intel® Core™ Solo	<code>pentium_4_sse3</code>

processors	
x86 processors not provided by Intel Corporation	generic

The following table lists the syntax for `cpuid-list`:

Syntax for <code>cpuid-list</code>
<code>cpuid</code>
<code>cpuid-list, cpuid</code>

The attributes are not case sensitive. The body of a function declared with `__declspec(cpu_dispatch)` must be empty, and is referred to as a stub (an empty-bodied function).

Manual CPU dispatch can disable some types of inlining, almost always results in larger code and executable sizes, and can introduce additional performance overhead because of the additional function calls. Test your application on all of the targeted platforms before release. Before using manual dispatch, consider whether the benefits outweigh the additional effort and possible performance issues.

The following example demonstrates using manual dispatch with both `cpu_specific` and `cpu_dispatch`.

Example

```
#include <stdio.h>
#include <mmintrin.h>
/* Pentium processor function does not use intrinsics
   to add two arrays. */
__declspec(cpu_specific(pentium))
void array_sum1(int *result, int *a, int *b, size_t len)
{
    for (; len > 0; len--)
        *result++ = *a++ + *b++;
}
/* Implementation for a Pentium processor with MMX technology uses
   an MMX instruction intrinsic to add four elements simultaneously. */
__declspec(cpu_specific(pentium_MMX))
void array_sum2(int *result, int const *a, int *b, size_t len)
{
    m64 *mmx_result = ( m64 *)result;
    __m64 const *mmx_a = (__m64 const *)a;
    __m64 const *mmx_b = (__m64 const *)b;
    for (; len > 3; len -= 4)
        *mmx_result++ = _mm_add_pi16(*mmx_a++, *mmx_b++);
    /* The following code, which takes care of excess elements, is not
       needed if the array sizes passed are known to be multiples of four. */
    result = (unsigned short *)mmx_result;
    a = (unsigned short const *)mmx_a;
    b = (unsigned short const *)mmx_b;
    for (; len > 0; len--)
        *result++ = *a++ + *b++;
}
__declspec(cpu_dispatch(pentium, pentium_MMX))
void array_sum3(int *result, int const *a, int *b, size_t len)
{

```

```
/* Empty function body informs the compiler to generate the
   CPU-dispatch function listed in the cpu dispatch clause. */
}
```

Use the following guidelines to implement processor dispatch support:

- The stub for `cpu_dispatch` must have a `cpuid` defined in `cpu_specific` elsewhere if the `cpu_dispatch` stub for a function `f` contains the `cpuid p`, then a `cpu_specific` definition of `f` with `cpuid p` must appear somewhere in the program; otherwise, an unresolved external error is reported.
  - A `cpu_specific` function definition need not appear in the same translation unit as the corresponding `cpu_dispatch` stub, unless the `cpu_specific` function is declared `static`. The `inline` attribute is disabled for all `cpu_specific` and `cpu_dispatch` functions.
- You must have a stub for `cpu_specific` function if a function `f` is defined as `__declspec(cpu_specific(p))`, then a `cpu_dispatch` stub must also appear for `f` within the program, and `p` must be in the `cpuid-list` of that stub; otherwise, that `cpu_specific` definition cannot be called nor generate an error condition.
- This overrides command line settings when a `cpu_dispatch` stub is compiled, its body is replaced with code that determines the processor on which the program is running, then dispatches the best `cpu_specific` implementation available as defined by the `cpuid-list`.
  - The `cpu_specific` function optimizes to the specified Intel processor regardless of command-line option settings.

## Symbol Visibility Attribute Options (Linux\* and Mac OS\*)

Applications that do not require symbol preemption or position-independent code can obtain a performance benefit by taking advantage of the generic ABI visibility attributes.

## Global Symbols and Visibility Attributes

A global symbol is a symbol that is visible outside the compilation unit in which it is declared (compilation unit is a single-source file with the associated include files). Each global symbol definition or reference in a compilation unit has a visibility attribute that controls how it may be referenced from outside the component in which it is defined.

The values for visibility are defined and described in the following topic:

- `-fvisibility` compiler option

### Note

Visibility applies to both references and definitions. A symbol reference's visibility attribute is an assertion that the corresponding definition will have that visibility.



## Symbol Preemption and Optimization

Sometimes programmers need to use some of the functions or data items from a shareable object, but at the same time, they need to replace other items with definitions of their own. For example, an application may need to use the standard run-time library shareable object, `libc.so`, but to use its own definitions of the heap management routines `malloc` and `free`.

### Note

In this case it is important that calls to `malloc` and `free` within `libc.so` use the user's definition of the routines and not the definitions in `libc.so`. The user's definition should then override, or *preempt*, the definition within the shareable object.

This functionality of redefining the items in shareable objects is called symbol preemption. When the run-time loader loads a component, all symbols within the component that have default visibility are subject to preemption by symbols of the same name in components that are already loaded. Note that since the main program image is always loaded first, none of the symbols it defines will be preempted (redefined).

The possibility of symbol preemption inhibits many valuable compiler optimizations because symbols with default visibility are not bound to a memory address until run-time. For example, calls to a routine with default visibility cannot be inlined because the routine might be preempted if the compilation unit is linked into a shareable object. A preemptable data symbol cannot be accessed using GP-relative addressing because the name may be bound to a symbol in a different component; and the GP-relative address is not known at compile time.

Symbol preemption is a rarely used feature and has negative consequences for compiler optimization. For this reason, by default the compiler treats all global symbol definitions as non-preemptable (protected visibility). Global references to symbols defined in another compilation unit are assumed by default to be preemptable (default visibility). In those rare cases where all global definitions as well as references need to be preemptable, you can override this default.

## Specifying Symbol Visibility Explicitly

The Intel® compiler has visibility attribute options that provide command-line control of the visibility attributes in addition to a source syntax to set the complete range of these attributes.

The options ensure immediate access to the feature without depending on header file modifications. The visibility options cause all global symbols to get the visibility specified by the option. There are two variety of options to specify symbol visibility explicitly.

**Example**

```
-fvisibility=keyword  
-fvisibility-keyword=file
```

The first form specifies the default visibility for global symbols. The second form specifies the visibility for symbols that are in a file (this form overrides the first form).

**Specifying Visibility without the Symbol File**

This option sets the visibility for symbols not specified in a visibility list file and that do not have `VISIBILITY` attribute in their declaration. If no symbol file option is specified, all symbols will get the specified attribute. Command line example:

**Example**

```
icc -fvisibility=protected a.c
```

You can set the default visibility for symbols using one of the following command line options:

**Examples**

```
-fvisibility=extern  
-fvisibility=default  
-fvisibility=protected  
-fvisibility=hidden  
-fvisibility=internal
```

## Interprocedural Optimizations Overview

Use `-ip` (Linux\*) or `/Qip` (Windows\*) and `-ipo` (Linux) or `/Qipo` (Windows) to enable Interprocedural Optimizations (IPO). IPO allows the compiler to analyze your code to determine where you can benefit from the optimizations listed in the following tables.

### IA-32 and Itanium®-based applications

Optimization	Affected Aspect of Program
Inline function expansion	Calls, jumps, branches and loops
Interprocedural constant propagation	Arguments, global variables and return values
Monitoring module-level static variables	Further optimizations and loop invariant code
Dead code elimination	Code size
Propagation of function characteristics	Call deletion and call movement
Multi-file optimization	Same aspects as <code>-ip</code> (Linux) or <code>/Qip</code> (Windows) but across multiple files

### IA-32 applications only

Optimization	Affected Aspect of Program
Passing arguments in registers	Calls and register usage
Loop-invariant code motion	Further optimizations and loop invariant code

Inline function expansion is one of the main optimizations performed during Interprocedural Optimization. For function calls that the compiler determines are frequently executed, the compiler might decide to replace the instructions of the call with code for the function itself.

With `-ip` (Linux and Mac OS\*) or `/Qip` (Windows), the compiler performs inline function expansion for calls to procedures defined within the current source file. However, when you use `-ipo` (Linux) or `/Qipo` (Windows) to specify multifile IPO, the compiler performs inline function expansion for calls to procedures defined in separate files.

#### Caution

The `-ip` and `-ipo` (Linux) or `/Qip` and `/Qipo` (Windows) options can in some cases significantly increase compile time and code size.

## **-auto-ilp32 (Linux\*) or /Qauto-ilp32 (Windows\*) for Intel® EM64T and Itanium®-based Systems**

On Itanium®-based systems, the `-auto-ilp32` (Linux) or `/Qauto-ilp32` (Windows) option specifies interprocedural analysis over the whole program. This optimization allows the compiler to use 32-bit pointers whenever possible as long as the application does not exceed a 32-bit address space. Using the `-auto-ilp32` (Linux) or `/Qauto-ilp32` (Windows) option on programs that exceed 32-bit address space may cause unpredictable results during program execution.

Because this optimization requires interprocedural analysis over the whole program, you must use this option with the `-ipo` (Linux) or `/Qipo` (Windows) option.

On Intel® EM64T systems, this option has no effect unless `-xP` or `-axP` (Linux) or `/QxP` or `/QaxP` (Windows) is also specified.

### **Note**

Mac OS\*: This option is not supported.

## **IPO Compilation Model**

For Intel® compilers Interprocedural Optimization (IPO) generally refers to multi-file IPO.

IPO benefits application performance primarily because it enables inlining. For information on inlining and the minimum inlining criteria, see [Criteria for Inline Function Expansion and Controlling Inline Expansion of User Functions](#).

Inlining and other optimizations are improved by profile information. For a description of how to use IPO with profile information for further optimization, see [Example of Profile-Guided Optimization](#).

When you use the `-ipo` (Linux\*) or `/Qipo` (Windows\*) option, the compiler collects information from individual program modules of a program. Using this information, the compiler performs optimizations across modules. The `-ipo` (Linux) or `/Qipo` (Windows) option is applied to both the compilation and the linkage phases.

### **Compilation Phase**

As each source file is compiled with IPO, the compiler stores an intermediate representation (IR) of the source code in the object file, which includes summary information used for optimization.

By default, the compiler produces mock object files during the compilation phase of IPO. The mock object files contain the IR, instead of object code, so generating mock files instead of real object files reduces the time spent in the IPO compilation phase.

Each mock object file contains the IR for its corresponding source file but does not contain real code or data. These mock objects must be linked with the Intel® compiler or by using the Intel linkers: `xild` (Linux) or `xilink` (Windows). See [Creating a Multifile IPO Executable](#).

### Caution

Failure to link the mock objects with the Intel linkers will result in linkage errors.

## Linkage Phase

When you link with the `-ipo` (Linux) or `/Qipo` (Windows) option the compiler is invoked a final time. The compiler performs IPO across all object files that have an IR equivalent.

The compiler first analyzes the summary information and then finishes compiling the pieces of the application for which it has IR. Having global information about the application while it is compiling individual pieces can improve optimization quality.

The `-ipo` (Linux) or `/Qipo` (Windows) option enables the driver and compiler to attempt detecting a whole program automatically. If a whole program is detected, the interprocedural constant propagation, stack frame alignment, data layout and padding of common blocks perform more efficiently, while more dead functions get deleted. This option is safe.

## Understanding IPO-Related Performance Issues

There are some general optimization guidelines for IPO that you should keep in mind:

- Applications where the compiler does not have sufficient Intermediate Language (IL) coverage to do Whole Program analysis may not perform as well as those where IL information is complete.
- Large IPO compilations might trigger internal limits of other compiler optimization phases.
- The combination of IPO and PGO can be key for C++ applications. The `-O2`, `-ipo`, and `-prof-use` (Linux\*) or `/O2`, `/Qipo`, `-/Qprof-use` (Windows\*) options can result in significant performance gains.
- IPO benefits C more than C++, since C++ compilations include intra-file inlining by default.

In addition to the general guidelines, there are some practices to avoid while using IPO. The following list summarizes the activities to avoid:

- Do not do the link phase of an IPO compilation using IL files produced by a different compiler.
- Do not include major optimizations on the compile line and not include them on the link line.
- Do not link IL files with the `-prof-use` (Linux\*) or `/Qprof-use` (Windows\*) option unless the IL files were generated with the `-prof-use` (Linux) or `/Qprof-use` (Windows) compile.

## Improving IPO Performance

There are two key aspects to achieving improved performance with IPO:

1. Manipulating inlining heuristics
2. Applicability of whole program analysis

Whole Program Analysis (WPS), when it can be done, enables many IP optimizations. It is sometimes misunderstood. During the WPS process, the compiler reads all .il, .obj, and .lib files to determine if all references are resolved and whether or not a given symbol is defined in an .il file.

Symbols that are included in an .il file for both data and functions are candidates for Whole Program manipulation. Currently, .il information is generated for files compiled with IPO. The compiler embeds .il information inside the objects so that IPO information can be included inside libraries, DLLs, and shared objects.

## A Key Relationship for Application Performance

A variable or function that is defined in a module with IL and is never referenced in a module without IL is subject to whole program optimizations. Other variables (and functions) are not subject to these optimizations. A number of important IPO optimizations can be controlled with internal options and IP optimization masks.

## Command Line for Creating an IPO Executable

The command line options to enable IPO for compilations targeted for IA-32, Intel® EM64T, and Itanium® architectures are identical.

To produce mock object files containing IR, compile your source files with `-ipo` (Linux\*) or `/Qipo` (Windows\*) as demonstrated below:

Platform	Example Command
Linux*	<code>icpc -ipo -c a.cpp b.cpp c.cpp</code>
Windows*	<code>icl /Qipo /c a.cpp b.cpp c.cpp</code>

The output of the above example command differs according to platform:

- Linux: The commands produce `a.o`, `b.o`, and `c.o` object files.
- Windows: The commands produce `a.obj`, `b.obj`, and `c.obj` object files.

Use `-c` (Linux) or `/c` (Windows) to stop compilation after generating `.o` (Linux) or `.obj` (Windows) files. The output files contain Intel® compiler intermediate representation (IR) corresponding to the compiled source files.

Optimize interprocedurally by linking with the Intel® compiler or with the Intel linkers: `xild` (Linux) or `xilink` (Windows). The following examples produce an executable named `app`:

Platform	Example Command
Linux	<code>icpc -oapp a.o b.o c.o</code>
Windows	<code>icl /Feapp a.obj b.obj c.obj</code>

This command invokes the compiler on the objects containing IR and creates a new list of objects to be linked. Alternately, you can use the `xild` (Linux) or `xilink` (Windows) tool instead of `icpc` (Linux) or `icl` (Windows) with the appropriate linker options.

The separate commands demonstrated above can be combined into a single command, as shown in the following examples:

Platform	Example Command
Linux	<code>icpc -ipo -oapp a.f b.f c.f</code>
Windows	<code>icl /Qipo /Feapp a.cpp b.cpp c.cpp</code>

The `icl/icpc` command, shown in the examples above, calls `gcc ld` (Linux only) or Microsoft\* `link.exe` (Windows only) to link the specified object files and produce the executable application, which is specified by the `-o` (Linux) or `/Fe` (Windows) option. Multifile IPO is applied only to the source files that have an IR; otherwise, the object file passes to link stage.

## Generating Multiple IPO Object Files

In most cases, IPO generates a single object file for the link-time compilation. This behavior is not optimal for very large applications, perhaps even making it impossible to use `-ipo` (Linux\*) or `/Qipo` (Windows\*) on the application. The compiler provides two ways to avoid this problem. The first way is a size-based heuristic, which automatically causes the compiler to generate multiple object files for large link-time compilations.

The second way is using one of the following options to instruct the compiler to perform multi-object IPO:

Windows*	Linux*	Description
<code>/QipoN</code>	<code>-ipoN</code>	For this option, <i>N</i> indicates the number of object files to generate.  For more information, see the following topic: <ul style="list-style-type: none"> <li><code>-ipo</code> compiler option</li> </ul>
<code>/Qipo-separate</code>	<code>-ipo-separate</code>	Instructs the compiler to generate a separate IPO object file for each source file.

		For more information, see the following topic:
		<ul style="list-style-type: none"> <li>• <code>-ipo-separate</code> compiler option</li> </ul>

These options are alternatives to the `-ipo` (Linux) or `/Qipo` (Windows) option; they indicate an IPO compilation. Explicitly requesting a multi-object IPO compilation turns the size-based heuristic off.

The number of files generated by the link-time compilation is invisible to the user unless either the `-ipo-c` or `-ipo-s` (Linux) or `/Qipo-c` or `/Qipo-s` (Windows) option is used.

In this case the compiler appends a number to the file name. For example, consider this command line:

Platform	Example Command
Linux	<code>icpc -ipo-separate -ipo-c a.o b.o c.o</code>
Windows	<code>icl a.obj b.obj c.obj /Qipo-separate /Qipo-c</code>

On Windows, the example command generates `ipo_out.obj`, `ipo_out1.obj`, `ipo_out2.obj`, and `ipo_out3.obj`. On Linux, the file names will be as listed; however, the file extension will be `.o`.

In the object files, the first object file contains global symbols. The other object files correspond to the source files. This naming convention is also applied to user-specified names.

Platform	Example Command
Linux	<code>icpc -ipo-separate -ipo-c -o appl.o a.o b.o c.o</code>
Windows	<code>icl a.obj b.obj c.obj /Qipo-separate /Qipo-c -o appl.obj</code>

## Capturing Intermediate Outputs of IPO

The `-ipo-c` (Linux\*) or `/Qipo-c` (Windows\*) and `-ipo-s` (Linux) or `/Qipo-s` (Windows) options are useful for analyzing the effects of multifile IPO or when experimenting with multifile IPO between modules that do not make up a complete program.

Use the `-ipo-c` (Linux) or `/Qipo-c` (Windows) option to optimize across files and produce an object file. This option performs optimizations as described for `ipo` but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.s` (Linux) or `ipo_out.obj` (Windows). You can use the `-o` (Linux) or `/Fe` (Windows) option to specify a different name. For example:

Platform	Example Command
Linux	<code>icpc -ipo-c -ofilename a.cpp b.cpp c.cpp</code>
Windows	<code>icl /Qipo-c /Fefilename a.cpp b.cpp c.cpp</code>



Use the `-ipo-S` (Linux) or `/Qipo-S` (Windows) option to optimize across files and produce an assembly file. This option performs optimizations as described for `ipo`, but stops prior to the final link stage, leaving an optimized assembly file. The default name for this file is `ipo_out.s` (Linux) or `ipo_out.asm` (Windows).

You can use the `-o` (Linux) or `/Fe` (Windows) option to specify a different name. For example:

Platform	Example Command
Linux	<code>icpc -ipo-S -ofilename a.cpp b.cpp c.cpp</code>
Windows	<code>icl /Qipo-S /Fe file a.cpp b.cpp c.cpp</code>

These options generate multiple outputs if multi-object IPO is being used. The name of the first file is taken from the value of the `-o` (Linux) or `/Fe` (Windows) option. The name of subsequent files is derived from this file by appending a numeric value to the file name. For example, if the first object file is named `foo.o` (Linux) or `foo.obj` (Windows), the second object file will be named `foo1.o` (Linux) or `foo1.obj` (Windows).

The compiler generates a message indicating the name of each object or assembly file it is generating. These files can be added to the real link step to build the final application.

For more information on inlining and the minimum inlining criteria, see [Criteria for Inline Function Expansion and Controlling Inline Expansion of User Functions](#).

## Creating a Multifile IPO Executable

This topic describes how to use the Intel® linker, `xild` (Linux\*) or `xilink` (Windows\*), instead of the method specified in [Command Line for Creating IPO Executable](#).

The Intel® linker behaves differently on different platforms. The following table summarizes the primary differences:

Linux* Behavior Summary
Invokes the compiler to perform IPO if objects containing <code>IR</code> are found. Invokes <code>GCC ld</code> to link the application.
The command-line syntax for <code>xild</code> is the same as that of the GCC linker:
<code>xild [&lt;options&gt;] &lt;LINK_commandline&gt;</code>
where:
<ul style="list-style-type: none"> <li><code>&lt;options&gt;</code> (optional) may include any GCC linker options or options supported only by <code>xild</code>.</li> <li><code>&lt;LINK_commandline&gt;</code> is the linker command line containing a set of valid</li> </ul>

arguments to the `ld`.

To create `app` using IPO, use the option `-ofile` as shown in the following example:

```
xild -oapp a.o b.o c.o
```

The linker calls the compiler to perform IPO for objects containing `IR` and creates a new list of object(s) to be linked. The linker then calls `ld` to link the object files that are specified in the new list and produce the application with the name specified by the `-o` option. The linker supports the `-ipo`, `-ipoN`, and `-ipo-separate` options.

## Windows\* Behavior Summary

Invokes the Intel compiler to perform multifile IPO if objects containing `IR` are found.  
Invokes Microsoft\* `link.exe` to link the application.

The command-line syntax for the Intel® linker is the same as that of the Microsoft linker:

```
xilink [<options>] <LINK_commandline>
```

where:

- [`<options>`] (optional) may include any Microsoft linker options or options supported only by `xilink.exe`.
- `<LINK_commandline>` is the linker command line containing a set of valid arguments to the Microsoft linker.

To place the multifile IPO executable in `ipo_file.exe`, use the linker option `/out:file`; for example:

```
xilink /out:ipo_file.exe a.obj b.obj c.obj
```

The linker calls the compiler to perform IPO for objects containing `IR` and creates a new list of object(s) to be linked. The linker calls Microsoft `link.exe` to link the object files that are specified in the new list and produce the application with the name specified by the `/out:file` option.

## Usage Rules

You must use the Intel® linker to link your application if the following conditions apply:

- Your source files were compiled with multifile IPO enabled. Multifile IPO is enabled by specifying the `-ipo` (Linux) or `/Qipo` (Windows) command-line option
- You normally would invoke either the GCC linker (`ld`) or the Microsoft linker (`link.exe`) to link your application.

## The Intel® linker Options

Use these additional options supported by the Intel® linker to examine the results of multifile IPO:

Windows	Linux	Option Description
/qipo_fa[{file dir/}]	-qipo_fa[file.s]	<p>Produces assembly listing for the multifile IPO compilation. You may specify an optional name for the listing file, or a directory (with the backslash) in which to place the file.</p> <p>The default listing name is depends on the platform:</p> <ul style="list-style-type: none"> <li>Linux: ipo_out.s</li> <li>Windows: ipo_out.asm</li> </ul>
/qipo_fo[{file dir/}]	-qipo_fo[file.o]	<p>Produces object file for the multifile IPO compilation. You may specify an optional name for the object file, or a directory (with the backslash) in which to place the file.</p> <p>The default object file name is depends on the platform:</p> <ul style="list-style-type: none"> <li>Linux: ipo_out.o</li> <li>Windows: ipo_out.obj</li> </ul>
/qipo_fas	No equivalent	Add source lines to assembly listing.
/qipo_fac	-ipo-fcode-asm	Adds code bytes to the assembly listing.
/qipo_facs	No equivalent	Add code bytes and source lines to assembly listing.
No equivalent	-ipo-fsource-asm	Adds high-level source code to the assembly listing.
/qv	No equivalent	Display version information.
No equivalent	-ipo-fverbose-asm, -ipo-fnoverbose-asm	Enables and disables, respectively, inserting comments containing version and options used in the assembly listing.

If the Intel® linker invocation leads to an IPO multi-object compilation (either because the application is big, or because the user explicitly asked for multiple objects), the first .s (Linux) or .asm (Windows) file takes its name from the -qipo\_fa (Linux) or /qipo\_fa (Windows) option.

The compiler derives the names of subsequent .s (Linux) or .asm (Windows) files by appending a number to the name, for example, foo.asm and foo1.asm for ipo\_fafoo.asm. The same is true for the -qipo\_fo (Linux) or /qipo\_fo (Windows) option.

## Understanding Code Layout and Multi-Object IPO

One of the optimizations performed during an IPO compilation is code layout.

IPO analysis determines a layout order for all of the routines for which it has IR. If a single object is being generated, the compiler generates the layout simply by compiling the routines in the desired order.

For a multi-object IPO compilation, the compiler must tell the linker about the desired order. The compiler first puts each routine in a named text section that varies depending on the platform.

### Linux\*:

- The first routine is placed in `.text00001`, the second is placed in `.text00002`, and so on. It then generates a linker script that tells the linker to first link contributions from `.text00001`, then `.text00002`.

### Windows\*:

- The first routine is placed in `.text$00001`, the second is placed in `.text$00002`, and so on. The Windows linker (`link`) automatically collates these sections in the desired order.

However, the linker script must be taken into account if you use either `-ipo-c` or `-ipo-s` (Linux\*) or `/Qipo-c` or `/Qipo-s` (Windows\*).

With these options, the IPO compilation and actual linking are done by different invocations of the compiler. When this occurs, the compiler issues a message indicating that it is generating an explicit linker script, `ipo_layout.script`.

When `ipo_layout.script` is generated, the typical response is to modify your link command to use this script:

#### Example

```
--script=ipo_layout.script
```

If your application already requires a custom linker script, you can place the necessary contents of `ipo_layout.script` in your script.

The layout-specific content of `ipo_layout.script` is at the beginning of the description of the `.text` section. For example, to describe the layout order for 12 routines:

#### Example output

```
.text      :
{
*(.text00001) *(.text00002) *(.text00003) *(.text00004) *(.text00005)
*(.text00006) *(.text00007) *(.text00008) *(.text00009) *(.text00010)
```

```
*(.text00011) *(.text00012)
...
```

For applications that already require a linker script, you can add this section of the `.text` section description to the customized linker script. If you add these lines to your linker script, it is desirable to add additional entries to account for future development. The addition is harmless, since the “\*(...)” syntax makes these contributions optional.

If you choose to not use the linker script your application will still build, but the layout order will be random. This may have an adverse affect on application performance, particularly for large applications.

## Implementing IL Files with Version Numbers

An IPO compilation consists of two parts: compile phase and link phase.

In the compile phase, the compiler produces an Intermediate Language (IL) version of the users' code. In the link phase, the compiler reads the IL and completes the compilation, producing a real object file or executable.

Generally, different compiler versions produce unique IL based on different definitions; therefore, the intermediate language files from different compilations can be incompatible.

A compiler assigns a unique version number with each IL definition for the compiler. If a compiler attempts to read IL in a file with a version number other than its own, the compilation will proceed; however, the compiler discards the IL and does not use it during compilation. The compiler then issues a warning message that an incompatible IL was detected and discarded.

## IL in Objects and Libraries: More Optimizations

The IL produced by the Intel® compiler is stored in a special section of the object file. The IL stored in the object file is then placed in the library. If this library is used in an IPO compilation invoked with the same compiler as produced the IL for the library, the compiler can extract the IL from the library and use it to optimize the program. For example, it is possible to inline functions defined in the libraries into the users' source code.

### Inline Expansion of Functions

## Criteria for Inline Function Expansion

While function inlining (function expansion) increases execution time by removing the runtime overhead of function calls, inlining can, in many cases, increase code size, code complexity, and compile times.

In the Intel compiler, inline function expansion typically favors relatively small user functions over functions that are relatively large; however, the compiler can inline functions only if the conditions in the three main components match the conditions listed below:

- **Call-site:** The call-site is the site of the call to the function that might be inlined. For each call site, the following conditions must exist:
  - The number of actual arguments must match the number of formal arguments of the callee.
  - The number of return values must match the number of return values of the callee.
  - The data types of the actual and formal arguments must be compatible.
  - Caller and callee must be written in the same source language. No multilingual inlining is permitted.
- **Caller:** The caller is the function that contains the call-site. The Caller must meet the following conditions:
  - Is approximately the right size; at most, 2000 intermediate statements will be inlined into the caller from all the call-sites being inlined into the caller.
  - Is static; the function must be called if it is declared as static; otherwise, it will be deleted.
- **Callee:** The callee is the function being called that might be inlined. The Callee, or target function, must meet the following conditions:
  - Does not have variable argument list.
  - Is not considered unsafe for other reasons.
  - Is not considered infrequent due to the name. For example, routines which contain the following substrings in their names are not inlined: abort, alloca, denied, err, exit, fail, fatal, fault, halt, init, interrupt, invalid, quit, rare, stop, timeout, trace, trap, and warn.

## Selecting Routines for Inlining

If the minimum criteria identified are met, the compiler picks the routines whose inline expansions will provide the greatest benefit to program performance. This is done using the default heuristics. The inlining heuristics used by the compiler differ based on whether or not you use Profile-Guided Optimizations (PGO): `-prof-use` (Linux\*) or `/Qprof-use` (Windows\*).

When you use PGO with `-ip` or `-ipo` (Linux) or `/Qip` or `/Qipo` (Windows), the compiler uses the following heuristics:

- The default heuristic focuses on the most frequently executed call sites, based on the profile information gathered for the program.
- The default inline heuristic will stop inlining when direct recursion is detected.
- The default heuristic always inlines very small functions that meet the minimum inline criteria.
- By default, the compiler does not inline functions with more than 230 intermediate statements. You can change this value by specifying the following:

Platform	Command
Linux	<code>-Qoption,c,-ip_ninl_max_stats=n</code>
Windows	<code>/Qoption,c,-ip_ninl_max_stats=n</code>

where  $n$  is a new value.

### Note

There is a higher limit for functions declared by the user as `inline` or `__inline`.

See [Using `Qoption` Specifiers and Profile-Guided Optimization Overview](#).

When you do not use PGO with `-ip` or `-ipo` (Linux) or `/Qip` or `/Qipo` (Windows), the compiler uses less aggressive inlining heuristics: it inlines a function if the inline expansion does not increase the size of the final program.

The compiler offers some alternatives to using the `Qoption` specifiers; see [Developer Directed Inline Expansion of User Functions](#) for a summary.

## Inlining (Linux\*)

If you do not use profile-guided optimizations with `-ip` or `-ipo`, the compiler uses less aggressive inlining heuristics:

- Inline a function if the inline expansion will not increase the size of the final program.
- Inline a function if it is declared with the `inline` or `__inline` keywords.

## PGO (Windows\*)

The compiler uses characteristics of the source code to estimate which function calls are executed most frequently. It applies these estimates to the PGO-based heuristics described previously. The estimation of frequency, based on static characteristics of the source, is not always accurate. Hence, use of `/Qip` and `/Qipo` without PGO can result in longer compilation times and even slower application execution.

## Using `Qoption` Specifiers

Use the `-Qoption` (Linux\*) or `/Qoption` (Windows\*) option with the applicable keywords to select particular inline expansions and loop optimizations. The option must be entered with a `-ip` or `-ipo` (Linux) or `/Qip` or `/Qipo` (Windows) specification, as shown in the following example command:

Platform	Example Command
Linux	<code>-ip or -ipo [-Qoption,tool,opts]</code>
Windows	<code>/Qip or /Qipo [/Qoption,tool,opts]</code>

In the command syntax *tool* is C++ (c) and *opts* are `-Qoption` (Linux) or `/Qoption` (Windows) option specifiers.

See Specifying Alternative Tools and Locations in Building Applications for the supported specifiers.

For more detailed information about the supported specifiers and arguments, see the following topic:

- `-Qoption` compiler option

If you specify `-ip` or `-ipo` (Linux) or `/Qip` or `/Qipo` (Windows) without any `-Qoption` (Linux) or `/Qoption` (Windows) qualification, the compiler performs the following actions:

- Expands functions in line
- Propagates constant arguments
- Passes arguments in registers
- Monitors module-level static variables

Refer to Criteria for Inline Function Expansion to see how these specifiers may affect the inlining heuristics of the compiler.

## Compiler Directed Inline Expansion of User Functions

The compiler provides options for inlining user functions based on specific criteria without any other direction. See Criteria for Inline Function Expansion for more information.

The following options are useful in situations where an application can benefit from user function inlining but does not need specific direction about inlining limits. Except where noted, these options are supported on IA-32, Intel® EM64T, and Intel® Itanium® architectures.

Windows*	Linux*	Effect
<code>/Ob</code>	<code>-Ob</code>	Specifies the level of inline function expansion.  For more information, see the following topic: <ul style="list-style-type: none"> <li>• <code>-Qb</code> compiler option</li> </ul>
<code>/Qip-no-inlining</code>	<code>-ip-no-inlining</code>	Useful if <code>-ip</code> (Linux*) or <code>/Qip</code> (Windows*) or <code>-ipo</code> (Linux) or <code>/Qipo</code> (Windows) is also specified. In such cases, this option disables inlining that would result from the <code>-ip</code> (Linux) or <code>/Qip</code> (Windows) or <code>-Ob2</code> (Linux) or <code>/Ob2</code> (Windows) IPO, but has no effect on other interprocedural optimizations.



		<p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-ip-no-inlining</code> compiler option</li> </ul>
<code>/Qip-no-pinlining</code>	<code>-ip-no-pinlining</code>	<p>Disables partial inlining; can be used if <code>-ip</code> (Linux) or <code>/Qip</code> (Windows) or <code>-ipo</code> (Linux) or <code>/Qipo</code> (Windows) is also specified.</p> <p><b>Note</b></p> <p>Itanium-based systems: This option is not supported.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-ip-no-pinlining</code> compiler option</li> </ul>
<code>/Qinline-debug-info</code>	<code>-inline-debug-info</code>	<p>Keeps source information for inlined functions. The additional source code can be used by the Intel® Debugger to resolve issues.</p> <p><b>Note</b></p> <p>Mac OS*: This option is not supported.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-inline-debug-info</code> compiler option</li> </ul>

## Developer Directed Inline Expansion of User Functions

In addition to the options that support compiler directed inline expansion of user functions, the compiler also provides compiler options that allow you to more precisely direct when and if inline function expansion occurs.

The compiler measures the relative size of a routine in an abstract value of intermediate language units, which is approximately equivalent to the number of instructions that will be generated. The compiler uses the intermediate language unit estimates to classify routines and functions as relatively small, medium, or large functions. The compiler then uses the estimates to determine when to inline a function; if the minimum criteria for inlining is met and all other things are equal, the compiler has an affinity for inlining relatively small functions and not inlining relative large functions.

The following developer directed inlining options provide the ability to change the boundaries used by the inlining optimizer to distinguish between small and large

functions. These options are supported on IA-32, Intel® EM64T, and Intel® Itanium® architectures.

Windows*	Linux*	Effect
/Qinline-forceinline	-inline-forceinline	<p>Instructs the compiler to inline marked functions, if the inlining can be done safely. Typically, the compiler targets functions that have been marked for inlining based on the presence of keywords, like <code>__forceinline</code>, in the source code; however, the such directives in the source code are treated as suggestions for inlining. The option instructs the compiler to view the inlining suggestion as mandatory and inline the marked function if it can be done legally.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-inline-forceinline</code> compiler option</li> </ul>
/Qinline-min-size	-inline-min-size	<p>Redefines the maximum size of small routines; routines that are equal to or smaller than the value specified are more likely to be inlined.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-inline-min-size</code> compiler option</li> </ul>
/Qinline-max-size	-inline-max-size	<p>Redefines the minimum size of large routines; routines that are equal to or larger than the value specified are less likely to be inlined.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-inline-max-size</code> compiler option</li> </ul>
/Qinline-max-total-size	-inline-max-total-size	<p>Limits the expanded size of inlined functions.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-inline-max-total-size</code> compiler option</li> </ul>
/Qinline-max-per-routine	-inline-max-per-routine	<p>Defines the number of times a called function can be inlined in each routine.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-inline-max-per-routine</code> compiler option</li> </ul>
/Qinline-max-per-	-inline-max-per-	Defines the number of times a called functions can be

compile	compile	<p>inlined in each compilation unit.</p> <p>The compilation unit limit depends on the whether or not you specify the <code>-ipo</code> (Linux) or <code>/Qipo</code> (Windows) option. If you enable IPO, all source files that are part of the compilation are considered one compilation unit. For compilations not involving IPO each source file is considered an individual compilation unit.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-inline-max-per-compile</code> compiler option</li> </ul>
---------	---------	---

## Inline Expansion of Library Functions

By default, the compiler automatically inlines (expands) a number of standard and math library functions at the point of the call to that function, which usually results in faster computation; however, the inlined library functions do not set the `errno` variable when being expanded inline.

In code that relies upon the setting of the `errno` variable, you should use the `-nolib-inline` (Linux\*) or `/Oi-` (Windows\*) option.

If one of your functions has the same name as one of the compiler-supplied library functions, then when this function is called, the compiler assumes that the call is to the library function and replaces the call with an inlined version of the library function. Therefore, if the program defines a function with the same name as one of the known library routines, you must use the `-nolib-inline` (Linux) or `/Oi-` (Windows) option to ensure that the user-supplied function is used.

The option disables inlining of all intrinsics.

## Profile-Guided Optimizations Overview

Profile-Guided Optimization (PGO) provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application. For example, using PGO can often enable the compiler to make better decisions about function inlining, which increases the effectiveness of interprocedural optimizations.

See [Basic PGO Options and Using PGO in a Microsoft Visual C++ .NET\\* Project](#).

## Instrumented Program

The first phase in using PGO is to instrument the program. In this phase, the compiler creates an instrumented program from your source code and special code from the compiler.

Each time this instrumented code is executed, the instrumented program generates a dynamic information file.

When you compile a second time, the dynamic information files are merged into a summary file. Using the profile information in this file, the compiler attempts to optimize the execution of the most heavily traveled paths in the program.

Unlike other optimizations, such as those strictly for size or speed, the results of IPO and PGO vary. This behavior is due to each program having a different profile and different opportunities for optimizations. The guidelines provided help you determine if you can benefit by using IPO and PGO. You need to understand the principles of the optimizations and the unique aspects of your source code.

## Added Performance with PGO

The Intel® compiler PGO feature provides the following benefits:

- Register allocation uses the profile information to optimize the location of spill code.
- For indirect function calls, branch prediction is improved by identifying the most likely targets.  
Both Pentium® 4 and Intel® Xeon® processors have longer pipelines, which improves branch prediction and translates into high performance gains.
- The compiler detects and does not vectorize loops that execute only a small number of iterations, reducing the run time overhead that vectorization might otherwise add.

## Understanding Profile-Guided Optimization

PGO works best for code with many frequently executed branches that are difficult to predict at compile time. An example is the code with intensive error-checking in which the error conditions are false most of the time. The infrequently executed (cold) error-handling code can be placed such that the branch is rarely predicted incorrectly. Minimizing cold code interleaved into the frequently executed (hot) code improves instruction cache behavior.

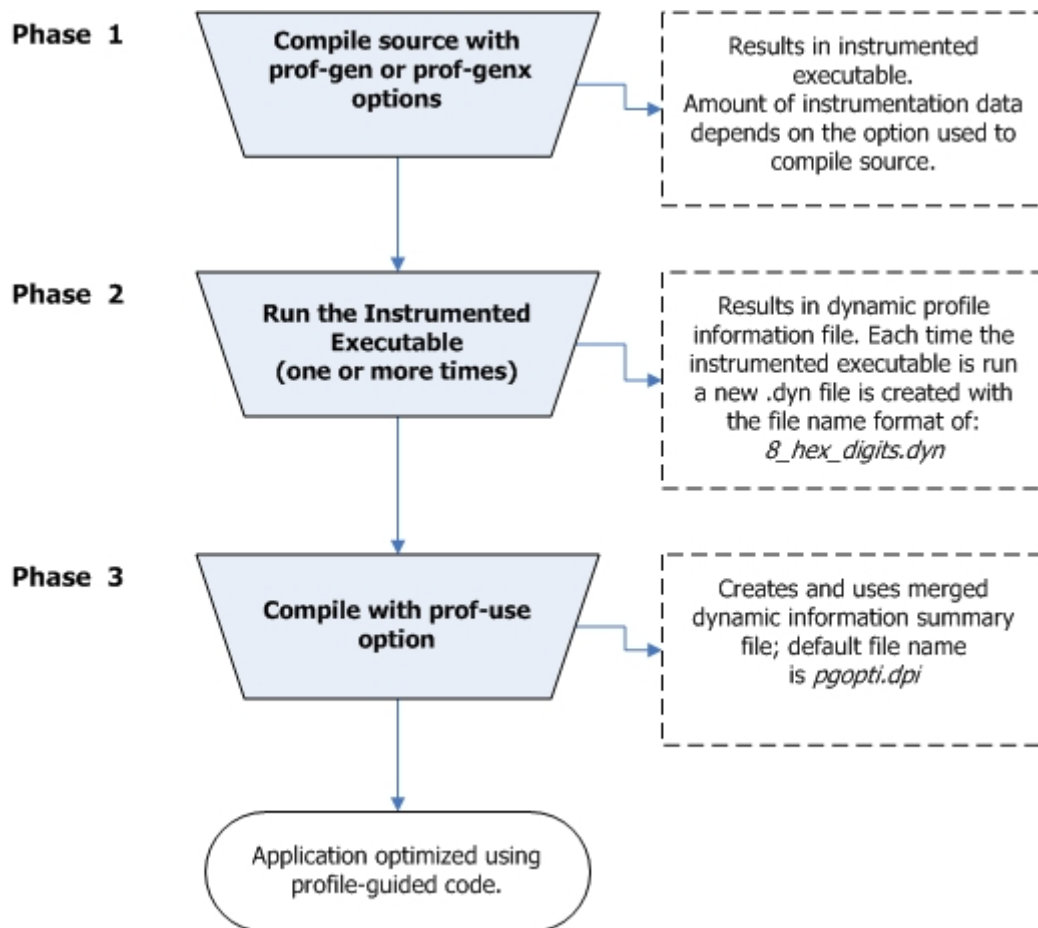
Using PGO often enables the compiler to make better decisions about function inlining, thereby increasing the effectiveness of IPO.

### PGO Phases

PGO requires the following three phases, or steps, and options:

- **Phase 1:** Instrumentation compilation and linking with the `-prof-gen` (Linux\*) or `/Qprof-gen` (Windows\*) option
- **Phase 2:** Instrumented execution by running the executable, which produced the dynamic-information ( `.dyn` ) files
- **Phase 3:** Feedback compilation with the `-prof-use` (Linux) or `/Qprof-use` (Windows) option

The flowchart (below) illustrates this process for IA-32 compilation and Itanium®-based compilation.



See Example of Profile-Guided Optimization for specific details on working with each phase.

A key factor in deciding whether or not to use PGO lies in knowing what sections of your code are the most heavily used. If the data set provided to your program is very consistent and it displays similar behavior on every execution, then PGO can probably help optimize your program execution.

However, different data sets can result in different algorithms being called. This can cause the behavior of your program to vary from one execution to the next. In cases where your code behavior differs greatly between executions, PGO may not provide noticeable benefits.

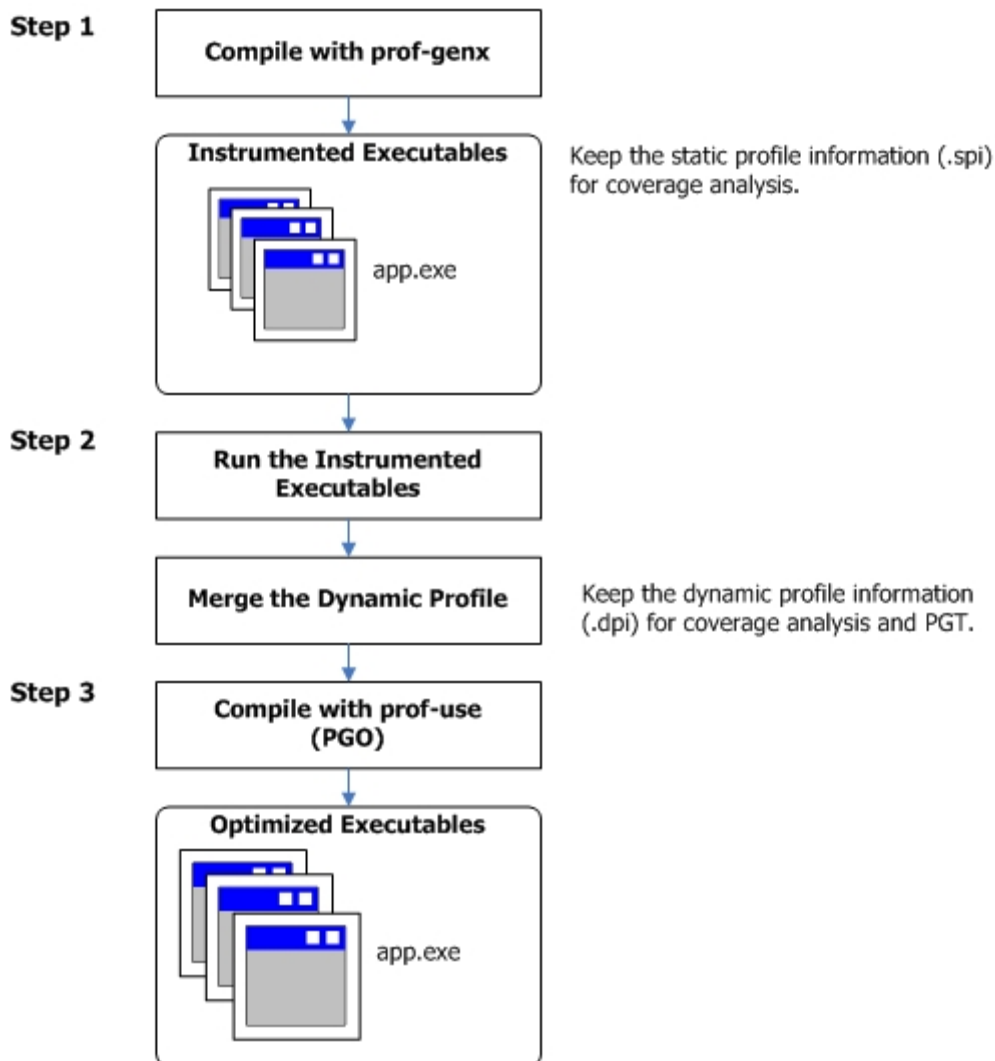
You have to ensure that the benefit of the profile information is worth the effort required to maintain up-to-date profiles.

Linux\* systems: When using `-prof-gen` with the `x` qualifier, extra source position is collected which provides support for tools, such as the Intel® compiler Code-coverage

Tool. Without such tools, `-prof-gen` (Linux) or `/Qprof-gen` (Windows) does not provide better optimization and may slow parallel compile times.

## PGO Usage Model

The following figure illustrates the general PGO process for preparing files for use with specific tools.



## Example of Profile-Guided Optimization

The sections in this topic provide examples of the three basic PGO phases, or steps:

- Instrumentation Compilation and Linking
- Instrumented Execution
- Feedback Compilation

When you use PGO, consider the following guidelines:

- Minimize the changes to your program after instrumented execution and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated.

#### Note

The compiler issues a warning that the dynamic information does not correspond to a modified function.

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.
- Specify the name of the profile summary file using the `-prof-file` (Linux\*) or `/Qprof-file` (Windows\*) option.

## Profile-guided Optimization (PGO) Phases

### Instrumentation Compilation and Linking

Use `-prof-gen` (Linux) or `/Qprof-gen` (Windows) to produce an executable with instrumented information included.

Use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option if the application includes the source files located in multiple directories. `-prof-dir` (Linux) or `/Qprof-dir` (Windows) insures the profile information is generated in one consistent place. The following example commands demonstrate how to combine these options:

Platform	Commands
Linux	<code>icpc -prof-gen -prof-dir /profdata -c a1.cpp a2.cpp a3.cpp</code> <code>icpc a1.o a2.o a3.o</code>
Windows	<code>icl /Qprof-gen /Qprof-dir:\profdata /c a1.cpp a2.cpp a3.cpp</code> <code>icl a1.obj a2.obj a3.obj</code>

In place of the second command, you can use the linker directly to produce the instrumented program.

#### Note

The compiler gathers extra information when you use the `-prof-genx` (Linux) or `/Qprof-genx` (Windows) qualifier; however, the additional static information gathered using the option can be used by specific tools only. See Basic PGO Options.



## Instrumented Execution

Run your instrumented program with a representative set of data to create one or more dynamic information files. The following examples demonstrate the command lines for running the executable generated by the example commands (listed above):

Platform	Command
Linux	<code>./a.out</code>
Windows	<code>a1.exe</code>

Executing the instrumented applications generates dynamic information file that has a unique name and `.dyn` suffix. A new `.dyn` file is created every time you execute the instrumented executable.

The instrumented file helps predict how the program runs with a particular set of data. You can run the program more than once with different input data.

## Feedback Compilation

The final phase compiles and links the sources files using the dynamic information generated in the instrumented execution phase. Compile and link the source files with `-prof-use` (Linux) or `/Qprof-use` (Windows) to use the dynamic information to guide the optimization of your program, according to its profile:

Platform	Examples
Linux	<code>icpc -prof-use -ipo a1.cpp a2.cpp a3.cpp</code>
Windows	<code>icl /Qprof-use /Qipo a1.cpp a2.cpp a3.cpp</code>

In addition to the optimized executable, the compiler produces a `pgopti.dpi` file.

You typically specify the default optimizations, `-O2` (Linux) or `/O2` (Windows), for phase 1, and specify more advanced optimizations, `-ipo` (Linux) or `/Qipo` (Windows), for phase 3. For example, the example shown above used `-O2` (Linux) or `/O2` (Windows) in phase 1 and `-ipo` (Linux) or `/Qipo` (Windows) in phase 3.

### Note

The compiler ignores the `-ipo` (Linux) or `/Qipo` (Windows) option with `-prof-gen` (Linux) or `/Qprof-gen` (Windows).

See Basic PGO Options.

## Basic PGO Options

This topic details the most commonly used options for profile-guided optimization.

The options listed below are used in the phases of the PGO. See Advanced PGO Options for options that extend PGO.

Windows*	Linux*	Effect
/Qprof-gen  /Qprof-genx	-prof-gen  -prof-genx	<p>Instruments the program for profiling to get the execution count of each basic block. The option is used in phase 1 of PGO (generating instrumented code) to instruct the compiler to produce instrumented code in your object files in preparation for instrumented execution. Results in dynamic-information (.dyn) file.</p> <p>With the <code>x</code> qualifier, the option gathers extra source information, which enables using specific tools, like the <code>proforder</code> tool and the Code-coverage Tool.</p> <p>If performing a parallel make, using this option will not affect it.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-prof-gen</code> and <code>-prof-genx</code> compiler options</li> </ul>
/Qprof-use	-prof-use	<p>Instruct the compiler to produce a profile-optimized executable and merges available dynamic-information (.dyn) files into a <code>pgopti.dpi</code> file. This option is used in phase 3 of PGO (generating a profile-optimized executable).</p> <p><b>Note</b></p> <p>The dynamic-information files are produced in phase 2 when you run the instrumented executable.</p> <p>If you perform multiple executions of the instrumented program, this option merges the dynamic-information files again and overwrites the previous <code>pgopti.dpi</code> file.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-prof-use</code> compiler option</li> </ul>
/Qprof-format-32	-prof-format-32	<p><b>Using 32-bit Counters:</b> This option has been deprecated in this release. The Intel® compiler, by default, produces profile data with 64-bit counters to handle large numbers of events</p>

		<p>in the <code>.dyn</code> and <code>.dpi</code> files.</p> <p><b>Note</b></p> <p>Mac OS*: This option is not supported.</p> <p>This option produces 32-bit counters for compatibility with the earlier compiler versions. If the format of the <code>.dyn</code> and <code>.dpi</code> files is incompatible with the format used in the current compilation, the compiler issues the following message:</p> <pre>Error: xxx.dyn has old or incompatible file format - delete file and redo instrumentation compilation/execution.</pre> <p>The 64-bit format for counters and pointers in <code>.dyn</code> and <code>.dpi</code> files eliminate the incompatibilities on various platforms due to different pointer sizes.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-prof-format-32</code> compiler option</li> </ul>
<code>/Qfnsplit-</code>	<code>-fnsplit-</code>	<p>Disables function splitting. Function splitting is enabled by <code>-prof-use</code> (Linux*) or <code>/Qprof-use</code> (Windows*) in phase 3 to improve code locality by splitting routines into different sections: (1) one section to contain the cold or very infrequently executed code, and (2) one section to contain the rest of the code (hot code).</p> <p>You may want to disable function splitting:</p> <ul style="list-style-type: none"> <li>• to get improved debugging capability. In the debug symbol table, it is difficult to represent a split routine, that is, a routine with some of its code in the hot code section and some of its code in the cold code section.</li> <li>• to disable function splitting when the profile data does not represent the actual program behavior, that is, when the routine is actually used frequently rather than infrequently.</li> </ul> <p><b>Note</b></p> <p>Windows* Only: This option behaves differently on IA-32 and Itanium®-based systems.</p> <p>Mac OS: This option is not supported.</p>

		<p><b>IA-32 systems:</b></p> <ul style="list-style-type: none"> <li>The option completely disables function splitting, placing all the code in one section.</li> </ul> <p><b>Itanium®-based systems:</b></p> <ul style="list-style-type: none"> <li>The option disables the splitting within a routine but enables function grouping, an optimization in which entire routines are placed either in the cold code section or the hot code section. Function grouping does not degrade debugging capability.</li> </ul> <p>See an example of using PGO.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li><code>-fnsplit</code> compiler option</li> </ul>
--	--	--

## Advanced PGO Options

The options listed below extend the ability of the basic PGO options and provide more control when using PGO. For information about the critical PGO options, see Basic PGO Options.

Windows*	Linux*	Effect
<code>/Qprof-dir</code>	<code>-prof-dir</code>	<p>Specifies the directory in which dynamic information (<code>.dyn</code>) files are created and stored; otherwise, the <code>.dyn</code> files are placed in the directory where the program is compiled.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li><code>-prof-dir</code> compiler option</li> </ul>
<code>/Qprof-file</code>	<code>-prof-file</code>	<p>Specifies file name for profiling summary file. If this option is not specified the summary information is stored in the default file: <code>pgopti.dpi</code>.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li><code>-prof-file</code> compiler option</li> </ul>
<code>/Qprof-gen-sampling</code>	<code>-prof-gen-sampling</code>	<p>IA-32 Only. Prepares application executables for hardware profiling (sampling) and causes the compiler to generate source code mapping information.</p>

		<p><b>Note</b></p> <p>Mac OS*: This option is not supported.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-prof-gen-sampling</code> compiler option</li> </ul>
<code>/Qssp</code>	<code>-ssp</code>	<p>IA-32 Only. Enables Software-based Speculative Pre-computation (SSP) optimization.</p> <p><b>Note</b></p> <p>Mac OS: This option is not supported.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-ssp</code> compiler option</li> </ul>

## Generating Function Order Lists

A function order list is text that specifies the order in which the linker should link the non-static functions of your program. The list improves the performance of your program by reducing paging and improving code locality. Profile-guided optimizations support the generation of a function order list to be used by the linker. The compiler determines the order using profile information.

To generate a function order list, use the `profmerge` and `proforder` utilities.

## Function Order List Usage Guidelines (Windows\*)

Use the following guidelines to create a function order list:

- The order list only affects the order of non-static functions.
- You must compile with `/Gy` to enable function-level linking. (This option is active if you specify either option `/O1` or `/O2`.)

## Comparison of Function Order Lists and IPO Code Layout

The Intel® compiler provides two methods of optimizing the layout of functions in the executable:

- Using a function order list
- Using the `-ipo` (Linux\*) or `/Qipo` (Windows\*) compiler option

Each method has its advantages. A function order list, created with `proforder`, lets you optimize the layout of non-static functions: that is, external and library functions whose names are exposed to the linker.

The linker cannot directly affect the layout order for static functions because the names of these functions are not available in the object files.

The compiler cannot affect the layout order for functions it does not compile, such as library functions. The function layout optimization is performed automatically when IPO is active.

Alternately, using the `-ipo` (Linux\*) or `/Qipo` (Windows\*) option allows you to optimize the layout of all `static` or `extern` functions compiled with the Intel® C++ Compiler. The compiler cannot affect the layout order for functions it does not compile, such as library functions. The function layout optimization is performed automatically when IPO is active.

### Function Order List Effects

Function Type	Code Layout with <code>-ipo</code> ( <code>/Qipo</code> )	Function Ordering with <code>proforder</code>
Static	X	No effect
Extern	X	X
Library	No effect	X

## Example of Generating a Function Order List

This section provide a general example of the process for generating a function order list. Assume you have a C++ program that consists of the following files:

- `file1.cpp`
- `file2.cpp`

Additionally, assume you have created a directory for the profile data files called `profdata`. You would enter commands similar to the following to generate and use a function order list for your application.

1. Compile your program using the `-prof-genx` (Linux) or `/Qprof-genx` (Windows) option:

Platform	Example commands
Linux	<code>icpc -o myprog -prof-genx -prof-dir ./profdata file1.cpp file2.cpp</code>
Windows	<code>icl /Femyprog /Qprof-genx /Qprof-dir c:\profdata file1.cpp file2.cpp</code>

This step creates an instrumented executable.

2. Run the instrumented program with one or more sets of input data. If you specified a location other than the current directory, change to the directory where the executables are located.

Platform	Example commands
Linux	<code>./myprog</code>
Windows	<code>myprog.exe</code>

The program produces a `.dyn` file each time it is executed. This process may take some time depending on the options specified.

3. Merge the data from instrumented program from one or more runs of the instrumented program using the `profmerge` tool to produce the `pgopti.dpi` file. Use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the location of the `.dyn` files.

Platform	Example commands
Linux	<code>profmerge -prof-dir ./profdat</code>
Windows	<code>profmerge /prof-dir c:\profdata</code>

4. Generate the function order list using the `proforder` utility. (By default, the function order list is produced in the file `proford.txt`.)

Platform	Example commands
Linux	<code>proforder -prof-dir ./profdata -o myprog.txt</code>
Windows	<code>proforder -prof-dir c:\profdata -o myprog.txt</code>

5. Compile the application with the generated profile feedback by specifying the `ORDER` option to the linker. Again, use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the location of the profile files.

Platform	Example commands
Linux	<code>icpc -o myprog -prof-dir ./profdata file1.cpp file2.cpp -Xlinker -ORDER:@myprog.txt</code>
Windows	<code>icl /Femyprog /Qprof-dir c:\profdata file1.cpp file2.cpp /link -ORDER:@myprog.txt</code>

## PGO API Support Overview

The Profile Information Generation Support (Profile IGS) lets you control the generation of profile information during the instrumented execution phase of Profile-guided Optimizations.

A set of functions and an environment variable comprise the Profile IGS. The remaining topics in this section describe the associated functions and environment variables.

The compiler sets a define for `_PGO_INSTRUMENT` when you compile with either `-prof-gen` (Linux\*) or `/Qprof-gen` (Windows\*) or `-prof-genx` (Linux) or `/Qprof-genx` (Windows). Without instrumentation, the Profile IGS functions cannot provide PGO API support.

Normally, profile information is generated by an instrumented application when it terminates by calling the standard `exit()` function.

To ensure that profile information is generated, the functions described in this section may be necessary or useful in the following situations:

- The instrumented application exits using a non-standard exit routine.
- The instrumented application is a non-terminating application: `exit()` is never called.
- The application requires control of when the profile information is generated.

You can use the Profile IGS functions in your application by including a header file at the top of any source file where the functions may be used.

### Example

```
#include <pgouser.h>
```

## The Profile IGS Environment Variable

The environment variable for Profile IGS is `PROF_DUMP_INTERVAL`. This environment variable may be used to initiate Interval Profile Dumping in an instrumented user application. See the recommended usage of `_PGOPTI_Set_Interval_Prof_Dump()` for more information.

## PGO Environment Variables

The environment variables determine the directory in which to store dynamic information files or whether to overwrite `pgopti.dpi`. The environment variables are described in the table below.

Variable	Description
<code>PROF_DIR</code>	Specifies the directory in which dynamic information files are created. This variable applies to all three phases of the profiling process.
<code>PROF_DUMP_INTERVAL</code>	Initiates interval profile dumping in an instrumented user application. This environment variable may be used to initiate Interval Profile Dumping in an instrumented application.  See Interval Profile Dumping for more information.
<code>PROF_NO_CLOBBER</code>	Alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges the data from all dynamic information files and creates a new <code>pgopti.dpi</code>



	<p>file, if the .dyn files are newer than an existing <code>pgopti.dpi</code> file.</p> <p>When this variable is set, the compiler does not overwrite the existing <code>pgopti.dpi</code> file. Instead, the compiler issues a warning and you must remove the <code>pgopti.dpi</code> file if you want to use additional dynamic information files.</p>
--	---

See the documentation for your operating system for instructions on how to specify environment variables and their values.

## Dumping Profile Information

The `_PGOPTI_Prof_Dump_All()` function dumps the profile information collected by the instrumented application. The prototype of the function call is listed below.

Syntax
<code>void _PGOPTI_Prof_Dump_All(void);</code>

An older version of this function, `_PGOPTI_Prof_Dump()`, which will also dump profile information is still available; the older function operates much like `_PGOPTI_Prof_Dump_All()`, except on Linux when used in connection with shared libraries (.so) and `_exit()` to terminate a program. When `_PGOPTI_Prof_Dump_All()` is called before `_exit()` to terminate the program, the new function insures that a .dyn file is created for all shared libraries needing to create a .dyn file. Use `_PGOPTI_Prof_Dump_All()` on Linux to insure portability and correct functionality.

The profile information is generated in a .dyn file (generated in phase 2 of the PGO).

## Recommended usage

Insert a single call to this function in the body of the function which terminates the user application. Normally, `_PGOPTI_Prof_Dump_All()` should be called just once. It is also possible to use this function in conjunction with `_PGOPTI_Prof_Reset()` function to generate multiple .dyn files (presumably from multiple sets of input data).

Example
<pre>#include &lt;pgouser.h&gt; void process_data(int foo) {} int get_input_data() { return 1; } int main(void) {     // Selectively collect profile information for the portion     // of the application involved in processing input data.     int input_data = get_input_data();     while (input_data) {         PGOPTI_Prof_Reset();         process_data(input_data);         PGOPTI_Prof_Dump_All();         input_data = get_input_data();     } }</pre>

```
    return 0;
}
```

## Resetting the Dynamic Profile Counters

The `_PGOPTI_Prof_Reset()` function resets the dynamic profile counters. The prototype of the function call is listed below.

### Syntax

```
void _PGOPTI_Prof_Reset(void);
```

## Recommended usage

Use this function to clear the profile counters prior to collecting profile information on a section of the instrumented application. See the example under Dumping Profile Information.

## Dumping and Resetting Profile Information

The `_PGOPTI_Prof_Dump_And_Reset()` function dumps the profile information to a new .dyn file and then resets the dynamic profile counters. Then the execution of the instrumented application continues.

The prototype of the function call is listed below.

### Syntax

```
void _PGOPTI_Prof_Dump_And_Reset(void);
```

This function is used in non-terminating applications and may be called more than once. Each call will dump the profile information to a new .dyn file.

## Recommended usage

Periodic calls to this function enables a non-terminating application to generate one or more profile information files (.dyn files). These files are merged during the feedback phase (phase 3) of profile-guided optimizations. The direct use of this function enables your application to control precisely when the profile information is generated.

## Interval Profile Dumping

The `_PGOPTI_Set_Interval_Prof_Dump()` function activates Interval Profile Dumping and sets the approximate frequency at which dumps occur. This function is used in non-terminating applications.

The prototype of the function call is listed below.

**Syntax**

```
void _PGOPTI_Set_Interval_Prof_Dump(int interval);
```

This function is used in non-terminating applications.

The *interval* parameter specifies the time interval at which profile dumping occurs and is measured in milliseconds. For example, if interval is set to 5000, then a profile dump and reset will occur approximately every 5 seconds. The interval is approximate because the time-check controlling the dump and reset is only performed upon entry to any instrumented function in your application.

Setting the interval to zero or a negative number will disable interval profile dumping, and setting a very small value for the interval may cause the instrumented application to spend nearly all of its time dumping profile information. Be sure to set interval to a large enough value so that the application can perform actual work and substantial profile information is collected.

The following example demonstrates one way of using interval profile dumping in non-terminating code.

**Example**

```
#include <stdio.h>
// The next include is to access
// _PGOPTI_Set_Interval_Prof_Dump_All
#include <pgouser.h>
int returnValue()
{ return 100; }
int main()
{
    int ans;
    printf("CTRL-C to quit.\n");
    _PGOPTI_Set_Interval_Prof_Dump(5000);
    while (1)
        ans = returnValue();
}
```

You can compile the code shown above by entering commands similar to the following:

Platform	Example
Linux*	<code>icc -prof-gen -o instrumented_number number.c</code>
Windows*	<code>icl /Qprof-gen /Feinstrumented_number number.c</code>

When compiled, the code shown above will dump profile information a .dyn file about every five seconds until the program is ended.

You can use the profmerge tool to merge the .dyn files.

## Recommended usage

Call this function at the start of a non-terminating user application to initiate Interval Profile Dumping. Note that an alternative method of initiating Interval Profile Dumping is by setting the environment variable, `PROF_DUMP_INTERVAL`, to the desired interval value prior to starting the application.

The intention of Interval Profile Dumping is to allow a non-terminating application to be profiled with minimal changes to the application source code.

## PGO Tools Overview

The compiler includes several tools and utilities that can take advantage of Profile-guided Optimizations (PGO). You should have a good understanding of PGO before using the following tools or utilities:

- Code-coverage tool
- Test-prioritization tool
- Profmerge utility
- Proforder utility
- Profrun utility

In addition to the topics listed above, this section includes information on some compiler options that are used primarily to prepare applications for these tools and utilities.

See Profile-guided optimizations (PGO) Overview for more information.

## Code-Coverage Tool

The code-coverage tool provides software developers with a view of how much application code is exercised when a specific workload is applied to the application. To determine which code is used, the code-coverage tool uses Profile-Guided Optimization technology.

The major features of the code-coverage tool are:

- Visually presenting code coverage information for an application with a customizable code-coverage coloring scheme
- Displaying dynamic execution counts of each basic block of the application
- Providing differential coverage, or comparison, profile data for two runs of an application

The tool analyzes static profile information generated by the compiler, as well as dynamic profile information generated by running an instrumented form of the application binaries on the workload. The tool can generate the in HTML-formatted report and export data in both text-, and XML-formatted files. The reports can be further customized to show color-coded, annotated, source-code listings that distinguish between used and unused code.

The code-coverage tool is available on IA-32, Intel® EM64T, and Intel® Itanium® architectures on Linux\* and Windows\*. The tool supports only IA-32 on Mac OS\*.

You can use the tool in a number of ways to improve development efficiency, reduce defects, and increase application performance:

- During the project testing phase, the tool can measure the overall quality of testing by showing how much code is actually tested.
- When applied to the profile of a performance workload, the code-coverage tool can reveal how well the workload exercises the critical code in an application. High coverage of performance-critical modules is essential to taking full advantage of the Profile-Guided Optimizations that Intel Compilers offer.
- The tool provides an option, useful for both coverage and performance tuning, enabling developers to display the dynamic execution count for each basic block of the application.
- The code-coverage tool can compare the profile of two different application runs. This feature can help locate portions of the code in an application that are unrevealed during testing but are exercised when the application is used outside the test space, for example, when used by a customer.

## Code-coverage tool Requirements

To run the code-coverage tool on an application, you must have following items:

- The application sources.
- The .spi file generated by the Intel® compiler when compiling the application for the instrumented binaries using the `-prof-genx` (Linux\*) or `/Qprof-genx` (Windows\*) options.
- A pgopti.dpi file that contains the results of merging the dynamic profile information (.dyn) files, which is most easily generated by the profmerge tool. This file is also generated implicitly by the Intel® compilers when compiling an application with `-prof-use` (Linux) or `/Qprof-use` (Windows) options with available .dyn and .dpi files.

See Understanding Profile-guided Optimization and Example of Profile-guided Optimization for general information on creating the files needed to run this tool.

## Using the Tool

In general, you must perform the following steps to use the code-coverage tool:

1. Compile the application using `-prof-genx` (Linux) or `/Qprof-genx` (Windows). This step generates an instrumented executable and a corresponding static profile information (pgopti.spi) file.
2. Run the instrumented application. This step creates the dynamic profile information (.dyn) file. Each time you run the instrumented application, the compiler generates a unique .dyn file either in the current directory or the directory specified in by `prof_dir`.

3. Use the profmerge tool to merge all the .dyn files into one .dpi (pgopti.dpi) file. This step consolidates results from all runs and represents the total profile information for the application, generates an optimized binary, and creates the dpi file needed by the code-coverage tool.

You can use the profmerge tool to merge the .dyn files into a .dpi file without recompiling the application. The profmerge tool can also merge multiple .dpi files into one .dpi file using the `profmerge -a` option. Select the name of the output .dpi file using the `profmerge -prof_dpi` option.

### Caution

The profmerge tool merges all .dyn files that exist in the given directory. Make sure unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code.

4. Run the code-coverage tool. (The valid syntax and tool options are shown below.)  
This step creates a report or exported data as specified. If no other options are specified, the code-coverage tool places a single HTML file (CODE\_COVERAGE.HTML) in the current directory. Open the file in a web browser to view the reports.

The tool uses the following syntax:

### Code-coverage tool Syntax

```
codecov [-codecov_option]
```

where `-codecov_option` is one or more optional parameters specifying the tool option passed to the tool.

The available tool options are listed under Code-coverage tool Options (below). If you do not use any additional tool options, the tool will provide the top-level code coverage for the entire application.

### Note

Windows\* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

The code-coverage tool allows you to name the project and specify paths to specific, necessary files. The following example demonstrates how to name a project and specify .dpi and .spi files to use:

### Example: specify .dpi and .spi files

```
codecov -prj myProject -spi pgopti.spi -dpi pgopti.dpi
```

The tool can add a contact name and generate an email link for that contact at the bottom of each HTML page. This provide a way to send an electronic message to the named contact. The following example demonstrates how to add specify a contact and the email links:

**Example: add contact information**

```
codecov -prj myProject -mname JoeSmith -maddr js@company.com
```

This following example demonstrates how to use the tool to specify the project name, specify the dynamic profile information file, and specify the output format and file name.

**Example: export data to text**

```
codecov -prj test1 -dpi test1.dpi -txtbcvrg test1_bcvrg.txt
```

## Code-coverage tool Options

The tool uses the options listed in the table:

Option	Default	Description
-help		Prints all the options of the code-coverage tool.
-spi <i>file</i>	pgopti.spi	Sets the path name of the static profile information file (.spi).
-dpi <i>file</i>	pgopti.dpi	Specifies the path name of the dynamic profile information file (.dpi).
-prj <i>string</i>		Sets the project name.
-counts		Generates dynamic execution counts.
-nopmeter		Turns off the progress meter. The meter is enabled by default.
-nopartial		Treats partially covered code as fully covered code.
-comp <i>file</i>		Specifies the file name that contains the list of files being covered.
-ref		Finds the differential coverage with respect to ref_dpi_file.
-demang		Demangles both function names and their arguments.
-mname <i>string</i>		Sets the name of the web-page owner.
-maddr <i>string</i>		Sets the email address of the web-page owner.
-bcolor <i>color</i>	#ffff99	Specifies the HTML color name for code in the uncovered blocks.

<code>-fcolor color</code>	<code>#ffcccc</code>	Specifies the HTML color name for code of the uncovered functions.
<code>-pcolor color</code>	<code>#fafad2</code>	Specifies the HTML color name or code of the partially covered code.
<code>-ccolor color</code>	<code>#ffffff</code>	Specifies the HTML color name or code of the covered code.
<code>-ucolor color</code>	<code>#ffffff</code>	Specifies the HTML color name or code of the unknown code.
<code>-xcolor color</code>	<code>#ffffff</code>	Specifies the HTML color of the code ignored.
<code>-beginblkdsbl string</code>		Specifies the comment that marks the beginning of the code fragment to be ignored by the coverage tool. If the string is not part of an inline comment, the string value must be surrounded by quotation marks (").
<code>-endblkdsbl string</code>		Specifies the comment that marks the end of the code fragment to be ignored by the coverage tool. If the string is not part of an inline comment, the string value must be surrounded by quotation marks (").
<code>-onelinedsbl string</code>		Specifies the comment that marks individual lines of code or the whole functions to be ignored by the coverage tool. If the string is not part of an inline comment, the string value must be surrounded by quotation marks (").
<code>-txtfcvrg file</code>		Export function coverage for covered function in text format. The file parameter must be in the form of a valid file name.
<code>-txtbcvrg file</code>		Export block-coverage for covered functions as text format. The file parameter must be in the form of a valid file name.
<code>-txtbcvrgfull file</code>		Export block-coverage for entire application in text and HTML formats. The file parameter must be in the form of a valid file name.
<code>-xmlbcvrg file</code>		Export the block-coverage for the covered function in XML format.
<code>-xmlfcvrg file</code>		Export function coverage for covered function in XML format. The file parameter must be in the form of a valid file name.
<code>-xmlbcvrgfull file</code>		Export function coverage for entire application in HTML and XML formats. The file parameter must be in the form of a valid file name.

## Visually Presenting Code Coverage for an Application

Based on the profile information collected from running the instrumented binaries when testing an application, the Intel® compiler will create HTML-formatted reports using the code-coverage tool. These reports indicate portions of the source code that were or were not exercised by the tests. When applied to the profile of the performance



workloads, the code-coverage information shows how well the training workload covers the application's critical code. High coverage of performance-critical modules is essential to taking full advantage of the profile-guided optimizations.

The code-coverage tool can create two levels of coverage:

- Top level (for a group of selected modules)
- Individual module source views

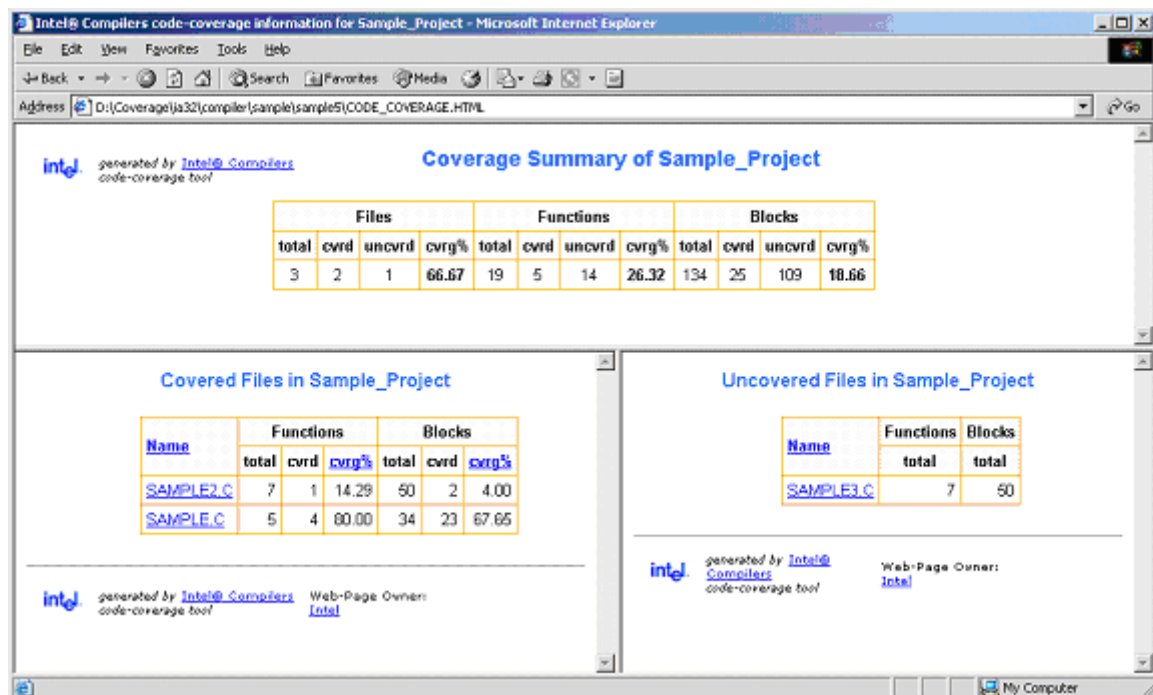
## Top Level Coverage

The top-level coverage reports the overall code coverage of the modules that were selected. The following options are provided:

- Select the modules of interest
- For the selected modules, the tool generates a list with their coverage information. The information includes the total number of functions and blocks in a module and the portions that were covered.
- By clicking on the title of columns in the reported tables, the lists may be sorted in ascending or descending order based on:
  - basic block coverage
  - function coverage
  - function name

By default, the code-coverage tool generates a single HTML file (CODE\_COVERAGE.HTML) and a subdirectory (CodeCoverage) in the current directory. The HTML file defines a frameset to display all of the other generated reports. Open the HTML file in a web-browser. The tool places all other generated report files in a CodeCoverage subdirectory.

If you choose to generate the html-formatted version of the report, you can view coverage source of that particular module directly from a browser. The following figure shows the top-level coverage report.



The coverage tool creates a frame set that allows quick browsing through the code to identify uncovered code. The top frame displays the list of uncovered functions while the bottom frame displays the list of covered functions. For uncovered functions, the total number of basic blocks of each function is also displayed. For covered functions, both the total number of blocks and the number of covered blocks as well as their ratio (that is, the coverage rate) are displayed.

For example, 66.67(4/6) indicates that four out of the six blocks of the corresponding function were covered. The block coverage rate of that function is thus 66.67%. These lists can be sorted based on the coverage rate, number of blocks, or function names. Function names are linked to the position in source view where the function body starts. So, just by one click, you can see the least-covered function in the list and by another click the browser displays the body of the function. You can scroll down in the source view and browse through the function body.

### Individual Module Source View

Within the individual module source views, the tool provides the list of uncovered functions as well as the list of covered functions. The lists are reported in two distinct frames that provide easy navigation of the source code. The lists can be sorted based on:

- Number of blocks within uncovered functions
- Block coverage in the case of covered functions
- Function names

## Setting the Coloring Scheme for the Code Coverage

The tool provides a visible coloring distinction of the following coverage categories: covered code, uncovered basic blocks, uncovered functions, partially covered code, and unknown code. The default colors that the tool uses for presenting the coverage information are shown in the tables that follows:

Category	Default	Description
Covered code	#FFFFFF	Indicates code was exercised by the tests. You can override the default color with the <code>-ccolor</code> tool option.
Uncovered basic block	#FFFF99	Indicates the basic blocks that were not exercised by any of the tests. However, these blocks were within functions that were executed during the tests. You can override the default color with the <code>-bcolor</code> tool option.
Uncovered function	#FFCCCC	Indicates functions that were never called during the tests. You can override the default color with the <code>-fcolor</code> tool option.
Partially covered code	#FAFAD2	Indicates that more than one basic block was generated for the code at this position. Some of the blocks were covered and some were not. You can override the default color with the <code>-pcolor</code> tool option.
Ignored code	#FFFFFF	Code that was specifically marked to be ignored. You can override this default color using the <code>-xcolor</code> tool option.
Unknown	#FFFFFF	No code was generated for this source line. Most probably, the source at this position is a comment, a header-file inclusion, or a variable declaration. You can override the default color with the <code>-ucolor</code> tool option.

The default colors can be customized to be any valid HTML color name or hexadecimal value using the options mentioned for each coverage category in the table above.

For code-coverage colored presentation, the coverage tool uses the following heuristic: source characters are scanned until reaching a position in the source that is indicated by the profile information as the beginning of a basic block. If the profile information for that basic block indicates that a coverage category changes, then the tool changes the color corresponding to the coverage condition of that portion of the code, and the coverage tool inserts the appropriate color change in the HTML-formatted report files.

### Note

You need to interpret the colors in the context of the code. For instance, comment lines that follow a basic block that was never executed would be colored in the same color as the uncovered blocks. Another example is the closing brackets in C/C++ applications.

## Coverage Analysis of Module Subsets

One of the capabilities of the code-coverage tool is efficient coverage analysis of a subset of modules for an application.

You can generate the profile information for the whole application, or a subset of it, and then break the covered modules into different components and use the coverage tool to obtain the coverage information of each individual component. If only a subset of the application modules is compiled with the `-prof-genx` (Linux) or `/Qprof-genx` (Windows) option, then the coverage information is generated only for those modules that are involved with this compiler option, thus avoiding the overhead incurred for profile generation of other modules.

To specify the modules of interest, use the `-comp` option. This option takes the name of a file as its argument. The file must be a text file that includes the name of modules or directories you would like to analyze.

### Note

Each line of component file should include one, and only one, module name.

For example:

<b>Example</b>
----------------

<code>codecov -prj Project_Name -comp component1</code>
---

Any module of the application whose full path name has an occurrence of any of the names in the component file will be selected for coverage analysis. For example, if a line of file `component1` in the above example contains `mod.cpp`, then all modules in the application that have that name will be analyzed. The user can specify a particular module by passing more specific path information. For example, if the line contains `/cmp1/mod1.cpp`, then only those modules with the name `mod.cpp` will be selected that are in a directory named `cmp1`. If no component file is specified, then all files that have been compiled with `-prof-genx` (Linux) or `/Qprof-genx` (Windows) are selected for coverage analysis.

## Dynamic Counters

The coverage tool can be configured to generate the information about the dynamic execution counts. This ability can display the dynamic execution count of each basic block of the application and is useful for both coverage and performance tuning.

The custom configuration requires using the `-counts` option. The counts information is displayed under the code after a "^" sign precisely under the source position where the corresponding basic block begins.

If more than one basic block is generated for the code at a source position (for example, for macros), then the total number of such blocks and the number of the blocks that were

executed are also displayed in front of the execution count. For example, line 11 in the code is an `IF` statement:

Example	
11	<code>IF ((N .EQ. 1) .OR. (N .EQ. 0))</code> <code>  ^ 10 (1/2)</code>
12	<code>  PRINT N</code> <code>    ^ 7</code>

The coverage lines under code lines 11 and 12 contain the following information:

- The `IF` statement in line 11 was executed 10 times.
- Two basic blocks were generated for the `IF` statement in line 11.
- Only one of the two blocks was executed, hence the partial coverage color.
- Only seven out of the ten times variable `n` had a value of 0 or 1.

In certain situations, it may be desirable to consider all the blocks generated for a single source position as one entity. In such cases, it is necessary to assume that all blocks generated for one source position are covered when at least one of the blocks is covered. This assumption can be configured with the `-nopartial` option. When this option is specified, decision coverage is disabled, and the related statistics are adjusted accordingly. The code lines 11 and 12 indicate that the `print` statement in line 12 was covered. However, only one of the conditions in line 11 was ever true. With the `-nopartial` option, the tool treats the partially covered code (like the code on line 11) as covered.

## Differential Coverage

Using the code-coverage tool, you can compare the profiles from two runs of an application: a reference run and a new run identifying the code that is covered by the new run but not covered by the reference run. Use this feature to find the portion of the applications code that is not covered by the applications tests but is executed when the application is run by a customer. It can also be used to find the incremental coverage impact of newly added tests to an applications test space.

### Generating Reference Data

Create the dynamic profile information for the reference data, which can be used in differential coverage reporting later, by using the `-ref` option. The following command demonstrate a typical command for generating the reference data:

Example: generating reference data
<code>codecov -prj Project_Name -dpi customer.dpi -ref appTests.dpi</code>

The coverage statistics of a differential-coverage run shows the percentage of the code exercised on a new run but missed in the reference run. In such cases, the tool shows only the modules that included the code that was not covered. Keep this in mind when viewing the coloring scheme in the source views.

The code that has the same coverage property (covered or not covered) on both runs is considered as covered code. Otherwise, if the new run indicates that the code was executed while in the reference run the code was not executed, then the code is treated as uncovered. On the other hand, if the code is covered in the reference run but not covered in the new run, the differential-coverage source view shows the code as covered.

## Running Differential Coverage

To run the code-coverage tool for differential coverage, you must have the application sources, the .spi file, and the .dpi file, as described in the Code-coverage tool Requirements section (above).

Once the required files are available, enter a command similar to the following begin the process of differential coverage analysis:

### Example

```
codecov -prj Project_Name -spi pgopti.spi -dpi pgopti.dpi
```

Specify the path to the .dpi and .spi using the `-spi` and `-dpi` options.

## Excluding Code from Coverage Analysis

The code-coverage tool allows you to exclude portions of your code from coverage analysis. This ability can be useful during development; for example, certain portions of code might include functions used for debugging only. The test case should not include tests for functionality that will unavailable in the final application.

Another example of code that can be excluded is code that might be designed to deal with internal errors unlikely to occur in the application. In such cases, not having a test case lack of a test case is preferred. You might want to ignore infeasible (dead) code in the coverage analysis. The code-coverage tool provides several options for marking portions of the code infeasible (dead) and ignoring the code at the file level, function level, line level, and arbitrary code boundaries indicated by user-specific comments. The following sections explain how to exclude code at different levels.

## Including and Excluding Coverage at the File Level

The code-coverage tool provides the ability to selectively include or exclude files for analysis. Create a component file and add the appropriate string values that indicate the file and directory name for code you want included or excluded. Pass the file name as a parameter of the `-comp` option. The following example shows the general command:

### Example: specifying a component file

```
codecov -comp file
```

where `file` is the name of a text file containing strings that ask as file and directory name masks for including and excluding file-level analysis. For example, assume that the following:

- You want to include all files with the string "source" in the file name or directory name.
- You create a component text file named `myComp.txt` with the selective inclusion string.

Once you have a component file, enter a command similar to the following:

### Example

```
codecov -comp myComp.txt
```

In this example, individual files name including the string "source" (like `source1.c` and `source2.c`) and files in directories where the name contains the string "source" (like `source/file1.c` and `source2\file2.c`) are include in the analysis.

Excluding files is done in the same way; however, the string must have a tilde (~) prefix. The inclusion and exclusion can be specified in the same component file.

For example, assume you want to analyze all individual files or files contained in a directory where the name included the string "source", and you wanted to exclude all individual file and files contained in directories where the name included the string "skip". You would add content similar to the following to the component file (`myComp.txt`) and pass it to the `-comp` option:

### Example: inclusion and exclusion strings

```
source
~skip
```

Entering the `codecov -comp myComp.txt` command with both instructions in the component file will instruct the tool to include individual files where the name contains "source" (like `source1.c` and `source2.c`) and directories where the name contains "source" (like `source/file1.c` and `source2\file2.c`), and exclude any individual files where the name contains "skip" (like `skipthis1.c` and `skipthis2.c`) or directories where the name contains "skip" (like `skipthese1\debug1.c` and `skipthese2\debug2.c`).

## Excluding Coverage at the Line and Function Level

You can mark individual lines for exclusion my passing string values to the `-onelinedsbl` option. For example, assume that you have some code similar to the following:

### Sample code

```
printf ("internal error 123 - please report!\n"); // INFEASIBLE
printf ("internal error 456 - please report!\n"); /* INF IA32 */
```

If you wanted to exclude all functions marked with the comments INFEASIBLE or INF IA32, you would enter a command similar to the following.

### Example

```
codecov -onelinedsbl INFEASIBLE -onelinedsbl "INF IA32"
```

You can specify multiple exclusion strings simultaneously, and you can specify any string values for the markers; however, you must remember the following guidelines when using this option:

- Inline comments must occur at the end of the statement.
- If the string is not part of an inline comment, the string value must be surrounded by quotation marks (").

An entire function can be excluded from coverage analysis using the same methods. For example, the following function will be ignored from the coverage analysis when you issue example command shown above.

### Sample code

```
void dumpInfo (int n)
{ // INFEASIBLE
  ...
}
```

Additionally, you can use the code-coverage tool to color the infeasible code with any valid HTML color code by combining the `-onelinedsbl` and `-xcolor` options. The following example commands demonstrate the combination:

### Example: combining tool options

```
codecov -onelinedsbl INF -xcolor lightgreen
codecov -onelinedsbl INF -xcolor #CCFFCC
```

## Excluding Code by Defining Arbitrary Boundaries

The code-coverage tool provides the ability to arbitrarily exclude code from coverage analysis. This feature is most useful where the excluded code either occur inside of a function or spans several functions.

Use the `-beginblkdsbl` and `-endblkdsbl` options to mark the beginning and end, respectively, of any arbitrarily defined boundary to exclude code from analysis.

Remember the following guidelines when using these options:

- Inline comments must occur at the end of the statement.
- If the string is not part of an inline comment, the string value must be surrounded by quotation marks (").

For example assume that you have the following code:



**Sample code**

```

void div (int m, int n)
{
  if (n == 0)
  /* BEGIN INF */
  {
    printf (internal error 314 please report\n);
    recover ();
  }
  /* END INF */
  else {
    ...
  }
  ...
  // BINF
  Void recover ()
  {
    ...
  }
  // EINF

```

The following example commands demonstrate how to use the `-beginblkdsbl` option to mark the beginning and the `-endblkdsbl` option to mark the end of code to exclude from the sample shown above.

**Example: arbitrary code marker commands**

```

codecov -xcolor #ccFFCC -beginblkdsbl BINF -endblkdsbl EINF
codecov -xcolor #ccFFCC -beginblkdsbl "BEGIN INF" -endblkdsbl "END INF"

```

Notice that you can combine these options in combination with the `-xcolor` option.

## Exporting Coverage Data

The code-coverage tool provides specific options to extract coverage data from the dynamic profile information (.dpi files) that result from running instrumented application binaries under various workloads. The tool can export the coverage data in various formats for post-processing and direct loading into databases: the default HTML, text, and XML. You can choose to export data at the function and basic block levels.

There are two basic methods for exporting the data: quick export and combined export. Each method has associated options supported by the tool

- **Quick export:** The first method is to export the data coverage to text- or XML-formatted files without generating the default HTML report. The application sources need not be present for this method. The code-coverage tool creates reports and provides statistics only about the portions of the application executed. The resulting analysis and reporting occurs quickly, which makes it practical to apply the coverage tool to the dynamic profile information (the .dpi file) for every test case in a given test space instead of applying the tool to the profile of individual test suites or the merge of all test suites. The `-xmlfcvrg`, `-txtfcvrg`, `-xmlbcvrg` and `-txtbcvrg` options support the first method.

- **Combined export:** The second method is to generate the default HTML and simultaneously export the data to text- and XML-formatted files. This process is slower than first method since the application sources are parsed and reports generated. The `-xmlbcvrgfull` and `-txtbcvrgfull` options support the second method.

These export methods provide the means to quickly extend the code coverage reporting capabilities by supplying consistently formatted output from the code-coverage tool. You can extend these by creating additional reporting tools on top of these report files.

## Quick Export

The profile of covered functions of an application can be exported quickly using the `-xmlfcvrg`, `-txtfcvrg`, `-xmlbcvrg`, and `-txtbcvrg` options. When using any of these options, specify the output file that will contain the coverage report. For example, enter a command similar to the following to generate a report of covered functions in XML formatted output:

### Example: quick export of function data

```
codecov -prj test1 -dpi test1.dpi -xmlfcvrg test1_fcvg.xml
```

The resulting report will show how many times each function was executed and the total number of blocks of each function together with the number of covered blocks and the block coverage of each function. The following example shows some of the content of a typical XML report.

### XML-formatted report example

```
<PROJECT name = "test1">
  <MODULE name = "D:\SAMPLE.C">
    <FUNCTION name="f0" freq="2">
      <BLOCKS total="6" covered="5" coverage="83.33%"></BLOCKS>
    </FUNCTION>
    ...
  </MODULE>
  <MODULE name = "D:\SAMPLE2.C">
    ...
  </MODULE>
</PROJECT>
```

In the above example, we note that function `f0`, which is defined in file `sample.c`, has been executed twice. It has a total number of six basic blocks, five of which are executed, resulting in an 83.33% basic block coverage.

You can also export the data in text format using the `-txtfcvrg` option. The generated text report, using this option, for the above example would be similar to the following example:

### Text-formatted report example

Covered Functions in File: "D:\SAMPLE.C"				
"f0"	2	6	5	83.33
"f1"	1	6	4	66.67

"f2"	1	6	3	50.00
...				

In the text formatted version of the report, the each line of the report should be read in the following manner:

Column 1	Column 2	Column 3	Column 4	Column 5
function name	execution frequency	line number of the start of the function definition	column number of the start of the function definition	percentage of basic-block coverage of the function

Additionally, the tool supports exporting the block level coverage data using the `-xmlbcvrg` option. For example, enter a command similar to the following to generate a report of covered blocks in XML formatted output:

<b>Example: quick export of block data to XML</b>
<code>codecov -prj test1 -dpi test1.dpi -xmlbcvrg test1_bcvrg.xml</code>

The example command shown above would generate XML-formatted results similar to the following:

<b>XML-formatted report example</b> <pre>&lt;PROJECT name = "test1"&gt;   &lt;MODULE name = "D:\SAMPLE.C"&gt;     &lt;FUNCTION name="f0" freq="2"&gt;       ...       &lt;BLOCK line="11" col="2"&gt;         &lt;INSTANCE id="1" freq="1"&gt; &lt;/INSTANCE&gt;       &lt;/BLOCK&gt;       &lt;BLOCK line="12" col="3"&gt;         &lt;INSTANCE id="1" freq="2"&gt; &lt;/INSTANCE&gt;         &lt;INSTANCE id="2" freq="1"&gt; &lt;/INSTANCE&gt;       &lt;/BLOCK&gt;</pre>
---

In the sample report, notice that one basic block is generated for the code in function f0 at the line 11, column 2 of the file sample.c. This particular block has been executed only once. Also notice that there are two basic blocks generated for the code that starts at line 12, column 3 of file. One of these blocks, which has id = 1, has been executed two times, while the other block has been executed only once. A similar report in text format can be generated through the `-txtbcvrg` option.

## Combined Exports

The coverage tool has also the capability of exporting coverage data in the default HTML format while simultaneously generating the text- and XML-formatted reports.

Use the `-xmlbcvrgfull` and `-txcbcvrgfull` options to generate reports in all supported formatted in a single run. These options export the basic-block level coverage data while simultaneously generating the HTML reports. These options generate more complete reports since they include analysis on functions that were not executed at all. However, exporting the coverage data using these options requires access to application source files and take much longer to run.

## Test-Prioritization Tool

The test-prioritization tool enables the profile-guided optimizations on IA-32, Intel® EM64T, and Itanium® architectures, on Linux\* and Windows\*, to select and prioritize test for an application based on prior execution profiles. The tool supports only IA-32 on Mac OS\*.

The tool offers a potential of significant time saving in testing and developing large-scale applications where testing is the major bottleneck.

Development often requires changing applications modules. As applications change, developers can have a difficult time retaining the quality of their functional and performance tests so they are current and on-target. The test-prioritization tool lets software developers select and prioritize application tests as application profiles change.

## Features and Benefits

The test-prioritization tool provides an effective testing hierarchy based on the code coverage for an application. The following list summarizes the advantages of using the tool:

- Minimizing the number of tests that are required to achieve a given overall coverage for any subset of the application: the tool defines the smallest subset of the application tests that achieve exactly the same code coverage as the entire set of tests.
- Reducing the turn-around time of testing: instead of spending a long time on finding a possibly large number of failures, the tool enables the users to quickly find a small number of tests that expose the defects associated with the regressions caused by a change set.
- Selecting and prioritizing the tests to achieve certain level of code coverage in a minimal time based on the data of the tests' execution time.

See Understanding Profile-guided Optimization and Example of Profile-guided Optimization for general information on creating the files needed to run this tool.

## Test-prioritization tool Requirements

The following items are files are required to run the test-prioritization tool on an applications tests:

- The .spi file generated by Intel® compilers when compiling the application for the instrumented binaries with the `-prof-genx` (Linux\*) or `/Qprof-genx` (Windows\*) option.
- The .dpi files generated by the profmerge tool as a result of merging the dynamic profile information .dyn files of each of the application tests. Run the profmerge tool on all .dyn files that are generated for each individual test and name the resulting .dpi in a fashion that uniquely identifies the test.

### Caution

The profmerge tool merges all .dyn files that exist in the given directory. Make sure unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code

- User-generated file containing the list of tests to be prioritized. For successful tool execution, you should:
  - Name each test .dpi file so the file names uniquely identify each test.
  - Create a .dpi list file, which is a text file that contains the names of all .dpi test files.

Each line of the .dpi list file should include one, and only one .dpi file name. The name can optionally be followed by the duration of the execution time for a corresponding test in the `dd:hh:mm:ss` format.

For example: `Test1.dpi 00:00:60:35` states that Test1 lasted 0 days, 0 hours, 60 minutes and 35 seconds.

The execution time is optional. However, if it is not provided, then the tool will not prioritize the test for minimizing execution time. It will prioritize to minimize the number of tests only.

The tool uses the following general syntax:

<b>Test-priorization tool Syntax</b>
<code>tselect -dpi_list file</code>

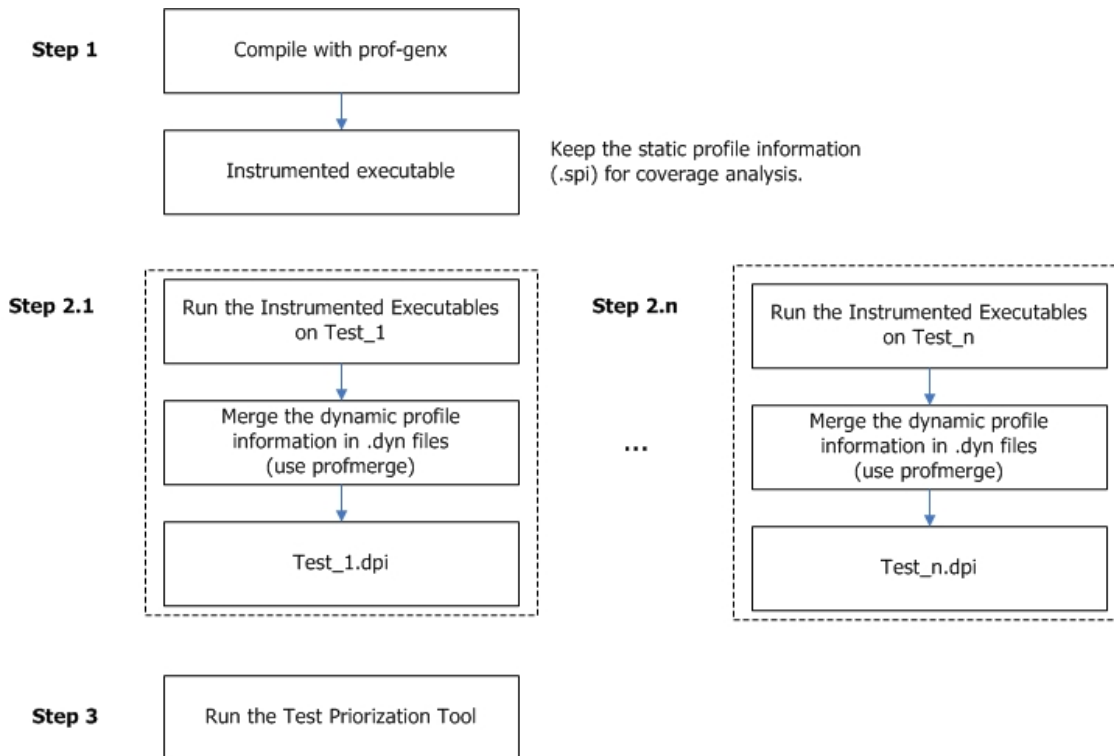
where `-dpi_list` is a required tool option that sets the path to the list file containing the list of the all .dpi files. All other commands are optional.

### Note

Windows\* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

## Usage Model

The following figure illustrates a test-prioritization tool usage model.



## Test-prioritization tool Options

The tool uses the options that are listed in the following table:

Option	Default	Description
-help		Prints options supported by the tool.
-spi <i>file</i>	pgopti.spi	Specifies the file name of the static profile information file (.SPI).
-dpi_list <i>file</i>		Specifies the file name of the file that contains the name of the dynamic profile information (.dpi) files.  Each line of the file must contain one .dpi name optionally followed by its execution time. The name must uniquely identify the test.
-prof_dpi <i>dir</i>		Specifies the path name of the output report file.
-o <i>file</i>		Specifies the file name of the output report file.
-comp <i>file</i>		Specifies the file name that contains the list of files of interest.

-cutoff <i>value</i>		Instructs the tool to terminate when the cumulative block coverage reaches a preset percentage, as specified by <i>value</i> , of pre-computed total coverage. <i>value</i> must be greater than 0.0 (for example, 99.00) but not greater than 100. <i>value</i> can be set to 100.
-nototal		Instructs the tool to ignore the pre-compute total coverage process.
-mintime		Instructs the tool to minimize testing execution time. The execution time of each test must be provided on the same line of <code>dpi_list</code> file, after the test name in <code>dd:hh:mm:ss</code> format.
-verbose		Instructs the tool to generate more logging information about program progress.

## Running the tool

The following steps demonstrate one simple example for running the tool on IA-32 architectures.

1. Specify the directory by entering a command similar to the following:

<b>Example</b>
<code>set prof-dir=c:\myApp\prof-dir</code>

2. Compile the program and generate instrumented binary by issuing commands similar to the following:

Platform	Command
Linux	<code>icpc -prof-genx myApp.c</code>
Windows	<code>icl /Qprof-genx myApp.c</code>

This commands shown above compiles the program and generates instrumented binary `myApp` as well as the corresponding static profile information `pgopti.spi`.

3. Make sure that unrelated `.dyn` files are not present by issuing a command similar to the following:

<b>Example</b>
<code>rm prof-dir \*.dyn</code>

4. Run the instrumented files by issuing a command similar to the following:

<b>Example</b>
<code>myApp &lt; data1</code>

The command runs the instrumented application and generates one or more new dynamic profile information files that have an extension `.dyn` in the directory specified by the `-prof-dir` step above.

5. Merge all `.dyn` file into a single file by issuing a command similar to the following:

<b>Example</b>
----------------

<pre>profmerge -prof_dpi Test1.dpi</pre>
--

The `profmerge` tool merges all the `.dyn` files into one file (`Test1.dpi`) that represents the total profile information of the application on `Test1`.

6. Again make sure there are no unrelated `.dyn` files present a second time by issuing a command similar to the following:

<b>Example</b>
----------------

<pre>rm prof-dir \*.dyn</pre>
-------------------------------

7. Run the instrumented application and generate one or more new dynamic profile information files that have an extension `.dyn` in the directory specified the `prof-dir` step above by issuing a command similar to the following:

<b>Example</b>
----------------

<pre>myApp &lt; data2</pre>
-----------------------------

8. Merge all `.dyn` files into a single file, by issuing a command similar to the following

<b>Example</b>
----------------

<pre>profmerge -prof_dpi Test2.dpi</pre>
--

At this step, the `profmerge` tool merges all the `.dyn` files into one file (`Test2.dpi`) that represents the total profile information of the application on `Test2`.

9. Make sure that there are no unrelated `.dyn` files present for the final time, by issuing a command similar to the following:

<b>Example</b>
----------------

<pre>rm prof-dir \*.dyn</pre>
-------------------------------

10. Run the instrumented application and generates one or more new dynamic profile information files that have an extension `.dyn` in the directory specified by `-prof-dir` by issuing a command similar to the following:

<b>Example</b>
----------------

<pre>myApp &lt; data3</pre>
-----------------------------



11. Merge all .dyn file into a single file, by issuing a command similar to the following:

### Example

```
profmerge -prof_dpi Test3.dpi
```

At this step, the profmerge tool merges all the .dyn files into one file (Test3.dpi) that represents the total profile information of the application on Test3.

12. Create a file named tests\_list with three lines. The first line contains Test1.dpi, the second line contains Test2.dpi, and the third line contains Test3.dpi.

## Tool Usage Examples

When these items are available, the test-prioritization tool may be launched from the command line in prof-dir directory as described in the following examples.

### Example 1: Minimizing the Number of Tests

The following example describes how minimize the number of test runs.

### Example Syntax

```
tselect -dpi_list tests_list -spi pgopti.spi
```

where the -spi option specifies the path to the .spi file.

The following sample output shows typical results from the test-prioritization tool.

### Sample Output

```
Total number of tests = 3
Total block coverage ~ 52.17
Total function coverage ~ 50.00
```

Num	%RatCvrg	%BlkCvrg	%FncCvrg	Test Name @ Options
1	87.50	45.65	37.50	Test3.dpi
2	100.00	52.17	50.00	Test2.dpi

In this example, the results provide the following information:

- By running all three tests, we achieve 52.17% block coverage and 50.00% function coverage.
- Test3 by itself covers 45.65% of the basic blocks of the application, which is 87.50% of the total block coverage that can be achieved from all three tests.
- By adding Test2, we achieve a cumulative block coverage of 52.17% or 100% of the total block coverage of Test1, Test2, and Test3.

- Elimination of Test1 has no negative impact on the total block coverage.

## Example 2: Minimizing Execution Time

Suppose we have the following execution time of each test in the `tests_list` file:

Sample Output	
Test1.dpi	00:00:60:35
Test2.dpi	00:00:10:15
Test3.dpi	00:00:30:45

The following command minimizes the execution time by passing the `-mintime` option:

Sample Syntax	
<code>tselect -dpi_list tests_list -spi pgopti.spi -mintime</code>	

The following sample output shows possible results:

Sample Output	
Total number of tests	= 3
Total block coverage	~ 52.17
Total function coverage	~ 50.00
Total execution time	= 1:41:35

Num	elapsedTime	%RatCvrg	%BlkCvrg	%FncCvrg	Test Name @ Options
1	10:15	75.00	39.13	25.00	Test2.dpi
2	41:00	100.00	52.17	50.00	Test3.dpi

In this case, the results indicate that the running all tests sequentially would require one hour, 45 minutes, and 35 seconds, while the selected tests would achieve the same total block coverage in only 41 minutes.

The order of tests, when prioritization, is based on minimizing time (first Test2, then Test3) could be different than when prioritization is done based on minimizing the number of tests. See Example 1 shown above: first Test3, then Test2. In Example 2, Test2 is the test that gives the highest coverage per execution time, so Test2 is picked as the first test to run.

## Using Other Options

The `-cutoff` option enables the tool to exit when it reaches a given level of basic block coverage. The following example demonstrates how to the option:

Example	
<code>tselect -dpi_list tests_list -spi pgopti.spi -cutoff 85.00</code>	

If the tool is run with the cutoff value of 85.00, as in the above example, only Test3 will be selected, as it achieves 45.65% block coverage, which corresponds to 87.50% of the total block coverage that is reached from all three tests.

The test-prioritization tool does an initial merging of all the profile information to figure out the total coverage that is obtained by running all the tests. The `-nototal` option enables you to skip this step. In such a case, only the absolute coverage information will be reported, as the overall coverage remains unknown.

## Profmerge and Proforder Utilities

### profmerge Utility

Use the profmerge utility to merge dynamic profile information (.dyn) files. The compiler executes profmerge automatically during the feedback compilation phase when you specify `-prof-use` (Linux\*) or `/Qprof-use` (Windows\*).

The command-line usage for profmerge is as follows:

Syntax
<code>profmerge [/nologo] [-prof_dir dir_name]</code>

This merges all .dyn files in the current directory or the directory specified by `-prof_dir` and produces the summary file `pgopti.dpi`.

The `-prof_dir` utility option is different than the compiler option: `-prof-dir` (Linux) or `/Qprof-dir` (Windows).

Since the profmerge utility merges all the .dyn files that exist in the given directory, you must insure unrelated .dyn files are not present; otherwise, profile information will be based on invalid profile data, which can negatively impact the performance of optimized code. The .dyn files can be merged to a .dpi file by the profmerge utility without recompiling the application.

### Profmerge Options

The profmerge utility supports the following options:

Option	Description
<code>-help</code>	List supported options.
<code>-nologo</code>	Disables version information.
<code>-exclude_funcs functions</code>	Excludes function from the profile. The list items must be separated by a comma (","); you can use a period (".") as a wild card character in function names.
<code>-prof_dir dir</code>	Specifies the directory from which to read .dyn and .dpi files
<code>-prof_dpi file</code>	Specifies the name of the .dpi file.

<code>-prof_file file</code>	Merges information from file matching: <code>dpi_file_and_dyn_tag</code>
<code>-dump</code>	Displays profile information.
<code>-src_old dir -src new dir</code>	Change the directory path stored within the .dpi.
<code>-a file1.dpi...fileN.dpi</code>	Merges .dpi files.

## Relocating source files using profmerge

The Intel® compiler uses the full path to the source file for each routine to look up the profile summary information associated with that routine. By default, this prevents you from:

- Using the profile summary file (.dpi) if you move your application sources.
- Sharing the profile summary file with another user who is building identical application sources that are located in a different directory.

To enable the movement of application sources, as well as the sharing of profile summary files, use `profmerge -src_old -src_new`. For example:

### Example: relocation command syntax

```
profmerge -prof_dir <dir1> -src_old <dir2> -src_new <dir3>
```

where `<dir1>` is the full path to dynamic information file (.dpi), `<dir2>` is the old full path to source files, and `<dir3>` is the new full path to source files. The example command (above) reads the `pgopti.dpi` file, in the location specified in `<dir1>`. For each function represented in the `pgopti.dpi` file, whose source path begins with the `<dir2>` prefix, `profmerge` replaces that prefix with `<dir3>`. The `pgopti.dpi` file is updated with the new source path information.

You can run `profmerge` more than once on a given `pgopti.dpi` file. You may need to do this if the source files are located in multiple directories. For example:

Platform	Command Examples
Linux	<pre>profmerge -prof_dir -src_old /src/prog 1 -src_new /src/prog 2 profmerge -prof_dir -src_old /proj_1 -src_new /proj_2</pre>
Windows	<pre>profmerge -src_old "c:/program files" -src_new "e:/program files" profmerge -src_old c:/proj/application -src_new d:/app</pre>

In the values specified for `-src_old` (Linux) or `/src_old` (Windows) and `-src_new` (Linux) or `/src_new` (Windows), uppercase and lowercase characters are treated as identical. Likewise, forward slash (/) and backward slash (\) characters are treated as identical.

Because the source relocation feature of `profmerge` modifies the `pgopti.dpi` file, you may wish to make a backup copy of the file prior to performing the source relocation.

## proforder Utility

The `proforder` utility is used as part of the feedback compilation phase, to improve program performance. Use `proforder` to generate a function order list for use with the `/ORDER` linker option. The tool uses the following syntax:

Syntax
<code>proforder [-prof_dir <i>dir</i>] [-o <i>file</i>]</code>

where *dir* is the directory containing the profile files (`.dpi` and `.spi`), and *file* is the optional name of the function order list file. The default name is `proford.txt`.

The `-prof_dir` utility option is different than the compiler option: `-prof-dir` (Linux) or `/Qprof-dir` (Windows).

## proforder Options

The `proforder` utility supports the following options:

Option	Description
<code>-help</code>	Lists supported options.
<code>-nologo</code>	Disables version information.
<code>-omit_static</code>	Instructs the tool to omit static functions from function ordering.
<code>-prof_dir <i>dir</i></code>	Specifies the directory where the <code>.spi</code> and <code>.dpi</code> file reside.
<code>-prof_file <i>string</i></code>	Selects the <code>.dpi</code> and <code>.spi</code> files that include the substring value in the file name matching the values passed as <i>string</i> .
<code>-prof_dpi <i>file</i></code>	Specifies the name of the <code>.dpi</code> file.
<code>-prof_spi <i>file</i></code>	Specifies the name of the <code>.spi</code> file.
<code>-o <i>file</i></code>	Specifies an alternate name for the output file.

## Profrun Utility

The `profrun` utility is a tool for collecting data from the Performance Monitoring Unit (PMU) hardware for a CPU.

### Note

Mac OS\*: The `profrun` utility is not supported.

## Profrun Utility Requirements and Behavior

This utility uses the Intel® VTune™ Performance Analyzer Driver to sample events. Therefore, you must have the VTune™ analyzer installed to use the `profrun` utility. The requirements differ depending on platform:

Platform	Requirements
Linux*	<ol style="list-style-type: none"> <li>1. Install Intel® VTune™ Performance Analyzer 3.0 for Linux (or later).</li> <li>2. Grant utility user to the appropriate access to the VTune™ Performance Analyzer Driver.</li> </ol> <p>All users running the utility must be a member of the same group used by the Intel® VTune Performance Analyzer for Linux Driver; the default group is vtune. If another group was specified during installation, add the user to the specified group.</p>
Windows*	<ol style="list-style-type: none"> <li>1. Install Intel® VTune™ Performance Analyzer 7.2 (or later).</li> <li>2. Grant utility user to the appropriate access to the VTune™ Performance Analyzer Driver.</li> </ol> <p>All users running the utility must have Profile Single Process and Profile System Performance rights on the system. Adding the user to either the Administrators or Power Users group should work. Refer to Notes on Windows-family installations in the Intel® VTune Performance Analyzer Release Notes for detailed information.</p> <ol style="list-style-type: none"> <li>3. Specify a local disk as the default directory.</li> </ol>

The utility, in coordination with the analyzer driver, collects samples of events monitored by the PMU and creates a hardware profiling information file (.hpi). The hardware profiling data contained in the file can be used by the Intel® compiler to further enhance optimizations for some programs.

During the initial target program analysis phase, VTune™ analyzer driver creates a file that contains event data for all system-wide processes. By default, the file is named `pgopti.tb5`. Eventually, the `profrun` utility will coalesce the data for the target executable into a significantly smaller `pgopti.hpi` file, and then deletes the `.tb5` file.

### Note

Your system must have sufficient hard disk space to hold the `.tb5` file temporarily. The file size can range widely, plan for as much as 400 MB.

The VTune™ analyzer driver can be used by only one process at a time. The `profrun` utility returns an error if the driver is already in use by another process. By default, `profrun` waits up to 10 minutes for the driver to become available before attempting to access it again.

## Using the profrun Utility

1. Compile your target application using the `-prof-gen-sampling` (Linux\*) or `/Qprof-gen-sampling` (Windows\*) option. The following examples illustrate possible combinations:

Platform	Command Examples
Linux	<code>icc -oMyApp -O2 -prof-gen-sampling source1.c source2.c</code>
Windows	<code>icl /FeMyApp.exe /O2 /Qprof-gen-sampling source1.c source2.c</code>

2. Run the resulting executable by entering a command similar to the following:

Platform	Command Examples
Linux	<code>profrun -dcache MyApp</code>
Windows	<code>profrun -dcache MyApp.exe</code>

This step uses the VTune™ analyzer driver to produce the necessary .hpi file.

3. Compile your target application again; however, during this compilation use the `-prof-use` (Linux) or `/Qprof-use` (Windows) option. The following examples illustrate possible, valid combinations:

Platform	Command Examples
Linux	<code>icc -oMyApp -O2 -prof-use source1.c source2.c</code>
Windows	<code>icl /FeMyApp.exe /O2 /Qprof-use source1.c source2.c</code>

The `-prof-use` (Linux) or `/Qprof-use` (Windows) option instructs the compiler to read the .hpi file and optimize the application using the collected branch sample data.

The `profrun` utility uses the following syntax:

Syntax
<code>profrun -command [argument...] application</code>

where *command* is one or more of the commands listed in the following table, *argument* is one or more of the extended options, and *application* is the command line for running the application, which is usually the application name.

#### Note

Windows\* systems: Unlike the compiler options, which are preceded by forward slash ("/"), the utility options are preceded by a hyphen ("-").

## Profrun Utility Options

The following table summarizes the available `profrun` utility commands, list defaults where applicable, and provides a brief description of each.

Command	Default	Description
<code>-help</code>		Lists the supported tool options.

<code>[sav]</code>	10007	<p>An optional argument supported by the <code>-branch</code>, <code>-dcache</code>, <code>-icache</code> or <code>-event</code> commands.</p> <p>This integer value specifies the sample-after value used for collecting samples. The default value is 10007, which is a moderately-sized prime number. If another value is not specified for <code>sav</code> when using any of the supported commands, the default value is used. Use a prime number for best results.</p> <p>When changing the value, keep the following guideline in mind:</p> <ul style="list-style-type: none"> <li>Decreasing the value to forces the utility to sample more frequently. Frequent sampling results in a more accurate profile, and it a larger output file size, when compared to the file size created by the default value.</li> <li>Increasing the value to forces the utility to sample less frequently. Less frequent sampling results in less accurate profiles, and it produces a relatively smaller output file size.</li> </ul>
<code>-branch[:sav]</code>	10007	<p>Collect branch samples. A sample is taken after every interval, as defined by the <code>sav</code> argument.</p> <p>Use this command to gather information that guides the compiler while doing hot/cold block layout, predicting which direction conditional branches directions, and deciding where it is most profitable to inline functions. Gathering information using this command is similar to using the <code>-prof-gen</code> (Linux) or <code>/Qprof-gen</code> (Windows) option to instrument your program, but using this command is much less intrusive.</p>
<code>-dcache[:sav]</code>	10007	<p>Collect samples of data cache misses. A sample is taken after every interval, as defined by the <code>sav</code> argument.</p> <p>Use this command to gather information to guide the compiler in placing prefetch operations and performing data layout.</p>
<code>-icache[:sav]</code>	10007	Collect samples of misses in the Instruction



		<p>cache. A sample is taken after every interval, as defined by the <i>sav</i> argument.</p> <p>Use this command to gather information to guide the compiler in placing prefetch operations and performing data layout.</p>
-event: <i>eventname</i> [: <i>sav</i> ]	10007	<p>Collect information about valid VTune™ analyzer events; <i>eventname</i> specifies a specific event name. Use this command when <i>-branch</i>, <i>-dcache</i>, or <i>-icache</i> do not apply.</p> <p>Some event names contain embedded spaces. In the case where you can use a period instead of a space. The utility will change periods to spaces before passing the event to VTune™ analyzer.</p> <p>Refer to Intel® VTune™ Performance Analyzer documentation for more information on valid events.</p>
-tb5: <i>file</i>	pgopti.tb5	<p>Specifies the name of the .tb5 file name generated by the VTune™ analyzer driver while it profiles the application. By default, the file resides in the current directory,</p> <p>The specified file will be deleted when <i>profrun</i> completes unless <i>-tb5only</i> is also specified.</p> <p>You might consider overriding this behavior and place the .tb5 file on a disk with more available space.</p>
-tb5only		<p>Produces only the .tb5 file. If this command is not specified the utility will reduce the data into a single .hpi file and delete the .tb5 file.</p>
-wait[: <i>time</i> ]	600 seconds (10 minutes)	<p>Forces the utility to wait for the specified time before attempting to access to the VTune™ analyzer driver. This option is most useful in cases where you anticipate the driver will be busy.</p> <p>Disable the command by specifying a value of 0 (zero).</p>
-hpi: <i>file</i>	pgopti.hpi	<p>Specifies name of the .hpi file containing the profile information for the application. The file resides in the current directory.</p>
-executable: <i>file</i>		<p>Specifies the name of the executable being profiled. By default, this is the first token of the command.</p>

<code>-bufsize: <i>size</i></code>	65536 KB (64 MB)	Specifies the buffer size used in kilobytes (KB).
<code>-sampint: <i>interval</i></code>		Specifies sampling interval in Milliseconds (ms). This command should rarely be needed since all sampling is event-based sampling.
<code>--</code>		Stop parsing options for the tool. Use this if the command name starts with a hyphen.

## Software-based Speculative Precomputation (IA-32)

Software-based Speculative Precomputation (SSP), which is also known as helper-threading optimization, is an optimization technique to improve the performance of some programs by creating helper threads to do data prefetching dynamically.

### Note

Mac OS\*: SSP is not supported, and the associated options listed in this topic are not available.

SSP can be effective for programs where the program performance is dominated by data-cache misses, typically due to pointer-chasing loops. Prefetching data reduces the impact of the long latency to access main memory. The resulting code must run on hardware with shared data cache and multi-threading capabilities to be effective, such as Intel® Pentium® 4 Processors with Hyper-Threading Technology.

## SSP Behavior

SSP is available only in the IA-32 Compiler. SSP directly executes a subset (or slice) of the original program instructions on separate helper threads in parallel with the main computation thread. The helper threads run ahead of the main thread, compute the addresses of future memory accesses, and trigger early cache misses. This behavior hides the memory latency from the main thread.

The command line option to turn on the SSP is `-ssp` (Linux\*) or `/Qssp` (Windows\*). SSP must be used after generating profile feedback information by running the application with a representative set of data. See `profrun` Utility for more information.

When invoked with SSP, the compiler moves through the following stages:

- **Delinquent load identification:** The compiler identifies the top cache-missing loads, which are known as delinquent loads, by examining the feedback information.
- **Loop selection:** The compiler identifies regions of code within which speculative loads will be useful. Delinquent loads typically occur within a heavily traversed loop nest.

- **Program slicing:** Within each region, the compiler looks at each delinquent load and identifies the slices of code required to compute the addresses of the delinquent loads.
- **Helper thread code generation:** The compiler generates code for the slices. Additionally, the compiler generates code to invoke and schedule the helper threads at run-time.

### Caution

Using SSP in conjunction with profiling and interprocedural optimization can degrade the performance of some programs. Experiment with SSP and closely watch the effect of SSP on the target applications before deploying applications using these techniques. Using SSP may also increase compile time.

## Using SSP Optimization

SSP optimization requires several steps; the following procedure demonstrates using SSP optimization in a typical manner.

For the following example, assume that you have the following source files: `a1.c`, `a2.c` and `a3.c`, which will be compiled into an executable named `go` (Linux) or `go.exe` (Windows).

1. Create instrumented code by compiling the application using the `-prof-gen` (Linux) or `/Qprof-gen` (Windows) option to produce an executable with instrumented code, as shown in the examples below:

Platform	Command Examples
Linux	<code>icc -prof-gen a1.c a2.c a3.c -o go</code>
Windows	<code>icl /Qprof-gen a1.c a2.c a3.c /Fego</code>

For more information about the option used in this step, see the following topic:

- o `-prof-gen` compiler option

2. Generate dynamic profile information by running the instrumented program with a representative set of data to create a dynamic profile information file.

Platform	Command Examples
Linux	<code>go</code>
Windows	<code>go.exe</code>

Executing the instrumented application generates a dynamic profile information file with a `.dyn` suffix. You can run the program more than once with different input data. The compiler will merge all of the `.dyn` files into a single `.dpi` file during a later step.

3. Prepare the application for the PMU by recompiling the application using both the `-prof-gen-sampling` and `-prof-use` (Linux) or `/Qprof-gen-sampling` and `/Qprof-use` (Windows) option to produce an executable that can gather information from the hardware Performance Monitoring Unit (PMU). The following command examples show how to combine the options during compilation:

Platform	Command Examples
Linux	<code>icc -prof-gen-sampling -prof-use -O3 -ipo a1.c a2.c a3.c -o go</code>
Windows	<code>icl /Qprof-gen-sampling /Qprof-use /O3 /Qipo a1.c a2.c a3.c /Fego</code>

For more information about the options used in this step, see the following topics:

- o `-prof-gen-sampling` and `-prof-use` compiler options

4. Run the application, using the `profrun` utility, again with a representative set of data to create a file with hardware profile information, including delinquent load information.

Platform	Command Examples
Linux	<code>profrun -dcache go</code>
Windows	<code>profrun -dcache go.exe</code>

This step executes the application and generates file containing hardware profile information; the file resides in the local directory and has a `.hpi` suffix. You can run the program more than once with different input data. The hardware profile information for all runs will be merged automatically.

5. Compile the application a final time using both the `-prof-use` and `-ssp` (Linux) or `/Qprof-use` and `/Qssp` (Windows) options to produce an executable with SSP optimization enabled. The following command examples show how to combine the options during compilation:

Platform	Command Examples
Linux	<code>icc -prof-use -ssp -O3 -ipo a1.c a2.c a3.c -o go</code>
Windows	<code>icl /Qprof-use /Qssp /O3 /Qipo a1.c a2.c a3.c -o go</code>

For more information about the `-ssp` (Linux) or `/Qssp` (Windows) option used in this step, see the following topic in Compiler Options:

- o `-ssp` compiler option

The final step compiles and links the source files with SSP, using the feedback from the instrumented execution phase and the cache miss information from the profiling execution phase.

Since SSP is a profiling- and sampling feedback-based optimization, if the compiler does not find delinquent loads or the performance benefit gained by using helper threads is negligible, the compiler does not generate SSP-specific. In such cases, the compiler will not generate a SSP status message.

See [Profile-guided Optimizations Overview](#).

## HLO Overview

High-level Optimizations (HLO) exploit the properties of source code constructs (for example, loops and arrays) in applications developed in high-level programming languages, such as C++. The high-level optimizations include loop interchange, loop fusion, loop unrolling, loop distribution, unroll-and-jam, blocking, prefetching, scalar replacement, and data layout optimizations.

While the `-O2` (Linux\*) or `/O2` (Windows\*) option performs some high-level optimizations (for example, prefetching, complete unrolling, etc.), using the `-O3` (Linux) or `/O3` (Windows) option provides the best chance for performing loop transformations to optimize memory accesses; the scope of optimizations enabled by these options is different for IA-32, Itanium®, and Intel® EM64T architectures. See Optimization Options Summary.

## IA-32 and Itanium®-based Applications

The `-O3` (Linux) or `/O3` (Windows) option enables the `-O2` (Linux) or `/O2` (Windows) option and adds more aggressive optimizations (like loop transformations); `O3` optimizes for maximum speed, but may not improve performance for some programs.

## IA-32 Applications

In conjunction with the vectorization options, `-ax` and `-x` (Linux) or `/Qax` and `/Qx` (Windows), the `-O3` (Linux) or `/O3` (Windows) option causes the compiler to perform more aggressive data dependency analysis than the default `-O2` (Linux) or `/O2` (Windows). This may result in longer compilation times.

## Tuning Itanium-based Applications

The `-ivdep-parallel` (Linux) or `/Qivdep-parallel` (Windows) option asserts there is no loop-carried dependency in the loop where an IVDEP directive is specified. This is useful for sparse matrix applications.

Follow these steps to tune applications on Itanium®-based systems:

1. Compile your program with `-O3` (Linux) or `/O3` (Windows) and `-ipo` (Linux) or `/Qipo` (Windows). Use profile guided optimization whenever possible. (See Understanding Profile-Guided Optimization.)
2. Identify hot spots in your code. (See Using Intel® Performance Analysis Tools.)
3. Generate a high-level optimization report.
4. Check why loops are not software pipelined.
5. Make the changes indicated by the results of the previous steps.
6. Repeat these steps until you have achieved a satisfactory optimization level.

## Tuning Applications

In general, you can use the following strategies to tune applications:

- Use `#pragma ivdep` to indicate there is no dependence. You might need to compile with the `-ivdep-parallel` (Linux) or `/Qivdep-parallel` (Windows) option to absolutely specify no loop-carried dependence.
- Use `#pragma swp` to enable software pipelining (useful for loop-sided controls and unknown loop count).
- Use `#pragma loop count(n)` when needed.
- Use of `-ansi-alias` (Linux) or `/Qansi-alias` (Windows) is helpful.
- Add the `restrict` keyword to insure there is no aliasing. Compile with `-restrict` (Linux) or `/Qrestrict` (Windows).
- Use `-alias-args` (Linux) or `/Qalias-args-` (Windows) to indicate arguments are not aliased.
- Use `#pragma distribute point` to split large loops (normally this is done automatically).
- For C code, do not use unsigned int for loop indexes. HLO may skip optimization due to possible subscripts overflow. If upper bounds are pointer references, assign it to a local variable whenever possible.
- Check that the prefetch distance is correct. Use `#pragma prefetch` to override the distance when it is needed.

## Loop Transformations

Within HLO, loop transformation techniques include:

- Loop Permutation or Interchange
- Loop Distribution
- Loop Fusion
- Loop Unrolling
- Data Prefetching
- Scalar Replacement
- Unroll and Jam
- Loop Blocking or Tiling
- Partial-Sum Optimization
- Loadpair Optimization
- Predicate Optimization
- Loop Versioning with Runtime Data-Dependence Check (Itanium®-based systems only)
- Loop Versioning with Low Trip-Count Check
- Loop Reversal
- Profile-Guided Loop Unrolling
- Loop Peeling
- Data Transformation: Malloc Combining and Memset Combining
- Loop Rerolling
- Memset and Malloc Recognition
- Statement Sinking for Creating Perfect Loopnests

## Scalar Replacement

The goal of scalar replacement, which is enabled by `-scalar-rep` (Linux\*) or `/Qscalar-rep` (Windows\*), is to reduce memory references. This is done mainly by replacing array references with register references.

While the compiler replaces some array references with register references when `-O1` or `-O2` (Linux) or `/O1` or `/O2` (Windows) is specified, more aggressive replacement is performed when `-O3` (Linux) or `/O3` (Windows) and `-scalar-rep` (Linux) or `/Qscalar-rep` (Windows) are specified. For example, with `-O3` (Linux) or `/O3` (Windows) the compiler attempts replacement when there are loop-carried dependencies or when data dependency analysis is required for memory disambiguation.

The `-scalar-rep` (Linux) or `/Qscalar-rep` (Windows) compiler option enables (default) scalar replacement performed during loop transformations.

## Absence of Loop-carried Memory Dependency with IVDEP Directive

For Itanium®-based applications, the `-ivdep-parallel` (Linux\*) or `/Qivdep-parallel` (Windows\*) option indicates there is no loop-carried memory dependency in the loop where an `IVDEP` pragma is specified. This technique is useful for some sparse matrix applications.

### Note

Mac OS\*: This option is not supported.

For example, the following loop requires the `parallel` option in addition to the directive `ivdep` to indicate there is no loop-carried dependencies:

### Example

```
#pragma ivdep
for (i=1; i<n; i++)
{
    e[ix[2][i]] = e[ix[2][i]]+1.0;
    e[ix[3][i]] = e[ix[3][i]]+2.0;
}
```

For example, the following loop requires the `parallel` option in addition to the `IVDEP` pragma to ensure there is no loop-carried dependency for the store into `a()`.

### Example

```
#pragma ivdep
for (j=0; j<n; j++)
{
    a[b[j]] = a[b[j]] + 1;
}
```



## prefetch Directive

The `prefetch` directive is supported on Itanium®-based systems only.

The syntax is

- `#pragma prefetch var:hint:distance`

where `hint` value can be 0 (T0), 1 (NT1), 2 (NT2), or 3 (NTA).

The following example demonstrates how to use the `prefetch` directive.

Example
<pre>for (i=i0; i!=i1; i+=is) {     float sum = b[i];     int ip = srow[i];     int c = col[ip];      #pragma noprefetch col     #pragma prefetch value:1:80     #pragma prefetch x:1:40      for(; ip&lt;srow[i+1]; c=col[++ip])         sum -= value[ip] * x[c];     y[i] = sum; }</pre>

See also Vectorization support.

## Loop Unrolling

The benefits of loop unrolling are as follows:

- Unrolling eliminates branches and some of the code.
- Unrolling enables you to aggressively schedule (or pipeline) the loop to hide latencies if you have enough free registers to keep variables live.
- The Pentium® 4 and Intel® Xeon® processors can correctly predict the exit branch for an inner loop that has 16 or fewer iterations, if that number of iterations is predictable and there are no conditional branches in the loop. Therefore, if the loop body size is not excessive, and the probable number of iterations is known, unroll inner loops for:
  - Pentium 4 processors, until they have a maximum of 16 iterations
  - Pentium III or Pentium II processors, until they have a maximum of 4 iterations

A potential limitation is that excessive unrolling, or unrolling of very large loops, can lead to increased code size. For more information on how to optimize with the `-unroll [n]` (Linux\*) or `/Qunroll [n]` (Windows\*) option, refer to the *Intel® Pentium® 4 and Intel® Xeon® Processor Optimization Reference Manual*.

The `-unroll[n]` (Linux\*) or `/Qunroll[n]` (Windows\*) option controls how the Intel® compiler handles loop unrolling. The following table summarizes how to use this option:

Windows	Linux	Description
<code>/Qunrolln</code>  Synonym: <code>/unroll[:n]</code>	<code>-unrolln</code>	Specifies the maximum number of times you want to unroll a loop. The following examples unrolls a loop at most four times:  <pre>icpc -unroll4 a.cpp (Linux)</pre> <pre>icl /Qunroll4 a.cpp (Windows)</pre> <p><b>Note</b></p> <p>The Itanium® compiler currently recognizes only <math>n = 0</math>; any other value is ignored.</p>
<code>/Qunroll</code>	<code>-unroll</code>	Omitting a value for $n$ lets the compiler decide whether to perform unrolling or not. This is the default; the compiler uses default heuristics or defines $n$ .
<code>/Qunroll0</code>	<code>-unroll0</code>	Disables the loop unroller. To disable loop unrolling, specify $n$ as 0. The following examples disables loop unrolling:  <pre>icpc -unroll0 a.cpp (Linux)</pre> <pre>icl /Qunroll0 a.cpp (Windows)</pre>

For more information about the option behaviors listed above, see the following topic:

- `-unroll` compiler option

## Loop Independence

Loop independence is important since loops that are independent can be parallelized. Independent loops can be parallelized in a number of ways, from the course-grained parallelism of OpenMP\*, to fine-grained Instruction Level Parallelism (ILP) of vectorization and software pipelining.

Loops are considered independent when the computation of iteration  $Y$  of a loop can be done independently of the computation of iteration  $X$ . In other words, if iteration 1 of a loop can be computed and iteration 2 simultaneously could be computed without using any result from iteration 1, then the loops are independent.

Occasionally, you can determine if a loop is independent by comparing results from the output of the loop with results from the same loop written with a decrementing index counter.

For example, the loop shown in example 1 might be independent if the code in example 2 generates the same result (in both cases, assume `MAX` is defined as 1024).

### Example

```
#define MAX 1024
void loop indep1(int a[MAX],int b[MAX])
{
    for (int j=0;j<MAX;j++)
        a[j] = b[j];
}
#define MAX 1024
void loop indep2(int a[MAX],int b[MAX])
{
    for (int j=MAX;j>0;j--)
        a[j] = b[j];
}
```

When loops are dependent, improving loop performance becomes much more difficult. Loops can be dependent in several, general ways.

- Flow Dependency
- Anti Dependency
- Output Dependency
- Reductions

The following sections illustrate the different loop dependencies.

## Flow Dependency - Read After Write

Cross-iteration flow dependence is created when variables are read then written in different iterations, as shown in the following example:

### Example

```
void flow dep(double A[])
{
    for (int j=1; j<1024; j++)
        A[j]=A[j-1];
}
```

The above example is equivalent to the following lines for the first few iterations:

### Sample iterations

```
A[1]=A[0];
A[2]=A[1];
```

Recurrence relations feed information forward from one iteration to the next.

### Example

```
void time stepping loops(double a[], double b[])
{
    for(int j=1; j<MAX; j++) {
```

```

    a[j] = a[j-1] + b[j];
}
}

```

Most recurrences cannot be made fully parallel. Instead, look for a loop further out or further in to parallelize. You might be able to get more performance gains through unrolling.

## Anti Dependency - Write After Read

Cross-iteration anti-dependence is created when variables are written then read in different iterations, as shown in the following example:

### Example

```

void anti_dep1(double A[])
{
    for (int j=1; j<1024; j++)
        A[j]=A[j+1];
}

```

The above example is equivalent to the following lines for the first few iterations:

### Sample iterations

```

A[1]=A[2];
A[2]=A[3];

```

## Output Dependency - Write After Write

Cross-iteration output dependence is where variables are written then rewritten in a different iteration. The following example illustrates this type of dependency:

### Example

```

void anti_dep2(double A[], double B[], double C[])
{
    for (int j=1; j<1024; j++) {
        A[j]=B[j];
        A[j+1]=C[j];
    }
}

```

The above example is equivalent to the following lines for the first few iterations:

### Sample iterations

```

A[1]=B[1];
A[2]=C[1];
A[2]=B[2];
A[3]=C[2];

```

## Reductions

The Intel® compiler can successfully vectorize or software pipeline (SWP) most loops containing reductions on simple math operators like multiplication (\*), addition (+), subtraction (-), and division (/). Reductions collapse array data to scalar data by using associative operations:

### Example

```
void reduction(double * sum, double c[])
{
    for (int j=0; j<MAX; j++) {
        *sum = *sum + c[j];
    }
}
```

The compiler might occasionally misidentify a reduction and report flow-, anti-, output-dependencies or sometimes loop-carried memory-dependency-edges; in such cases, the compiler will not vectorize or SWP the loop. In such cases, recognize that the programming construct is simply a reduction, and direct the compiler through the use of pragmas to vectorize or SWP the loop; you can pragmas (like, `#pragma ivdep` or `#pragma swp`) to help the compiler in these cases.

## Prefetching with Options

The goal of prefetch insertion optimization is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache. The prefetch optimization is enabled or disabled by the `-prefetch` (Linux\*) or `/Qprefetch` (Windows\*) compiler option.

### Note

Mac OS\*: This option is not supported.

To facilitate compiler optimization:

- Minimize use of global variables and pointers.
- Minimize use of complex control flow.
- Choose data types carefully and avoid type casting.

To use this option, you must also specify `-O3` (Linux) or `/O3` (Windows).

For more information on how to optimize using this option, refer to the *Intel® Pentium® 4 and Intel® Xeon® Processor Optimization Reference Manual*. Additionally, see the following topic:

- `-prefetch` compiler option

## Floating-point Arithmetic Optimizations Overview

This section summarizes compiler options that provide optimizations for floating-point data and floating-point precision on IA-32, Intel® EM64T, and Intel® Itanium® architectures. The topics include the following:

- Floating-point options for multiple architectures
- Floating-point options for IA-32 and Intel® EM64T architectures
- Floating-point options for Itanium® architectures
- Improving or restricting floating-point arithmetic precision
- Understanding Floating-Point Performance

## Floating-point Options for Multiple Architectures

The options described in this topic provide optimizations with varying degrees of precision in floating-point calculations for the IA-32, Intel® EM64T, and Itanium® architectures. Where options are not universally supported on all architectures, the description lists the supported architecture.

Using the options listed below can reduce application performance. In general, to achieve greater application performance you might need to sacrifice some degree of floating-point accuracy.

The floating point options listed in this topic provide optimizations with varying degrees of precision in floating-point arithmetic; `-fpm` (Linux\*) or `/fp` (Windows\*) disables these optimizations.

Windows*	Linux*	Description
<code>/fp</code>	<code>-fp-model</code>	<p>Specifies semantics used in floating-point calculations.</p> <p>The default model is <code>fast</code>, which enables aggressive optimizations when implementing floating-point calculations. The optimizations increase speed, but might affect floating-point computation accuracy.</p> <p>See the following topic for detailed descriptions of the different models and examples:</p> <ul style="list-style-type: none"> <li>• <code>-fp-model</code> compiler option</li> </ul>
<code>/Op</code>	<code>-mp</code>	<p>Limits floating point optimizations and maintains declared precision. The option might slightly reduce execution speed. Use option <code>-fp-model</code> (Linux) or <code>/fp</code> (Windows) instead.</p> <p>IA-32 and Intel® EM64T:</p> <ul style="list-style-type: none"> <li>• In general, the option maintains maximum precision</li> </ul>

		<p>not the declared precision.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-mp</code> compiler option</li> </ul> <p>Using <code>-fp-model</code> (Linux) or <code>/fp</code> (Windows) affects the behavior of this option.</p>
<code>/Qprec</code>	<code>-mp1</code>	<p>Improves floating-point precision; this option has less impact to performance and disables fewer optimizations than the <code>-fp-model precise</code> (Linux) or <code>/fp:precise</code> (Windows) option.</p> <p>This option prevents the compiler from performing optimizations which change NaN comparison semantics; also, the option causes all values used in comparisons to be truncated to declared precision prior to use in the comparison. Furthermore, the option insures the use of library routines, which provide more accurate results compared to the X87 transcendental instructions. Finally, the option causes the Intel® Compiler to use precise divide and square root operations.</p> <p>The <code>-fp-model precise</code> (Linux) or <code>/fp:precise</code> (Windows) option implies this option; however, <code>-fp-model fast</code> (Linux) or <code>/fp:fast</code> (Windows) disables this option; however, <code>-fp-model fast</code> (Linux) or <code>/fp:fast</code> (Windows) will not override this option if both are specified on the same command line.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-mp1</code> compiler option</li> </ul>

## Floating-point Options for IA-32 and Intel® EM64T

The following table lists options that enable you to control the compiler optimizations for floating-point computations on IA-32 and Intel® EM64T systems. The options listed here are not valid for Itanium®-based systems.

Windows*	Linux*	Effect
<code>/Qprec-div</code>	<code>-prec-div</code>	Attempts to use faster but less accurate implementation of single precision floating-point divide. Use this option to disable the divide optimizations in cases where it is important to maintain the full range and precision for floating-point division. Using this option results in greater accuracy with some loss of performance.

		<p>When using <code>-fp-model precise</code> (Linux*) or <code>/fp:precise</code> (Windows*) the divide optimization is disabled.</p> <p>Use <code>-no-prec-div</code> (Linux) or <code>/Qprec-div-</code> (Windows) to enable the divide optimizations. Additionally, <code>-no-prec-div</code> enables the division-to-multiplication by reciprocal optimization; <code>-fast</code> implies <code>-no-prec-div</code>.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-prec-div</code> compiler option</li> </ul>
<code>/Qpc</code>	<code>-pc</code>	<p>Changes the floating point precision control when the <code>PROGRAM</code> function is compiled. The program that uses this option must use <code>PROGRAM</code> as its entry point, and the file containing <code>PROGRAM</code> must be compiled using this option.</p> <p>A change of the default precision control or rounding mode (for example, by using the <code>-pc32</code> (Linux) or <code>/Qpc32</code> (Windows) flag or by user intervention) may affect the results returned by some of the mathematical functions. Some flags are not compatible with specific <code>-fp-model</code> (Linux) or <code>/fp</code> (Windows) settings, which use expression evaluation following C99 <code>FLT_EVAL_METHOD=1</code> or <code>FLT_EVAL_METHOD=2</code>:</p> <ul style="list-style-type: none"> <li>• For <code>-pc32</code> (Linux) or <code>/Qpc32</code> (Windows), the non-compatible options are <code>-fp-model precise</code>, <code>-fp-model double</code>, <code>-fp-model extended</code>, and <code>-fp-model strict</code> (Linux) or <code>/fp:precise</code>, <code>/fp:double</code>, <code>/fp:extended</code>, and <code>/fp:strict</code> (Windows).</li> <li>• For <code>-pc64</code> (Linux) or <code>/Qpc64</code> (Windows), the non-compatible option is <code>-fp-model extended</code> (Linux) or <code>/fp:extended</code> (Windows).</li> </ul> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-pc</code> compiler option</li> </ul>
<code>/Qrcd</code>	<code>-rcd</code>	<p>Disables rounding mode changes for floating-point-to-integer conversions.</p> <p>The system default floating point rounding mode is round-to-nearest. This means that values are rounded during floating point calculations; however, the C/C++ language requires floating point values to be truncated when a conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating point conversion and change it back afterwards.</p>



		<p>This option disables the change to truncation of the rounding mode for all floating point calculations, including floating-point-to-integer conversions. This behavior means that all floating point calculations must use the default round-to-nearest, including floating point-to-integer conversions. Turning on this option can improve performance, but floating-point conversions to integer will not conform to C/C++ semantics.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-rcd</code> compiler option</li> </ul>
<code>/Qfp-port</code>	<code>-fp-port</code>	<p>Provides a rounding control of the results of the floating-point data operations. Using this option might cause some speed impact, but it also makes sure that rounding to the user-declared precision at assignments is always done.</p> <ul style="list-style-type: none"> <li>• Linux: <code>-fp-port</code> is supported on both IA-32 and Intel® EM64T systems.</li> <li>• Windows: <code>/Qfp-port</code> is supported on IA-32 only.</li> </ul> <p>The <code>-fp-model precise</code> (Linux) or <code>/fp:precise</code> (Windows) option implies this option; however, <code>-fp-model fast</code> (Linux) or <code>/fp:fast</code> (Windows) disables this option; however, <code>-fp-model fast</code> (Linux) or <code>/fp:fast</code> (Windows) will not override this option if both are specified on the same command line.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-fp-port</code> compiler option</li> </ul>
<code>/Qprec-sqrt</code>	<code>-prec-sqrt</code>	<p>Improves precision of square root implementations, but using the option might impact speed.</p> <p>The compiler uses a fast but less accurate implementation of single precision floating-point square root. In cases where it is important to maintain full accuracy for square root calculations, use this option to disable the square root optimization.</p> <p>When using <code>-fp-model precise</code> (Linux) or <code>/fp:precise</code> (Windows) the square root optimization is disabled. Use <code>-no-prec-sqrt</code> (Linux) or <code>/Qprec-sqrt-</code> (Windows) to enable the square root optimization.</p>

		<p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-prec-sqrt</code> compiler option</li> </ul>
--	--	---

## Floating-point Options for Itanium®-based Systems

The following table lists options that enable you to control the compiler optimizations for floating-point computations on Itanium®-based systems. The options listed here are not valid for IA-32 and Intel® EM64T systems.

### Note

Mac OS\*: The options listed in this topic are not supported.

Windows*	Linux*	Effect
/Qftz	-ftz	<p>Use this option if the denormal values are not critical to application behavior. Flushes denormal results to zero (0) when the application is in the gradual underflow mode. Flushing the denormal values to zero with this option may improve overall application performance.</p> <p>Use this option only on the source that contains the <code>main</code> program to turn the FTZ mode on. The initial thread, and any threads subsequently created by that process, will operate in FTZ mode.</p> <p>There may be performance issues when using SSE instructions. Refer to the compiler option topic below for other SSE-specific behaviors.</p> <ul style="list-style-type: none"> <li>• This option is disabled when using <code>-fp-model precise</code> (Linux) or <code>/fp:precise</code> (Windows*). Combine this option with <code>with -fp-model precise</code> (Linux) or <code>/fp:precise</code> (Windows) to flush denormal results to zero.</li> <li>• This option affects the result of abrupt underflow by setting the floating underflow to zero (0) and allowing the execution to continue. The option instructs the compiler to treat denormal values used in a computation as zero (0) so no floating invalid exception occurs.</li> <li>• Using the <code>-O3</code> (Linux) or <code>/O3</code> (Windows) option sets the abrupt underflow to zero (enables this option). At lower optimization levels, gradual underflow to zero (0) is the default behavior.</li> </ul>

		<ul style="list-style-type: none"> <li>Gradual underflow to zero (0) can degrade performance. Using higher optimization levels to get the default abrupt underflow or explicitly setting this option improves performance. The option may improve performance on Itanium® 2 processor, even in the absence of actual underflow, most frequently for single-precision code.</li> </ul> <p>If this option produces undesirable results of the numerical behavior of your program, you can turn the FTZ mode off by using this option in the command line while still benefiting from other optimizations.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li><code>-ftz</code> compiler option</li> </ul>
<code>/QIPF-fma</code>	<code>-IPF-fma</code>	<p>Enables or disables the contraction of floating-point multiply and add/subtract operations into a single operation. Unless <code>-fp-model strict</code> (Linux) or <code>/fp:strict</code> (Windows) is specified, the compiler contracts these operations whenever possible. The <code>-fp-model strict</code> (Linux) or <code>/fp:strict</code> (Windows) option disables the contractions.</p> <p>For example, a combination of <code>-fp-model strict</code> (Linux) or <code>/fp:strict</code> (Windows) and <code>-IPF-fma</code> (Linux) or <code>/QIPF-fma</code> (Windows) enables the compiler to contract operations:</p> <ul style="list-style-type: none"> <li><code>icpc -fp-model strict -IPF-fma prog.cpp</code> (Linux)</li> <li><code>icl /fp:strict /QIPF-fma prog.cpp</code> (Windows)</li> </ul> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li><code>-IPF-fma</code> compiler option</li> </ul>
<code>/QIPF-fp-speculation</code>	<code>-IPF-fp-speculation</code>	<p>Default. Sets the compiler to speculate on floating-point operations.</p> <p>When using the <code>-fp-model strict</code> or <code>-fp-model except</code> (Linux) or <code>/fp:strict</code> or <code>/fp:except</code> options, the speculation mode is set to strict and cannot be overridden; however, when using the <code>-fp-model fast</code> (Linux) or <code>/fp:fast</code> (Windows) option, you can use the <code>-IPF-fp-speculation</code> (Linux) or <code>/QIPF-fp-speculation</code> (Windows) option to restrict speculation.</p> <p>For more information, see the following topic:</p>

		<ul style="list-style-type: none"> <li>• <code>-IPF-fp-speculation</code> compiler option</li> </ul>
<code>/QIPF-fp-relaxed</code>	<code>-IPF-fp-relaxed</code>	<p>Enables use of faster but slightly less accurate code sequences for math functions, such as the <code>sqrt()</code> function and the divide operation. As compared to strict IEEE* precision, using this option slightly reduces the accuracy of floating-point calculations performed by these functions, usually limited to the least significant digit.</p> <p>When using any <code>-fp-model</code> (Linux) or <code>/fp</code> (Windows) option setting, this option is disabled: <code>-no-IPF-fp-relaxed</code> (Linux) or <code>/QIPF_fp_relaxed-</code> (Windows); however, <code>-fp-model</code> (Linux) or <code>/fp</code> (Windows) does not override the explicit setting.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-IPF-fp-relaxed</code> compiler option</li> </ul>

## Improving or Restricting FP Arithmetic Precision

For most programs, using options to improve floating-point (FP) precision adversely affects performance. In general, to achieve greater performance, it may be necessary to sacrifice some degree of floating-point accuracy.

If you are not sure whether your application needs the compiler supported options, try compiling and running your program both with and without the options to evaluate the effects on performance versus precision.

While there are several floating-point related options, the recommended method to control the semantics of floating-point calculations is to use the `-fp-model` (Linux\*) or `/fp` (Windows\*) option. See the following topic for detailed descriptions of the different models and examples:

- `-fp-model` compiler option

## Understanding Floating-point Performance

### Denormal Computations

A denormal number is where the mantissa is non zero, but the exponent value is zero in an IEEE\* floating-point representation. The smallest normal single precision floating point number greater than zero is about `1.175494350822288e-38`. Smaller numbers are possible, but are denormal and take hardware or operating system intervention to handle them, which can cost hundreds of clock cycles.

In many cases, denormal numbers are evidence of an algorithm problem where a poor choice of algorithms is causing excessive computation in the denormal range. There are several ways to handle denormal numbers. For example, you can translate to normal, which means to multiply by a large scalar number, do the remaining computations in the normal space, then scale back down to denormal range. Do this whenever the small denormal values benefit the program design. In many cases, denormals that can be considered to be zero may be flushed to zero.

Denormals are computed in software on Itanium® processors. Hundreds of clock cycles are required, resulting in excessive kernel time. Attempt to understand why denormal results occur, and determine if they are justified. If you determine they are not justified, then use the following suggestions to handle the results:

- Translate to normal problem by scaling values.
- Increase precision and range by using a wider data type.
- Set flush-to-zero mode in floating-point control register: `-ftz` (Linux\*) or `/Qftz` (Windows\*).

Denormal numbers always indicate a loss of precision, an underflow condition, and usually an error (or at least a less than desirable condition). On the Intel® Pentium® 4 processor and the Intel Itanium® processor, floating-point computations that generate denormal results can be set to zero, improving the performance.

The Intel compiler disables the FTZ and DAZ bits when you specify value-safe options, including the `strict`, `precise`, `source`, `double`, and `extended` models supported by the `-fp-model` (Linux\*) or `/fp` (Windows\*) option.

## IA-32 compiler

The IA-32 compiler does not support the `-ftz` (Linux) or `/Qftz` (Windows) option; however, `-xK` or `-xW` (Linux) or `/QxK` or `/QxW` (Windows) will set the Flush-To-Zero mode in the SSE control register, which is the preferred approach. The only other way to enable Flush-to-Zero mode on an Intel® Pentium® 4 processor is to manually program the SSE2 Control Register as illustrated in the following example:

### Example

```
void SIMDFlushToZero (void)
{
    DWORD SIMDCtrl;
    asm
    {
        STMXCSR SIMDCtrl
        mov eax, SIMDCtrl
        // flush-to-zero = bit 15
        // mask underflow = bit 11
        // denormals are zero = bit 6
        or eax, 08840h
        mov SIMDCtrl, eax
        LDMXCSR SIMDCtrl
    }
}
```

## Note

Mac OS\*: The `-ftz` option is not supported.

Refer to IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture (<http://www.intel.com/design/pentiumii/manuals/243190.htm>) for more details about flush to zero or specific bit field settings.

## Itanium® compiler

The Itanium® compiler supports the `-ftz` (Linux) or `/Qftz` (Windows) option used to flush denormal results to zero when the application is in the gradual underflow mode. Use this option if the denormal values are not critical to application behavior. The default status of the option is OFF. By default, the compiler lets results gradually underflow.

Use the `-ftz` (Linux) or `/Qftz` (Windows) on the source containing `main()`; the option turns on the Flush-to-Zero (FTZ) mode for the process started by `main()`. The initial thread, and any threads subsequently created by that process, will operate in FTZ mode.

By default, the `-O3` (Linux) or `/O3` (Windows) option enables FTZ; in contrast, the `-O2` (Linux) or `/O2` (Windows) option disables FTZ. Alternately, you can use `-no-ftz` (Linux) or `/Qftz-` (Windows) to disable flushing denormal results to zero (DAZ).

For detailed optimization information related to microarchitectural optimization and cycle accounting, refer to *Introduction to Microarchitectural Optimization for Itanium® 2 Processors Reference Manual* also known as “Software Optimization book” document number 251464-001 located at [http://www.intel.com/software/products/vtune/techtopic/software\\_optimization.pdf](http://www.intel.com/software/products/vtune/techtopic/software_optimization.pdf).

## Inexact Floating Point Comparisons

Some floating point applications exhibit extremely poor performance by not terminating. The applications do not terminate, in many cases, because exact floating-point comparisons were made against a given value. The following examples demonstrate the concept:

### Example

```
if (foo() == 2.0)
```

Where `foo()` may be as close to 2.0 as can be imagined without actually exactly matching 2.0. You can improve the performance of such codes by using inexact floating point comparisons or fuzzy comparisons to test a value to within a certain tolerance, as shown below:

### Example

```
epsilon = 1E-8;  
if (abs(foo() - 2.0) <= epsilon)
```

## Compiler Reports Overview

The Intel® compiler provides several reports that can help identify non-optimal performance. Some of these optimizer reports are platform-specific, others are more general. Start with the general reports that are common to all platforms, then use the reports unique to a given architecture. This section discusses the following concepts and reports:

- Optimizer Report Generation
- High-Level Optimization (HLO) Report
- Interprocedural Optimizations (IPO) Report
- Software Pipelining (SWP) Report
- Vectorization Report

## Optimizer Report Generation

This topic discusses the compiler options related to creating optimization reports, typical report generation syntax, and optimization phases available for reporting.

The Intel® compiler provides the following options to generate and manage optimization reports:

Windows*	Linux*	Description
/Qopt-report	-opt-report	Generates an optimization report and directs it to <code>stderr</code> . By default, the compiler does not generate optimization reports.  For more information, see the following topic: <ul style="list-style-type: none"> <li>• <code>-opt-report</code> compiler option</li> </ul>
/Qopt-report-phase	-opt-report-phase	Specifies the optimization phase to use when generating reports. See <a href="#">Specifying Optimizations to Generate Reports</a> (below) for information about supported phases.  For additional information, see the following topic: <ul style="list-style-type: none"> <li>• <code>-opt-report-phase</code> compiler option</li> </ul>
/Qopt-report-file	-opt-report-file	Generates an optimization report and directs the report output to the

		<p>specified file name. If the file is not in the local directory, supply the full path to the output file.</p> <p>This option overrides the <code>opt-report</code> option.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-opt-report-file</code> compiler option</li> </ul>
<code>/Qopt-report-level</code>	<code>-opt-report-level</code>	<p>Specifies the detail level in the optimization report. The <i>min</i> argument provides the minimal summary and <i>max</i> produces the full report.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-opt-report-level</code> compiler option</li> </ul>
<code>/Qopt-report-routine</code>	<code>-opt-report-routine</code>	<p>Generates reports from all routines with names containing a string as part of their name; pass the string as an argument to this option. If not specified, the compiler will generate reports on all routines.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <li>• <code>-opt-report-routine</code> compiler option</li> </ul>

Use syntax similar to the following to generate optimization reports.

Platform	Sample Syntax
Linux	<code>icc -opt-report -opt-report-phase all prog.cpp</code>
Windows	<code>icl /Qopt-report /Qopt-report-phaseall prog.cpp</code>

These example commands instruct the compiler to generate a report and send the results to `stderr` and specifies that the reports should include information about all available optimizers.



In most cases, specifying `all` as the phase will generate too much information to be useful.

If you want to capture the report in an output file instead of sending it to `stderr`, specify `-opt-report-file` (Linux) or `/Qopt-report-file` (Windows) and indicate an output file name. Specify `-c` (Linux) or `/c` (Windows) to instruct the compiler to not invoke the linker; the compiler stops after generating object code.

## Specifying Optimizations to Generate Reports

The compiler can generate reports for the optimizer you specify in the *phase* argument of the `-opt-report-phase` (Linux) or `/Qopt-report-phase` (Windows) option. The following optimizers are supported:

Optimizer Phase	High-level Categories
<code>all</code>	All phases
<code>ecg</code>	Itanium® Compiler Code Generator  <b>Note</b>  Mac OS*: This phase is not supported.
<code>hlo</code>	High-level Language Optimizer
<code>ilo</code>	Intermediate Language Scalar Optimizer
<code>ipo</code>	Interprocedural Optimizer
<code>pgo</code>	Profile-Guided Optimizer

See Interprocedural Optimizations (IPO) Report, High-Level Optimization (HLO) Report, and Software Pipelining (SWP) Report for examples of using these optimization reports.

When one of the logical names is specified, as shown above, the compiler generates all reports from that optimizer. The option can be used multiple times on the same command line to generate reports for multiple optimizers. For example, for if you specified `-opt-report-phase ipo -opt-report-phase hlo` (Linux) or `/Qopt-report-phase ipo /Qopt-report-phase hlo` (Windows) the compiler generates reports from the interprocedural optimizer and the high-level optimizer code generator.

The following table shows the optimizations available by architecture.

Architecture	Supported Optimizers
IA-32, Intel® EM64T, and Itanium®-based systems	<ul style="list-style-type: none"> <li><code>ilo</code> and <code>pgo</code></li> <li><code>hlo</code>: supported only if <code>-O3</code> (Linux) or <code>/O3</code> (Windows) option is specified.</li> <li><code>ipo</code>: supported only if interprocedural optimizer is invoked with <code>-ip</code> or <code>-ipo</code> (Linux) or <code>/Qip</code> or <code>/Qipo</code> (Windows).</li> <li><code>all</code>: the above optimizers if <code>-O3</code> (Linux) or <code>/O3</code></li> </ul>

	(Windows) and <code>-ip</code> or <code>-ipo</code> (Linux) or <code>/Qip</code> or <code>/Qipo</code> (Windows) are specified.
Itanium®-based systems only	<ul style="list-style-type: none"> <li>all: the above optimizers if <code>-O3</code> (Linux) or <code>/O3</code> (Windows) and <code>-ip</code> or <code>-ipo</code> (Linux) or <code>/Qip</code> or <code>/Qipo</code> (Windows) are specified</li> <li>hlo and ipo: if either a hlo or ipo is specified, but the controlling option, <code>-O3</code> (Linux) or <code>/O3</code> (Windows) or <code>-ip</code> and <code>-ipo</code> (Linux) or <code>/Qip</code> and <code>/Qipo</code> (Windows), is not enabled, the compiler generates an empty report.</li> <li>ecg</li> </ul>

Each of the optimizer logical names supports many specific, targeted optimizations within them. However, each of the targeted optimizations have the prefix of the optimizer name. Enter `-opt-report-help` (Linux) or `/Qopt-report-help` (Windows) to list the names of optimizers that are supported. The following table lists some examples:

Optimizer optimization	Description
ipo_inl	Interprocedural Optimizer, inline expansion of functions
ipo_cp	Interprocedural Optimizer, constant propagation
hlo_unroll	High-level Optimizer, loop unrolling
hlo_prefetch	High-level Optimizer, prefetching
ecg_swp	Itanium®-based Compiler Code Generator, software pipelining

The entire name for a particular optimization within an optimizer need not be fully specified; in many cases, the first few characters should suffice to generate reports. All optimization reports that have a matching prefix are generated.

## Viewing the Optimization Reports Graphically (Linux)

In addition to the text-based optimization reports, the Intel® compiler can now generate the necessary information to allow the OptReport feature in Intel® VTune™ Linux 8.x to display the optimization reports graphically. See the VTune™ Performance Analyzer documentation for details.

To generate the graphical report display, you must do the following:

1. Set the `VT_ENABLE_OPTIMIZATION_REPORT` environment variable to ON. (This is an VTune™ Performance Analyzer environment setting, which is not required for the compiler to generate text-based optimization reports.)
2. Compile the application using the following optimization reports options, at a minimum: `-opt-report-phase` and `-opt-report-file`.

As with the text-based reports, the graphical report information can be generated on all architectures; however, you can only view the graphical reports on IA-32 and Intel EM64T architectures.

## High-Level Optimization (HLO) Report

Run the High-level Optimization (HLO) report by entering a command similar to the following:

Platform	Example Command
Linux*	<code>icpc -opt-report -opt-report-phase hlo -O3 a.c b.c</code>
Windows*	<code>icl /Qopt-report /Qopt-report-phase hlo /O3 a.c b.c</code>

HLO performs specific optimizations based on the usefulness and applicability of each optimization. See Loop Transformations for specific optimizations HLO performs.

The HLO report provides information on all relevant areas plus structure splitting and loop-carried scalar replacement. The following is an example of the HLO report for a matrix multiply program:

Example
<pre>multiply d lx.c HLO REPORT LOG OPENED ===== Total #of lines prefetched in multiply d for loop in line 15=2, dist=74 Block, Unroll, Jam Report: (loop line numbers, unroll factors and type of transformation) Loop at line 15 unrolled without remainder by 8 ===== ... -out:matrix1.exe</pre>

These report results demonstrate the following information:

- There were 2 cache lines prefetched 74 loop iterations ahead, that is, with a distance of 74. The prefetch instruction corresponds to line 15 of the source code.
- The compiler has unrolled the loop at line 15 eight times.

Manual optimization techniques, like manual cache blocking, should be generally avoided and used only as a last resort.

The HLO report tells you explicitly what loop transformations the compiler performed. By not mentioning a given loop transformation, the report might imply, by omission, that there are transformations the developer might perform. Some transformation that you might want to try are listed in the following table.

Transformation	Description
Distribution	Distribute or split up one large loop into two smaller loops. This

	may be advantageous when too many registers are being consumed in a given large loop.
Interchange	Swap the order of execution of two nested loops to gain a cache locality or Unit Stride access performance advantage.
Fusion	Fuse two smaller loops with the same trip count together to improve data locality
Block	Cache blocking arranges a loop so that it will perform as many computations as possible on data already residing in cache. The next block of data is not read into cache until all computations with the first block are finished.
Unroll	Disassemble the loop structure. Unrolling is a way of partially disassembling a loop structure so that fewer numbers of iterations of the loop are required, at the expense of each loop iteration being larger. It can be used to hide instruction and data latencies, to take advantage of floating point loadpair instructions, to increase the ratio of real work done per memory operation.
Prefetch	Makes request to bring data in from relatively slow memory to a faster cache several loop iterations ahead of when the data is actually needed.
LoadPair	Makes use of an instruction to bring two floating point data elements in from memory at a time.

See Optimizer Report Generation for more information about options you can use to generate reports.

## Interprocedural Optimizations (IPO) Report

The IPO report provides information on the functions that have been inlined. Some loops will not software pipeline (SWP) and others will not vectorize if function calls are embedded inside your loops. One way to get these loops to SWP or to vectorize is to inline the functions using IPO.

The IPO report can help to identify the problem loops.

### Note

Software Pipelining is available only on Itanium®-based systems

You must enable Interprocedural Optimizations to generate the IPO report.

The following command examples demonstrate how to run the IPO reports.

Platform	Syntax Examples
Linux*	<code>icpc -ipo -opt-report -opt-report-phase ipo a.c b.c</code>
Windows*	<code>icl /Qipo /Qopt-report /Qopt-report-phase ipo a.c b.c</code>

where `-ipo` (Linux) or `/Qipo` (Windows) tells the compiler to perform inline function expansion in multiple files, `-opt-report` (Linux) or `/Qopt-report` (Windows) invokes the report generator, and `-opt-report-phase ipo` (Linux) or `/Qopt-report-phase ipo` (Windows) indicates the phase (ipo) for which to generate the report.

You can specify an output file to capture the report results. IPO report results can be very extensive and technical; specifying a file to capture the results can help to reduce analysis time.

### Note

Linux\* only: The space between the option and the phase is optional.

See Optimizer Report Generation for more information about options you can use to generate reports.

## Software Pipelining (SWP) Report (Linux\* and Windows\*)

The SWP report can provide details information about loops currently taking advantage of software pipelining available on Itanium®-based systems. Additionally, the report suggests reasons for the loops not being pipelined.

The following command syntax examples demonstrates how to generate a SWP report for the Itanium® Compiler Code Generator (ECG) Software Pipeliner (SWP).

Platform	Syntax Examples
Linux*	<code>icc -c -opt-report -opt-report-phase ecg_swp swp.cpp</code>
Windows*	<code>icl /c /Qopt-report /Qopt-report-phase ecg_swp swp.cpp</code>

where `-c` (Linux) or `/c` (Windows) tells the compiler to stop at generating the object code (no linking occurs), `-opt-report` (Linux) or `/Qopt-report` (Windows) invokes the report generator, and `-opt-report-phase ecg_swp` (Linux) or `/Qopt-report-phase ecg_swp` (Windows) indicates the phase (ecg) for which to generate the report.

### Note

Linux\* only: The space between the option and the phase is optional.

Typically, loops that software pipeline will have a line that indicates the compiler has scheduled the loop for SWP in the report. If the `-O3` (Linux) or `/O3` (Windows) option is specified, the SWP report merges the loop transformation summary performed by the loop optimizer.

You can compile this example code to generate a sample SWP report, but you must use compile the example using a combination of `-c -restrict` (Linux) and `/c /Qrestrict` (Windows). The sample reports is also shown below.

**Example**

```
#define NUM 1024
void multiply d(double a[] [NUM], double b[] [NUM], double
c[restrict] [NUM]) {
    int i,j,k;
    double temp;
    for(i=0;i<NUM;i++) {
        for(j=0;j<NUM;j++) {
            for(k=0;k<NUM;k++) {
                c[i] [j] = c[i] [j] + a[i] [k] * b[k] [j];
            }
        }
    }
}
```

The following sample report shows the report phase that results from compiling the example code shown above (when using the `ecg_swp` phase).

**Sample SWP Report**

```
Swp report for loop at line 8 in  Z10multiply dPA1024 dS0 S0  in file
SWP report.cpp
Resource II      = 2
Recurrence II   = 2
Minimum II      = 2
Scheduled II    = 2

Estimated GCS II    = 7

Percent of Resource II needed by arithmetic ops      = 100%
Percent of Resource II needed by memory ops          = 50%
Percent of Resource II needed by floating point ops  = 50%

Number of stages in the software pipeline = 6
```

**Reading the Reports**

One fast way to determine if specific loops are software pipelining is to search the report output for the phrase “Number of stages in the software pipeline”; if this phrase is present in the report, it indicates that software pipelining succeeded for the associated loop.

To understand the SWP report results, you must know something about the terminology used and the related concepts. The following table describes some of the terminology used in the SWP report.

Term	Definition
II	<p>Initiation Interval (II). The number of cycles between the start of one iteration and the next in the SWP. The presence of the term II in any SWP report indicates that SWP succeeded for the loop in question.</p> <p>II can be used in a quick calculation to determine how many cycles your loop will take, if you also know the number of iterations. Total cycle time of the loop is approximately <math>N * \text{Scheduled II} + \text{number Stages}</math> (Where N is</p>

	the number of iterations of the loop). This is an approximation because it does not take into account the ramp-up and ramp-down of the prolog and epilog of the SWP, and only considers the kernel of the SWP loop. As you modify your code, it is generally better to see scheduled II go down, though it is really $N * (\text{Scheduled II}) + \text{Number of stages in the software pipeline}$ that is ultimately the figure of merit.
Resource II	Resource II implies what the Initiation Interval should be when considering the number of functional units available.
Recurrence II	Recurrence II indicates what the Initiation Interval should be when there is a recurrence relationship in the loop. A recurrence relationship is a particular kind of a data dependency called a flow dependency like $a[i] = a[i-1]$ where $a[i]$ cannot be computed until $a[i-1]$ is known. If Recurrence II is non-zero and there is no flow dependency in the code, then this indicates either Non-Unit Stride Access or memory aliasing.  See Helping the Compiler for more information.
Minimum II	Minimum II is the theoretical minimum Initiation Interval that could be achieved.
Scheduled II	Scheduled II is what the compiler actually scheduled for the SWP.
number of stages	Indicates the number of stages. For example, in the report results below, the line <code>Number of stages in the software pipeline = 3</code> indicates there were three stages of work, which will show, in assembly, to be a load, an FMA instruction and a store.
loop-carried memory dependence edges	The loop-carried memory dependence edges means the compiler avoided WAR (Write After Read) dependency.  Loop-carried memory dependence edges can indicate problems with memory aliasing. See Helping the Compiler.

## Using the Report to Resolve Issues

The most efficient path to solve problems is to analyze the loops that did not SWP in order to determine how to enable SWP.

If the compiler reports the `Loop was not SWP because...`, see the following table for suggestions about how to mitigate the problems:

Message in Report	Suggested Action
acyclic global scheduler can achieve a better schedule: => loop not pipelined	Indicates that the most likely cause is memory aliasing issues. For memory alias problems see memory aliasing (restrict, #pragma ivdep).  Might also indicate that the application might be accessing memory in a non-Unit Stride fashion. Non-Unit Stride issues may be indicated by an artificially high recurrence II; If you know there is no recurrence relationship ( $a[i] = a[i-1] + b[i]$ for example) in the loop, then a high recurrence II (greater than 0)

	is a sign that you are accessing memory non-Unit Stride. Rearranging code, perhaps a loop interchange, might help mitigate this problem.
Loop body has a function call	Indicates that inlining the function might help solve the problem.
Not enough static registers	Indicates that you should distribute the loop by separating it into two or more loops.  On Itanium®-based systems you may use <code>#pragma distribute point</code> .
Not enough rotating registers	Indicates that the loop carried values use the rotating registers. Distribute the loop. On Itanium-based systems you may use <code>#pragma distribute point</code> .
Loop too large	Indicates that you should distribute the loop.  On Itanium-based systems you may use the <code>#pragma distribute point</code> .
Loop has a constant trip count < 4	Indicates that unrolling was insufficient. Attempt to fully unroll the loop. However, with small loops fully unrolling the loop is not likely to affect performance significantly.
Too much flow control	Indicates complex loop structure. Attempt to simplify the loop.

Index variable type used can greatly impact performance. In some cases, using loop index variables of type short or unsigned int can prevent software pipelining. If the report indicates performance problems in loops where the index variable is not int and if there are no other obvious causes, try changing the loop index variable to type int.

See Optimizer Report Generation for more information about options you can use to generate reports.

## Vectorization Report

The vectorization report can provide information on what loops take advantage of Streaming SIMD Extensions 2 (SSE2) and Streaming SIMD Extensions 3 (SSE3) vectorization and which ones do not. The vectorization report is available on IA-32 and Intel® EM64T systems.

The `-vec-report` (Linux\*) or `/Qvec-report` (Windows\*) options directs the compiler to generate the vectorization reports with different levels of information. You can use this option to control the diagnostic message set to the reports. If you want to redirect the results to a text file you would use a command similar to the following:

Platform	Command
Linux	<code>icc -xW -vec-report3 matrix1.c &gt; report.txt</code>



Windows	icl /QxW /Qvec-report3 matrix1.c > report.txt
---------	---

For more information about this option, see the following topic:

- `-vec-report` compiler option

See Parallelism Overview for information on other vectorizer options.

The following example results illustrate the type of information generated by the vectorization report:

### Example results

```
multiply.c(10) : (col. 2) remark: loop was not vectorized: not
inner loop.
multiply.c(11) : (col. 6) remark: loop was not vectorized: not
inner loop.
multiply.c(12) : (col. 5) remark: vector dependence: assumed
FLOW dependence between c line 13 and b line 13.
multiply.c(12) : (col. 5) remark: vector dependence: assumed
FLOW dependence between c line 13 and a line 13.
multiply.c(12) : (col. 5) remark: vector dependence: assumed
FLOW dependence between c line 13 and c line 13.
multiply.c(12) : (col. 5) remark: loop was not vectorized:
existence of vector dependence.
```

If the compiler reports “Loop was not vectorized” because of the existence of vector dependence, then you need to do a vector dependence analysis of the loop.

If you are convinced that no legitimate vector dependence exists, then the above message indicates that the compiler was likely assuming the pointers or arrays in the loop were dependent, which is another way of saying that the pointers or arrays were aliased. Memory disambiguation techniques should be used to get the compiler to vectorize in these cases.

There are three major types of vector dependence: `FLOW`, `ANTI`, and `OUTPUT`.

See Loop Independence to determine if you have a valid vector dependence. Many times the compiler report will assert a vector dependence where none exists – this is because the compiler assumes memory aliasing. The action to take in these cases is to check code for dependencies; if there are none, inform the compiler using methods described in memory aliasing including `restrict` or `pragma ivdep`.

There are a number of situations where the vectorization report may indicate vector dependencies. The following situations will sometimes be reported as vector dependencies, Non-Unit Stride, Low Trip Count, Complex Subscript Expression.

## Non-Unit Stride

The report might indicate that a loop could not be vectorized when the memory is accessed in a non-Unit Stride fashion. This means that nonconsecutive memory locations are being accessed in the loop. In such cases, see if loop interchange can help

or if it is practical. If not, then sometimes you can force vectorization through `vector always pragma`; however, you should verify improvement.

See Understanding Runtime Performance for more information about non-unit stride conditions.

## Usage with Other Options

The vectorization reports are generated during the final compilation phase, which is when the executable is generated; therefore, there are certain option combinations you cannot use if you are attempting to generate a report. If you use the following option combinations, the compiler issues a warning and does not generate a report:

- `-c` or `-ipo` or `-x` with `-vec-report` (Linux\*) and `/c` or `/Qipo` or `/Qx` with `/Qvec-report` (Windows\*)
- `-c` or `-ax` with `-vec-report` (Linux) and `/c` or `/Qax` with `/Qvec-report` (Windows)

The following example commands can generate vectorization reports:

Platform	Command Examples
Linux	<p>The following commands generate a vectorization report:</p> <pre>icpc -xK -vec report3 file.cpp icpc -xK -ipo -ipo obj -vec report3 file.cpp icpc -c -xK -ipo -ipo_obj -vec_report3 file.cpp</pre> <p>The following commands will not generate a vectorization report:</p> <pre>icpc -c -xK -vec report3 file.cpp icpc -xK -ipo -vec report3 file.cpp icpc -c -xK -ipo -vec_report3 file.cpp</pre>
Windows	<p>The following commands generate a vectorization report:</p> <pre>icl /QxK /Qvec_report3 file.cpp icl /QxK /Qipo /Qipo obj /Qvec report3 file.cpp icl /c /QxK /Qipo /Qipo_obj /Qvec_report3 file.cpp</pre> <p>The following commands will not generate a vectorization report:</p> <pre>icl /c /QxK /Qvec report3 file.cpp icl /QxK /Qipo /Qvec_report3 file.cpp icl /c /QxK /Qipo /Qvec_report3 file.cpp</pre>

## Changing Code Based on Report Results

You might consider changing existing code to allow vectorization under the following conditions:

- The vectorization report indicates that the program contains unvectorizable statement at line XXX; eliminate conditions such as, a printf() or user defined foo() the loop.
- The vectorization report states there is a vector dependence: proven FLOW dependence between 'variable' line XXX, and 'variable' line XXX or loop was not vectorized: existence of vector dependence. Generally, these conditions indicate true loop dependencies are stopping vectorization. In such cases, consider changing the loop algorithm.

For example, consider the two equivalent algorithms producing identical output below. "Foo" will not vectorize due to the FLOW dependence but "bar" does vectorize.

### Example

```
void foo(double *y)
{
    for(int i=1;i<10;i++) { // a loop that puts sequential numbers into
        array y
        y[i] = y[i-1]+1;
    }
}
void bar(double *y)
{
    for(int i=1;i<10;i++) { // a loop that puts sequential numbers into
        array y
        y[i] = y[0]+i;
    }
}
```

Unsupported loop structures may prevent vectorization. An example of an unsupported loop structure is a loop index variable that requires complex computation. Change the structure to remove function calls to loop limits and other excessive computation for loop limits.

### Example

```
int function(int n)
{
    return (n*n-1);
}
void unsupported loop structure(double *y, int n)
{
    for (int i=0; i<function(n); i++) {
        *y = *y * 2.0;
    }
}
```

Non-unit stride access might cause the report to state that vectorization possible but seems inefficient. Try to restructure the loop to access the data in a unit-stride manner (for example, apply loop interchange), or try pragma vector always.

Using mixed data types in the body of a loop might prevent vectorization. In the case of mixed data types, the vectorization report might state something similar to loop was not vectorized: condition too complex.

The following example code demonstrates a loop that cannot vectorize due to mixed data types within the loop. For example, `withinborder` is an `int` while all other data types in loop are defined as `double`. Simply changing the `withinborder` data type to `double` will allow this loop to vectorize.

### Example

```
int howmany_close(double *x, double *y)
{
    int withinborder=0;
    double dist;
    for(int i=0;i<100;i++) {
        dist=sqrtf(x[i]*x[i] + y[i]*y[i]);
        if (dist<5) withinborder++;
    }
    return 0;
}
```

## Parallelism Overview

This section discusses the three major features of parallel programming supported by the Intel® compiler:

- Parallelization with OpenMP\*
- Auto-parallelization
- Auto-vectorization

Each of these features contributes to application performance depending on the number of processors, target architecture (IA-32 or Itanium® architecture), and the nature of the application. These features of parallel programming can be combined to contribute to application performance.

Parallel programming can be *explicit*, that is, defined by a programmer using OpenMP directives. Parallel programming can also be *implicit*, that is, detected automatically by the compiler. Implicit parallelism implements auto-parallelization of outer-most loops and auto-vectorization of innermost loops (or both).

Parallelism defined with OpenMP and auto-parallelization directives is based on thread-level parallelism (TLP). Parallelism defined with auto-vectorization techniques is based on instruction-level parallelism (ILP).

The Intel® compiler supports OpenMP and auto-parallelization for IA-32, Intel EM64T, and Itanium architectures for multiprocessor systems, dual-core processors systems, and systems with Hyper-Threading Technology (HT Technology) enabled.

Auto-vectorization is supported on the families of the Pentium®, Pentium with MMX™ technology, Pentium II, Pentium III, and Pentium 4 processors. To enhance the compilation of the code with auto-vectorization, users can also add vectorizer directives to their program. A closely related technique software pipelining (SWP) is available on the Itanium-based systems.

The following table summarizes the different ways in which parallelism can be exploited with the Intel® Compiler.

Parallelism	Description
<b>Explicit</b>	Parallelism programmed by the user
OpenMP* (thread-level parallelism)  IA-32 and Itanium® architectures	Supported on: <ul style="list-style-type: none"> <li>• IA-32, Intel EM64T, and Itanium-based multiprocessor systems and dual-core processors</li> <li>• Hyper-Threading Technology-enabled systems</li> </ul>
<b>Implicit</b>	Parallelism generated by the compiler and by user-supplied hints

Auto-parallelization (thread-level parallelism) of outer-most loops; IA-32 and Itanium architectures	Supported on: <ul style="list-style-type: none"> <li>IA-32, Intel EM64T, and Itanium-based multiprocessor systems and dual-core processors</li> <li>Hyper-Threading Technology-enabled systems</li> </ul>
Auto-vectorization (instruction-level parallelism) of inner-most loops; IA-32 and Itanium architectures	Supported on: <ul style="list-style-type: none"> <li>Pentium®, Pentium with MMX™ Technology, Pentium II, Pentium III, and Pentium 4 processors</li> </ul>

## Parallel Program Development

### OpenMP

The Intel® compiler supports the OpenMP\* C/C++ version 2.5 API specification available from the OpenMP\* (<http://www.openmp.org>) web site. The OpenMP directives relieve the user from having to deal with the low-level details of iteration space partitioning, data sharing, and thread scheduling and synchronization.

### Auto-Parallelization

The Auto-parallelization feature of the Intel® compiler automatically translates serial portions of the input program into semantically equivalent multithreaded code. Automatic parallelization determines the loops that are good worksharing candidates, performs the dataflow analysis to verify correct parallel execution, and partitions the data for threaded code generation as is needed in programming with OpenMP directives. The OpenMP and Auto-parallelization applications provide the performance gains from shared memory on multiprocessor and dual-core systems and IA-32 processors with the Hyper-Threading Technology.

### Auto-Vectorization

Auto-vectorization detects low-level operations in the program that can be done in parallel, and then converts the sequential program to process 2, 4, 8 or up to 16 elements in one operation, depending on the data type. In some cases auto-parallelization and vectorization can be combined for better performance results.

The following example demonstrates how code can be designed to explicitly benefit from parallelization and vectorization. Assuming you compile the code shown below using `-parallel -xP` (Linux\*) or `/Qparallel /QxP` (Windows\*), the compiler will parallelize the outer loop and vectorize the innermost loop.

<b>Example</b>
<pre>#include &lt;stdio.h&gt; #define ARR_SIZE 500 //Define array int main()</pre>

```

{
  int matrix[ARR_SIZE][ARR_SIZE];
  int arrA[ARR_SIZE]={10};
  int arrB[ARR_SIZE]={30};
  int i, j;
  for(i=0;i<ARR_SIZE;i++)
  {
    for(j=0;j<ARR_SIZE;j++)
    {
      matrix[i][j] = arrB[i]*(arrA[i]%2+10);
    }
  }
}

```

Compiling the example code with the correct options, the compiler should report results similar to the following:

```
vectorization.c(18) : (col. 6) remark: LOOP WAS VECTORIZED.
```

```
vectorization.c(16) : (col. 3) remark: LOOP WAS AUTO-PARALLELIZED.
```

Auto-vectorization can help improve performance of an application that runs on systems based on Pentium®, Pentium with MMX™ technology, Pentium II, Pentium III, and Pentium 4 processors.

The following tables summarize the options that enable auto-vectorization, auto-parallelization, and OpenMP support.

### Auto-vectorization: IA-32 only

Windows*	Linux*	Description
/Qx	-x	Generates specialized code to run exclusively on processors with the extensions specified by {K,W,N,B,P,T}. P is the only valid value on Mac OS* systems.  See the following topic in Compiler Options:  <ul style="list-style-type: none"> <li>-x</li> </ul>
/Qax	-ax	Generates, in a single binary, code specialized to the extensions specified by {K,W,N,B,P,T} and also generic IA-32 code. P is the only valid value on Mac OS systems.  The generic code is usually slower.  See the following topic in Compiler Options:  <ul style="list-style-type: none"> <li>-ax</li> </ul>
/Qvec-report	-vec-report	Controls the diagnostic messages from the vectorizer, see subsection that follows the table.

		<p>See the following topic in Compiler Options:</p> <ul style="list-style-type: none"> <li>• <code>-vec-report</code></li> </ul>
--	--	--

### Auto-parallelization: IA-32 and Itanium® architectures

Windows*	Linux*	Description
<code>/Qparallel</code>	<code>-parallel</code>	<p>Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.</p> <p>Intel® Itanium®-based systems only:</p> <ul style="list-style-type: none"> <li>• Implies <code>-opt-mem-bandwidth1</code> (Linux) or <code>/Qopt-mem-bandwidth1</code> (Windows).</li> </ul> <p>See the following topic in Compiler Options:</p> <ul style="list-style-type: none"> <li>• <code>-parallel</code></li> </ul>
<code>/Qpar-threshold[:n]</code>	<code>-par-threshold{n}</code>	<p>Sets a threshold for the auto of loops based on the probability of profitable execution of the loop in parallel, n=0 to 100.</p> <p>See the following topic in Compiler Options:</p> <ul style="list-style-type: none"> <li>• <code>-par-threshold</code></li> </ul>
<code>/Qpar-report</code>	<code>-par-report</code>	<p>Controls the auto-parallelizer's diagnostic levels.</p> <p>See the following topic in Compiler Options:</p> <ul style="list-style-type: none"> <li>• <code>-par-report</code></li> </ul>

### OpenMP: IA-32 and Itanium® architectures

Windows*	Linux*	Description
<code>/Qopenmp</code>	<code>-openmp</code>	<p>Enables the parallelizer to generate multithreaded code based on the OpenMP directives.</p> <p>Intel® Itanium®-based systems only:</p> <ul style="list-style-type: none"> <li>• Implies <code>-opt-mem-bandwidth1</code> (Linux) or <code>/Qopt-mem-bandwidth1</code> (Windows).</li> </ul> <p>See the following topic in Compiler Options:</p>



		<ul style="list-style-type: none"> <li>• <code>-openmp</code></li> </ul>
<code>/Qopenmp-report</code>	<code>-openmp-report</code>	<p>Controls the OpenMP parallelizer's diagnostic levels.</p> <p>See the following topic in Compiler Options:</p> <ul style="list-style-type: none"> <li>• <code>-openmp-report</code></li> </ul>
<code>/Qopenmp-stubs</code>	<code>-openmp-stubs</code>	<p>Enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.</p> <p>See the following topic in Compiler Options:</p> <ul style="list-style-type: none"> <li>• <code>-openmp-stubs</code></li> </ul>

### Note

When both `-openmp` (Linux) or `/Qopenmp` (Windows) and `-parallel` (Linux) or `/Qparallel` (Windows) are specified on the command line, the `-parallel` (Linux) or `/Qparallel` (Windows) option is only applied in routines that do not contain OpenMP directives. For routines that contain OpenMP directives, only the `-openmp` (Linux) or `/Qopenmp` (Windows) option is applied.

With the right choice of options, you can:

- Increase the performance of your application with minimum effort
- Use compiler features to develop multithreaded programs faster

Additionally, with the relatively small effort of adding OpenMP directives to existing code you can transform a sequential program into a parallel program.

The following example demonstrates one method of using the OpenMP pragmas within code.

### Example

```
#include <stdio.h>
#define ARR_SIZE 100 //Define array
void foo(int ma[][ARR_SIZE], int mb[][ARR_SIZE], int *a, int *b, int *c);
int main()
{
    int arr a[ARR_SIZE];
    int arr b[ARR_SIZE];
    int arr c[ARR_SIZE];
    int i,j;
    int matrix a[ARR_SIZE][ARR_SIZE];
    int matrix b[ARR_SIZE][ARR_SIZE];

    #pragma omp parallel for
    // Initialize the arrays and matrices.
```

```

for(i=0; i<ARR_SIZE; i++)
{
    arr a[i]= i;
    arr b[i]= i;
    arr c[i]= ARR_SIZE-i;
    for(j=0; j<ARR_SIZE; j++)
    {
        matrix a[i][j]= j;
        matrix b[i][j]= i;
    }
}
foo(matrix a, matrix b, arr a, arr b, arr c);
}
void foo(int ma[][ARR_SIZE], int mb[][ARR_SIZE], int *a, int *b, int *c)
{
    int i, num, arr x[ARR_SIZE];
    #pragma omp parallel for private(num)
    // Expresses the parallelism using the OpenMP pragma: parallel for.
    // The pragma guides the compiler generating multithreaded code.
    // Array arr X, mb, b, and c are shared among threads based on OpenMP
    // data sharing rules. Scalar num is specified as private
    // for each thread.
    for(i=0; i<ARR_SIZE; i++)
    {
        num = ma[b[i]][c[i]];
        arr x[i]= mb[a[i]][num];
        printf("Values: %d\n", arr x[i]); //prints values 0-ARR_SIZE-1
    }
}

```

## Parallelization with OpenMP\* Overview

The Intel® compiler supports the OpenMP\* version 2.5 API specification and an automatic parallelization capability. OpenMP provides symmetric multiprocessing (SMP) with the following major features:

- Relieves the user from having to deal with the low-level details of iteration space partitioning, data sharing, and thread scheduling and synchronization.
- Provides the benefit of the performance available from shared memory, multiprocessor and dual-core processor systems and on IA-32 processors with Hyper-Threading Technology (HT Technology).

### Note

For information on HT Technology, refer to the IA-32 Intel® Architecture Optimization Reference Manual ([http://developer.intel.com/design/pentium4/manuals/index\\_new.htm](http://developer.intel.com/design/pentium4/manuals/index_new.htm)).

The compiler performs transformations to generate multithreaded code based on the user's placement of OpenMP directives in the source program making it easy to add threading to existing software. The Intel compiler supports all of the current industry-standard OpenMP directives, except `WORKSHARE`, and compiles parallel programs annotated with OpenMP directives.

### Note

As with many advanced features of compilers, you must properly understand the functionality of the OpenMP directives in order to use them effectively and avoid unwanted program behavior. See parallelization options summary for all of the options of the OpenMP feature in the Intel C++ Compiler.

In addition, the compiler provides Intel-specific extensions to the OpenMP C/C++ version 2.5 specification including run-time library routines and environment variables.

For complete information on the OpenMP standard, visit the OpenMP\* (<http://www.openmp.org>) web site. For complete C++ language specifications, see the OpenMP C/C++ version 2.5 specifications (<http://www.openmp.org/specs>).

## Parallel Processing with OpenMP

To compile with OpenMP, you need to prepare your program by annotating the code with OpenMP directives. The Intel compiler first processes the application and produces a multithreaded version of the code which is then compiled. The output is an executable with the parallelism implemented by threads that execute parallel regions or constructs.

### Windows\* Considerations

The OpenMP specification does not define interoperability of multiple implementations; therefore, the OpenMP implementation supported by other compilers and OpenMP support in Intel compilers for Windows might not be interoperable. To avoid possible linking or run-time problems, keep the following guidelines in mind:

- Avoid using multiple copies of the OpenMP runtime libraries from different compilers.
- Compile all the OpenMP sources with one compiler, or compile the parallel region and entire call tree beneath it using the same compiler.
- Use dynamic libraries for OpenMP.

## Performance Analysis

For performance analysis of your program, you can use the Intel® VTune™ Performance Analyzer and/or the Intel® Threading Tools to show performance information. You can obtain detailed information about which portions of the code that require the largest amount of time to execute and where parallel performance problems are located.

## Targeting a Processor Run-time Check

While parallelizing a loop, the Intel compiler's loop parallelizer, OpenMP, tries to determine the optimal set of configurations for a given processor. At run time, a check is performed to determine which processor OpenMP should optimize a given loop. See detailed information in Processor-specific Runtime Checks for IA-32 Systems.

## Parallel Processing Thread Model

This topic explains the processing of the parallelized program and adds more definitions of the terms used in the parallel programming.

### The Execution Flow

A program containing OpenMP C++ API compiler directives begins execution as a single process, called the master thread of execution. The master thread executes sequentially until the first parallel construct is encountered.

In the OpenMP C++ API, the `#pragma omp parallel` directive defines the parallel construct. When the master thread encounters a parallel construct, it creates a team of threads, with the master thread becoming the master of the team. The program statements enclosed by the parallel construct are executed in parallel by each thread in the team. These statements include routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the static extent of the construct. The dynamic extent includes the static extent as well as the routines called from within the construct. When the `#pragma omp parallel` directive is encountered, the threads in the team synchronize at that point, the team is dissolved, and only the master thread continues execution. The other threads in the team enter a wait state. You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

### General Performance Guidelines

For applications where the workload depends on application input that can vary widely, delay the decision about the number of threads to employ until runtime when the input sizes can be examined. Examples of workload input parameters that affect the thread count include things like matrix size, database size, image/video size and resolution, depth/breadth/bushiness of tree based structures, and size of list based structures. Similarly, for applications designed to run on systems where the processor count can vary widely, defer the number of threads to employ decision till application run-time when the machine size can be examined.

For applications where the amount of work is unpredictable from the input data, consider using a calibration step to understand the workload and system characteristics to aid in choosing an appropriate number of threads. If the calibration step is expensive, the calibration results can be made persistent by storing the results in a permanent place like the file system. Avoid creating more threads than the number of processors on the system, when all the threads can be active simultaneously; this situation causes the operating system to multiplex the processors and typically yields sub-optimal performance.

When developing a library as opposed to an entire application, provide a mechanism whereby the user of the library can conveniently select the number of threads used by

the library, because it is possible that the user has higher-level parallelism that renders the parallelism in the library unnecessary or even disruptive.

Finally, for OpenMP, use the `num_threads` clause on parallel regions to control the number of threads employed and use the `if` clause on parallel regions to decide whether to employ multiple threads at all. The `omp_set_num_threads` function can also be used but it is not recommended except in specialized well-understood situations because its affect is global and persists even after the current function ends, possibly affecting parents in the call tree. The `num_threads` clause is local in its effect and so does not impact the calling environment.

## Using Orphaned Directives

In routines called from within parallel constructs, you can also use directives. Directives that are not in the lexical extent of the parallel construct, but are in the dynamic extent, are called orphaned directives. Orphaned directives allow you to execute major portions of your program in parallel with only minimal changes to the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program call tree and use directives to control execution in any of the called routines. For example:

Example 1
<pre>int main(void) {     ...     #pragma omp parallel     {         phase1();     } }  void phase1(void) {     ...     #pragma omp for private(i) shared(n)     for(i=0; i &lt; n; i++)     {         some work(i);     } }</pre>

This is an orphaned directive because the parallel region is not lexically present.

## Data Environment Directive

A data environment directive controls the data environment during the execution of parallel constructs.

You can control the data environment within parallel and worksharing constructs. Using directives and data environment clauses on directives, you can:

- Privatize named common blocks by using `THREADPRIVATE` directive

- Control data scope attributes by using the `THREADPRIVATE` directive's clauses.
- The data scope attribute clauses are:
  - `COPYIN`
  - `DEFAULT`
  - `PRIVATE`
  - `FIRSTPRIVATE`
  - `LASTPRIVATE`
  - `REDUCTION`
  - `SHARED`

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them. If you do not specify a data scope attribute clause on a directive, the default is `SHARED` for those variables affected by the directive.

For detailed descriptions of the clauses, see the OpenMP C/C++ version 2.5 specifications (<http://www.openmp.org/specs>).

### Example 2: Pseudo Code of the Parallel Processing Model

```
main() {
    ...                // Begin serial execution
    #pragma omp parallel // Only the master thread executes
    {                  // Begin a Parallel Construct, form
        ...            // a team. This is Replicated Code
        ...            // (each team member executes
        ...            // the same code)
        #pragma omp sections // Begin a Worksharing Construct
        {
            #pragma omp section // One unit of work
            { ... }
            #pragma omp section // Another unit of work
            { ... }
        }
        ...            // Wait until both units of work complete
        ...            // More Replicated Code
        #pragma omp for nowait // Begin a Worksharing Construct
        for(...)
        {
            ...        // Each iteration is unit of work
            ...        // Work is distributed among the team members
        }              // End of Worksharing Construct
        ...            // nowait was specified, so threads proceed
        #pragma omp critical // Begin a Critical Section
        {
            ...        // Replicated Code, but only one
            ...        // thread can execute it at a
            ...        // given time
        }
        ...            // More Replicated Code
        #pragma omp barrier // Wait for all team members to arrive
        ...            // More Replicated Code
    }                  // End of Parallel Construct
    ...                // disband team and continue
    ...                // serial execution
    ...                // Possibly more Parallel constructs
}                      // End serial execution
```

## OpenMP\* and Hyper-Threading Technology

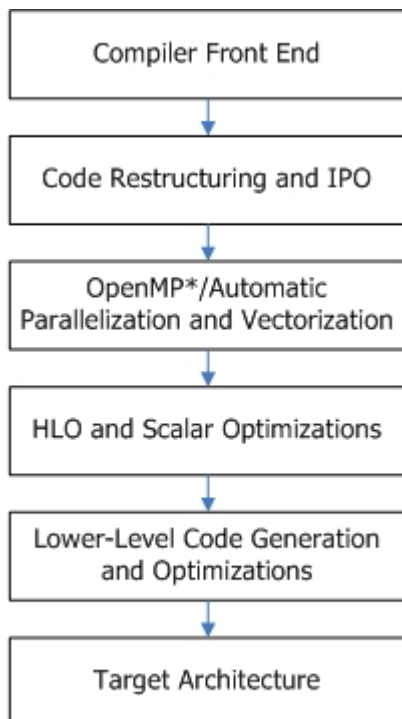
The compiler incorporates many well-known and advanced optimization techniques designed to leverage Intel® processor features for higher performance on IA-32- and Itanium®-based systems.

The Intel® compiler has a common intermediate representation for the supported languages, so that the pragma-guided parallelization and a majority of optimization techniques are applicable through a single high-level code transformation, irrespective of the source language.

The code transformations and optimizations in the Intel compiler can be categorized into the following functional areas:

- Code restructuring and interprocedural optimizations (IPO)
- OpenMP-based and automatic parallelization and vectorization
- High-Level Optimizations (HLO) and scalar optimizations including memory optimizations such as loop control and data transformations, partial redundancy elimination (PRE), and partial dead store elimination (PDSE)
- Low-level machine code generation and optimizations such as register allocation and instruction scheduling

The figure illustrates the interrelation of the different areas.



Parallelization guided by OpenMP pragmas or derived by automatic data dependency and control-flow analysis is a high-level code transformation that exploits both medium- and coarse-grained parallelism for multiprocessor systems, systems with dual-core processors, and Hyper-Threading Technology (HT Technology) enabled systems to achieve better performance and higher throughput.

The Intel® compiler has a common intermediate language representation (called IL0) into which applications are translated by the front-ends. Many optimization phases in the compiler work on the IL0 representation.

The IL0 has been extended to express the OpenMP pragmas. Implementing the OpenMP phase at the IL0 level allows the same implementation to be used across languages and architectures. The Intel compiler-generated code references a high-level multithreaded library API, which allows the compiler OpenMP transformation phase to be independent of the underlying operating systems.

The Intel compiler integrates OpenMP parallelization with advanced compiler optimizations to generate efficient multithreaded code that is typically faster than optimized uniprocessor code. An effective optimization phase ordering has been designed in the Intel compiler to make sure that all optimizations, such as IPO inlining, code restructuring; lgoto optimizations, and constant propagation, which are effectively enabled before the OpenMP parallelization, preserve legal OpenMP program semantics and necessary information for parallelization.

The integration also ensures that all optimizations after the OpenMP parallelization, such as automatic vectorization, loop transformation, PRE, and PDSE, can effectively help achieve a better cache locality and help minimize the number of computations and the number of references to memory. For example, given a double-nested OpenMP parallel loop, the parallelization methods are able to generate multithreaded code for the outer loop, while maintaining the loop structure, memory reference behavior, and symbol table information for the innermost loop. This behavior enables subsequent intra-register vectorization of the innermost loop to fully leverage the HT Technology and SIMD Streaming Extension features of Intel processors.

OpenMP parallelization in the Intel compiler includes:

- A pre-pass that transforms OpenMP parallel sections into parallel loop and work-sharing sections into work-sharing loops.
- A work-region graph builder that builds a region hierarchical graph based on the OpenMP-aware control-flow graph.
- A loop analysis phase for building the loop structure that consists of loop control variable, loop lower-bound, loop upper-bound, loop pre-header, loop header, and control expression.
- A variable classification phase that performs analysis of shared and private variables.
- A multithreaded code generator that generates multithreaded code at compiler intermediate code level based on Guide, which is a multithreaded run-time library API.
- A privatizer that performs privatization to handle `firstprivate`, `private`, `lastprivate`, and `reduction` variables.



- A post-pass that generates code to cache in thread local storage for handling `threadprivate` variables.

OpenMP, a compiler-based threading method, provides a high-level interface to the underlying thread libraries. With OpenMP, you can use pragmas to describe parallelism to the compiler. Using the supplied pragmas removes much of the complexity of explicit threading because the compiler handles the details. OpenMP is less invasive, so significant source code modifications are not usually necessary. A non-OpenMP compiler simply ignores the pragmas, leaving the underlying serial code intact.

As in every other aspect of optimization, the key to attaining good parallel performance is choosing the right granularity for your application. Within the context of this discussion, granularity is the amount of work in the parallel task. If granularity is too fine performance can suffer from increased communication overhead. Conversely, if granularity is too coarse performance can suffer from load imbalance. The design goal is to determine the right granularity for the parallel tasks while avoiding load imbalance and communication overhead.

The amount of work for each parallel task, or granularity, of a multithreaded application greatly affects its parallel performance. When threading an application, the first step is to partition the problem into as many parallel tasks as possible. The second step is to determine the necessary communication in terms of data and synchronization. The third step is to consider the performance of the algorithm. Since communication and partitioning are not free operations, the operations often need to combine partitions. This overcomes the overheads and achieve the most efficient implementation. The combination step is the process of determining the best granularity for the application.

The granularity is often related to how balanced the workload is between threads. It is easier to balance the workload of a large number of small tasks but too many small tasks can lead to excessive parallel overhead. Therefore, coarse granularity is usually best. Increasing granularity too much can create load imbalance; tools like the Intel® Thread Profiler can help identify the right granularity for your application.

### Note

For detailed information on Hyper-Threading Technology, refer to the IA-32 Intel® Architecture Optimization Reference Manual ( [http://developer.intel.com/design/pentium4/manuals/index\\_new.htm](http://developer.intel.com/design/pentium4/manuals/index_new.htm)).

## Compiling with OpenMP, Directive Format, and Diagnostics

To run the Intel® compiler in OpenMP mode, invoke the compiler with the `-openmp` (Linux\*) or `/Qopenmp` (Windows\*) option using a command structured similar to the following:

Platform	Description
Linux	<code>icc -openmp input_file</code>
Windows	<code>icl /Qopenmp input_file</code>

Before you run the multithreaded code, you can set the number of desired threads in the OpenMP environment variable, `OMP_NUM_THREADS`. For more information, see the OpenMP Environment Variables section. The Intel Extension Routines topic describes the OpenMP extensions to the specification that have been added by Intel to the Intel® compiler.

### OpenMP Option

The `-openmp` (Linux\*) or `/Qopenmp` (Windows\*) option enables the parallelizer to generate multithreaded code based on the OpenMP directives. The code can be executed in parallel on uniprocessor, multiprocessor, and dual-core processor systems.

The `-openmp` (Linux) or `/Qopenmp` (Windows) option works with both `-O0` (Linux) and `/Od` (Windows), or with any optimization level of `-O1`, `-O2` and `-O3`. (Linux) or `/O1`, `/O2` and `/O3` (Windows). Specifying `-O0` (Linux) or `/Od` (Windows) with the OpenMP option helps to debug OpenMP applications.

#### Note

Intel® Itanium®-based systems: Specifying this option implies `-opt-mem-bandwidth1` (Linux) or `/Qopt-mem-bandwidth1` (Windows).

When you use the `openmp` option, the compiler sets the `-auto` (Linux) or `/Qauto` (Windows) option, which causes all variables to be allocated on the stack, rather than in local static storage.

### OpenMP Pragma Format and Syntax

OpenMP pragmas use a specific format and syntax. The following syntax and example help illustrate how to use the pragmas with your source.

Syntax
<code>#pragma omp directive-name [clause, ...] newline</code>

where:

- `#pragma omp` - Required for all OpenMP directives.
- `directive-name` - A valid OpenMP directive. Must appear after the `pragma` and before any clauses.
- `clause` - Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.
- `newline` - Required. Proceeds the structured block which is enclosed by this directive.

The following example demonstrates one way of using an OpenMP\* `pragma` to parallelize loops within parallel regions.

### Example

```
int i;
void simple omp(int *a){
    #pragma omp parallel for
    for (i=0; i<1024; i++)
        a[i] = i*2;
}
```

Assume that you compile the sample above, using the commands similar to the following, where `-c` (Linux) or `/c` (Windows) instructs the compiler to compile the code without generating an executable:

Platform	Description
Linux	<code>icpc -openmp -c parallel.c</code>
Windows	<code>icl /Qopenmp /c parallel.c</code>

The compiler might return a message similar to the following:

```
parallel.c(20) : (col. 3) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
```

## OpenMP Diagnostic Reports

The `-openmp-report` (Linux) or `/Qopenmp-report` (Windows) option controls the OpenMP\* parallelizer's diagnostic levels 0, 1, or 2 as follows:

Windows*	Linux*	Description
<code>/Qopenmp-report0</code>	<code>-openmp-report0</code>	No diagnostic information is displayed.
<code>/Qopenmp-report1</code>	<code>-openmp-report1</code>	Display diagnostics indicating loops, regions, and sections successfully parallelized.
<code>/Qopenmp-report2</code>	<code>-openmp-report2</code>	Same as specifying 1 plus diagnostics indicating constructs like <code>MASTER</code> , <code>SINGLE</code> , <code>CRITICAL</code> , <code>ORDERED</code> , <code>ATOMIC</code> directives, and so forth are successfully handled.

For more information about the option behaviors listed above, see the following topic in Compiler Options:

- `-openmp-report`

## OpenMP\* Directives and Clauses Summary

This topic provides a summary of the OpenMP directives and clauses. For detailed descriptions, see the OpenMP C/C++ version 2.5 specifications (<http://www.openmp.org/specs>).

### OpenMP Directives

Directive	Description
PARALLEL END PARALLEL	Defines a parallel region.
DO END DO	Identifies an iterative worksharing construct in which the iterations of the associated loop should be executed in parallel.
SECTIONS END SECTIONS	Identifies a non-iterative worksharing construct that specifies a set of structured blocks that are to be divided among threads in a team.
SECTION	Indicates that the associated structured block should be executed in parallel as part of the enclosing sections construct.
SINGLE END SINGLE	Identifies a construct that specifies that the associated structured block is executed by only one thread in the team.
PARALLEL FOR	A shortcut for a <code>parallel</code> region that contains a single <code>for</code> directive.  <b>Note</b>  The <code>parallel</code> or <code>for</code> OpenMP directive must be immediately followed by a <code>for</code> statement. If you place other statement or an OpenMP directive between the <code>parallel</code> or <code>for</code> directive and the <code>for</code> statement, the Intel C++ Compiler issues a syntax error.
FOR	Identifies an iterative work-sharing construct that specifies a region in which the iterations of the associated loop should be executed in parallel.
PARALLEL SECTIONS END PARALLEL SECTIONS	Provides a shortcut form for specifying a parallel region containing a single <code>SECTIONS</code> construct.
MASTER END MASTER	Identifies a construct that specifies a structured block that is executed by only the master thread of the team.
CRITICAL [ <i>lock</i> ] END CRITICAL [ <i>lock</i> ]	Identifies a construct that restricts execution of the associated structured block to a single thread at a time. Each thread waits at the beginning of the critical construct until no other thread is executing a critical construct with the same <i>lock</i> argument.
BARRIER	Synchronizes all the threads in a team. Each thread waits until all of the other threads in that team have reached this point.
ATOMIC	Ensures that a specific memory location is updated atomically, rather

	than exposing it to the possibility of multiple, simultaneously writing threads.
FLUSH [(list)]	Specifies a "cross-thread" sequence point at which the implementation is required to ensure that all the threads in a team have a consistent view of certain objects in memory. The optional <i>list</i> argument consists of a comma-separated list of variables to be flushed.
ORDERED END ORDERED	The structured block following an <code>ORDERED</code> directive is executed in the order in which iterations would be executed in a sequential loop.
THREADPRIVATE	Makes the named file-scope or namespace-scope variables specified private to a thread but file-scope visible within the thread.

## OpenMP Clauses

Clause	Description
PRIVATE	Declares variables to be <code>private</code> to each thread in a team.
FIRSTPRIVATE	Provides a superset of the functionality provided by the <code>private</code> clause.
LASTPRIVATE	Provides a superset of the functionality provided by the <code>private</code> clause.
SHARED	Shares variables among all the threads in a team.
DEFAULT	Enables you to affect the data-scope attributes of variables.
REDUCTION	Performs a reduction on scalar variables.
ORDERED	The structured block following an <code>ordered</code> directive is executed in the order in which iterations would be executed in a sequential loop.
IF	If the <code>if(scalar_logical_expression)</code> clause is present, the enclosed code block is executed in parallel only if the <code>scalar_logical_expression</code> evaluates to <code>TRUE</code> . Otherwise the code block is serialized.
SCHEDULE	Specifies how iterations of the <code>for</code> loop are divided among the threads of the team.
COPYIN	Provides a mechanism to assign the same name to <code>threadprivate</code> variables for each thread in the team executing the parallel region.

## OpenMP\* Support Libraries

The Intel® compiler with OpenMP\* support provides a production support library: `libguide.a` (Linux\*) and `libguide.lib` (Windows\*). This library enables you to run an application under different execution modes. It is used for normal or performance-critical runs on applications that have already been tuned.

### Note

The support library is linked dynamically, regardless of command-line options, to avoid performance issues that are hard to debug.

## Execution modes

The compiler with OpenMP enables you to run an application under different execution modes that can be specified at run time. The libraries support the serial, turnaround, and throughput modes. These modes are selected by using the `KMP_LIBRARY` environment variable at run time.

### Serial

The serial mode forces parallel applications to run on a single processor.

### Throughput

In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously. In this mode, the worker threads will yield to other threads while waiting for more parallel work.

The throughput mode is designed to make the program aware of its environment (that is, the system load) and to adjust its resource usage to produce efficient execution in a dynamic environment. This mode is the default.

After completing the execution of a parallel region, threads wait for new parallel work to become available. After a certain period of time has elapsed, they stop waiting and sleep. Sleeping allows the threads to be used, until more parallel work becomes available, by non-OpenMP threaded code that may execute between parallel regions, or by other applications. The amount of time to wait before sleeping is set either by the `KMP_BLOCKTIME` environment variable or by the `KMP_SET_BLOCKTIME` function. A small `KMP_BLOCKTIME` value may offer better overall performance if your application contains non-OpenMP threaded code that executes between parallel regions. A larger `KMP_BLOCKTIME` value may be more appropriate if threads are to be reserved solely for use for OpenMP execution, but may penalize other concurrently-running OpenMP or threaded applications.

### Turnaround

In a dedicated (batch or single user) parallel environment where all processors are exclusively allocated to the program for its entire run, it is most important to effectively utilize all of the processors all of the time. The turnaround mode is designed to keep active all of the processors involved in the parallel computation in order to minimize the execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads.

#### Note

Avoid over-allocating system resources. This occurs if either too many threads have been specified, or if too few processors are available at run time. If system

resources are over-allocated, this mode will cause poor performance. The throughput mode should be used instead if this occurs.

## OpenMP\* Environment Variables

This topic describes the OpenMP\* environment variables (with the `OMP_` prefix) and Intel-specific environment variables (with the `KMP_` prefix) that are Intel extensions.

### Standard Environment Variables

Variable	Default	Description
<code>OMP_SCHEDULE</code>	STATIC, no chunk size specified	Sets the run-time schedule type and chunk size.
<code>OMP_NUM_THREADS</code>	Number of processors	Sets the number of threads to use during execution.
<code>OMP_DYNAMIC</code>	.FALSE.	Enables (.TRUE.) or disables (.FALSE.) the dynamic adjustment of the number of threads.
<code>OMP_NESTED</code>	.FALSE.	Enables (.TRUE.) or disables (.FALSE.) nested parallelism.

### Intel Extension Environment Variables

Environment Variable	Default	Description
<code>KMP_ALL_THREADS</code>	<code>max(32, 4 * OMP_NUM_THREADS, 4 * number of processors)</code>	Sets the maximum number of threads that can be used by any parallel region.
<code>KMP_BLOCKTIME</code>	200 milliseconds	Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping.  See also the throughput execution mode and the <code>KMP_LIBRARY</code> environment variable. Use the optional character suffix s, m, h, or d, to specify seconds, minutes, hours, or days.
<code>KMP_LIBRARY</code>	throughput (execution mode)	Selects the OpenMP run-time library execution mode. The options for the variable value are <code>throughput</code> or <code>turnaround</code> .
<code>KMP_MONITOR_STACKSIZE</code>	<code>max(32k, system minimum thread stack size)</code>	Sets the number of bytes to allocate for the monitor thread, which is used for book-keeping during program execution. Use the optional suffix b, k, m, g, or t, to specify bytes, kilobytes, megabytes,

		gigabytes, or terabytes.
KMP_STACKSIZE	2m: IA-32 compiler  4m: Itanium® and Intel® EM64T compilers	Sets the number of bytes to allocate for each parallel thread to use as its private stack. Use the optional suffix b, k, m, g, or t, to specify bytes, kilobytes, megabytes, gigabytes, or terabytes.
KMP_VERSION	Disabled	Enables (set) or disables (unset) the printing of OpenMP run-time library version information during program execution.

## OpenMP\* Run-time Library Routines

OpenMP\* provides several run-time library routines to help you manage your program in parallel mode. Many of these run-time library routines have corresponding environment variables that can be set as defaults. The run-time library routines let you dynamically change these factors to assist in controlling your program. In all cases, a call to a run-time library routine overrides any corresponding environment variable.

The following tables specify the interfaces to these routines. The names for the routines are in user name space. Header files are provided in the `INCLUDE` directory of your compiler installation:

- The `omp.h` and `omp_lib.h` header files are provided in the `INCLUDE` directory of your compiler installation.

There are definitions for two different locks, `OMP_LOCK_KIND` and `OMP_NEST_LOCK_KIND`, which are used by the functions in the table that follows.

This topic provides a summary of the OpenMP run-time library routines. For detailed descriptions, see the OpenMP C/C++ version 2.5 specifications (<http://www.openmp.org/specs>).

## Execution Environment Routines

Function	Description
<code>omp_set_num_threads(<i>nthreads</i>)</code>	Sets the number of threads to use for subsequent parallel regions.
<code>omp_get_num_threads()</code>	Returns the number of threads that are being used in the current parallel region.
<code>omp_get_max_threads()</code>	Returns the maximum number of threads that are available for parallel execution.
<code>omp_get_thread_num()</code>	Returns the unique thread number of the thread currently executing this section of code.
<code>omp_get_num_procs()</code>	Returns the number of processors available to the program.



<code>omp_in_parallel()</code>	Returns <code>TRUE</code> if called within the dynamic extent of a parallel region executing in parallel; otherwise returns <code>FALSE</code> .
<code>omp_set_dynamic(dynamic_threads)</code>	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If <code>dynamic_threads</code> is <code>TRUE</code> , dynamic threads are enabled. If <code>dynamic_threads</code> is <code>FALSE</code> , dynamic threads are disabled. Dynamics threads are disabled by default.
<code>omp_get_dynamic()</code>	Returns <code>TRUE</code> if dynamic thread adjustment is enabled, otherwise returns <code>FALSE</code> .
<code>omp_set_nested(nested)</code>	Enables or disables nested parallelism. If <code>nested</code> is <code>TRUE</code> , nested parallelism is enabled. If <code>nested</code> is <code>FALSE</code> , nested parallelism is disabled. Nested parallelism is disabled by default.
<code>omp_get_nested()</code>	Returns <code>TRUE</code> if nested parallelism is enabled, otherwise returns <code>FALSE</code>

## Lock Routines

Function	Description
<code>omp_init_lock(lock)</code>	Initializes the lock associated with <code>lock</code> for use in subsequent calls.
<code>omp_destroy_lock(lock)</code>	Causes the lock associated with <code>lock</code> to become undefined.
<code>omp_set_lock(lock)</code>	Forces the executing thread to wait until the lock associated with <code>lock</code> is available. The thread is granted ownership of the lock when it becomes available.
<code>omp_unset_lock(lock)</code>	Releases the executing thread from ownership of the lock associated with <code>lock</code> . The behavior is undefined if the executing thread does not own the lock associated with <code>lock</code> .
<code>omp_test_lock(lock)</code>	Attempts to set the lock associated with <code>lock</code> . If successful, returns <code>TRUE</code> , otherwise returns <code>FALSE</code> .
<code>omp_init_nest_lock(lock)</code>	Initializes the nested lock associated with <code>lock</code> for use in the subsequent calls.
<code>omp_destroy_nest_lock(lock)</code>	Causes the nested lock associated with <code>lock</code> to become undefined.
<code>omp_set_nest_lock(lock)</code>	Forces the executing thread to wait until the nested lock associated with <code>lock</code> is available. The thread is granted ownership of the nested lock when it becomes available.

<code>omp_unset_nest_lock(lock)</code>	Releases the executing thread from ownership of the nested lock associated with <i>lock</i> if the nesting count is zero. Behavior is undefined if the executing thread does not own the nested lock associated with <i>lock</i> .
<code>omp_test_nest_lock(lock)</code>	Attempts to set the nested lock associated with <i>lock</i> . If successful, returns the nesting count, otherwise returns zero.

## Timing Routines

Function	Description
<code>omp_get_wtime()</code>	Returns a double-precision value equal to the elapsed wallclock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution.
<code>omp_get_wtick()</code>	Returns a double-precision value equal to the number of seconds between successive clock ticks.

## Intel Extension Routines/Functions

The Intel® compiler implements the following group of routines as an extensions to the OpenMP\* run-time library:

- Getting and setting stack size for parallel threads
- Memory allocation

The Intel extension routines described in this section can be used for low-level debugging to verify that the library code and application are functioning as intended. It is recommended to use these routines with caution because using them requires the use of the `-openmp-stubs` (Linux\*) or `/Qopenmp-stubs` (Windows\*) command-line option to execute the program sequentially. These routines are also generally not recognized by other vendor's OpenMP-compliant compilers, which may cause the link stage to fail for these other compilers.

### Note

The following functions require the pre-processor directive `#include <omp.h>`.

## Stack Size

In most cases, environment variables can be used in place of the extension library routines. For example, the stack size of the parallel threads may be set using the `KMP_STACKSIZE` environment variable rather than the `KMP_SET_STACKSIZE()` or `KMP_SET_STACKSIZE_S()` library routine.

**Note**

A run-time call to an Intel extension routine takes precedence over the corresponding environment variable setting.

The routines `KMP_SET_STACKSIZE()` and `KMP_GET_STACKSIZE()` take a 32-bit argument only. The routines `KMP_SET_STACKSIZE_S()` and `KMP_GET_STACKSIZE_S()` take a `SIZE_T` argument, which can hold 64-bit integers.

On Itanium®-based systems, it is recommended to always use `KMP_SET_STACKSIZE_S()` and `KMP_GET_STACKSIZE_S()`. These `_S()` variants must be used if you need to set a stack size  $\geq 2^{32}$  bytes (4 gigabytes).

**Stack Size**

Function	Description
<code>kmp_get_stacksize_s()</code>	Returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed with <code>kmp_set_stacksize_s()</code> prior to the first parallel region or with the <code>KMP_STACKSIZE</code> environment variable.
<code>kmp_get_stacksize()</code>	This function is provided for backwards compatibility only. Use <code>kmp_get_stacksize_s()</code> for compatibility across different families of Intel processors.
<code>kmp_set_stacksize_s(size)</code>	Sets to <i>size</i> the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be set via the <code>KMP_STACKSIZE</code> environment variable. In order for <code>kmp_set_stacksize_s()</code> to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.
<code>kmp_set_stacksize(size)</code>	This function is provided for backward compatibility only; use <code>kmp_set_stacksize_s()</code> for compatibility across different families of Intel processors.

**Memory Allocation**

The Intel® compiler implements a group of memory allocation routines as an extension to the OpenMP\* run-time library to enable threads to allocate memory from a heap local to each thread. These routines are: `KMP_MALLOC`, `KMP_CALLOC`, and `KMP_REALLOC`.

The memory allocated by these routines must also be freed by the `KMP_FREE` routine. While it is legal for the memory to be allocated by one thread and freed (using

`kmp_free`) by a different thread, this mode of operation has a slight performance penalty.

## Memory Allocation

Function	Description
<code>kmp_malloc(size)</code>	Allocate memory block of <i>size</i> bytes from thread-local heap.
<code>kmp_calloc(nelem, elsize)</code>	Allocate array of <i>nelem</i> elements of size <i>elsize</i> from thread-local heap.
<code>kmp_realloc(ptr, size)</code>	Reallocate memory block at address <i>ptr</i> and <i>size</i> bytes from thread-local heap.
<code>kmp_free(ptr)</code>	Free memory block at address <i>ptr</i> from thread-local heap. Memory must have been previously allocated with <code>kmp_malloc()</code> , <code>kmp_calloc()</code> , or <code>kmp_realloc()</code> .

## Examples of OpenMP\* Usage

The following examples show how to use the OpenMP\* features.

See more examples in the OpenMP C/C++ version 2.5 specifications (<http://www.openmp.org/specs>).

## A Simple Difference Operator

This example shows a simple parallel loop where the amount of work in each iteration is different. Dynamic scheduling is used to get good load balancing. The `for` has a `nowait` because there is an implicit `barrier` at the end of the parallel region.

Example
<pre>void for1(float a[], float b[], int n) {     int i, j;     #pragma omp parallel shared(a,b,n) private(i,j)     {         #pragma omp for schedule(dynamic,1) nowait         for (i = 1; i &lt; n; i++)             for (j = 0; j &lt;= i; j++)                 b[j + n*i] = (a[j + n*i] + a[j + n*(i-1)]) / 2.0;     } }</pre>

## Two Difference Operators

This example uses two parallel loops fused to reduce fork/join overhead. The first `for` has a `nowait` because all the data used in the second loop is different than all the data used in the first loop.

**Example**

```

void for2(float a[], float b[], float c[], float d[],
          int n, int m)
{
    int i, j;
    #pragma omp parallel shared(a,b,c,d,n,m) private(i,j)
    {
        #pragma omp for schedule(dynamic,1) nowait
        for (i = 1; i < n; i++)
            for (j = 0; j <= i; j++)
                b[j + n*i] = ( a[j + n*i] + a[j + n*(i-1)] )/2.0;
        #pragma omp for schedule(dynamic,1) nowait
        for (i = 1; i < m; i++)
            for (j = 0; j <= i; j++)
                d[j + m*i] = ( c[j + m*i] + c[j + m*(i-1)] )/2.0;
    }
}

```

## OpenMP\* Advanced Issues

This topic discusses how to use the OpenMP\* library functions and environment variables, and discusses some guidelines for enhancing performance with OpenMP\*.

OpenMP\* provides specific function calls, and environment variables. See the following topics to refresh your memory about the primary functions and environment variable used in this topic:

- OpenMP\* Run-time Library Routines
- OpenMP\* Environment Variables

To use the function calls, include the `omp.h` header file and compile the application using the `-openmp` (Linux\*) or `/Qopenmp` (Windows\*) option. No additional libraries are required for linking. See [Compiling with OpenMP\\*](#), [Directive Format](#), and [Diagnostics](#) for more information.

The following example, which demonstrates how to use the OpenMP\* functions to print the alphabet, also illustrates several important concepts.

First, when using function instead of pragmas your code must be rewritten; any rewrite can mean extra debugging, testing, and maintenance efforts.

Second, it becomes difficult to compile without OpenMP support.

Third, it is very easy to introduce bugs, as in the loop (below) that fails to print all the letters of the alphabet when the number of threads is not a multiple of 26.

Fourth and finally, you lose the ability to adjust loop scheduling without creating your own work-queue algorithm, which is a lot of extra effort. You are limited by your own scheduling, which is mostly likely static scheduling as shown in the example.

**Example**

```
#include <stdio.h>
#include <omp.h>
int main(void)
{
    omp_set_num_threads(4);
    int i;
    #pragma omp parallel private(i)
    { // OMP_NUM_THREADS is not a multiple of 26,
      // which can be considered a bug in this code.
      int LettersPerThread = 26 / omp_get_num_threads();
      int ThisThreadNum = omp_get_thread_num();
      int StartLetter = 'a'+ThisThreadNum*LettersPerThread;
      int EndLetter = 'a'+ThisThreadNum*LettersPerThread+LettersPerThread;
      for (i=StartLetter; i<EndLetter; i++)
          printf("%c", i);
    }
    printf("\n");
    return 0;
}
```

Debugging threaded applications is a complex process, because debuggers change the runtime performance, which can mask race conditions. Even print statements can mask issues, because they use synchronization and operating system functions. OpenMP\* adds even more complications. OpenMP\* inserts private variables, shared variables, and additional code that is impossible to examine and step through without a specialized debugger that supports OpenMP\*. When using OpenMP, your key debugging tool is the process of elimination.

Remember that most mistakes are race conditions. Most race conditions are caused by shared variables that really should have been declared private. Start by looking at the variables inside the parallel regions and make sure that the variables are declared private when necessary. Next, check functions called within parallel constructs. By default, variables declared on the stack are private, but the C/C++ keyword `static` will change the variable to be placed on the global heap and therefore shared for OpenMP loops.

The `default(none)` clause, shown below, can be used to help find those hard-to-spot variables. If you specify the `default(none)`, then every variable must be declared with a data-sharing attribute clause.

**Example**

```
#pragma omp parallel for default(none) private(x,y) shared(a,b)
```

Another common mistake is using uninitialized variables. Remember that private variables do not have initial values upon entering a parallel construct. Use the `firstprivate` and `lastprivate` clauses to initialize them only when necessary, because doing so adds extra overhead.

If you still can't find the bug, then consider the possibility of reducing the scope. Try a binary-hunt. Force parallel sections to be serial again with `if(0)` on the parallel construct or commenting out the pragma altogether. Another method is to force large chunks of a parallel region to be critical sections. Pick a region of the code that you think

contains the bug and place it within a critical section. Try to find the section of code that suddenly works when it is within a critical section and fails when it is not. Now look at the variables, and see if the bug is apparent. If that still doesn't work, try setting the entire program to run in serial by setting the compiler-specific environment variable

```
KMP_LIBRARY=serial.
```

If the code is still not working, compile it without the `-openmp` (Linux) or `/Qopenmp` (Windows) option to make sure the serial version works.

## Performance

OpenMP threaded application performance is largely dependent upon the following things:

- The underlying performance of the single-threaded code.
- CPU utilization, idle threads, and poor load balancing.
- The percentage of the application that is executed in parallel.
- The amount of synchronization and communication among the threads.
- The overhead needed to create, manage, destroy, and synchronize the threads, made worse by the number of single-to-parallel or parallel-to-single transitions called fork-join transitions.
- Performance limitations of shared resources such as memory, bus bandwidth, and CPU execution units.
- Memory conflicts caused by shared memory or falsely shared memory.

Threaded code performance is affected by two things:

- How well the single-threaded version runs
- How well you divide the work among multiple processors with the least amount of overhead.

Performance always begins with a properly constructed parallel algorithm or application. It should be obvious that parallelizing a bubble-sort, even one written in hand-optimized assembly language, is not a good place to start. Keep scalability in mind; creating a program that runs well on two CPUs is not as efficient as creating one that runs well on  $n$  CPUs. With OpenMP\* the number of threads is chosen by the compiler, so programs that work well regardless of the number of threads are highly desirable. Producer/consumer architectures are rarely efficient, because they are made specifically for two threads.

Once the algorithm is in place, make sure that the code runs efficiently on the targeted Intel® architecture; a single-threaded version can be a big help. Turn off the `-openmp` (Linux\*) or `/Qopenmp` (Windows\*) option to generate a single-threaded version, and run the single-threaded version through the usual set of optimizations. See Worksharing Using OpenMP\* for more information.

Once you have gotten the single-threaded performance, it is time to generate the multi-threaded version and start doing some analysis.

Start by looking at the amount of time spent in the idle loop of the operating system. The VTune™ Performance Analyzer is a great tool to help with the investigation. Idle time can indicate unbalanced loads, lots of blocked synchronization, and serial regions. Fix those issues, and then go back to the VTune™ Analyzer to look for excessive cache misses and memory issues like false-sharing. Solve these basic problems, and you will have a well-optimized parallel program that will run well on Hyper-Threading Technology as well as multiple physical CPUs.

Optimizations are really a combination of patience, experimentation, and practice. Make little test programs that mimic the way your application uses the computer's resources to get a feel for what things are faster than others. Be sure to try the different scheduling clauses for the parallel sections. If the overhead of a parallel region is large compared to the compute time, you may want to use an if clause to execute the section serially.

## THREADPRIVATE Directive

You can make named common blocks private to a thread, but global within the thread, by using the `THREADPRIVATE` directive.

Each thread gets its own copy of the common block with the result that data written to the common block by one thread is not directly visible to other threads. During serial portions and `MASTER` sections of the program, accesses are to the master thread copy of the common block.

You cannot use a thread private common block or its constituent variables in any clause other than the `COPYIN` clause.

In the following example, common blocks `BLK1` and `FIELDS` are specified as thread private:

### Example

```
COMMON /BLK1/ SCRATCH
COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE (/BLK1/, /FIELDS/)
```



## Auto-parallelization Overview

The auto-parallelization feature of the Intel® compiler automatically translates serial portions of the input program into equivalent multithreaded code. The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multithreaded code for those loops which can safely and efficiently be executed in parallel.

This behavior enables the potential exploitation of the parallel architecture found in symmetric multiprocessor (SMP) systems.

Automatic parallelization relieves the user from:

- Dealing with the details of finding loops that are good worksharing candidates
- Performing the dataflow analysis to verify correct parallel execution
- Partitioning the data for threaded code generation as is needed in programming with OpenMP\* directives.

The parallel run-time support provides the same run-time features as found in OpenMP, such as handling the details of loop iteration modification, thread scheduling, and synchronization.

While OpenMP directives enable serial applications to transform into parallel applications quickly, a programmer must explicitly identify specific portions of the application code that contain parallelism and add the appropriate compiler directives.

Auto-parallelization, which is triggered by the `-parallel` (Linux\*) or `/Qparallel` (Windows\*) option, automatically identifies those loop structures that contain parallelism. During compilation, the compiler automatically attempts to deconstruct the code sequences into separate threads for parallel processing. No other effort by the programmer is needed.

### Note

Intel® Itanium®-based systems: Specifying these options implies `-opt-mem-bandwidth1` (Linux) or `/Qopt-mem-bandwidth1` (Windows).

Serial code can be divided so that the code can execute concurrently on multiple threads. For example, consider the following serial code example.

#### Example 1: Original Serial Code

```
void ser(int *a, int *b, int *c)
{
    for (int i=0; i<100; i++)
        a[i] = a[i] + b[i] * c[i];
}
```

The following example illustrates one method showing how the loop iteration space, shown in the previous example, might be divided to execute on two threads.

### Example 2: Transformed Parallel Code

```
void par(int *a, int *b, int *c)
{
    int i;
    // Thread 1
    for (i=0; i<50; i++)
        a[i] = a[i] + b[i] * c[i];
    // Thread 2
    for (i=50; i<100; i++)
        a[i] = a[i] + b[i] * c[i];
}
```

## Programming with Auto-parallelization

The auto-parallelization feature implements some concepts of OpenMP\*, such as the worksharing construct (with the `PARALLEL FOR` directive). See Programming with OpenMP for worksharing construct. This section provides details on auto-parallelization.

## Guidelines for Effective Auto-parallelization Usage

A loop can be parallelized if it meets the following criteria:

- The loop is countable at compile time: this means that an expression representing how many times the loop will execute (also called "the loop trip count") can be generated just before entering the loop.
- There are no `FLOW` (`READ after WRITE`), `OUTPUT` (`WRITE after WRITE`) or `ANTI` (`WRITE after READ`) loop-carried data dependencies. A loop-carried data dependency occurs when the same memory location is referenced in different iterations of the loop. At the compiler's discretion, a loop may be parallelized if any assumed inhibiting loop-carried dependencies can be resolved by run-time dependency testing.

The compiler may generate a run-time test for the profitability of executing in parallel for loop with loop parameters that are not compile-time constants.

## Coding Guidelines

Enhance the power and effectiveness of the auto-parallelizer by following these coding guidelines:

- Expose the trip count of loops whenever possible; specifically use constants where the trip count is known and save loop parameters in local variables.
- Avoid placing structures inside loop bodies that the compiler may assume to carry dependent data, for example, procedure calls, ambiguous indirect references or global references.

## Auto-parallelization Data Flow

For auto-parallelization processing, the compiler performs the following steps:

1. Data flow analysis: Computing the flow of data through the program.
2. Loop classification: Determining loop candidates for parallelization based on correctness and efficiency, as shown by threshold analysis.
3. Dependency analysis: Computing the dependency analysis for references in each loop nest.
4. High-level parallelization: Analyzing dependency graph to determine loops which can execute in parallel, and computing run-time dependency.
5. Data partitioning: Examining data reference and partition based on the following types of access: `SHARED`, `PRIVATE`, and `FIRSTPRIVATE`.
6. Multi-threaded code generation: Modifying loop parameters, generating entry/exit per threaded task, and generating calls to parallel run-time routines for thread creation and synchronization.

## Programming for Multithread Platform Consistency

For applications where most of the computation is carried out in simple loops, Intel compilers may be able to generate a multithreaded version automatically. This information applies to applications built for deployment on symmetric multiprocessors (SMP), systems with Hyper-Threading Technology (HT Technology) enabled, and dual-core processor systems.

The compiler can analyze dataflow in loops to determine which loops can be safely and efficiently executed in parallel. Automatic parallelization can sometimes result in shorter execution times. Compiler enabled auto-parallelization can help reduce the time spent performing several common tasks:

- searching for loops that are good candidates for parallel execution
- performing dataflow analysis to verify correct parallel execution
- adding parallel compiler directives manually

Parallelization is subject to certain conditions, which are described in the next section. If `-openmp` and `-parallel` (Linux\*) or `/Qopenmp` and `/Qparallel` (Windows\*) are both specified on the same command line, the compiler will only attempt to parallelize those functions that do not contain OpenMP\* directives.

The following program contains a loop with a high iteration count:

### Example

```
#include <math.h>
void no_dep()
{
    int a, n = 100000000;
    float c[n];
    for (int i = 0; i < n; i++) {
```

```

    a = 2 * i - 1;
    c[i] = sqrt(a);
}

```

Dataflow analysis confirms that the loop does not contain data dependencies. The compiler will generate code that divides the iterations as evenly as possible among the threads at runtime. The number of threads defaults to the number of processors but can be set independently using the `OMP_NUM_THREADS` environment variable. The increase in parallel speed for a given loop depends on the amount of work, the load balance among threads, the overhead of thread creation and synchronization, etc., but generally will be less than the number of threads. For a whole program, speed increases depend on the ratio of parallel to serial computation.

For builds with separate compiling and linking steps, be sure to link the OpenMP\* runtime library when using automatic parallelization. The easiest way to do this is to use the Intel® compiler driver for linking.

## Parallelizing Loops

Three requirements must be met for the compiler to parallelize a loop.

1. The number of iterations must be known before entry into a loop so that the work can be divided in advance. A while-loop, for example, usually cannot be made parallel.
2. There can be no jumps into or out of the loop.
3. The loop iterations must be independent.

In other words, correct results must not logically depend on the order in which the iterations are executed. There may, however, be slight variations in the accumulated rounding error, as, for example, when the same quantities are added in a different order. In some cases, such as summing an array or other uses of temporary scalars, the compiler may be able to remove an apparent dependency by a simple transformation.

Potential aliasing of pointers or array references is another common impediment to safe parallelization. Two pointers are aliased if both point to the same memory location. The compiler may not be able to determine whether two pointers or array references point to the same memory location. For example, if they depend on function arguments, run-time data, or the results of complex calculations. If the compiler cannot prove that pointers or array references are safe and that iterations are independent, the compiler will not parallelize the loop, except in limited cases when it is deemed worthwhile to generate alternative code paths to test explicitly for aliasing at run-time. If you know parallelizing a particular loop is safe and that potential aliases can be ignored, you can instruct the compiler to parallelize the loop using the `#pragma parallel pragma`.

An alternative way in C to assert that a pointer is not aliased is to use the `restrict` keyword in the pointer declaration, along with the `-restrict` (Linux) or `/Qrestrict` (Windows) command-line option. The compiler will never parallelize a loop that it can prove to be unsafe.

The compiler can only effectively analyze loops with a relatively simple structure. For example, the compiler cannot determine the thread safety of a loop containing external function calls because it does not know whether the function call might have side effects that introduce dependences. You can invoke interprocedural optimization with the `-ipo` (Linux) or `/Qipo` (Windows) compiler option. Using this option gives the compiler the opportunity to analyze the called function for side effects.

When the compiler is unable to parallelize automatically loops you know to be parallel use OpenMP\*. OpenMP\* is the preferred solution because you, as the developer, understand the code better than the compiler and can express parallelism at a coarser granularity. On the other hand, automatic parallelization can be effective for nested loops, such as those in a matrix multiply. Moderately coarse-grained parallelism results from threading of the outer loop, allowing the inner loops to be optimized for fine-grained parallelism using vectorization or software pipelining.

If a loop can be parallelized, it's not always the case that it should be parallelized. The compiler uses a threshold parameter to decide whether to parallelize a loop. The `-par-threshold` (Linux) or `/Qpar-threshold` (Windows) compiler option adjusts this behavior. The threshold ranges from 0 to 100, where 0 instructs the compiler to always parallelize a safe loop and 100 instructs the compiler to only parallelize those loops for which a performance gain is highly probable. Use the `-par-report` (Linux) or `/Qpar-report` (Windows) compiler option to determine which loops were parallelized. The compiler will also report which loops could not be parallelized indicate a probably reason why it could not be parallelized. See Auto-parallelization: Threshold Control and Diagnostics for more information on the using these compiler options.

The following example illustrates using the options in combination. Assume you have the following code:

### Example code

```
void add (int k, float *a, float *b)
{
  for (int i = 1; i < 10000; i++)
    a[i] = a[i+k] + b[i];
}
```

Entering a command-line compiler command similar to the following will result in the compiler issuing parallelization messages:

Platform	Example Command
Linux*	<code>icpc -c -parallel -par-report3 add.cpp</code>
Windows*	<code>icl /c /Qparallel /Qpar-report3 add.cpp</code>

The compiler might report results similar to those listed below:

### Sample results

```
add.cpp
  procedure:
    add serial loop: line 2
      anti data dependence assumed from line 2 to line 2, due to "a"
```

```
flow data dependence assumed from line 2 to line 2, due to "a"
flow data dependence assumed from line 2 to line 2, due to "a"
```

Because the compiler does not know the value of  $k$ , the compiler assumes the iterations depend on each other, for example if  $k$  equals -1, even if the actual case is otherwise. You can override the compiler inserting `#pragma parallel`:

### Example

```
void add(int k, float *a, float *b)
{
    #pragma parallel
    for (int i = 0; i < 10000; i++)
        a[i] = a[i+k] + b[i];
}
```

As the developer, it's your responsibility to not call this function with a value of  $k$  that is less than 10000; passing a value less than 10000 could to incorrect results.

## Thread Pooling

Thread pools offer an effective approach to managing threads. A thread pool is a group of threads waiting for work assignments. In this approach, threads are created once during an initialization step and terminated during a finalization step. This simplifies the control logic for checking for failures in thread creation midway through the application and amortizes the cost of thread creation over the entire application. Once created, the threads in the thread pool wait for work to become available. Other threads in the application assign tasks to the thread pool. Typically, this is a single thread called the thread manager or dispatcher. After completing the task, each thread returns to the thread pool to await further work. Depending upon the work assignment and thread pooling policies employed, it is possible to add new threads to the thread pool if the amount of work grows. This approach has the following benefits:

- Possible runtime failures midway through application execution due to inability to create threads can be avoided with simple control logic.
- Thread management costs from thread creation are minimized. This in turn leads to better response times for processing workloads and allows for multithreading of finer-grained workloads.

A typical usage scenario for thread pools is in server applications, which often launch a thread for every new request. A better strategy is to queue service requests for processing by an existing thread pool. A thread from the pool grabs a service request from the queue, processes it, and returns to the queue to get more work.

Thread pools can also be used to perform overlapping asynchronous I/O. The I/O completion ports provided with the Win32\* API allow a pool of threads to wait on an I/O completion port and process packets from overlapped I/O operations.

OpenMP\* is strictly a fork/join threading model. In some OpenMP implementations, threads are created at the start of a parallel region and destroyed at the end of the parallel region. OpenMP applications typically have several parallel regions with

intervening serial regions. Creating and destroying threads for each parallel region can result in significant system overhead, especially if a parallel region is inside a loop; therefore, the Intel OpenMP implementation uses thread pools. A pool of worker threads is created at the first parallel region. These threads exist for the duration of program execution. More threads may be added automatically if requested by the program. The threads are not destroyed until the last parallel region is executed.

Thread pools can be created on Windows and Linux using the thread creation API. For instance, a custom thread pool using Win32 threads may be created as follows:

### Example

```
// Initialization method/function
{
    DWORD tid;
    // Create initial pool of threads
    for (int i = 0; i < MIN_THREADS; i++)
    {
        HANDLE *ThHandle = CreateThread (NULL, 0, CheckPoolQueue, NULL, 0, &tid);
        if (ThHandle == NULL)
            // Handle Error
        else
            RegisterPoolThread (ThHandle);
    }
}
```

The function `CheckPoolQueue` executed by each thread in the pool is designed to enter a wait state until work is available on the queue. The thread manager can keep track of pending jobs in the queue and dynamically increase the number of threads in the pool based on the demand.

## Auto-parallelization: Enabling, Options, Directives, and Environment Variables

To enable the auto-parallelizer, use the `-parallel` (Linux\*) or `/Qparallel` (Windows\*) option. This option detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops. An example of the command using auto-parallelization is as follows:

Platform	Description
Linux	<code>icpc -c -parallel prog.cpp</code>
Windows	<code>icl /c /Qparallel prog.cpp</code>

### Auto-parallelization Options

The `-parallel` (Linux) or `/Qparallel` (Windows) option enables the auto-parallelizer if the `-O2` or `-O3` (Linux) or `/O2` or `/O3` (Windows) optimization option is also specified. This option detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops.

Windows	Linux	Description
<code>/Qparallel</code>	<code>-parallel</code>	Enables the auto-parallelizer.
<code>/Qpar-threshold</code>	<code>-par-threshold</code>	Controls the work threshold needed for auto-parallelization.
<code>/Qpar-report</code>	<code>-par-report</code>	Controls the diagnostic messages from the auto-parallelizer, see later subsection.

#### Note

See Parallelism Overview for more information about the options listed above. Intel® Itanium®-based systems: Specifying these options implies `-opt-mem-bandwidth1` (Linux) or `/Qopt-mem-bandwidth1` (Windows).

### Auto-parallelization Environment Variables

Option Variable	Default	Description
<code>OMP_NUM_THREADS</code>	Number of processors currently installed in the system while generating the executable	Controls the number of threads used.
<code>OMP_SCHEDULE</code>	Static	Specifies the type of run-time scheduling.



## Auto-parallelization: Threshold Control and Diagnostics

### Threshold Control

The `-par-threshold{n}` (Linux\*) or `/Qpar-threshold[:n]` (Windows\*) option sets a threshold for auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. The value of  $n$  can be from 0 to 100.

### Diagnostics

The `-par-report` (Linux) or `/Qpar-report` (Windows) option controls the diagnostic levels 0, 1, 2, or 3 of the auto-parallelizer. Specify level 3 to get the most information possible from the option.

For example, assume you want a full diagnostic report on the following example code:

#### Example 1: Sample code

```
void no_par(void)
{
    int i;
    int a[1000];
    for (i=1; i<1000; i++) {
        a[i] = (i * 2) % i * 1 + sqrt(i);
        a[i] = a[i-1] + i;
    }
}
```

You can use `-par-report3` (Linux) or `/Qpar-report3` (Windows) by entering a command similar to the following:

Platform	Commands
Linux	<code>icpc -parallel -par-report3 -c diag_prog.c</code>
Windows	<code>icl /Qparallel /Qpar-report3 /c diag_prog.c</code>

where `-c` (Linux) or `/c` (Windows) instructs the compiler to compile the example without generating an executable.

The following example output illustrates the diagnostic report generated by the compiler for the example code shown above.

#### Example 2: Sample Code Report Output

```
procedure: no_par
serial loop: line 29
    flow data dependence from line 30 to line 31, due to "a"
    flow data dependence from line 31 to line 31, due to "a"
```

## Troubleshooting Tips

- Use `-par-threshold0` (Linux) or `/Qpar-threshold:0` (Windows) to auto-parallelize loops regardless of computational work.
- Use `-par-report3` (Linux) or `/Qpar-report3` (Windows) to view diagnostics (see example above).
- Use `-ipo[value]` (Linux) or `/Qipo` (Windows) to eliminate assumed side-effects done to function calls.

## Vectorization Overview (IA-32 and Intel® EM64T)

The vectorizer is a component of the Intel® compiler that automatically uses SIMD instructions in the MMX™, SSE, SSE2, and SSE3 instruction sets. The vectorizer detects operations in the program that can be done in parallel, and then converts the sequential operations like one SIMD instruction that processes 2, 4, 8 or up to 16 elements in parallel, depending on the data type.

This section provides options description, guidelines, and examples for Intel® compiler vectorization implemented by IA-32 compiler only.

The section discusses the following topics, among others:

- High-level discussion of compiler options used to control or influence vectorization
- Vectorization Key Programming Guidelines
- Loop parallelization and vectorization
- Descriptions of the C++ language features to control vectorization
- Discussion and general guidelines on vectorization levels:
  - automatic vectorization
  - vectorization with user intervention
- Examples demonstrating typical vectorization issues and resolutions

The compiler supports a variety of directives that can help the compiler to generate effective vector instructions. See Vectorization Support.

See The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance, A.J.C. Bik. Intel Press, June, 2004, for a detailed discussion of how to vectorize code using the Intel® compiler. Additionally, see the Related Publications topic in this document for other resources.

## Vectorizer Options

Vectorization within the Intel® compiler depends upon its ability to disambiguate memory references. Certain options may enable the compiler to do better vectorization.

These options can enable other optimizations in addition to vectorization. Keep the following guidelines in mind when using these options:

- When either the `-x` or `-ax` (Linux\*) or `/Qx` or `/Qax` (Windows\*) options are used and `-O2` (Linux) or `/O2` (Windows) is also specified, vectorizer is enabled.
- The `-x` (Linux) or `/Qx` (Windows) or `-ax` (Linux) or `/Qax` (Windows) options also enable vectorization with either the `-O1` and `-O3` (Linux) or `/O1` and `/O3` (Windows) options.
- `-xP` and `-axP` are the only valid processor values for Mac OS\* systems.

See Parallelism Overview for more information about options used in vectorization. See Vectorization Report for information on generating vectorization reports.

## Key Programming Guidelines for Vectorization

The goal of vectorizing compilers is to exploit single-instruction multiple data (SIMD) processing automatically. Users can help however by supplying the compiler with additional information; for example, by using directives.

### Guidelines

You will often need to make some changes to your loops. Follow these guidelines for loop bodies.

#### Use:

- straight-line code (a single basic block)
- vector data only; that is, arrays and invariant expressions on the right hand side of assignments. Array references can appear on the left hand side of assignments.
- only assignment statements

#### Avoid:

- function calls
- unvectorizable operations (other than mathematical)
- mixing vectorizable types in the same loop
- data-dependent loop exit conditions

To make your code vectorizable, you will often need to make some changes to your loops. However, you should make only the changes needed to enable vectorization and no others. In particular, you should avoid these common changes:

- loop unrolling; the compiler does it automatically.
- decomposing one loop with several statements in the body into several single-statement loops.

### Restrictions

There are a number of restrictions that you should be consider. Vectorization depends on two major factors: hardware and style of source code.

Factor	Description
Hardware	The compiler is limited by restrictions imposed by the underlying hardware. In the case of Streaming SIMD Extensions, the vector memory operations are limited to stride-1 accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes

	a loop as vectorizable, it still might not vectorize it for a distinct target architecture.
Style of source code	The style in which you write source code can inhibit optimization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove that two memory references refer to distinct locations. Consequently, this prevents certain reordering transformations.

Many stylistic issues that prevent automatic vectorization by compilers are found in loop structures. The ambiguity arises from the complexity of the keywords, operators, data references, and memory operations within the loop bodies.

However, by understanding these limitations and by knowing how to interpret diagnostic messages, you can modify your program to overcome the known limitations and enable effective vectorization. The following sections summarize the capabilities and restrictions of the vectorizer with respect to loop structures.

## Loop Parallelization and Vectorization

Combine the `-parallel` (Linux\*) or `/Qparallel` (Windows\*) and `-x` (Linux) or `/Qx` (Windows) options to instructs the compiler to attempt both automatic loop parallelization and automatic loop vectorization in the same compilation.

In most cases, the compiler will consider outermost loops for parallelization and innermost loops for vectorization. If deemed profitable, however, the compiler may even apply loop parallelization and vectorization to the same loop.

See [Guidelines for Effective Auto-parallelization Usage and Vectorization Key Programming Guidelines](#).

In some rare cases successful loop parallelization (either automatically or by means of OpenMP\* directives) may affect the messages reported by the compiler for a non-vectorizable loop in a non-intuitive way; for example, in the cases where `-vec-report2` (Linux) or `/Qvec-report2` (Windows) option indicating loops were not successfully vectorized.

See [Vectorization Report](#).

## Types of Vectorized Loops

For integer loops, the 64-bit MMX™ technology and 128-bit Streaming SIMD Extensions (SSE) provide SIMD instructions for most arithmetic and logical operators on 32-bit, 16-bit, and 8-bit integer data types.

Vectorization may proceed if the final precision of integer wrap-around arithmetic will be preserved. A 32-bit shift-right operator, for instance, is not vectorized in 16-bit mode if the final stored value is a 16-bit integer. Also, note that because the MMX™ and SSE

instruction sets are not fully orthogonal (shifts on byte operands, for instance, are not supported), not all integer operations can actually be vectorized.

For loops that operate on 32-bit single-precision and 64-bit double-precision floating-point numbers, SSE provides SIMD instructions for the following arithmetic operators: addition (+), subtraction (-), multiplication (\*), and division (/).

Additionally, the Streaming SIMD Extensions provide SIMD instructions for the binary `MIN` and `MAX` and unary `SQRT` operators. SIMD versions of several other mathematical operators (like the trigonometric functions `SIN`, `COS`, and `TAN`) are supported in software in a vector mathematical run-time library that is provided with the Intel® compiler of which the compiler takes advantage.

## Statements in the Loop Body

The vectorizable operations are different for floating-point and integer data.

## Floating-point Array Operations

The statements within the loop body may contain float operations (typically on arrays). The following arithmetic operations are supported: addition, subtraction, multiplication, division, negation, square root, `MAX`, `MIN`, and mathematical functions such as `SIN` and `COS`.

Operation on `DOUBLE PRECISION` types is not valid, unless optimizing for a Pentium® 4 and Intel® Xeon® processors system, using the `-xW` (Linux\*) or `/QxW` (Windows\*) or `-axW` (Linux) or `/QaxW` (Windows) compiler option.

## Integer Array Operations

The statements within the loop body may contain `char`, `unsigned char`, `short`, `unsigned short`, `int`, and `unsigned int`. Calls to functions such as `sqrt` and `fabs` are also supported. Arithmetic operations are limited to addition, subtraction, bitwise `AND`, `OR`, and `XOR` operators, division (16-bit only), multiplication (16-bit only), `min`, and `max`. You can mix data types only if the conversion can be done without a loss of precision. Some example operators where you can mix data types are multiplication, shift, or unary operators.

## Other Operations

No statements other than the preceding floating-point and integer operations are allowed. In particular, note that the special `__m64` and `__m128` datatypes are not vectorizable. The loop body cannot contain any function calls. Use of the Streaming SIMD Extensions intrinsics (`_mm_add_ps`) are not allowed.

## Data Dependency

Data dependency relations represent the required ordering constraints on the operations in serial loops. Because vectorization rearranges the order in which operations are executed, any auto-vectorizer must have at its disposal some form of data dependency analysis.

An example where data dependencies prohibit vectorization is shown below. In this example, the value of each element of an array is dependent on the value of its neighbor that was computed in the previous iteration.

### Example 1: Data-dependent Loop

```
int i;
void dep(float *data)
{
    for (i=1; i<100; i++)
        data[i] = data[i-1]*0.25 + data[i]*0.5 + data[i+1]*0.25;
}
```

The loop in the above example is not vectorizable because the `WRITE` to the current element `DATA(I)` is dependent on the use of the preceding element `DATA(I-1)`, which has already been written to and changed in the previous iteration. To see this, look at the access patterns of the array for the first two iterations as shown below.

### Example 2: Data-dependency Vectorization Patterns

```
for(i=0; i<100; i++)
a[i]=b[i];
has access pattern
read b[0]
write a[0]
read b[1]
write a[1]
i=1: READ data[0]
READ data[1]
READ data[2]
WRITE data[1]
i=2: READ data[1]
READ data[2]
READ data[3]
WRITE data[2]
```

In the normal sequential version of this loop, the value of `DATA(1)` read from during the second iteration was written to in the first iteration. For vectorization, it must be possible to do the iterations in parallel, without changing the semantics of the original loop.

## Data dependency Analysis

Data dependency analysis involves finding the conditions under which two memory accesses may overlap. Given two references in a program, the conditions are defined by:

- whether the referenced variables may be aliases for the same (or overlapping) regions in memory
- for array references, the relationship between the subscripts

For IA-32, data dependency analyzer for array references is organized as a series of tests, which progressively increase in power as well as in time and space costs.

First, a number of simple tests are performed in a dimension-by-dimension manner, since independency in any dimension will exclude any dependency relationship. Multidimensional arrays references that may cross their declared dimension boundaries can be converted to their linearized form before the tests are applied.

Some of the simple tests that can be used are the fast greatest common divisor (GCD) test and the extended bounds test. The GCD test proves independency if the GCD of the coefficients of loop indices cannot evenly divide the constant term. The extended bounds test checks for potential overlap of the extreme values in subscript expressions.

If all simple tests fail to prove independency, the compiler will eventually resort to a powerful hierarchical dependency solver that uses Fourier-Motzkin elimination to solve the data dependency problem in all dimensions.

## Loop Constructs

Loops can be formed with the usual `for` and `while` constructs. The loops must have a single entry and a single exit to be vectorized. The following examples illustrate loop constructs that can and cannot be vectorized.

### Example: Vectorizable structure

```
void vec(float a[], float b[], float c[])
{
    int i = 0;
    while (i < 100) {
// The if branch is inside body of loop.
        a[i] = b[i] * c[i];
        if (a[i] < 0.0)
            a[i] = 0.0;
        i++;
    }
}
```

The following example shows a loop that cannot be vectorized because of the inherent potential for an early exit from the loop.

### Example: Non-vectorizable structure

```
void no_vec(float a[], float b[], float c[])
{
    int i = 0;
    while (i < 100) {
        if (i < 50)
// The next statement is a second exit
// that allows an early exit from the loop.
            break;
    }
}
```



```

    ++i;
}
}

```

## Loop Exit Conditions

Loop exit conditions determine the number of iterations a loop executes. For example, fixed indexes for loops determine the iterations. The loop iterations must be countable; in other words, the number of iterations must be expressed as one of the following:

- A constant
- A loop invariant term
- A linear function of outermost loop indices

In the case where a loops exit depends on computation, the loops are not countable. The examples below show loop constructs that are countable and non-countable.

### Example: Countable Loop

```

void cnt1(float a[], float b[], float c[],
          int n, int lb)
{
    // Exit condition specified by "N-lb+1"
    int cnt=n, i=0;
    while (cnt >= lb) {
        // lb is not affected within loop.
        a[i] = b[i] * c[i];
        cnt--;
        i++;
    }
}

```

The following example demonstrates a different countable loop construct.

### Example: Countable Loop

```

void cnt2(float a[], float b[], float c[],
          int m, int n)
{
    // Number of iterations is "(n-m+2)/2".
    int i=0, l;
    for (l=m; l<n; l+=2) {
        a[i] = b[i] * c[i];
        i++;
    }
}

```

The following examples demonstrates a loop construct that is non-countable due to dependency loop variant count value.

### Example: Non-Countable Loop

```

void no cnt(float a[], float b[], float c[])
{
    int i=0;
    // Iterations dependent on a[i].
}

```

```

while (a[i]>0.0) {
    a[i] = b[i] * c[i];
    i++;
}

```

## Strip-mining and Cleanup

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

- It increases the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- It reduces the number of iterations of the loop by a factor of the length of each vector, or number of operations being performed per SIMD operation. In the case of Streaming SIMD Extensions, this vector or strip-length is reduced by 4 times: four floating-point data items per single Streaming SIMD Extensions single-precision floating-point SIMD operation are processed.

First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine.

The compiler automatically strip-mines your loop and generates a cleanup loop. For example, assume the compiler attempts to strip-mine the following loop:

### Example1: Before Vectorization

```

i=0;
while(i<n)
{
    // Original loop code
    a[i]=b[i]+c[i];
    ++i;
}

```

The compiler might handle the strip mining and loop cleaning by restructuring the loop in the following manner:

### Example 2: After Vectorization

```

// The vectorizer generates the following two loops
i=0;
while(i<(n-n%4))
{
    // Vector strip-mined loop
    // Subscript [i:i+3] denotes SIMD execution
    a[i:i+3]=b[i:i+3]+c[i:i+3];
    i=i+4;
}
while(i<n)
{
    // Scalar clean-up loop
}

```

```

a[i]=b[i]+c[i];
++i;
}

```

## Loop Blocking

It is possible to treat loop blocking as strip-mining in two or more dimensions. Loop blocking is a useful technique for memory performance optimization. The main purpose of loop blocking is to eliminate as many cache misses as possible. This technique transforms the memory domain into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse.

Consider the following example, loop blocking allows arrays **A** and **B** to be blocked into smaller rectangular chunks so that the total combined size of two blocked (A and B) chunks is smaller than cache size, which can improve data reuse.

### Example 3: Original loop

```

#include <time.h>
#include <stdio.h>
#define MAX 7000
void add(int a[][MAX], int b[][MAX]);
int main()
{
    int i, j;
    int A[MAX][MAX];
    int B[MAX][MAX];
    time_t start, elaspe;
    int sec;
    //Initialize array
    for(i=0;i<MAX;i++)
    {
        for(j=0;j<MAX;j++)
        {
            A[i][j]=j;
            B[i][j]=j;
        }
    }
    start= time(NULL);
    add(A, B);
    elaspe=time(NULL);
    sec = elaspe - start;
    printf("Time %d",sec); //List time taken to complete add function
}
void add(int a[][MAX], int b[][MAX])
{
    int i, j;
    for(i=0;i<MAX;i++)
    {
        for(j=0;j<MAX;j++)
        {
            a[i][j] = a[i][j] + b[j][i]; //Adds two matrices
        }
    }
}

```

The following example illustrates loop blocking the `add` function (from the previous example). In order to benefit from this optimization you might have to increase the cache size.

#### Example 4: Transformed Loop after blocking

```
#include <stdio.h>
#include <time.h>
#define MAX 7000
void add(int a[][MAX], int b[][MAX]);
int main()
{
#define BS 8 //Block size is selected as the loop-blocking factor.
int i, j;
int A[MAX][MAX];
int B[MAX][MAX];
time_t start, elaspe;
int sec;
//initialize array
for(i=0;i<MAX;i++)
{
    for(j=0;j<MAX;j++)
    {
        A[i][j]=j;
        B[i][j]=j;
    }
}
start= time(NULL);
add(A, B);
elaspe=time(NULL);
sec = elaspe - start;
printf("Time %d",sec); //Display time taken to complete loopBlocking
function
}
void add(int a[][MAX], int b[][MAX])
{
int i, j, ii, jj;
for(i=0;i<MAX;i+=BS)
{
    for(j=0; j<MAX;j+=BS)
    {
        for(ii=i; ii<i+BS; ii++)//outer loop
        {
            for(jj=j;jj<j+BS; jj++) //Array B experiences one cache miss
            { //for every iteration of outer loop
                a[ii][jj] = a[ii][jj] + b[jj][ii]; //Add the two arrays
            }
        }
    }
}
}
```

## Vectorization Examples

This section contains simple examples of some common issues in vector programming.

### Argument Aliasing: A Vector Copy

The loop in the following example, a vector copy operation, vectorizes because the compiler can prove `dest[i]` and `src[i]` are distinct.

#### Example: Vectorizable Copy Due To Unproven Distinction

```
void vec copy multi version(float *dest, float *src, int len)
{
    for (int i=0; i<len; i++)
        dest[i] = src[i];
}
```

The `restrict` keyword in the next example indicates that the pointers refer to distinct objects. Therefore, the compiler allows vectorization without generation of multi-version code.

#### Example: Using restrict to Prove Vectorizable Distinction

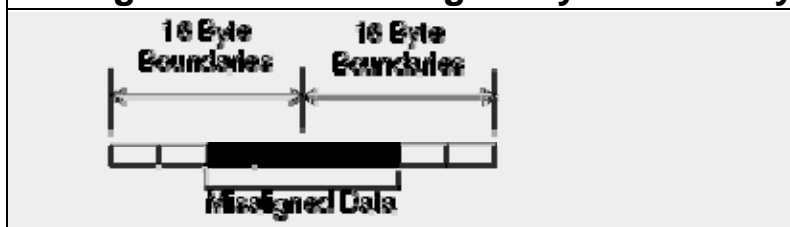
```
void vec copy(float *restrict dest, float *restrict src, int len)
{
    for (int i=0; i<len; i++)
        dest[i] = src[i];
}
```

## Data Alignment

A 16-byte (Linux\*) or 64-byte (Windows\*) or greater data structure or array should be aligned so that the beginning of each structure or array element is aligned in a way that its base address is a multiple of 16 (Linux) or 32 (Windows).

The figure (below) shows the effect of a data cache unit (DCU) split due to misaligned data. The code loads the misaligned data across a 16-byte boundary, which results in an additional memory access causing a six- to twelve-cycle stall. You can avoid the stalls if you know that the data is aligned and you specify to assume alignment

#### Misaligned Data Crossing 16-Byte Boundary



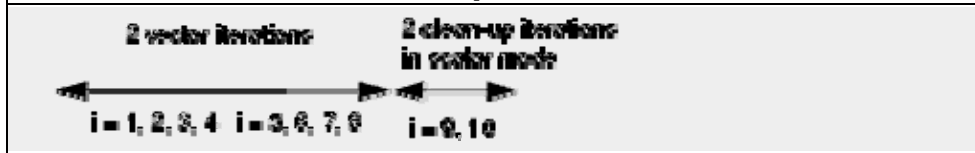
For example, if you know that elements `a[0]` and `b[0]` are aligned on a 16-byte boundary, then the following loop can be vectorized with the alignment option on (`#pragma vector aligned`):

### Example: Alignment of Pointers is Known

```
void aligned(float *a, float *b, int len)
{
    for (int i=0; i<len; i++)
        a[i] = b[i];
}
```

After vectorization, the loop is executed as shown in figure below

### Vector and Scalar Clean-up Iterations



Both the vector iterations `a[0:3] = b[0:3]`; and `a[4:7] = b[4:7]`; can be implemented with aligned moves if both the elements `a[0]` and `b[0]` (or, likewise, `a[4]` and `b[4]`) are 16-byte aligned.

#### Caution

If you use the vectorizer with incorrect alignment options the compiler will generate code with unexpected behavior. Specifically, using aligned moves on unaligned data, will result in an illegal instruction exception.

## Data Alignment Examples

This example contains a loop that vectorizes but only with unaligned memory instructions. The compiler can align the local arrays, but because `lb` is not known at compile-time. The correct alignment cannot be determined.

### Example: Loop Unaligned Due to Unknown Variable Value at Compile Time

```
void unaligned(int lb, float *a, float x, float *y, int len)
{
    for (int i=lb; i<len; i++)
        a[i] = a[i] * x + y[i];
}
```

If you know that `lb` is a multiple of 4, you can align the loop with `#pragma vector aligned` as shown in the example that follows:

### Example: Alignment Due to Assertion of Variable as Multiple of 4

```
#include <assert.h>
void assert_aligned(int lb, float *a, float x, float *y, int len)
{
    assert(lb%4 == 0);
    #pragma vector aligned
    for (int i=lb; i<len; i++)
        a[i] = a[i] * x + y[i];
}
```

## Loop Interchange and Subscripts: Matrix Multiply

Loop interchange need unit-stride constructs to be vectorized. Matrix multiplication is commonly written as shown in the following example:

### Example: Typical Matrix Multiplication

```
void matmul_slow(float *a[], float *b[], float *c[])
{
    int N = 100;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

The use of  $B(K, J)$  is not a stride-1 reference and therefore will not normally be vectorizable.

If the loops are interchanged, however, all the references will become stride-1 as shown in the following example.

### Example: Matrix Multiplication with Stride-1

```
void matmul_fast(float *a[], float *b[], float *c[])
{
    int N = 100;
    for (int i = 0; i < N; i++)
        for (int k = 0; k < N; k++)
            for (int j = 0; j < N; j++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

Interchanging is not always possible because of dependencies, which can lead to different results.

## Language Support and Directives

This topic addresses language features that better help to vectorize code. The `__declspec(align(n))` declaration enables you to overcome hardware alignment constraints. The restrict qualifier and the pragmas address the stylistic issues due to lexical scope, data dependency, and ambiguity resolution.

## Language Support

Feature	Description
<code>__declspec(align(n))</code>	Directs the compiler to align the variable to an <i>n</i> -byte boundary. Address of the variable is <i>address mod n=0</i> .
<code>__declspec(align(n,off))</code>	Directs the compiler to align the variable to an <i>n</i> -byte boundary with offset <i>off</i> within each <i>n</i> -byte boundary. Address of the variable is <i>address mod n=off</i> .
<code>restrict</code>	Permits the disambiguator flexibility in alias assumptions, which enables more vectorization.
<code>__assume_aligned(a,n)</code>	Instructs the compiler to assume that array <i>a</i> is aligned on an <i>n</i> -byte boundary; used in cases where the compiler has failed to obtain alignment information.
<code>#pragma ivdep</code>	Instructs the compiler to ignore assumed vector dependencies.
<code>#pragma vector {aligned unaligned always}</code>	Specifies how to vectorize the loop and indicates that efficiency heuristics should be ignored.
<code>#pragma novector</code>	Specifies that the loop should never be vectorized,

## Multi-version Code

Multi-version code is generated by the compiler in cases where data dependency analysis fails to prove independence for a loop due to the occurrence of pointers with unknown values. This functionality is referred to as dynamic dependency testing.

## Pragma Scope

These pragmas control the vectorization of only the subsequent loop in the program, but the compiler does not apply them to any nested loops. Each nested loop needs its own pragma preceding it in order for the pragma to be applied. You must place a pragma only before the loop control statement.

### #pragma vector always

Syntax
<code>#pragma vector always</code>

This pragma instructs the compiler to override any efficiency heuristic during the decision to vectorize or not. `#pragma vector always` will vectorize non-unit strides or very unaligned memory accesses.

Example
<pre>void vec_always(int *a, int *b, int m) {     #pragma vector always     for(int i = 0; i &lt;= m; i++)         a[32*i] = b[99*i]; }</pre>



## #pragma ivdep

### Syntax

```
#pragma ivdep
```

This pragma instructs the compiler to ignore assumed vector dependencies. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This pragma overrides that decision. Only use this when you know that the assumed loop dependencies are safe to ignore.

The loop in this example will not vectorize without the `ivdep` pragma, since the value of `k` is not known (vectorization would be illegal if `k < 0` ).

### Example

```
void ignore vec dep(int *a, int k, int c, int m)
{
    #pragma ivdep
    for (int i = 0; i < m; i++)
        a[i] = a[i + k] * c;
}
```

## #pragma vector

### Syntax

```
#pragma vector{aligned | unaligned}
```

The `vector` loop pragma means the loop should be vectorized, if it is legal to do so, ignoring normal heuristic decisions about profitability. When the `aligned` (or `unaligned`) qualifier is used with this pragma, the loop should be vectorized using `aligned` (or `unaligned`) operations. Specify one and only one of `aligned` or `unaligned`.

### Caution

If you specify `aligned` as an argument, you must be absolutely sure that the loop will be vectorizable using this instruction. Otherwise, the compiler will generate incorrect code.

The loop in the following example uses the `aligned` qualifier to request that the loop be vectorized with aligned instructions, as the arrays are declared in such a way that the compiler could not normally prove this would be safe to do so.

### Example

```
void vec aligned(float *a, int m, int c)
{
    int i;
    // Instruct compiler to ignore assumed vector dependencies.
    #pragma vector aligned
    for (i = 0; i < m; i++)
        a[i] = a[i] * c;
    // Alignment unknown but compiler can still align.
    for (i = 0; i < 100; i++)
        a[i] = a[i] + 1.0f;
```

```
}
```

The compiler has at its disposal several alignment strategies in cases where the alignment of data structures is not known at compile-time. A simple example is shown. If, in the loop, the alignment is unknown, the compiler will generate a prelude loop that iterates until the array reference that occurs most often hits an aligned address. The behavior makes the alignment properties of a known, and the vector loop is optimized accordingly.

### Alignment Strategies Example

```
float *a;
// alignment unknown
for (i = 0; i < 100; i++)
{
    a[i] = a[i] + 1.0f;
}
// dynamic loop peeling
p = a & 0x0f;
if (p != 0)
{
    p = (16 - p) / 4;
    for (i = 0; i < p; i++)
    {
        a[i] = a[i] + 1.0f;
    }
}
// loop with a aligned (will be vectorized accordingly)
for (i = p; i < 100; i++)
{
    a[i] = a[i] + 1.0f;
}
```

### #pragma novector

#### Syntax

```
#pragma novector
```

The `novector` loop pragma specifies that the loop should never be vectorized, even if it is legal to do so. In this example, suppose you know the trip count is too low to make vectorization worthwhile. You can use `#pragma novector` to tell the compiler not to vectorize, even if the loop is considered vectorizable.

### Example

```
void novec(int lb, int ub, float *a, float *b)
{
    #pragma novector
    for (int j = lb; j < ub; j++)
        a[j] = a[j] + b[j];
}
```

### #pragma vector nontemporal

#### Syntax

```
#pragma vector nontemporal
```

`#pragma vector nontemporal` results in streaming stores on Pentium® 4 based systems. An example loop (float type) together with the generated assembly are shown in the example. For large N, significant performance improvements result on a Pentium 4 systems over a non-streaming implementation.

### Example

```
#pragma vector nontemporal
for (i = 0; i < N; i++)
    a[i] = 1;
.B1.2:
movntps XMMWORD PTR a[eax], xmm0
movntps XMMWORD PTR a[eax+16], xmm0
add eax, 32
cmp eax, 4096
jnl .B1.2
```

### Dynamic Dependence Testing Example

#### Example

```
float *p, *q;
for (i = L; i <= U; i++)
{
    p[i] = q[i];
}
...
pL = p * 4*L;
pH = p + 4*U;
qL = q + 4*L;
qH = q + 4*U;
if (pH < qL || pL > qH)
{
    // loop without data dependency
    for (i = L; i <= U; i++)
    {
        p[i] = q[i];
    } else {
        for (i = L; i <= U; i++)
        {
            p[i] = q[i];
        }
    }
}
```

## Optimization Support Features Overview

This section describes the Intel® compiler features such as pragmas, directives, intrinsics, and run-time library routines, which enhance your application performance in support of compiler optimizations. These features are language extensions that enable you optimize your source code directly.

This section includes examples of optimizations supported by Intel extended directives and intrinsics or library routines that enhance and help analyze performance. Start with the Compiler Directives Overview.

See C++ Intrinsics Reference for more information about the available intrinsics.

## Compiler Directives Overview

This section discusses the language extended directives used in:

- Software Pipelining for Itanium®-based Applications
- Loop Count and Loop Distribution
- Loop Unrolling Support
- Vectorization Support
- Prefetch Support
- Optimization Restriction Support

## Pipelining for Itanium®-based Applications

The `SWP` pragma indicates preference for loops to be software pipelined. The pragma does not help data dependency, but overrides heuristics based on profile counts or unequal control flow.

The syntax for this pragma is shown below:

Syntax
<code>#pragma swp</code> <code>#pragma noswp</code>

The Software Pipelining optimization triggered by the `SWP` directive applies instruction scheduling to certain innermost loops, allowing instructions within a loop to be split into different stages, allowing increased instruction level parallelism.

This strategy can reduce the impact of long-latency operations, resulting in faster loop execution. Loops chosen for software pipelining are always innermost loops that do not contain procedure calls that are not inlined. Because the optimizer no longer considers fully unrolled loops as innermost loops, fully unrolling loops can allow an additional loop to become the innermost loop (see loop unrolling options).

You can view an optimization report to see whether software pipelining was applied (see Optimizer Report Generation).

The following example demonstrates on way of using the pragma to instruct the compiler to attempt software pipelining.

Example: Using SWP
<pre>void swp(int a[], int b[]) {     #pragma swp     for (int i = 0; i &lt; 100; i++)         if (a[i] == 0)             b[i] = a[i] + 1;         else             b[i] = a[i] * 2; }</pre>

## Loop Count and Loop Distribution

### Loop Count

The `loop count` pragma indicates the loop count is likely to be an integer constant. The syntax for this pragma is shown below:

#### Syntax

```
#pragma loop count (n)
```

where  $n$  is an integer value.

The value of loop count affects heuristics used in software pipelining and data prefetch.

#### Example: Using loop count

```
void loop count(int a[], int b[])
{
    #pragma loop count (10000)
    for (int i=0; i<1000; i++)
    // This should enable
    // software pipelining for this loop.
        a[i] = b[i] + 1.2;
}
```

### Loop Distribution

The `distribute point` pragma indicates a preference for performing loop distribution. The syntax for this pragma is shown below:

#### Syntax

```
#pragma distribute point
```

Loop distribution may cause large loops be distributed into smaller ones. This strategy might enable more loops to get software-pipelined.

- If the pragma is placed inside a loop, the distribution is performed after the directive and any loop-carried dependency is ignored.
- If the pragma is placed before a loop, the compiler will determine where to distribute and data dependency is observed. Multiple distribute pragma are supported if they are placed inside the loop.
- When the pragmas are placed inside the loop, they cannot be put inside an IF statement.

#### Example: Using distribute point

```
void dist1(int a[], int b[], int c[], int d[])
{
    #pragma distribute point
    // Compiler will automatically decide where to
```

```
// distribute. Data dependency is observed.
for (int i=1; i<1000; i++) {
    b[i] = a[i] + 1;
    c[i] = a[i] + b[i];
    d[i] = c[i] + 1;
}

void dist2(int a[], int b[], int c[], int d[])
{
    for (int i=1; i<1000; i++) {
        b[i] = a[i] + 1;
        #pragma distribute point
// Distribution will start here,
// ignoring all loop-carried dependency.
        c[i] = a[i] + b[i];
        d[i] = c[i] + 1;
    }
}
```

## Loop Unrolling Support

The `unroll[n]` pragma tells the compiler how many times to unroll a counted loop.

The general syntax for this pragma is shown below:

### Syntax

```
#pragma unroll
#pragma unroll(n)
#pragma nounroll
```

where  $n$  is an integer constant from 0 through 255.

The `unroll` pragma must precede the `FOR` statement for each `FOR` loop it affects. If  $n$  is specified, the optimizer unrolls the loop  $n$  times. If  $n$  is omitted or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop.

This pragma is supported only when option `-O3` (Linux\*) or `/O3` (Windows\*) is used. The `unroll` pragma overrides any setting of loop unrolling from the command line.

Currently, the pragma can be applied only for the innermost loop nest. If applied to the outer loop nests, it is ignored. The compiler generates correct code by comparing  $n$  and the loop count.

### Example

```
void unroll(int a[], int b[], int c[], int d[])
{
    #pragma unroll(4)
    for (int i = 1; i < 100; i++) {
        b[i] = a[i] + 1;
        d[i] = c[i] + 1;
    }
}
```

## Vectorization Support

The `vector` directives control the vectorization of the subsequent loop in the program, but the compiler does not apply them to nested loops. Each nested loop needs its own directive preceding it. You must place the vector directive before the loop control statement.

### vector always Directive

The `vector always` directive instructs the compiler to override any efficiency heuristic during the decision to vectorize or not, and will vectorize non-unit strides or very unaligned memory accesses.

#### Example: vector always Directive

```
#pragma vector always
for(i=0; i<=N; i++)
{
    a[32*i]=b[99*i];
}
```

### ivdep Directive

The `ivdep` directive instructs the compiler to ignore assumed vector dependences. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This directive overrides that decision. Use `ivdep` only when you know that the assumed loop dependences are safe to ignore. The loop in the example that follows will not vectorize with the `ivdep`, since the value of `k` is not known (vectorization would be illegal if  $k < 0$  ).

#### Example: ivdep Directive

```
#pragma ivdep
for(i=0; i<m; i++)
{
    a[i]=a[i+k]*c;
}
```

### vector aligned Directive

#### Syntax

```
#pragma vector{aligned | unaligned}
```

The vector loop pragma means the loop should be vectorized, if it is legal to do so, ignoring normal heuristic decisions about profitability. When the aligned (or unaligned) qualifier is used with this pragma, the loop should be vectorized using aligned (or unaligned) operations. Specify one and only one of aligned or unaligned.

#### Caution

If you specify aligned as an argument, you must be sure that the loop will be vectorizable using this instruction. Otherwise, the compiler will generate incorrect code.

The loop in the example that follows uses the aligned qualifier to request that the loop be vectorized with aligned instructions, as the arrays are declared in such a way that the compiler could not normally prove this would be safe to do so.

### Example

```
void foo (float *a)
{
    #pragma vector aligned
    for (i = 0; i < m; i++)
    {
        a[i] = a[i] * c;
    }
}
```

The compiler has at its disposal several alignment strategies in case the alignment of data structures is not known at compile-time. A simple example is shown (but several other strategies are supported as well). If, in the loop shown, the alignment of a is unknown, the compiler will generate a prelude loop that iterates until the array reference that occurs the most hits an aligned address. This makes the alignment properties of a known, and the vector loop is optimized accordingly.

### Example: Alignment Strategies

```
float *a;

//Alignment unknown
for(i=0; i<100; i++)
{
    a[i]=a[i]+1.0f;
}

//Dynamic loop peeling
p=a & 0x0f;
if(p!=0)
{
    p=(16-p)/4;
    for(i=0; i<p; i++)
    {
        a[i]=a[i]+1.0f;
    }
}

//Loop with a aligned.
//Will be vectorized accordingly.
for(i=p; i<100; i++)
{
    a[i]=a[i]+1.0f;
}
```



## novector Directive

The novector directive specifies that the loop should never be vectorized, even if it is legal to do so. In this example, suppose you know the trip count (ub - lb) is too low to make vectorization worthwhile. You can use novector to tell the compiler not to vectorize, even if the loop is considered vectorizable.

### Example: novector Directive

```
void foo(int lb, int ub)
{
    #pragma novector
    for(j=lb; j<ub; j++)
    {
        a[j]=a[j]+b[j];
    }
}
```

## vector nontemporal Directive (Windows\*)

### Syntax

```
#pragma vector nontemporal
```

#pragma vector nontemporal results in streaming stores on Pentium® 4 based systems. An example loop (float type) together with the generated assembly are shown in the example that follows. For large N, significant performance improvements result on a Pentium 4 systems over a non-streaming implementation.

### Example

```
#pragma vector nontemporal
for (i = 0; i < N; i++)
    a[i] = 1;
.B1.2:
movntps XMMWORD PTR _a[eax], xmm0
movntps XMMWORD PTR _a[eax+16], xmm0
add eax, 32
cmp eax, 4096
jl .B1.2
```

### Example: Dynamic Dependence Testing Example

```
float *p, *q;
for (i = L; i <= U; i++)
{
    p[i] = q[i];
}
...
pL = p * 4*L;
pH = p + 4*U;
qL = q + 4*L;
qH = q + 4*U;
```

```

if (pH < qL || pL > qH)
{
    // loop without data dependence
    for (i = L; i <= U; i++)
    {
        p[i] = q[i];
    } else {
        for (i = L; i <= U; i++)
        {
            p[i] = q[i];
        }
    }
}

```

## Prefetching Support

Data prefetching refers to loading data from a relatively slow memory into a relatively fast cache before the data is needed by the application. Data prefetch behavior depends on the architecture:

- **Itanium® processors:** the Intel® compiler generally issues prefetch instructions when you specify `-O1`, `-O2`, and `-O3` (Linux\*) or `/O1`, `/O2`, and `/O3` (Windows\*).
- **Pentium® 4 processors:** these processors do a hardware prefetch so the compiler will not issue prefetch instructions when targeted for Pentium® 4 processors.
- **Pentium® III processors:** the Intel® compiler issues prefetches when you specify `-xK` (Linux) or `/QxK` (Windows).

Issuing prefetches improves performance in most cases; there are cases where issuing prefetch instructions might slow application performance. Experiment with prefetching; it might be helpful to specifically turn prefetching on or off with a compiler option while leaving all other optimizations unaffected to isolate a suspected prefetch performance issue. See Prefetching with Options for information on using compiler options for prefetching data.

There are two primary methods of issuing prefetch instructions. One is by using compiler directives and the other is by using compiler intrinsics.

## Pragmas

### prefetch and noprefetch

The `prefetch` and `noprefetch` directives are supported by Itanium® processors only. These directives assert that the data prefetches be generated or not generated for some memory references. This affects the heuristics used in the compiler. The general syntax for these pragmas is shown below:

Syntax
<code>#pragma noprefetch</code>
<code>#pragma prefetch</code>
<code>#pragma prefetch a,b</code>

If loop includes expression  $A(j)$ , placing `prefetch A` in front of the loop, instructs the compiler to insert prefetches for  $A(j + d)$  within the loop.  $d$  is the number of iterations ahead to prefetch the data and is determined by the compiler. This directive is supported only when option `-O3` (Linux\*) or `/O3` (Windows\*) is on. These directives are also supported when you specify options `-O1` and `-O2` (Linux) or `/O1` and `/O2` (Windows). Remember that `-O2` or `/O2` is the default optimization level.

### Example

```
#pragma noprefetch b
#pragma prefetch a
for(i=0; i<m; i++)
{
    a[i]=b[i]+1;
}
```

The following example, which is for Itanium®-based systems only, demonstrates how to use the `prefetch`, `noprefetch`, and `memref_control` pragmas together:

### Example

```
#define SIZE 10000
int prefetch(int *a, int *b)
{
    int i, sum = 0;
    #pragma memref_control a:l2
    #pragma noprefetch a
    #pragma prefetch b
    for (i = 0; i<SIZE; i++)
        sum += a[i] * b[i];
    return sum;
}
#include <stdio.h>
int main()
{
    int i, arr1[SIZE], arr2[SIZE];
    for (i = 0; i<SIZE; i++) {
        arr1[i] = i;
        arr2[i] = i;
    }
    printf("Demonstrating the use of prefetch, noprefetch,\n"
           "and memref_control pragma together.\n");
    prefetch(arr1, arr2);
    return 0;
}
```

## memref\_control

The `memref_control` pragma is supported on by Itanium® processors only. This pragma provides a method for controlling load latency and temporal locality at the variable level. The `memref_control` pragma allows you to specify locality and latency at the array level. For example, using this pragma allows you to control the following:

- The location (cache level) to store data for future access.
- The most appropriate latency value to be used for a load, or the latency that has to be overlapped if a `prefetch` is issued for this reference.

The syntax for this pragma is shown below:

Syntax
<code>#pragma memref_control [name1[:&lt;locality&gt;[:&lt;latency&gt;]], [name2...]</code>

The following table lists the supported arguments.

Argument	Description
<i>name1, name2</i>	Specifies the name of array or pointer. You must specify at least one name; however, you can specify names with associated locality and latency values.
<i>locality</i>	<p>An optional integer value that indicates the desired cache level to store data for future access. This will determine the load/store hint (or <code>prefetch</code> hint) to be used for this reference. The value can be one of the following:</p> <ul style="list-style-type: none"> <li>• <code>l1 = 0</code></li> <li>• <code>l2 = 1</code></li> <li>• <code>l3 = 2</code></li> <li>• <code>mem = 3</code></li> </ul> <p>To use this argument, you must also specify name.</p>
<i>latency</i>	<p>An optional integer value that indicates the load (or the latency that has to be overlapped if a prefetch is issued for this address). The value can be one of the following:</p> <ul style="list-style-type: none"> <li>• <code>l1_latency = 0</code></li> <li>• <code>l2_latency = 1</code></li> <li>• <code>l3_latency = 2</code></li> <li>• <code>mem_latency = 3</code></li> </ul> <p>To use this argument, you must also specify name and locality.</p>

When you specify source-level and the data locality information at a high level for a particular data access, the compiler decides how best to use this information. If the compiler can prefetch profitably for the reference, then it issues a `prefetch` with a distance that covers the specified latency specified and then schedules the corresponding load with a smaller latency. It also uses the hints on the prefetch and load appropriately to keep the data in the specified cache level.

If the compiler cannot compute the address in advance, or decides that the overheads for prefetching are too high, it uses the specified latency to separate the load and its use (in a pipelined loop or a Global Code Scheduler loop). The hint on the load/store will correspond to the cache level passed with the locality argument.

You can use this with the `prefetch` and `noprefetch` to further tune the hints and prefetch strategies. When using the `memref_control` with `noprefetch`, keep the following guidelines in mind:

- Specifying `noprefetch` along with the `memref_control` causes the compiler to not issue prefetches; instead the latency values specified in the `memref_control` is used to schedule the load.
- There is no ordering requirements for using the two pragmas together. Specify the two pragmas in either order as long as both are specified consecutively just before the loop where it is to be applied. Issuing a `prefetch` with one hint and loading it later using a different hint can provide greater control over the hints used for specific architectures.
- `memref_control` is handled differently from the `prefetch` or `noprefetch`. Even if the load cannot be prefetched, the reference can still be loaded using a non-default load latency passed to the `latency` argument.

This following example illustrates the case where the address is not known in advance, so prefetching is not possible. In this case, the compiler will schedule the loads of the `tab` array with an L3 load latency of 15 cycles (inside a software pipelined loop or GCS loop).

### Example: gather

```
#pragma memref control tab : l2 : l3 latency
for (i=0; i<n; i++)
{
    x = <generate 64 random bits inline>;
    dum += tab[x&mask]; x>>=6;
    dum += tab[x&mask]; x>>=6;
    dum += tab[x&mask]; x>>=6;
}
```

The following example illustrates one way of using `memref_control`, `prefetch`, and `noprefetch` together.

### Example: sparse matrix

```
if( size <= 1000 ) {
#pragma noprefetch cp, vp
#pragma memref control x:l2:l3 latency

#pragma noprefetch yp, bp, rp
#pragma noprefetch xp
    for (iii=0; iii<rag1m0; iii++) {
        if( ip < rag2 ) {
            sum -= vp[ip]*x[cp[ip]];
            ip++;
        } else {
            xp[i] = sum*yp[i];
            i++;
            sum = bp[i];
            rag2 = rp[i+1];
        }
    }
    xp[i] = sum*yp[i];
} else {

#pragma prefetch cp, vp
#pragma memref_control x:l2:mem_latency

#pragma prefetch yp, bp, rp
#pragma noprefetch xp
    for (iii=0; iii<rag1m0; iii++) {
```

```

    if( ip < rag2 ) {
        sum -= vp[ip]*x[cp[ip]];
        ip++;
    } else {
        xp[i] = sum*yp[i];
        i++;
        sum = bp[i];
        rag2 = rp[i+1];
    }
    xp[i] = sum*yp[i];
}

```

## Intrinsics

Before inserting compiler intrinsics, experiment with all other supported compiler options and pragmas. Compiler intrinsics are less portable and less flexible than either a compiler option or compiler pragmas.

Pragmas enable compiler optimizations while intrinsics perform optimizations. As a result, programs with pragmas are more portable, because the compiler can adapt to different processors, while the programs with intrinsics may have to be rewritten/ported for different processors. This is because intrinsics are closer to assembly programming.

Some prefetching intrinsics are:

Intrinsic	Description
<code>__lfetch</code>	Generate the <code>lfetch.lfhint</code> instruction.
<code>__lfetch_fault</code>	Generate the <code>lfetch.fault.lfhint</code> instruction.
<code>__lfetch_excl</code>	Generate the <code>lfetch.excl.lfhint</code> instruction.
<code>__lfetch_fault_excl</code>	Generate the <code>lfetch.fault.excl.lfhint</code> instruction.
<code>__mm_prefetch</code>	Loads one cache line of data from address <code>a</code> to a location closer to the processor.

See Operating System Related Intrinsics and Cacheability Support Using Streaming SIMD Extensions in the Compiler Reference for more information about these intrinsics.

The following example demonstrates how to generate an `lfetch.nt2` instruction using prefetch intrinsics:

Example
<pre> for (i=i0; i!=i1; i+=is) {     float sum = b[i];     int ip = srow[i];     int c = col[ip];     for(; ip&lt;srow[i+1]; c=col[++ip])         lfetch(2, &amp;value[ip+40]);         // mm prefetch(&amp;value[ip+40], 2);     sum -= value[ip] * x[c];     y[i] = sum; } </pre>

For SSE-enabled processors you could also use the following SSE intrinsics:

- `_mm_prefetch`
- `_mm_stream_pi`
- `_mm_stream_ps`
- `_mm_sfence`

See Intel® Itanium® Architecture Software Developer's Manual, Volume 3: Instruction Set Reference, Revision 2.1. Part I: Intel® Itanium® Instruction Set Descriptions.

## Optimization Restriction Support

### optimize pragma

You can turn off all optimizations for specific functions by using `#pragma optimize`. Valid second arguments for `#pragma optimize` are as shown below:

- `#pragma optimize("", off)` - disables optimization
- `#pragma optimize("", on)` - enables optimization

The compiler ignores first argument values.

Specifying `#pragma optimize("", off)` disables optimization until either the compiler finds a matching `#pragma optimize("", on)` statement or until the compiler reaches the end of the source file. For example, in example 1 (below), optimization is disabled for function `alpha()` but not for function `omega()`.

#### Example 1: Disabling optimization for a single function

```
#pragma optimize("", off)
alpha() {
...
}
#pragma optimize("", on)
omega() {
...
}
```

In example 2, optimizations are disabled for both the `alpha()` and `omega()` functions.

#### Example 2: Disabling optimization for all functions

```
#pragma optimize("", off)
alpha() {
...
}
omega() {
...
}
```

## optimization\_level pragma

You can control optimization for a specific function using `#pragma optimization_level`. This pragma affects optimization for the specified function only. You can use this pragma to restrict optimization for a specific function while optimizing the remaining application using a different, higher optimization level. For example, if you specify `-O3` (Linux\*) or `/O3` (Windows\*) for the application and specify `#pragma optimization_level 1`, the marked function will be optimized at the `-O1` (Linux) or `/O1` (Windows) level, while the remaining application will be optimized at the higher level.

The pragma uses the following syntax:

Syntax
<code>#pragma optimization_level n</code>

where *n* is an integer value. The valid values and descriptions are shown below.

Value	Description
0	Same as <code>-O0</code> (Linux) or <code>/Od</code> (Windows)
1	Same as <code>-O1</code> (Linux) or <code>/O1</code> (Windows)
2	Same as <code>-O2</code> (Linux) or <code>/O2</code> (Windows)
3	Same as <code>-O3</code> (Linux) or <code>/O3</code> (Windows)

For more information on the optimizations levels, see Optimization Options Summary.

Place the pragma immediately prior to the function, as shown below.

Example
<pre>#pragma optimization_level 1 gamma() {     ... }</pre>

In general, the pragma will optimize the function at the level specified for *n*; however, certain compiler optimizations, like IPO, are not enabled or disabled during translation unit compilation. For example, if you enable IPO and a specific optimization level, IPO is enabled even for the function targeted by the `optimization_level` pragma, although it might not be fully implemented regardless of the optimization level specified at the command line. The reverse is also true.



## Optimizing Applications Glossary

Term	Definition
<b>A</b>	
alignment constraint	The proper boundary of the stack where data must be stored.
alternate loop transformation	An optimization in which the compiler generates a copy of a loop and executes the new loop depending on the boundary size.
<b>B</b>	
branch probability database	The database generated by the branch count profiler. The database contains the number of times each branch is executed.
<b>C</b>	
cache hit	The situation when the information the processor wants is in the cache.
call site	A call site consists of the instructions immediately preceding a call instruction and the call instruction itself.
common subexpression elimination	An optimization in which the compiler detects and combines redundant computations.
conditionals	Any operation that takes place depending on whether or not a certain condition is true.
constant argument propagation	An optimization in which the compiler replaces the formal arguments of a routine with actual constant values. The compiler then propagates constant variables used as actual arguments.
constant branches	Conditionals that always take the same branch.
constant folding	An optimization in which the compiler, instead of storing the numbers and operators for computation when the program executes, evaluates the constant expression and uses the result.
copy propagation	An optimization in which the compiler eliminates unnecessary assignments by using the value assigned to a variable instead of using the variable itself.
<b>D</b>	
dataflow	The movement of data through a system, from entry to destination.
dead-code elimination	An optimization in which the compiler eliminates any code that generates unused values or any code that will never be executed in the program.
denormal values	Computed floating-point values that have absolute values smaller than the smallest normalized floating-point number.
dynamic linking	The process in which a shared object is mapped into the virtual address space of your program at run time.
<b>E</b>	
empty declaration	A semicolon and nothing before it.

<b>F</b>	
frame pointer	A pointer that holds a base address for the current stack and is used to access the stack frame.
<b>G</b>	
Gradual underflow	<p>Gradual underflow occurs when computed floating-point values have absolute values smaller than the smallest normalized floating-point number. Such floating-point values are called denormal values.</p> <p>Gradual underflow can degrade the performance of an application.</p>
<b>H</b>	
HLO	High-Level Optimization
Hyper-Threading Technology	<p>Hyper-Threading Technology enables the operation of multiple logical processors to share execution resources in each physical processor package. It increases system throughput when executing multithreaded applications or when multitasked workloads are running concurrently.</p> <p>Hyper-Threading Technology enables you to use simultaneous multithreading on the IA-32 systems. This technology makes a single physical processor appear as two logical processors. Each logical processor can execute a software thread, allowing a maximum of two software threads to execute simultaneously on one physical processor. The two software threads execute simultaneously by the execution engine.</p>
<b>I</b>	
IPO	Interprocedural Optimization. An optimization that applies to the entire program except for library routines.
in-line function expansion	An optimization in which the compiler replaces each function call with the function body expanded in place.
induction variable simplification	An optimization in which the compiler reduces the complexity of an array index calculation by using only additions.
instruction scheduling	An optimization in which the compiler reorders the generated machine instructions so that more than one can execute in parallel.
instruction sequencing	An optimization in which the compiler eliminates less efficient instructions and replaces them with instruction sequences that take advantage of a particular processor's features.
<b>L</b>	
load balancing	The equal division of work among threads. If a load is balanced, it ensures processors are busy most, if not all, of the time. If a load is not balanced, some threads may finish significantly before others, leaving processor resources idle and wasting performance opportunities.
loop blocking	An optimization in which the compiler reorders the execution sequence of instructions so that the compiler can execute iterations from outer loops before completing all the iterations of the inner loop.
loop unrolling	An optimization in which the compiler duplicates the executed

	statements inside a loop to reduce the number of loop iterations.
loop-invariant code movement	An optimization in which the compiler detects multiple instances of a computation that does not change within a loop.
<b>P</b>	
padding	The addition of bytes or words at the end of each data type in order to meet size and alignment constraints.
PGO	Profile-guided Optimization
PMU	Performance Monitor Unit
preloading	An optimization in which the compiler loads the vectors, one cache at a time, so that during the loop computation the number of external bus turnarounds is reduced.
privatization of scalars	Privatization of scalars is an operation of re-assigning the storage of scalars from the static or parent stack area to the local stack of a thread to enable parallelization. This operation requires a WRITE permission and is usually performed to remove a data dependency between concurrently executing threads.
profiling	(PGO) A process in which detailed information is produced about the program's execution.
<b>R</b>	
register variable detection	An optimization in which the compiler detects the variables that never need to be stored in memory and places them in register variables.
<b>S</b>	
side effects	Results of the optimization process that might increase the code size and/or processing time.
static linking	The process in which a copy of the object file that contains a function used in your program is incorporated in your executable file at link time.
strength reduction	An optimization in which the compiler reduces the complexity of an array index calculation by using only additions.
strip mining	An optimization in which the compiler creates an additional level of nesting to enable inner loop computations on vectors that can be held in the cache. This optimization reduces the size of inner loops so that the amount of data required for the inner loop can fit the cache size.
SWP	Software Pipelining
<b>T</b>	
token pasting	The process in which the compiler treats two tokens separated by a comment as one (for example, <code>a/**/b</code> become <code>ab</code> ).
transformation	A rearrangement of code. In contrast, an optimization is a rearrangement of code where improved run-time performance is guaranteed.
<b>U</b>	
unreachable code	Instructions that are never executed by the compiler.
unused code	Instructions that produce results that are not used in the program.
<b>V</b>	
variable	An optimization in which the compiler renames instances of a variable

renaming	that refer to distinct entities.
----------	----------------------------------

# Index

.

.dpi file..... 78, 94, 101  
.dyn file..... 78, 94, 101  
.hpi file..... 103  
.spi file..... 71, 78, 94  
.tb5 file..... 103

/

/Qprof-gen compiler option  
    using with SSP ..... 108  
  
/Qprof-gen-sampling compiler option  
    using with profrun ..... 103  
    using with SSP ..... 108  
  
/Qprof-genx compiler option  
    code-coverage tool..... 78  
    test-priorization tool ..... 94  
  
/Qprof-use compiler option  
    code-coverage tool..... 78  
    profmerge utility..... 101  
    using with profrun ..... 103  
    using with SSP ..... 108  
  
/Qssp compiler option  
    using with SSP ..... 108

—

\_\_assume\_aligned ..... 193  
\_\_declspec ..... 193

## A

accuracy  
    controlling..... 124  
  
advanced PGO options ..... 65, 70  
  
aliases ..... 191  
  
aligning data ..... 24  
  
alignment  
    example ..... 191  
    strategy ..... 191  
  
alignment..... 24, 46  
  
alignment..... 191  
  
allocating temporary arrays ..... 156  
  
analyzing  
    effects of multifile IPO ..... 50  
    programming ..... 2  
  
analyzing applications  
    Intel® Debugger ..... 3  
    Intel® Threading Tools..... 3  
    VTune™ Performance Analyzer ..... 3  
  
analyzing applications ..... 3  
  
analyzing applications ..... 6

analyzing applications .....	9	enabling .....	178
analyzing hotspots .....	4	overview .....	171
application		programming with .....	172
basic block.....	78	threshold .....	179
code coverage .....	78	auto-parallelization .....	143
OpenMP* .....	143	auto-parallelization .....	171
pipelining .....	198	auto-parallelized loops.....	179
tests .....	78	auto-parallelizer	
visual presentation.....	78	controls .....	143, 179
application characteristics .....	9	enabling .....	143
application performance.....	9	auto-parallelizer .....	143
application tests .....	94	auto-parallelizer .....	171
argument aliasing.....	191	auto-vectorization .....	143
arithmetic precision		auto-vectorizer .....	181
improving .....	126	<b>B</b>	
restricting .....	126	basic PGO options.....	68
arrays		browsing frames using the coverage tool	
alignment in vectorization .....	191	.....	78
loop blocking.....	188	<b>C</b>	
operations in a loop body .....	184	clauses	
automatic		summary table of .....	158
optimization for IA-32 systems .....	37	cleanup of loops .....	188
auto-parallelization		code-coverage tool	
diagnostic .....	179	coloring scheme for.....	78
		dynamic counters in .....	78

- export data..... 78
- options ..... 78
- options in ..... 78
- syntax of ..... 78
- visual presentation of..... 78
- code-coverage tool..... 78
- compilation
  - phase..... 46
- compilation ..... 55
- compilation units ..... 59
- compiler
  - intermediate language files produced by ..... 55
- compiler reports
  - High-Level Optimization (HLO)..... 133
  - Interprocedural Optimizations (IPO) ..... 134
  - report generation ..... 129
  - software pipelining..... 135
  - vectorization ..... 138
- compiler reports ..... 129
- computing denormals..... 126
- controlling
  - auto-parallelizer diagnostics. 143, 179
  - inline expansion..... 58
  - rounding ..... 121
- correct usage of countable loop
  - countable loop..... 187
- correct usage of countable loop ..... 186
- correct usage of countable loop ..... 187
- COS
  - SIMD version of ..... 183
- countable loop
  - correct usage of ..... 186, 187
- counters for dynamic profile ..... 76
- CPU dispatch..... 40
- CPU time ..... 25
- CPUID values ..... 40
- creating
  - DPI list..... 94
  - multifile IPO executable ..... 48, 51
- criteria
  - for inline function expansion ..... 55
- D**
- data alignment ..... 24
- data format
  - alignment ..... 191
  - dependence ..... 113, 114, 179, 198
  - partitioning ..... 172

prefetching.....	112, 199	inlining.....	58
sharing.....	143	optimizations with pragmas.....	209
structure.....	191	DISTRIBUTE POINT	
type.....	143, 181	using .....	199
data prefetches .....	204	division-to-multiplication optimization	121
dataflow analysis.....	143, 171	DO constructs.....	186
denormals .....	126	double-precision	
denormals-are-zero.....	38	numbers .....	162
dependence of data .....	185	dual-core.....	143
determining parallelization .....	143	Dual-Core Intel® Itanium® 2 Processor 9000 .....	33
diagnostic reports.....	156, 179	dumping profile information .....	75, 76
diagnostics		dyn files .....	65, 68, 70, 74, 75, 76
auto-parallelizer.....	143, 179	dynamic counters .....	78
OpenMP* .....	156	dynamic information	
diagnostics .....	143	directory for files.....	70
diagnostics .....	182	dumping profile information.....	75
difference operators .....	166	files.....	65
differential coverage.....	78	resetting profile counters.....	76
directives for language support.....	193	threads .....	162
directory		dynamic information .....	62
specifying for dynamic information files.....	70	dynamic information .....	74
directory .....	70	dynamic-information files .....	68
disabling		<b>E</b>	
function splitting.....	68	effects of multifile IPO.....	50



- enabling
  - auto-parallelizer ..... 143
  - inlining ..... 58
  - parallelizer ..... 143
  - PGO options ..... 68
  - SIMD-encodings ..... 188
- enhancing optimization ..... 9
- enhancing performance ..... 9
- environment variables
  - and OpenMP\* extension routines. 164
  - for auto-parallelization ..... 178
  - OpenMP\* ..... 161
  - PROF\_DUMP\_INTERVAL ..... 76
  - routines overriding ..... 162
- errno variable ..... 61
- example of
  - auto-parallelization ..... 178
  - auto-vectorization ..... 201
  - dumping profile information ..... 75
  - loop constructs ..... 186
  - parallel program development ..... 143
  - using OpenMP\* ..... 166
  - using profile-guided optimization .... 65
  - vectorization ..... 191
- exclude code
  - code-coverage tool ..... 78
- exclude code ..... 78
- execution environment routines ..... 162
- execution flow ..... 172
- execution mode ..... 164
- exit conditions ..... 187
- F**
- files
  - .dpi ..... 68, 78, 94, 101
  - .dyn ..... 68, 70, 74, 75, 76, 78, 94, 101
  - .hpi ..... 103
  - .spi ..... 78, 94
  - .tb5 ..... 103
  - IR ..... 46
  - object files ..... 48, 51
  - OpenMP\* header ..... 162
  - real object ..... 55
  - source ..... 48, 65
- floating-point applications
  - comparisons ..... 126
- floating-point arithmetic
  - array operations ..... 184
  - on IA-32 systems ..... 121

on Itanium®-based systems .....	124	for vectorization.....	181, 182
options for.....	120	<b>H</b>	
overview .....	120	helper thread optimization .....	108
performance .....	126	heuristics	
restricting precision of.....	126	affecting data prefetches.....	199, 204
floating-point arithmetic.....	27	affecting software pipelining..	198, 199
floating-point arithmetic.....	126	for inlining functions .....	55, 57
flow dependency in loops.....	116	high performance.....	1
function expansion .....	59	high performance programming .....	2, 62
function order list.....	71	high-level optimizer.....	112, 129
function splitting		HLO	
enabling or disabling.....	68	reports.....	133
<b>G</b>		HLO .....	112
general compiler directives .....	200, 204	hotspots .....	4, 6
generating		Hyper-Threading Technology	
instrumented code .....	68	parallel loops.....	173
processor-specific code.....	37	thread pools .....	173
profile-optimized executable.....	68	using OpenMP* .....	153
profiling information .....	73	<b>I</b>	
reports .....	129	IA-32 architecture	
guidelines		applications for .....	112
for auto-parallelization .....	172	dispatch options for .....	37
for high performance programming ..	2	options for .....	36, 37
for profile-guided optimization ..	62, 70	options targeting .....	33

- processors for ..... 36, 37, 143
- report generation ..... 129
- targeting ..... 33, 36
- ILO ..... 129
- improving
  - floating-point arithmetic precision . 126
- inlining
  - compiler directed ..... 58
  - developer directed ..... 59
- inlining ..... 58
- inlining ..... 59
- inlining ..... 61
- inlining ..... 62
- instruction-level parallelism ..... 143
- instrumentation
  - compilation ..... 65
  - repeat ..... 70
- instrumentation ..... 75
- instrumented code
  - execution - run ..... 65
  - generating ..... 68
  - program ..... 62
- Intel® Core™ Duo processors ..... 33
- Intel® Core™ Solo processors ..... 33
- Intel® Debugger ..... 3
- Intel® extension routines ..... 164
- Intel® Itanium® 2 processors ..... 33
- Intel® Itanium® processors ..... 33
- Intel® Pentium® 4 processors ..... 33
- Intel® Pentium® II processors ..... 33
- Intel® Pentium® III processors ..... 33
- Intel® Pentium® Pro processors ..... 33
- Intel® Pentium® processors ..... 33
- Intel® Threading Tools ..... 3
- Intel® Xeon® processors ..... 33
- intermediate language files (IL)
  - implementing with version number.. 55
- intermediate language scalar optimizer
  - ..... 129
- interprocedural optimizations
  - code layout ..... 71
- interprocedural optimizations. 27, 45, 58, 62
- interprocedural optimizations ..... 129
- interval profile dumping
  - initiating ..... 76
- intrinsic ..... 204
- introduction to Optimizing Applications. 1
- IPO

code layout .....	54	OpenMP* run-time routines ..	162, 164
generating multiple IPO object files	49	libraries .....	25, 46
issues .....	47	library functions .....	61
overview .....	46	library routines	
performance .....	47	inline expansion of library functions	61
reports .....	134	Intel extension .....	164
Itanium®-based applications		OpenMP* run-time routines .....	162
auto-vectorization in .....	143	lock routines .....	162
floating point options.....	120, 124	LOOP COUNT	
HLO .....	112	and loop distribution .....	199
options targeting .....	33	loop interchange .....	15
pipelining for .....	198	loop unrolling	
report generation .....	129	limitations of .....	115
targeting.....	33	support for.....	200
IVDEP		using the HLO optimizer .....	112, 129
effect in loop transformations .....	113	loop unrolling .....	182
effect of compiler option on .....	114	loop unrolling .....	198
effect when tuning applications ....	112	loops	
<b>L</b>		anti dependency.....	116
language support		arrays within .....	184
__declspec .....	193	blocking.....	188, 211
language support .....	193	body .....	184
libraries		constructs.....	186
inline expansion of library functions	61	count .....	199, 200

- data dependency ..... 185
- dependencies ..... 172, 201
- distribution ..... 112, 113, 199
- exit conditions ..... 187
- flow dependency ..... 116
- independence ..... 116
- interchange ..... 15, 18, 112, 113, 193
- manual transformation ..... 18
- output dependency ..... 116
- parallelization ..... 143, 172, 183
- reductions ..... 116
- sectioning ..... 188
- transformations. 15, 27, 112, 113, 188
- types vectorized ..... 183
- unrolling ..... 115, 200
- vectorization ..... 183
- vectorized ..... 183
- loops ..... 184
- loops ..... 186
- loops ..... 199
- M**
- manual dispatch ..... 40
- manual transformations ..... 18
- matrix multiplication
  - example of ..... 193
- memory
  - allocation ..... 164
  - dependency ..... 113, 114
- memory aliasing ..... 15
- misaligned data ..... 191
- mixing vectorizable types in a loop... 182
- MMX(TM) ..... 181
- multifile IPO
  - analyzing the effects of ..... 50
  - creating and using an executable for ..... 48, 51
  - optimization ..... 45
  - overview ..... 49
- multithreaded programs ..... 143, 171
- multithreading ..... 159, 172
- N**
- non-unit memory access ..... 15
- NOPREFETCH
  - using ..... 204
- NOSWP
  - using ..... 198
- O**
- obj files ..... 48
- OMP directives ..... 143, 166

OpenMP*	OS-related..... 9
advanced issues..... 167	reports..... 129, 197, 198
debugging..... 167	strategies ..... 9
directives ..... 153, 170	system-related ..... 9
environment variables ..... 161	targeting processors ..... 33
Hyper-Threading Technology ..... 153	tuning tools..... 4
parallel processing thread model.. 150	optimization support ..... 1
performance ..... 167	optimization_level pragma ..... 209
pragmas..... 153	optimizations
run-time library routines..... 162	compilation process ..... 27
support libraries ..... 159	floating-point ..... 120
OpenMP* Fortran directives	for specific processors ..... 33
clauses for ..... 158	helper thread ..... 108
examples of ..... 166	high-level language ..... 112
Intel extensions for ..... 164	interprocedural ..... 45
syntax of ..... 156	IPO ..... 46
OpenMP* Fortran directives..... 158	multiple IPO ..... 45
optimization	options for IA-32..... 33
analyzing applications ..... 9	options for Itanium® architecture .... 33
application-specific ..... 9	overview of ..... 27, 62
hardware-related ..... 9	parallelization ..... 143
library-related..... 9	PGO methodology ..... 63
methodology ..... 6	profile-guided ..... 62
options ..... 27	SSP ..... 108

- support features for ..... 197
- optimizations ..... 1
- optimizations ..... 27
- optimize pragma..... 209
- optimizer report generation ..... 129
- optimizing
  - applications..... 9
  - helping the compiler ..... 15
  - overview ..... 1
  - technical applications ..... 9
- optimizing ..... 1
- optimizing performance..... 2
- OptReport support..... 129
- ORDERED
  - overview of OpenMP\* directives and clauses..... 158
- overflow ..... 120
- overriding
  - call to a runtime library routine ..... 162
  - loop unrolling ..... 200
  - software pipelining..... 198
  - vectorization ..... 201
- overview
  - of optimizing different application types ..... 27
  - of optimizing for specific processors33
  - of parallelism ..... 143
  - of programming for high performance ..... 2
- overview ..... 1
- P**
- packed structures ..... 24
- PARALLEL DO
  - summary of OpenMP\* directives and clauses ..... 158
- parallel invocations with makefile . 51, 68
- PARALLEL OpenMP\* directive ..... 158
- parallel processing
  - thread model ..... 150
- parallel programming ..... 1, 143
- parallel regions ..... 158
- PARALLEL SECTIONS
  - summary of OpenMP\* directives... 158
- parallelism ..... 143, 148, 162, 171
- parallelization
  - diagnostic..... 179
- parallelization..... 143, 148, 171, 172
- parallelization..... 183
- passing
  - options to other tools..... 57

passing.....	55	pipelining .....	143
performance analyzer .....	25, 148	pragmas	
performance issues with IPO .....	47	ivdep .....	193
PGO .....	62, 63	novector .....	193
PGO API		optimization_level .....	209
_PGOPTI_Prof_Dump_And_Reset	76	optimize.....	209
_PGOPTI_Prof_Reset.....	76	restrict .....	193
_PGOPTI_Set_Interval_Prof_Dump	76	using with vectorization .....	193
enable.....	73	vector .....	193
PGO tools		vector always .....	193
code-coverage tool.....	78	vector nontemporal .....	193
helper threads.....	108	pragmas for language support.....	193
profmerge .....	101	PREFETCH	
proforder .....	101	options used for .....	119
profrun .....	103	using .....	204
software-based precomputation ...	108	prefetches of data	
test-prioritization tool .....	94	optimizations for .....	119
PGO tools.....	78	prefetches of data .....	204
pgopti.dpi file .....	68, 74	preloading.....	211
pgopti.spi file .....	63, 78, 94	prioritizing application tests .....	94
pgouser.h header file .....	73	PRIVATE	
pipelining		summary of data scope attribute	
affect of LOOP COUNT on .....	199	clauses .....	158
for Itanium®-based applications ...	198	processor	
		manual dispatch .....	40



- optimizing for specific ..... 33, 36
  - run-time checks for IA-32 systems . 38
  - targeting..... 33
- processor-based optimizations ..... 33
- processors..... 33
- processor-specific runtime checks ..... 38
- PROF\_DIR environment variable..... 74
- PROF\_DUMP\_INTERVAL environment variable ..... 74
- PROF\_NO\_CLOBBER environment variable ..... 74
- prof-gen compiler option
  - using with SSP ..... 108
- prof-gen-sampling compiler option
  - using with profrun ..... 103
  - using with SSP ..... 108
- prof-genx compiler option
  - code-coverage tool ..... 78
  - test-priorization tool ..... 94
- profile data
  - dumping ..... 75, 76
  - resetting dynamic counters for ..... 76
- profile-guided optimization
  - API support..... 73
  - dumping profile information ..... 76
  - environment variables ..... 74
  - example of ..... 65
  - interval profile dumping ..... 76
  - methodology ..... 63
  - options ..... 68, 70
  - overview ..... 62
  - resetting dynamic profile counters .. 76
  - resetting profile information..... 76
  - support ..... 73
- profile-guided optimization..... 62
- profile-guided optimization..... 63
- profile-optimized code ..... 68, 73
- profiling
  - generating information ..... 73
  - specifying a summary ..... 70
- profmerge
  - code-coverage tool ..... 78
- profmerge ..... 101
- profrun
  - .hpi file..... 103
  - .tb5 file ..... 103
  - requirements ..... 103
  - SSP ..... 108
- profrun ..... 103

-prof-use compiler option		routines	
code-coverage tool .....	78	Intel extension .....	164
profmerge utility .....	101	OpenMP* run-time .....	162
using with profrun .....	103	timing .....	162
using with SSP .....	108	run-time checks	
program loops .....	171	processor-specific .....	38
programs		<b>S</b>	
high performance.....	2	sample of timing .....	25
interprocedural optimization of .....	45	scalar clean-up iterations.....	191
pseudo code		scalar replacement .....	114
parallel processing model.....	150	shared scalars .....	166
<b>R</b>		significand.....	121
real object files		SMP systems.....	171
compiling with .....	55	software pipelining	
REDUCTION		affect of LOOP COUNT on .....	199
summary of data scope attribute		for Itanium®-based applications ...	198
clauses.....	158	optimization.....	198
reductions in loops .....	116	reports.....	135
report generation.....	129	software pipelining.....	135
report software pipelining (SWP) .....	135	software pipelining.....	143
resetting		specialized code .....	37, 143
dynamic profile counters.....	76	specifiers	
profile information .....	76	/Qoption compiler option .....	57
restrict .....	193	-Qoption compiler option .....	57
restricting optimization .....	209		

- specifiers ..... 57
- specifying symbol visibility ..... 42
- SSE ..... 181
- SSE2 ..... 181
- SSP
  - profun ..... 108
  - using with /Qprof-gen ..... 108
  - using with /Qprof-use..... 108
  - using with /Qssp ..... 108
  - using with -prof-gen ..... 108
  - using with -prof-use ..... 108
  - using with -ssp..... 108
- SSP ..... 108
- ssp compiler option
  - using with SSP ..... 108
- statements
  - in the loop body ..... 184
- strategies for optimization ..... 9
- Streaming SIMD Extensions .... 182, 188
- stripmining..... 188
- subroutines in the OpenMP\* run-time library..... 162
- support
  - for loop unrolling ..... 200
  - for OpenMP\* ..... 159
  - for optimization..... 197
  - for prefetching ..... 204
  - for vectorization..... 201
  - parallel run-time ..... 171
- SWP
  - SWP reports..... 135
  - using ..... 198
- symbol visibility
  - specifying ..... 42
- symbol visibility ..... 42
- symbol visibility on Linux\* ..... 42
- symbol visibility on Mac OS\* ..... 42
- synchronization..... 143, 171
- T**
  - targeting..... 33
  - targeting optimizations..... 33
  - targeting processors
    - Itanium®..... 33
    - Itanium® 2..... 33
  - targeting processors ..... 33
  - technical applications ..... 9
  - testing applications ..... 94
  - test-prioritization tool
    - examples..... 94

options .....	94	tuning .....	4
requirements.....	94	tools .....	78
usage .....	94	transformations	
test-prioritization tool .....	94	loop .....	113
thread pooling .....	173	reordering.....	182
threads		tuning .....	4
parallel processing model for.....	150	types of loop vectorized .....	183
thread sleep time .....	164	<b>U</b>	
thread-level parallelism.....	143	underflow .....	120
threshold control for auto-parallelization		unvectorizable copy .....	182
.....	179	usage rules .....	51
timeout .....	55	user functions .....	58, 59
timing		using	
OpenMP* routines for .....	162	advanced PGO .....	70
tool options		auto-parallelization .....	143, 171
code-coverage tool .....	78	dynamic libraries .....	162
profmerge .....	101	OpenMP* .....	166
proforder .....	101	profile-guided optimization .....	65
profrun .....	103	timing for an application .....	25
test-prioritization tool .....	94	utilities	
tool options.....	78	profmerge.....	101
tools		proforder .....	101
code-coverage tool .....	78	utilities.....	78
strategies .....	4		
test-prioritization tool .....	94		

**V**

## variables

PGO environment..... 74

renaming..... 27

vector copy..... 191

vector dependencies..... 201

## vectorization

examples ..... 191

options ..... 181

options for..... 143

overview ..... 181

patterns..... 185, 201

programming guidelines ..... 181, 182

reports ..... 138

support for ..... 201

## vectorizing

loops ..... 62, 186

vectorizing ..... 182

## VTune™ Performance Analyzer

profrun..... 103

VTune™ Performance Analyzer..... 3

**W**

worker thread..... 159

worksharing ..... 143, 158, 171

**X**

xild ..... 48, 51

xilink..... 48, 51

**Z**

zero denormal values ..... 124