

Intel® Fortran Compiler Optimizing Applications

Document Number: 307781-003US

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.....	3
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.....	5
Gather performance data	6
Analyze the data.....	7
Generate alternatives	7
Implement enhancements	7
Test the results	8
Applying Performance Enhancement Strategies	9
Understanding Run-time Performance.....	14
Helping the Compiler	14
Memory Aliasing on Itanium®-based Systems.....	14
Applying Optimization Strategies	17
Loop Interchange.....	17
Unrolling	18

Cache Blocking.....	19
Loop Distribution.....	20
Load Pair (Itanium® Compiler).....	21
Manual Loop Transformations.....	22
Understanding Data Alignment	22
Timing Your Application	23
Considerations on Timing Your Application.....	23
Methods of Timing Your Application.....	24
Sample Timing.....	24
Coding Guidelines for Intel® Architectures	26
Memory Access.....	26
Memory Layout.....	27
Optimizing for Floating-point Applications	28
Auto-vectorization (IA-32 Only)	29
Creating Multithreaded Applications.....	29
Setting Data Type and Alignment	29
Causes of Unaligned Data and Ensuring Natural Alignment.....	29
Checking for Inefficient Unaligned Data	32
Ordering Data Declarations to Avoid Unaligned Data	33
Using Arrays Efficiently	36
Accessing Arrays Efficiently	37
Passing Array Arguments Efficiently	40
Improving I/O Performance	41
Use Unformatted Files Instead of Formatted Files	42

Use of Variable Format Expressions	43
Efficient Use of Record Buffers and Disk I/O.....	44
Specify RECL	45
Use the Optimal Record Type	45
Reading from a Redirected Standard Input File	46
Improving Run-time Efficiency	46
Avoid Small Integer and Small Logical Data Items.....	46
Avoid Mixed Data Type Arithmetic Expressions	46
Use Efficient Data Types	47
Avoid Using Slow Arithmetic Operators.....	47
Avoid Using EQUIVALENCE Statements.....	48
Use Statement Functions and Internal Subprograms.....	48
Code DO Loops for Efficiency	48
Using Intrinsics for Itanium®-based Systems	48
CACHESIZE Intrinsic (Itanium®-based Compiler)	49
Compiler Optimizations Overview	50
Optimization Options Summary	50
Setting Optimization Levels	51
Restricting Optimization.....	54
Diagnostic Options	55
Optimizing for Specific Processors Overview	55
Targeting a Processor.....	56
Options for IA-32 and Intel® EM64T Processors	56
Options for Itanium® Processors	57

Processor-specific Optimization (IA-32 only)	58
Automatic Processor-specific Optimization (IA-32 Only)	59
Processor-specific Run-time Checks for IA-32 Systems.....	60
Check for Supported Processor	60
Setting FTZ and DAZ Flags.....	61
Optimizing the Compilation Process Overview	62
Efficient Compilation	62
Efficient Compilation Techniques	62
Options That Increase Run-time Performance	63
Options That Decrease Run-time Performance	64
Stacks: Automatic Allocation and Checking.....	65
Automatic Allocation of Variables	65
Checking the Floating-point Stack State	67
Checking and Setting Space	68
Aliases	68
Little-endian-to-Big-endian Conversion (IA-32).....	70
Little-to-Big Endian Conversion Environment Variable.....	70
Usage Examples	71
Alignment Options.....	73
Symbol Visibility Attribute Options (Linux* and Mac OS*).....	74
Global Symbols and Visibility Attributes	74
Symbol Preemption and Optimization	74
Specifying Symbol Visibility Explicitly	75
Interprocedural Optimizations Overview	77

IA-32 and Itanium®-based applications.....	77
IA-32 applications only	77
-auto-ilp32 (Linux*) or /Qauto-ilp32 (Windows*) for Intel® EM64T and Itanium®-based Systems.....	78
IPO Compilation Model	78
Understanding IPO-Related Performance Issues	79
Improving IPO Performance	79
Command Line for Creating an IPO Executable	80
Generating Multiple IPO Object Files	81
Capturing Intermediate Outputs of IPO	82
Creating a Multifile IPO Executable	83
Usage Rules	84
Understanding Code Layout and Multi-Object IPO	86
Implementing IL Files with Version Numbers	87
IL in Objects and Libraries: More Optimizations	87
Criteria for Inline Function Expansion	87
Selecting Routines for Inlining	88
Using Qoption Specifiers.....	89
Compiler Directed Inline Expansion of User Functions.....	90
Developer Directed Inline Expansion of User Functions.....	91
Profile-Guided Optimizations Overview	94
Instrumented Program.....	94
Added Performance with PGO	94
Understanding Profile-Guided Optimization	95
PGO Phases.....	95

PGO Usage Model	97
Example of Profile-Guided Optimization	97
Profile-guided Optimization (PGO) Phases	98
Basic PGO Options	99
Advanced PGO Options.....	102
Generating Function Order Lists	103
Function Order List Usage Guidelines (Windows*)	103
Comparison of Function Order Lists and IPO Code Layout	103
Example of Generating a Function Order List	104
PGO API Support Overview.....	105
The Profile IGS Environment Variable.....	106
PGO Environment Variables	106
Dumping Profile Information.....	107
Recommended usage	107
Dumping Profile Data	107
Resetting the Dynamic Profile Counters	108
Recommended usage	108
Dumping and Resetting Profile Information	108
Recommended usage	109
Interval Profile Dumping.....	109
Recommended usage	109
PGO Tools Overview	110
Code-Coverage Tool.....	110
Code-coverage tool Requirements.....	111

Visually Presenting Code Coverage for an Application	114
Excluding Code from Coverage Analysis	119
Exporting Coverage Data	122
Test-Prioritization Tool	125
Features and Benefits	125
Test-prioritization tool Requirements.....	126
Tool Usage Examples	130
Using Other Options	132
Profmerge and Proforder Utilities.....	132
profmerge Utility	132
proforder Utility	134
Profrun Utility.....	135
Profrun Utility Requirements and Behavior	135
Using the profrun Utility	136
Profrun Utility Options.....	137
Software-based Speculative Precomputation (IA-32)	139
SSP Behavior	140
Using SSP Optimization	140
HLO Overview.....	143
IA-32 and Itanium®-based Applications	143
IA-32 Applications.....	143
Tuning Itanium-based Applications	143
Loop Transformations	144
Scalar Replacement.....	144

Absence of Loop-carried Memory Dependency with IVDEP Directive.....	145
Loop Unrolling	145
Loop Independence	146
Flow Dependency - Read After Write	147
Anti Dependency - Write After Read	148
Output Dependency - Write After Write	148
Reductions.....	149
Prefetching with Options	149
Floating-point Arithmetic Optimizations Overview.....	151
Floating-point Options for Multiple Architectures	151
Floating-point Options for IA-32 and Intel® EM64T	154
Floating-point Options for Itanium®-based Systems.....	156
Improving or Restricting FP Arithmetic Precision.....	158
Understanding Floating-point Performance	158
Denormal Computations.....	158
Inexact Floating Point Comparisons.....	160
Compiler Reports Overview	160
Optimizer Report Generation	160
Specifying Optimizations to Generate Reports.....	162
High-Level Optimization (HLO) Report	164
Interprocedural Optimizations (IPO) Report.....	165
Software Pipelining (SWP) Report (Linux* and Windows*).....	166
Reading the Reports.....	167
Vectorization Report.....	169

Usage with Other Options	171
Changing Code Based on Report Results.....	171
Parallelism Overview.....	174
Parallel Program Development.....	175
Parallelization with OpenMP* Overview	178
Parallel Processing with OpenMP	179
Performance Analysis.....	179
Parallel Processing Thread Model	179
The Execution Flow	180
Using Orphaned Directives.....	181
OpenMP* and Hyper-Threading Technology	182
Programming with OpenMP*	185
Parallel Region	185
Worksharing Construct	185
Parallel Processing Directive Groups	186
Data Sharing.....	187
Orphaned Directives.....	188
Preparing Code for OpenMP Processing	189
Compiling with OpenMP, Directive Format, and Diagnostics.....	190
OpenMP Option.....	190
OpenMP Directive Format and Syntax	191
OpenMP Diagnostic Reports	192
OpenMP* Directives and Clauses Summary	192
OpenMP Directives.....	193

OpenMP Clauses	194
Directives and Clauses Cross-reference	195
OpenMP* Support Libraries	196
Execution modes	196
OpenMP* Environment Variables	197
Standard Environment Variables	197
Intel Extension Environment Variables	197
OpenMP* Run-time Library Routines	198
Execution Environment Routines	199
Lock Routines	199
Timing Routines	200
Intel Extension Routines/Functions	200
Stack Size	201
Memory Allocation	202
Examples of OpenMP* Usage	203
DO: A Simple Difference Operator	203
DO: Two Difference Operators	203
SECTIONS: Two Difference Operators	204
SINGLE: Updating a Shared Scalar	204
Combined Parallel and Worksharing Constructs	205
PARALLEL DO and END PARALLEL DO	205
PARALLEL SECTIONS and END PARALLEL SECTIONS	206
WORKSHARE and PARALLEL WORKSHARE	206
Parallel Region Directives	206

Clauses Used	207
Synchronization Constructs	209
ATOMIC Directive.....	209
BARRIER Directive.....	210
CRITICAL and END CRITICAL	210
FLUSH Directive.....	211
MASTER and END MASTER	212
ORDERED and END ORDERED	212
THREADPRIVATE Directive	212
Worksharing Construct Directives.....	213
DO and END DO	213
SECTIONS, SECTION and END SECTIONS	214
SINGLE and END SINGLE.....	215
Data Scope Attribute Clauses Overview	215
COPYIN Clause	216
DEFAULT Clause.....	216
PRIVATE, FIRSTPRIVATE, and LASTPRIVATE Clauses.....	217
PRIVATE	217
FIRSTPRIVATE.....	217
LASTPRIVATE	218
REDUCTION Clause.....	218
SHARED Clause.....	220
Specifying Schedule Type and Chunk Size	221
Auto-parallelization Overview.....	222

Programming with Auto-parallelization.....	223
Guidelines for Effective Auto-parallelization Usage.....	223
Auto-parallelization Data Flow	224
Programming for Multithread Platform Consistency.....	224
Auto-parallelization: Enabling, Options, Directives, and Environment Variables	228
Auto-parallelization Options.....	228
Auto-parallelization Directives	228
Auto-parallelization Environment Variables.....	229
Auto-parallelization: Threshold Control and Diagnostics	230
Threshold Control	230
Diagnostics	230
Vectorization Overview (IA-32 and Intel® EM64T)	231
Vectorizer Options.....	231
Key Programming Guidelines for Vectorization	232
Guidelines.....	232
Restrictions.....	232
Loop Parallelization and Vectorization	233
Types of Vectorized Loops.....	233
Statements in the Loop Body	234
Floating-point Array Operations.....	234
Integer Array Operations	234
Other Operations	234
Data Dependency	235
Data dependency Analysis	235

Loop Constructs	236
Loop Exit Conditions	237
Strip-mining and Cleanup.....	238
Loop Blocking	239
Vectorization Examples.....	239
Argument Aliasing: A Vector Copy	240
Data Alignment	240
Loop Interchange and Subscripts: Matrix Multiply	241
Optimization Support Features Overview	242
Compiler Directives Overview	242
Pipelining for Itanium®-based Applications.....	243
Loop Count and Loop Distribution.....	244
Loop Count.....	244
Loop Distribution.....	244
Loop Unrolling Support	245
Vectorization Support.....	246
IVDEP Directive.....	246
Overriding the Efficiency Heuristics in the Vectorizer	247
Prefetching Support	249
Directives.....	250
Intrinsics	251
Optimizing Applications Glossary.....	252
Index	257

Introduction to Optimizing Applications

This document explains how to use the Intel® Fortran 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
Use programming strategies to improve performance	Setting Data Type and Alignment Using Arrays Efficiently Improving I/O Performance Improving Run-time Efficiency Using Intrinsics for Itanium®-based Systems Coding Guidelines for Intel Architectures
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, 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 the How to Use This Document topics, which are located with the introductory topics in each book in the Intel® Fortran Compiler Documentation.

Programming for High Performance Overview

This section provides information on the following:

- **Optimizing Performance:** This section discusses optimization-related strategies and methods.
- **Programming Guidelines:** This section discusses programming guidelines that can enhance application performance and includes specific coding practices that

use the Intel® architecture features. This section also details the correlation between the programming practices and related compiler options.

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, and compiler options 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 http://www.intel.com/software/products/vtune/.</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 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 http://www.intel.com/software/products/threading/tcwin/.</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.

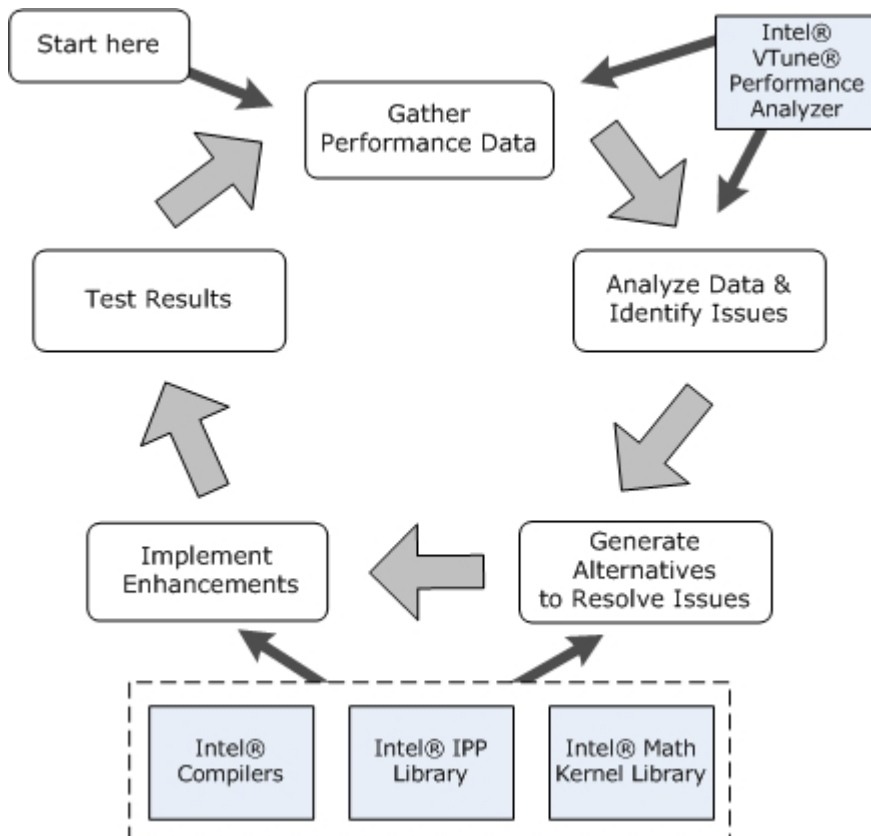
Intel compilers provide support for auto-parallelization and substantial support of OpenMP* as described in the OpenMP Fortran version 2.5 specification.

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 (_).

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.

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
Technical Applications		
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"> • If recurrences are listed in the report that you suspect do not exist, eliminate aliasing problems, or use <code>IVDEP</code> directive on the loop. • 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> directive. • 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> directive on the loop.
(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"> • Change the data type to a larger data type. • Depending on the target architecture, use flush-to-zero or vectorization options. <p>IA-32 and Intel® EM64T:</p> <ul style="list-style-type: none"> • Flush-to-zero mode is enabled by default for SSE2 instructions. The Intel® EM64T compiler generates SSE2 instructions by

		<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"> See Vectorization Support for more information. <p>Itanium®:</p> <ul style="list-style-type: none"> 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 PROGRAM. Selecting <code>-O3</code> (Linux) or <code>/O3</code> (Windows) automatically enables <code>-ftz</code> (Linux) or <code>/Qftz</code> (Windows). <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> directive 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>
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>
Other Application Types		
Applications with many small functions that are called from multiple locations	Low	<p>Use <code>-ip</code> (Linux and Mac OS) or <code>/Qip</code> (Windows) to enable inter-procedural inlining within a single source module.</p> <p>Streamlines code execution for simple functions by duplicating the code within the code block that originally called the function. This will increase</p>

		<p>application size.</p> <p>As a general rule, do not inline large, complicated functions.</p> <p>See Interprocedural Optimizations Overview.</p>
(same as above)	Low	<p>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).</p> <p>Using this option will increase link time due to the extended program flow analysis that occurs.</p> <p>Interprocedural Optimization (IPO) can perform whole program analysis to help memory pointer disambiguation.</p>

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	<p>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.</p> <p>See Cache Blocking.</p>
Compiler directives for better alias analysis	Medium	<p>Ignore vector dependencies. Use <code>IVDEP</code> and other directives to increase application speed.</p> <p>See Vectorization Support.</p>
Memory pointer disambiguation compiler options	Medium	<p>Experiment with the following compiler options:</p> <ul style="list-style-type: none"> • <code>-fno-fnalias</code> (Linux) • <code>-ansi-alias</code> (Linux) or <code>/Qansi-alias</code> (Windows) • <code>-safe-cray-ptr</code> (Linux) or <code>/Qsafe-cray-ptr</code> (Windows)
Math functions	Low	<p>Call Math Kernel Library (MKL) instead of user code.</p> <p>Call F90 intrinsics instead of user code (to enable</p>

		optimizations).
--	--	-----------------

Library/OS Recommendations

Area	Impact	Description
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, directives, 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 directives 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 directives to get the best performance from your application.

Use the Math Kernel Library (MKL) instead of user code, or calling F90 intrinsics instead of user code.

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 `IDVEP` directive. Alternatively, you might use a directive that applies to a specified loop in a function. This is more precise than specifying an entire function. The directive 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.

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

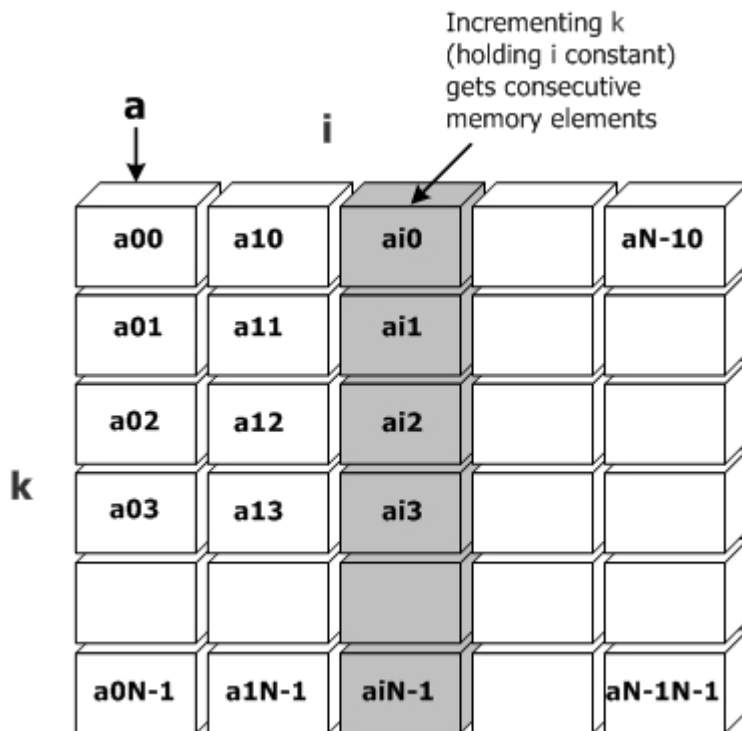
```
!Non-Unit Stride Memory Access
subroutine non_unit_stride_memory_access(a,b,c, NUM)
  implicit none
  integer :: i,j,k,NUM
  real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
! loop before loop interchange
  do i=1,NUM
    do j=1,NUM
      do k=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
      end do
    end do
  end do
end subroutine non_unit_stride_memory_access
```

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 $b[k=1][j=0]$, the loop has skipped over NUM memory locations having skipped $b[k][1]$, $b[k][2]$.. $b[k][NUM]$.

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:

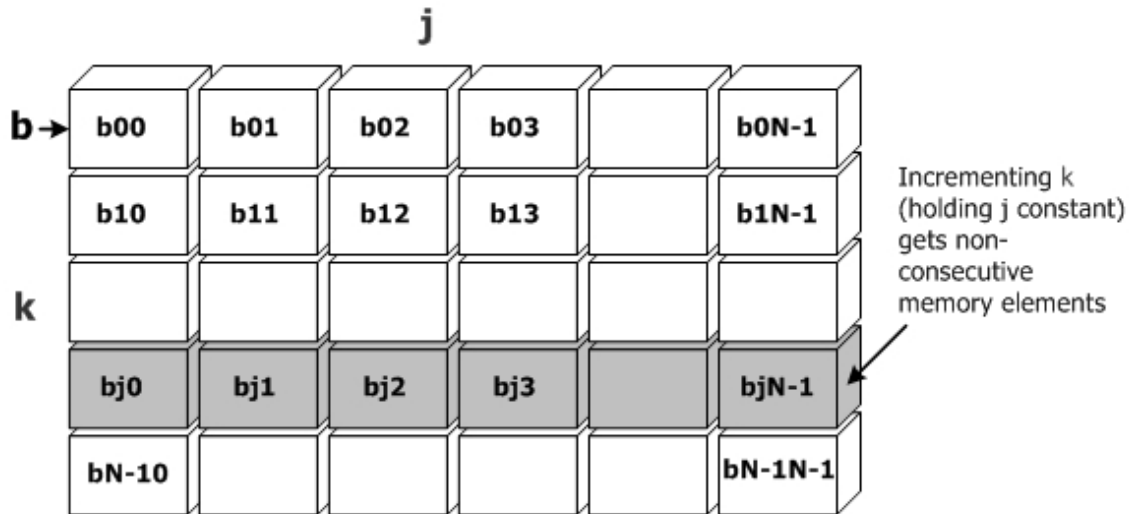


Assume you modify the example code listed above by making the following changes to introduce loop interchange:

Example

```
subroutine unit_stride_memory_access(a,b,c, NUM)
  implicit none
  integer :: i,j,k,NUM
  real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
  ! loop after interchange
  do i=1,NUM
    do k=1,NUM
      do j=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
      end do
    end do
  end do
end subroutine unit_stride_memory_access
```


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, and LoadPair. These transformations are discussed in the following sections, including how to transform loops manually and how to control them with directives 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

```
subroutine loop_interchange(a,b,c, NUM)
implicit none
integer :: i,j,k,NUM
real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
! loop before loop interchange
do i=1,NUM
  do j=1,NUM
    do k=1,NUM
      c(j,i) = c(j,i) + a(j,k) * b(k,i)
    end do
  end do
end do
! loop after interchange
do i=1,NUM
```

```

do k=1,NUM
  do j=1,NUM
    c(j,i) = c(j,i) + a(j,k) * b(k,i)
  end do
end do
end subroutine loop_interchange

```

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.

Example

```

subroutine loop_unroll_before(a,b,c,N,M)
implicit none
integer :: i,j,N,M
real :: a(N,M), b(N,M), c(N,M)
N=1025
M=5
do i=1,N
  do j=1,M
    a(j,i) = b(j,i) + c(j,i)
  end do
end do
end subroutine loop_unroll_before

```

Example

```

subroutine loop_unroll_after(a,b,c,N,M)
implicit none
integer :: i,j,K,N,M
real :: a(N,M), b(N,M), c(N,M)
N=1025
M=5
K=MOD(N,4) !K= N MOD 4
! main part of loop
do i=1,N-K,4
  do j=1,M
    a(j,i) = b(j,i) + c(j,i)
    a(j,i+1) = b(j,i+1) + c(j,i+1)
    a(j,i+2) = b(j,i+2) + c(j,i+2)
    a(j,i+3) = b(j,i+3) + c(j,i+3)
  end do
end do
! post conditioning part of loop
do i= N-K+2, N, 4
  do j=1,M
    a(j,i) = b(j,i) + c(j,i)
  end do
end do
end subroutine loop_unroll_after

```

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.

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

```
subroutine cache_blocking_before(a,b,N)
  implicit none
  integer :: i,j,k,N
  real :: a(N,N,N), b(N,N,N), c(N,N,N)
  N=1000
  do i = 1, N
    do j = 1, N
      do k = 1, N
        a(i,j,k) = a(i,j,k) + b(i,j,k)
      end do
    end do
  end do
end subroutine cache_blocking_before

subroutine cache_blocking_after(a,b,N)
  implicit none
  integer :: i,j,k,u,v,N
  real :: a(N,N,N), b(N,N,N), c(N,N,N)
  N=1000
  do v = 1, N, 20
    do u = 1, N, 20
      do k = v, v+19
        do j = u, u+19
          do i = 1, N
```

```

        a(i,j,k) = a(i,j,k) + b(i,j,k)
      end do
    end do
  end do
end do
end subroutine cache_blocking_after

```

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

```

subroutine loop_distribution_before(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
  do i = 1, N
    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
  end do
end subroutine loop_distribution_before

subroutine loop_distribution_after(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
  do i = 1, N
    a(i) = a(i) + i
    b(i) = b(i) + i
    c(i) = c(i) + i
  end do
  do i = 1, N
    x(i) = x(i) + i
    y(i) = y(i) + i
  end do
end subroutine loop_distribution_after

```

```

    z(i) = z(i) + i
  end do
end subroutine loop_distribution_after

```

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

Example

```
!DEC$ 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

```

subroutine loop_distribution_pragma1(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
!DEC$ distribute point
  do i = 1, N
    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
  end do
end subroutine loop_distribution_pragma1

```

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 directive within a loop to precisely indicate where the split should take place:

Example

```

subroutine loop_distribution_pragma2(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
  do i = 1, N
    a(i) = a(i) + i
    b(i) = b(i) + i
    c(i) = c(i) + i
!DEC$ distribute point
    x(i) = x(i) + i
    y(i) = y(i) + i
    z(i) = z(i) + i
  end do
end subroutine loop_distribution_pragma2

```

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 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.

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.

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.

See Setting Data Type and Alignment for more detailed information on aligning 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.

Methods of Timing Your Application

To perform application timings, use the VTune™ Performance Analyzer or a version of the `TIME` command in a `.BAT` file (or the function timing profiling option). You might consider modifying the program to call routines within the program to measure execution time (possibly using conditionally compiled lines).

For example:

- Intel Fortran intrinsic procedures, such as `SECNDS`, `CPU_TIME`, `SYSTEM_CLOCK`, `TIME`, and `DATE_AND_TIME`. See "Intrinsic Procedures" in the *Intel® Fortran Language Reference*.
- Portability library routines, such as `DCLOCK`, `ETIME`, `SECNDS`, or `TIME`. See "Portability Functions" in the *Intel® Fortran Libraries Reference*.

Whenever possible, perform detailed performance analysis on a system that closely resembles the system(s) that will be used for actual application use.

Sample Timing

The following program can be run by a `.BAT` file that executes the `TIME` command both before and after execution, to provide an approximate wall-clock time for the execution of the entire program. The Fortran intrinsic `CPU_TIME` can be used at selected points in the program to collect the CPU time between the start and end of the task to be timed.

Example

```
REAL time begin, time end
...
CALL CPU TIME ( time begin )
!
!task to be timed
!
CALL CPU TIME ( time end )
PRINT *, 'Time of operation was ', &
time_end - time_begin, ' seconds'
```

Considerations for Linux*

In the following example timings, the sample program being timed displays the following line:

Bourne* shell example

```
Average of all the numbers is:      4368488960.000000
```

Using the Bourne* shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

Bourne* shell example

```
$ time a.out
Average of all the numbers is:
4368488960.000000
real      0m2.46s
user      0m0.61s
sys       0m0.58s
```

Using the C shell, the following program timing reports 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use), about 4 seconds (0:04) of elapsed time, the use of 28% of available CPU time, and other information:

C shell example

```
% time a.out
Average of all the numbers is:      4368488960.000000

0.61u 0.58s 0:04 28% 78+424k 9+5io 0pf+0w
```

Using the bash shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

bash shell example

```
[user@system user]$ time ./a.out
Average of all the numbers is:      4368488960.000000
elapsed    0m2.46s
user       0m0.61s
sys        0m0.58s
```

Timings that indicate a large amount of system time is being used may suggest excessive I/O, a condition worth investigating.

If your program displays a lot of text, you can redirect the output from the program on the time command line. Redirecting output from the program will change the times reported because of reduced screen I/O.

For more information, see `time(1)`.

In addition to the `time` command, you might consider modifying the program to call routines within the program to measure execution time. For example, use the Intel Fortran intrinsic procedures, such as `SECNDS`, `DCLOCK`, `CPU_TIME`, `SYSTEM_CLOCK`, `TIME`,

and `DATE_AND_TIME`. See "Intrinsic Procedures" in the *Intel® Fortran Language Reference*.

Coding Guidelines for Intel® Architectures

This topic provides general guidelines for coding practices and techniques for using:

- IA-32 architecture supporting MMX™ technology and Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), and Streaming SIMD Extensions 3 (SSE3)
- Itanium® architecture

This section describes practices, tools, coding rules and recommendations associated with the architecture features that can improve the performance on IA-32 and Itanium processor families. For all details about optimization for IA-32 processors, see Intel® Architecture Optimization Reference Manual (<http://developer.intel.com/design/pentiumii/manuals/245127.htm>). For all details about optimization for Itanium processor family, see the Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization (<http://developer.intel.com/design/itanium2/manuals/251110.htm>).

Note

If a guideline refers to a particular architecture only, this architecture is explicitly named. The default is for IA-32 and Itanium architectures.

Performance of compiler-generated code may vary from one compiler to another. Intel® Visual Fortran Compiler generates code that is highly optimized for Intel architectures. You can significantly improve performance by using various compiler optimization options. In addition, you can help the compiler to optimize your Fortran program by following the guidelines described in this section.

To achieve optimum processor performance in your Fortran application, do the following:

- avoiding memory access stalls
- ensuring good floating-point performance
- ensuring good SIMD integer performance
- using vectorization

The following sections summarize and describe coding practices, rules and recommendations associated with the features that will contribute to optimizing the performance on Intel architecture-based processors.

Memory Access

The Intel compiler lays out arrays in column-major order. For example, in a two-dimensional array, elements `A(22, 34)` and `A(23, 34)` are contiguous in memory. For best performance, code arrays so that inner loops access them in a contiguous manner.

Consider the following examples. The code in example 1 will likely have higher performance than the code in example 2.

Example 1

```
subroutine contiguous(a, b, N)
  integer :: i, j, N, a(N,N), b(N,N)
  do j = 1, N
    do i = 1, N
      b(i, j) = a(i, j) + 1
    end do
  end do
end subroutine contiguous
```

The code above illustrates access to arrays A and B in the inner loop I in a contiguous manner which results in good performance; however, the following example illustrates access to arrays A and B in inner loop J in a non-contiguous manner which results in poor performance.

Example 2

```
subroutine non_contiguous(a, b, N)
  integer :: i, j, N, a(N,N), b(N,N)
  do i = 1, N
    do j = 1, N
      b(i, j) = a(i, j) + 1
    end do
  end do
end subroutine non_contiguous
```

The compiler can transform the code so that inner loops access memory in a contiguous manner. To do that, you need to use advanced optimization options, such as `-O3` (Linux*) or `/O3` (Windows*) for both IA-32 and Itanium architectures, and `-O3` (Linux) or `/O3` (Windows) and `-ax` (Linux) or `/Qax` (Windows) for IA-32 only; `-axP` is the default on Mac OS* systems if you specify `-O3`.

Memory Layout

Alignment is an increasingly important factor in ensuring good performance. Aligned memory accesses are faster than unaligned accesses. If you use the interprocedural optimization on multiple files, the `-ipo` (Linux) or `/Qipo` (Windows) option, the compiler analyzes the code and decides whether it is beneficial to pad arrays so that they start from an aligned boundary. Multiple arrays specified in a single common block can impose extra constraints on the compiler.

For example, consider the following `COMMON` statement:

Example 3

```
COMMON /AREA1/ A(200), X, B(200)
```

If the compiler added padding to align `A(1)` at a 16-byte aligned address, the element `B(1)` would not be at a 16-byte aligned address. So it is better to split `AREA1` as follows.

Example 4

```
COMMON /AREA1/ A(200)
COMMON /AREA2/ X
COMMON /AREA3/ B(200)
```

The above code provides the compiler maximum flexibility in determining the padding required for both A and B.

Optimizing for Floating-point Applications

To improve floating-point performance, follow these rules:

- Avoid exceeding representable ranges during computation since handling these cases can have a performance impact. Use REAL variables in single precision format unless the extra precision obtained through `DOUBLE` or `REAL*8` with a larger precision formation will also increase memory size and bandwidth requirements.
- **For IA-32:** Avoid repeatedly changing rounding modes between more than two values, which can lead to poor performance when the computation is done using non-SSE instructions. Hence avoid using `FLOOR` and `TRUNC` instructions together when generating non-SSE code. The same applies for using `CEIL` and `TRUNC`. Another way to avoid the problem is to use the `-x` (Linux) or `/Qx` (Windows) option to do the computation using SSE instructions.
- Reduce the impact of denormal exceptions for both architectures as described below.

Denormal Exceptions

Floating point computations with underflow can result in denormal values that have an adverse impact on performance.

- **For IA-32:** take advantage of the SIMD capabilities of Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), and Streaming SIMD Extensions 3 (SSE3) instructions.

The `-x` (Linux) or `/Qx` (Windows) options enable the flush-to-zero (FTZ) mode in SSE and SSE2 instructions, whereby underflow results are automatically converted to zero, which improves application performance. In addition, the `-xP` (Linux) or `/QxP` (Windows) option also enables the denormals-are-zero (DAZ) mode, whereby denormals are converted to zero on input, further improving performance. An application developer willing to trade pure IEEE-754 compliance for speed would benefit from these options. For more information on FTZ and DAZ, see Setting FTZ and DAZ Flags and "Floating-point Exceptions" in the Intel® Architecture Optimization Reference Manual (<http://developer.intel.com/design/pentiumii/manuals/245127.htm>).

- **For Itanium® architecture:** enable flush-to-zero (FTZ) mode with the `-ftz` (Linux) or `/Qftz` (Windows) set by `-O3` (Linux) or `/O3` (Windows) option.

Auto-vectorization (IA-32 Only)

Many applications significantly increase their performance if they can implement vectorization, which uses streaming SIMD SSE2 instructions for the main computational loops. The Intel Compiler turns vectorization on (auto-vectorization) or you can implement it with compiler directives. See Auto-vectorization (IA-32 Only) section for complete details.

Creating Multithreaded Applications

The Intel Fortran Compiler and the Intel® Threading Toolset have the capabilities that make developing multithreaded application easy. See Parallelism Overview. Multithreaded applications can show significant benefit on dual-core processor and multiprocessor Intel symmetric multiprocessing (SMP) systems or on Intel processors with Hyper-Threading technology.

Setting Data Type and Alignment

Alignment of data affects these kinds of variables:

- Those that are dynamically allocated. (Dynamically allocated data allocated with `ALLOCATE` is 8-byte aligned.)
- Those that are members of a data structure
- Those that are global or local variables
- Those that are parameters passed on the stack

For best performance, align data as follows:

- Align 8-bit data at any address.
- Align 16-bit data to be contained within an aligned four byte word.
- Align 32-bit data so that its base address is a multiple of four.
- Align 64-bit data so that its base address is a multiple of eight.
- Align 128-bit data so that its base address is a multiple of sixteen (8-byte boundaries).

Causes of Unaligned Data and Ensuring Natural Alignment

For optimal performance, make sure your data is aligned naturally. A natural boundary is a memory address that is a multiple of the data item's size. For example, a `REAL (KIND=8)` data item aligned on natural boundaries has an address that is a multiple of 8. An array is aligned on natural boundaries if all of its elements are so aligned.

All data items whose starting address is on a natural boundary are naturally aligned. Data not aligned on a natural boundary is called unaligned data.

Although the Intel® compiler naturally aligns individual data items when it can, certain Fortran statements can cause data items to become unaligned.

You can use the `align` command-option to ensure naturally aligned data, but you should check and consider reordering data declarations of data items within common blocks, derived-type structures, and record structures as follows:

- Carefully specify the order and sizes of data declarations to ensure naturally aligned data.
- Start with the largest size numeric items first, followed by smaller size numeric items, and then non-numeric (character) data.

The `!DEC$ ATTRIBUTES ALIGN` directive specifies the byte alignment for a variable. It is not supported for `ALLOCATABLE/POINTER` variables.

Common blocks (`COMMON` statement), derived-type data, and Fortran 77 record structures (`RECORD` statement) usually contain multiple items within the context of the larger structure.

The following statements can cause unaligned data:

Statement	Options	Description
Common blocks (COMMON statement)	<code>commons</code> or <code>dcommons</code>	<p>The order of variables in the <code>COMMON</code> statement determines their storage order. Unless you are sure that the data items in the common block will be naturally aligned, specify either <code>-align commons</code> or <code>-align dcommons</code> (Linux*) or <code>/align:commons</code> or <code>/align:dcommons</code> (Windows*), depending on the largest data size used.</p> <p>Refer to the User's Guide for detailed information on using these statements and options.</p>
Derived-type (user-defined) data	<code>records</code> or <code>sequence</code>	<p>Derived-type data items are declared after a <code>TYPE</code> statement.</p> <p>If your data includes derived-type data structures, you should use the <code>-align records</code> (Linux) or <code>/align:records</code> (Windows) option, unless you are sure that the data items in the derived-type structures will be naturally aligned.</p> <p>If you omit the <code>SEQUENCE</code> statement, the <code>-align records</code> (Linux) or <code>/align:records</code> (Windows) option (default) ensures all data items are naturally aligned.</p> <p>If you specify the <code>SEQUENCE</code> statement, the <code>-align records</code> (Linux) or <code>/align:records</code> (Windows) option is prevented from adding necessary padding</p>

		to avoid unaligned data (data items are packed) unless you specify <code>SEQUENCE</code> . When you use <code>SEQUENCE</code> , you should specify data declaration order so that all data items are naturally aligned.
Record structures (RECORD and STRUCTURE statements)	<code>record</code> or <code>structure</code>	Intel Fortran record structures usually contain multiple data items. The order of variables in the <code>STRUCTURE</code> statement determines their storage order. The <code>RECORD</code> statement names the record structure. Record structures are an Intel Fortran language extension. If your data includes record structures, you should use the <code>-align records</code> (Linux) or <code>/align:records</code> (Windows) option, unless you are sure that the data items in the record structures will be naturally aligned.
EQUIVALENCE statements		<code>EQUIVALENCE</code> statements can force unaligned data or cause data to span natural boundaries. For more information, see the <i>Intel® Fortran Language Reference</i> .

To avoid unaligned data in a common block, derived-type data, or record structure (extension), use one or both of the following:

- For new programs or for programs where the source code declarations can be modified easily, plan the order of data declarations with care. For example, you should order variables in a `COMMON` statement such that numeric data is arranged from largest to smallest, followed by any character data (see the data declaration rules in Ordering Data Declarations to Avoid Unaligned Data below).
- For existing programs where source code changes are not easily done or for array elements containing derived-type or record structures, you can use command line options to request that the compiler align numeric data by adding padding spaces where needed.

Other possible causes of unaligned data include unaligned actual arguments and arrays that contain a derived-type structure or Intel Fortran record structure as detailed below.

- When actual arguments from outside the program unit are not naturally aligned, unaligned data access occurs. Intel Fortran assumes all passed arguments are naturally aligned and has no information at compile time about data that will be introduced by actual arguments during program execution.
- For arrays where each array element contains a derived-type structure or Intel Fortran record structure, the size of the array elements may cause some elements (but not the first) to start on an unaligned boundary.
- Even if the data items are naturally aligned within a derived-type structure without the `SEQUENCE` statement or a record structure, the size of an array element might require use of the `-align records` (Linux) or `/align:records` (Windows) option to supply needed padding to avoid some array elements being unaligned.
- If you specify `-align norecords` (Linux) or `/align:norecords` (Windows) or specify `-vms` (Linux) or `/Qvms` (Windows) without `RECORDS` no padding bytes are

added between array elements. If array elements each contain a derived-type structure with the `SEQUENCE` statement, array elements are packed without padding bytes regardless of the Fortran command options specified. In this case, some elements will be unaligned.

- When the `-align records` (Linux) or `/align:records` (Windows) option is in effect, the number of padding bytes added by the compiler for each array element is dependent on the size of the largest data item within the structure. The compiler determines the size of the array elements as an exact multiple of the largest data item in the derived-type structure without the `SEQUENCE` statement or a record structure. The compiler then adds the appropriate number of padding bytes. For instance, if a structure contains an 8-byte floating-point number followed by a 3-byte character variable, each element contains five bytes of padding (16 is an exact multiple of 8). However, if the structure contains one 4-byte floating-point number, one 4-byte integer, followed by a 3-byte character variable, each element would contain one byte of padding (12 is an exact multiple of 4).

Checking for Inefficient Unaligned Data

During compilation, the Intel® compiler naturally aligns as much data as possible. Exceptions that can result in unaligned data are described above.

Because unaligned data can slow run-time performance, it is worthwhile to:

- Double-check data declarations within common block, derived-type data, or record structures to ensure all data items are naturally aligned (see the data declaration rules in the subsection below). Using modules to contain data declarations can ensure consistent alignment and use of such data.
- Avoid the `EQUIVALENCE` statement or use it in a manner that cannot cause unaligned data or data spanning natural boundaries.
- Ensure that passed arguments from outside the program unit are naturally aligned.
- Check that the size of array elements containing at least one derived-type data or record structure (extension) cause array elements to start on aligned boundaries (see the previous subsection).
- There are two ways unaligned data might be reported:
 - During compilation, warning messages are issued for any data items that are known to be unaligned (unless you specify the `-warn noalignments` (or `-warn none`) (Linux) or `/warn:noalignments` (or `/warn:none`) (Windows) option that suppresses all warnings).
 - During program execution, warning messages are issued for any data that is detected as unaligned. The message includes the address of the unaligned access.

Consider the following run-time message:

Example

```
Unaligned access pid=24821 <a.out> va=140000154, pc=3ff80805d60, ra=1200017bc
```


This message shows that:

- The statement accessing the unaligned data (program counter) is located at 3ff80805d60
- The unaligned data is located at address 140000154

Ordering Data Declarations to Avoid Unaligned Data

For new programs or when the source declarations of an existing program can be easily modified, plan the order of your data declarations carefully to ensure the data items in a common block, derived-type data, record structure, or data items made equivalent by an `EQUIVALENCE` statement will be naturally aligned.

Use the following rules to prevent unaligned data:

- Always define the largest size numeric data items first.
- If your data includes a mixture of character and numeric data, place the numeric data first.
- Add small data items of the correct size (or padding) before otherwise unaligned data to ensure natural alignment for the data that follows.

When declaring data, consider using explicit length declarations, such as specifying a `KIND` parameter. For example, specify `INTEGER(KIND=4)` (or `INTEGER(4)`) rather than `INTEGER`. If you do use a default size (such as `INTEGER`, `LOGICAL`, `COMPLEX`, and `REAL`), be aware that the following compiler options can change the size of an individual field's data declaration size and thus can alter the data alignment of a carefully planned order of data declarations:

Platform	Compiler Option
Windows*	/4I or /4R
Linux*	-integer_size or -real_size

Using the suggested data declaration guidelines minimizes the need to use the `-align keyword` (Linux) or `/align:keyword` (Windows) options to add padding bytes to ensure naturally aligned data. In cases where the `-align keyword` (Linux) or `/align:keyword` (Windows) options are still needed, using the suggested data declaration guidelines can minimize the number of padding bytes added by the compiler.

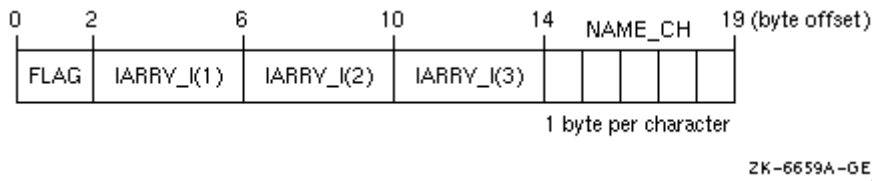
Arranging Data Items in Common Blocks

The order of data items in a `COMMON` statement determines the order in which the data items are stored. Consider the following declaration of a common block named `x`:

Example
<pre>logical (kind=2) flag integer iarray i(3) character(len=5) name ch common /x/ flag, iarray i(3), name ch</pre>

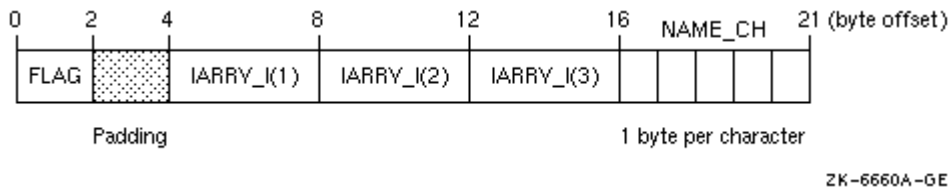
As shown in Figure 1-1, if you omit the appropriate Fortran command options, the common block will contain unaligned data items beginning at the first array element of `IARRY_I`.

Figure 1-1 Common Block with Unaligned Data



As shown in Figure 1-2, if you compile the program units that use the common block with the `-align commons` (Linux) or `/align:commons` (Windows) option, data items will be naturally aligned.

Figure 1-2 Common Block with Naturally Aligned Data



Because the common block `x` contains data items whose size is 32 bits or smaller, specify the `-align commons` (Linux) or `/align:commons` (Windows) option. If the common block contains data items whose size might be larger than 32 bits (such as `REAL (KIND=8)` data), use the `-align commons` (Linux) or `/align:commons` (Windows) option.

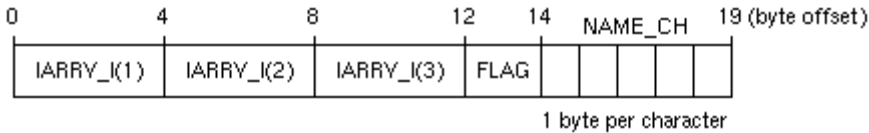
If you can easily modify the source files that use the common block data, define the numeric variables in the `COMMON` statement in descending order of size and place the character variable last. This provides more portability, ensures natural alignment without padding, and does not require the Fortran command options `-align commons` (Linux) or `-align dcommons` (Windows) or `-align commons` (Linux) or `/align:commons` (Windows):

Example

```
LOGICAL (KIND=2) FLAG
INTEGER          IARRY_I(3)
CHARACTER(LEN=5) NAME_CH
COMMON /X/ IARRY_I(3), FLAG, NAME_CH
```

As shown in Figure 1-3, if you arrange the order of variables from largest to smallest size and place character data last, the data items will be naturally aligned.

Figure 1-3 Common Block with Naturally Aligned Reordered Data



ZK-7915A-GE

When modifying or creating all source files that use common block data, consider placing the common block data declarations in a module so the declarations are consistent. If the common block is not needed for compatibility (such as file storage or Fortran 77 use), you can place the data declarations in a module without using a `COMMON` block.

Arranging Data Items in Derived-Type Data

Like common blocks, derived-type data may contain multiple data items (members).

Data item components within derived-type data will be naturally aligned on up to 64-bit boundaries, with certain exceptions related to the use of the `SEQUENCE` statement and Fortran options. See Alignment Options for links to resources about these exceptions.

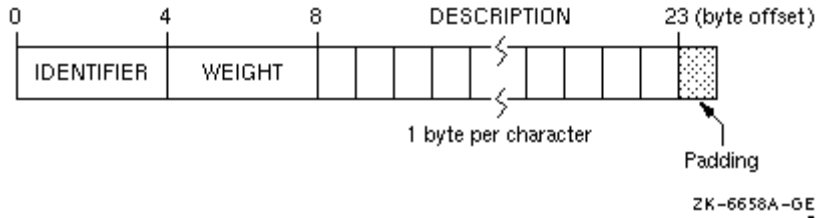
Intel Fortran stores a derived data type as a linear sequence of values, as follows:

- If you specify the `SEQUENCE` statement, the first data item is in the first storage location and the last data item is in the last storage location. The data items appear in the order in which they are declared. The Fortran options have no effect on unaligned data, so data declarations must be carefully specified to naturally align data. The `-align sequence (Linux)` or `/align:sequence (Windows)` option specifically aligns data items in a `SEQUENCE` derived-type on natural boundaries.
- If you omit the `SEQUENCE` statement, the Intel Fortran adds the padding bytes needed to naturally align data item components, unless you specify the `-align norecords (Linux)` or `/align:norecords (Windows)` option.

Consider the following declaration of array `CATALOG_SPRING` of derived-type `PART_DT`:

Example
<pre>MODULE DATA_DEFS TYPE PART_DT INTEGER IDENTIFIER REAL WEIGHT CHARACTER(LEN=15) DESCRIPTION END TYPE PART_DT TYPE (PART_DT) CATALOG_SPRING(30) ... END MODULE DATA_DEFS</pre>

As shown in Figure 1-4, the largest numeric data items are defined first and the character data type is defined last. There are no padding characters between data items and all items are naturally aligned. The trailing padding byte is needed because `CATALOG_SPRING` is an array; it is inserted by the compiler when the `-align records (Linux)` or `/align:records (Windows)` option is in effect.

Figure 1-4 Derived-Type Naturally Aligned Data (in CATALOG_SPRING : (,))

Arranging Data Items in Intel Fortran Record Structures

Intel Fortran supports record structures provided by Intel Fortran. Intel Fortran record structures use the `RECORD` statement and optionally the `STRUCTURE` statement, which are extensions to the Fortran 77 and Fortran standards. The order of data items in a `STRUCTURE` statement determines the order in which the data items are stored.

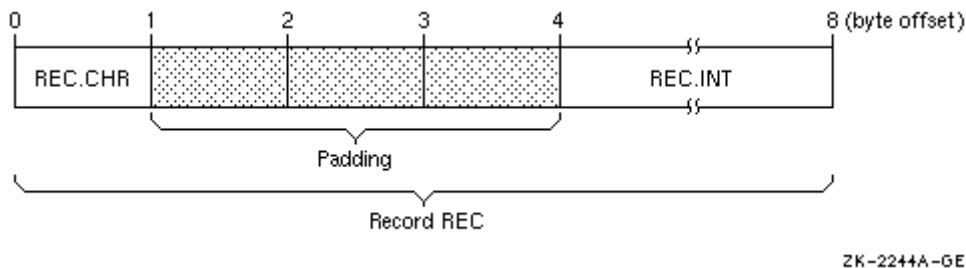
Intel Fortran stores a record in memory as a linear sequence of values, with the record's first element in the first storage location and its last element in the last storage location. Unless you specify `-align norecords` (Linux) or `/align:norecords` (Windows) padding bytes are added if needed to ensure data fields are naturally aligned.

The following example contains a structure declaration, a `RECORD` statement, and diagrams of the resulting records as they are stored in memory:

Example

```
STRUCTURE /STRA/
  CHARACTER*1 CHR
  INTEGER*4 INT
END STRUCTURE
...
RECORD /STRA/ REC
```

Figure 1-5 shows the memory diagram of record `REC` for naturally aligned records.

Figure 1-5 Memory Diagram of `REC` for Naturally Aligned Records

Using Arrays Efficiently

This topic discusses how to efficiently access arrays and pass array arguments.

Accessing Arrays Efficiently

Many of the array access efficiency techniques described in this section are applied automatically by the Intel Fortran loop transformation optimizations. Several aspects of array use can improve run-time performance; the following sections discuss the most important aspects.

Perform the fewest operations necessary

The fastest array access occurs when contiguous access to the whole array or most of an array occurs. Perform one or a few array operations that access all of the array or major parts of an array instead of numerous operations on scattered array elements. Rather than use explicit loops for array access, use elemental array operations, such as the following line that increments all elements of array variable `A`:

Example

```
A = A + 1
```

When reading or writing an array, use the array name and not a DO loop or an implied DO-loop that specifies each element number. Fortran 95/90 array syntax allows you to reference a whole array by using its name in an expression.

For example:

Example

```
REAL :: A(100,100)
A = 0.0
A = A + 1           ! Increment all elements
                   ! of A by 1
...
WRITE (8) A         ! Fast whole array use
```

Similarly, you can use derived-type array structure components, such as:

Example

```
TYPE X
  INTEGER A(5)
END TYPE X
...
TYPE (X) Z
WRITE (8) Z%A       ! Fast array structure
                   ! component use
```

Access arrays using the proper array syntax

Make sure multidimensional arrays are referenced using proper array syntax and are traversed in the natural ascending storage order, which is column-major order for Fortran. With column-major order, the leftmost subscript varies most rapidly with a stride of one. Whole array access uses column-major order.

Avoid row-major order, as is done by C, where the rightmost subscript varies most rapidly. For example, consider the nested DO loops that access a two-dimension array with the J loop as the innermost loop:

Example

```

INTEGER  X(3,5), Y(3,5), I, J
Y = 0
DO I=1,3
  DO J=1,5
    X (I,J) = Y(I,J) + 1
  END DO
END DO
...
END PROGRAM

```

! I outer loop varies slowest
! J inner loop varies fastest
! Inefficient row-major storage order
! (rightmost subscript varies fastest)

Since J varies the fastest and is the second array subscript in the expression X (I,J), the array is accessed in row-major order. To make the array accessed in natural column-major order, examine the array algorithm and data being modified. Using arrays X and Y, the array can be accessed in natural column-major order by changing the nesting order of the DO loops so the innermost loop variable corresponds to the leftmost array dimension:

Example

```

INTEGER  X(3,5), Y(3,5), I, J
Y = 0
DO J=1,5
  DO I=1,3
    X (I,J) = Y(I,J) + 1
  END DO
END DO
...
END PROGRAM

```

! J outer loop varies slowest
! I inner loop varies fastest
! Efficient column-major storage order
! (leftmost subscript varies fastest)

The Intel Fortran whole array access (X = Y + 1) uses efficient column major order. However, if the application requires that J vary the fastest or if you cannot modify the loop order without changing the results, consider modifying the application to use a rearranged order of array dimensions. Program modifications include rearranging the order of:

- Dimensions in the declaration of the arrays X(5,3) and Y(5,3)
- The assignment of X(J,I) and Y(J,I) within the DO loops
- All other references to arrays X and Y

In this case, the original DO loop nesting is used where J is the innermost loop:

Example

```

INTEGER  X(5,3), Y(5,3), I, J
Y = 0
DO I=1,3
  DO J=1,5
    X (J,I) = Y(J,I) + 1
  END DO
END DO

```

! I outer loop varies slowest
! J inner loop varies fastest
! Efficient column-major storage order
! (leftmost subscript varies fastest)

```
...
END PROGRAM
```

Code written to access multidimensional arrays in row-major order (like C) or random order can often make inefficient use of the CPU memory cache. For more information on using natural storage order during record, see Improving I/O Performance.

Use available intrinsics

Use the available Fortran 95/90 array intrinsic procedures rather than create your own.

Whenever possible, use Fortran 95/90 array intrinsic procedures instead of creating your own routines to accomplish the same task. Fortran 95/90 array intrinsic procedures are designed for efficient use with the various Intel Fortran run-time components.

Using the standard-conforming array intrinsics can also make your program more portable.

Avoid leftmost array dimensions

With multidimensional arrays where access to array elements will be noncontiguous, avoid leftmost array dimensions that are a power of two (such as 256, 512).

Since the cache sizes are a power of 2, array dimensions that are also a power of 2 may make inefficient use of cache when array access is noncontiguous. If the cache size is an exact multiple of the leftmost dimension, your program will probably make use of the cache less efficient. This does not apply to contiguous sequential access or whole array access.

One work-around is to increase the dimension to allow some unused elements, making the leftmost dimension larger than actually needed. For example, increasing the leftmost dimension of A from 512 to 520 would make better use of cache:

Example

```
REAL A (512,100)
DO I = 2,511
  DO J = 2,99
    A(I,J) = (A(I+1,J-1) + A(I-1, J+1)) * 0.5
  END DO
END DO
```

In this code, array A has a leftmost dimension of 512, a power of two. The innermost loop accesses the rightmost dimension (row major), causing inefficient access. Increasing the leftmost dimension of A to 520 (REAL A (520,100)) allows the loop to provide better performance, but at the expense of some unused elements.

Because loop index variables I and J are used in the calculation, changing the nesting order of the DO loops changes the results.

For more information on arrays and their data declaration statements, see the *Intel® Fortran Language Reference*.

Passing Array Arguments Efficiently

In Fortran, there are two general types of array arguments:

- Explicit-shape arrays (introduced with Fortran 77); for example, `A(3,4)` and `B(0:*)`

These arrays have a fixed rank and extent that is known at compile time. Other dummy argument (receiving) arrays that are not deferred-shape (such as assumed-size arrays) can be grouped with explicit-shape array arguments.

- Deferred-shape arrays (introduced with Fortran 95/90); for example, `C(:,:)`

Types of deferred-shape arrays include array pointers and allocatable arrays. Assumed-shape array arguments generally follow the rules about passing deferred-shape array arguments.

When passing arrays as arguments, either the starting (base) address of the array or the address of an array descriptor is passed:

- When using explicit-shape (or assumed-size) arrays to receive an array, the starting address of the array is passed.
- When using deferred-shape or assumed-shape arrays to receive an array, the address of the array descriptor is passed (the compiler creates the array descriptor).

Passing an assumed-shape array or array pointer to an explicit-shape array can slow run-time performance. This is because the compiler needs to create an array temporary for the entire array. The array temporary is created because the passed array may not be contiguous and the receiving (explicit-shape) array requires a contiguous array. When an array temporary is created, the size of the passed array determines whether the impact on slowing run-time performance is slight or severe.

The following table summarizes what happens with the various combinations of array types. The amount of run-time performance inefficiency depends on the size of the array.

	Dummy Argument Array Types (choose one)	
Actual Argument Array Types (choose one)	Explicit-Shape Arrays	Deferred-Shape and Assumed-Shape Arrays
Explicit-Shape	<u>Result when using this</u>	<u>Result when using this</u>

Arrays	<p><u>combination</u>: Very efficient. Does not use an array temporary. Does not pass an array descriptor.</p> <p>Interface block optional.</p>	<p><u>combination</u>: Efficient. Only allowed for assumed-shape arrays (not deferred-shape arrays).</p> <p>Does not use an array temporary. Passes an array descriptor.</p> <p>Requires an interface block.</p>
Deferred-Shape and Assumed-Shape Arrays	<p><u>Result when using this combination</u>: When passing an allocatable array, very efficient. Does not use an array temporary. Does not pass an array descriptor. Interface block optional.</p> <p>When not passing an allocatable array, not efficient. Instead use allocatable arrays whenever possible.</p> <p>Uses an array temporary. Does not pass an array descriptor. Interface block optional.</p>	<p><u>Result when using this combination</u>: Efficient. Requires an assumed-shape or array pointer as dummy argument.</p> <p>Does not use an array temporary. Passes an array descriptor.</p> <p>Requires an interface block.</p>

Improving I/O Performance

Improving overall I/O performance can minimize both device I/O and actual CPU time. The techniques listed in this topic can significantly improve performance in many applications.

An I/O flow problems limit the maximum speed of execution by being the slowest process in an executing program. In some programs, I/O is the bottleneck that prevents an improvement in run-time performance. The key to relieving I/O problems is to reduce the actual amount of CPU and I/O device time involved in I/O.

The problems can be caused by one or more of the following:

- A dramatic reduction in CPU time without a corresponding improvement in I/O time.
- Such coding practices as:
 - Unnecessary formatting of data and other CPU-intensive processing
 - Unnecessary transfers of intermediate results
 - Inefficient transfers of small amounts of data
 - Application requirements

Improved coding practices can minimize actual device I/O, as well as the actual CPU time. Intel offers software solutions to system-wide problems like minimizing device I/O delays.

Use Unformatted Files Instead of Formatted Files

Use unformatted files whenever possible. Unformatted I/O of numeric data is more efficient and more precise than formatted I/O. Native unformatted data does not need to be modified when transferred and will take up less space on an external file.

Conversely, when writing data to formatted files, formatted data must be converted to character strings for output, less data can transfer in a single operation, and formatted data may lose precision if read back into binary form.

To write the array `A(25,25)` in the following statements, `S1` is more efficient than `S2`:

Example	
<code>S1</code>	<code>WRITE (7) A</code>
<code>S2</code>	<code>WRITE (7,100) A</code>
<code>100</code>	<code>FORMAT (25(' ',25F5.21))</code>

Although formatted data files are more easily ported to other systems, Intel Fortran can convert unformatted data in several formats; see Little-endian-to-Big-endian Conversion (IA-32).

Write Whole Arrays or Strings

To eliminate unnecessary overhead, write whole arrays or strings at one time rather than individual elements at multiple times. Each item in an I/O list generates its own calling sequence. This processing overhead becomes most significant in implied-`DO` loops. When accessing whole arrays, use the array name (Fortran array syntax) instead of using implied-`DO` loops.

Write Array Data in the Natural Storage Order

Use the natural ascending storage order whenever possible. This is column-major order, with the leftmost subscript varying fastest and striding by 1. (See Accessing Arrays Efficiently.) If a program must read or write data in any other order, efficient block moves are inhibited.

If the whole array is not being written, natural storage order is the best order possible.

If you must use an unnatural storage order, in certain cases it might be more efficient to transfer the data to memory and reorder the data before performing the I/O operation.

Use Memory for Intermediate Results

Performance can improve by storing intermediate results in memory rather than storing them in a file on a peripheral device. One situation that may not benefit from using intermediate storage is when there is a disproportionately large amount of data in relation to physical memory on your system. Excessive page faults can dramatically impede virtual memory performance.

Linux* Only: If you are primarily concerned with the CPU performance of the system, consider using a memory file system (mfs) virtual disk to hold any files your code reads or writes.

Enable Implied-DO Loop Collapsing

DO loop collapsing reduces a major overhead in I/O processing. Normally, each element in an I/O list generates a separate call to the Intel Fortran run-time library (RTL). The processing overhead of these calls can be most significant in implied-DO loops.

Intel Fortran reduces the number of calls in implied-DO loops by replacing up to seven nested implied-DO loops with a single call to an optimized run-time library I/O routine. The routine can transmit many I/O elements at once.

Loop collapsing can occur in formatted and unformatted I/O, but only if certain conditions are met:

- The control variable must be an integer. The control variable cannot be a dummy argument or contained in an `EQUIVALENCE` or `VOLATILE` statement. Intel Fortran must be able to determine that the control variable does not change unexpectedly at run time.
- The format must not contain a variable format expression.

For information on the `VOLATILE` attribute and statement, see the *Intel® Fortran Language Reference*.

For loop optimizations, see Loop Transformations, Loop Unrolling, and Optimization Levels.

Use of Variable Format Expressions

Variable format expressions (an Intel Fortran extension) are almost as flexible as run-time formatting, but they are more efficient because the compiler can eliminate run-time parsing of the I/O format. Only a small amount of processing and the actual data transfer are required during run time.

On the other hand, run-time formatting can impair performance significantly. For example, in the following statements, `S1` is more efficient than `S2` because the formatting is done once at compile time, not at run time:

Example	
<code>S1</code>	<code>WRITE (6,400) (A(I), I=1,N)</code>
<code>400</code>	<code>FORMAT (1X, <N> F5.2)</code>
<code>...</code>	
<code>S2</code>	<code>WRITE (CHFMT,500) ' (1X, ',N, ' F5.2) '</code>
<code>500</code>	<code>FORMAT (A,I3,A)</code>
	<code>WRITE (6,FMT=CHFMT) (A(I), I=1,N)</code>

Efficient Use of Record Buffers and Disk I/O

Records being read or written are transferred between the user's program buffers and one or more disk block I/O buffers, which are established when the file is opened by the Intel Fortran RTL. Unless very large records are being read or written, multiple logical records can reside in the disk block I/O buffer when it is written to disk or read from disk, minimizing physical disk I/O.

You can specify the size of the disk block physical I/O buffer by using the `OPEN` statement `BLOCKSIZE` specifier; the default size can be obtained from `fstat(2)`. If you omit the `BLOCKSIZE` specifier in the `OPEN` statement, it is set for optimal I/O use with the type of device the file resides on (with the exception of network access).

The `OPEN` statement `BUFFERCOUNT` specifier specifies the number of I/O buffers. The default for `BUFFERCOUNT` is 1. Any experiments to improve I/O performance should increase the `BUFFERCOUNT` value and not the `BLOCKSIZE` value, to increase the amount of data read by each disk I/O.

If the `OPEN` statement has `BLOCKSIZE` and `BUFFERCOUNT` specifiers, then the internal buffer size in bytes is the product of these specifiers. If the `open` statement does not have these specifiers, then the default internal buffer size is 8192 bytes. This internal buffer will grow to hold the largest single record, but will never shrink.

The default for the Fortran run-time system is to use unbuffered disk writes. That is, by default, records are written to disk immediately as each record is written instead of accumulating in the buffer to be written to disk later.

To enable buffered writes (that is, to allow the disk device to fill the internal buffer before the buffer is written to disk), use one of the following:

- The `OPEN` statement `BUFFERED` specifier
- The `-assume buffered_io` (Linux*) or `/assume:buffered_io` (Windows*) option of the `ASSUME` command
- The `FORT_BUFFERED` run-time environment variable

The `OPEN` statement `BUFFERED` specifier takes precedence over the `-assume buffered_io` (Linux) or `/assume:buffered_io` (Windows) option. If neither one is set (which is the default), the `FORT_BUFFERED` environment variable is tested at run time.

The `OPEN` statement `BUFFERED` specifier applies to a specific logical unit. In contrast, the `-assume nobuffered_io` (Linux) or `/assume:nobuffered_io` (Windows) option and the `FORT_BUFFERED` environment variable apply to all Fortran units.

Using buffered writes usually makes disk I/O more efficient by writing larger blocks of data to the disk less often. However, a system failure when using buffered writes can cause records to be lost, since they might not yet have been written to disk. (Such records would have been written to disk with the default unbuffered writes.)

When performing I/O across a network, be aware that the size of the block of network data sent across the network can impact application efficiency. When reading network data, follow the same advice for efficient disk reads, by increasing the `BUFFERCOUNT`. When writing data through the network, several items should be considered:

- Unless the application requires that records be written using unbuffered writes, enable buffered writes by a method described above.
- Especially with large files, increasing the `BLOCKSIZE` value increases the size of the block sent on the network and how often network data blocks get sent.
- Time the application when using different `BLOCKSIZE` values under similar conditions to find the optimal network block size.

Completion of a `WRITE` statement, even when not buffered, does not guarantee that the data has been written to disk; the operating system may delay the actual disk write. To ensure that the data has been written to disk, you can call the `FLUSH` routine. Because this can cause a significant slowdown of the application, `FLUSH` should be used only when absolutely needed.

Linux* Only: When writing records, be aware that I/O records are written to unified buffer cache (UBC) system buffers. To request that I/O records be written from program buffers to the UBC system buffers, use the `FLUSH` library routine. Be aware that calling `FLUSH` also discards read-ahead data in user buffer. For more information, see `FLUSH` in the *Intel® Fortran Libraries Reference*.

Specify RECL

The sum of the record length (`RECL` specifier in an `OPEN` statement) and its overhead is a multiple or divisor of the blocksize, which is device-specific. For example, if the `BLOCKSIZE` is 8192 then `RECL` might be 24576 (a multiple of 3) or 1024 (a divisor of 8).

The `RECL` value should fill blocks as close to capacity as possible (but not over capacity). Such values allow efficient moves, with each operation moving as much data as possible; the least amount of space in the block is wasted. Avoid using values larger than the block capacity, because they create very inefficient moves for the excess data only slightly filling a block (allocating extra memory for the buffer and writing partial blocks are inefficient).

The `RECL` value unit for formatted files is always 1-byte units. For unformatted files, the `RECL` unit is 4-byte units, unless you specify the `-assume byterecl` (Linux) or `/assume:byterecl` (Windows) option for the `ASSUME` specifier to request 1-byte units.

Use the Optimal Record Type

Unless a certain record type is needed for portability reasons, choose the most efficient type, as follows:

- For sequential files of a consistent record size, the fixed-length record type gives the best performance.

- For sequential unformatted files when records are not fixed in size, the variable-length record type gives the best performance, particularly for `BACKSPACE` operations.
- For sequential formatted files when records are not fixed in size, the `Stream_LF` record type gives the best performance.

Reading from a Redirected Standard Input File

Due to certain precautions that the Fortran run-time system takes to ensure the integrity of standard input, reads can be very slow when standard input is redirected from a file. For example, when you use a command such as `myprogram.exe < myinput.data>`, the data is read using the `READ(*)` or `READ(5)` statement, and performance is degraded. To avoid this problem, do one of the following:

- Explicitly open the file using the `OPEN` statement. For example:
`OPEN(5, STATUS='OLD', FILE='myinput.dat')`
- Use an environment variable to specify the input file.

To take advantage of these methods, be sure your program does not rely on sharing the standard input file.

For more information on Intel Fortran data files and I/O, see "Files, Devices, and I/O" in the User's Guide; on `OPEN` statement specifiers and defaults, see "Open Statement" in the *Intel® Fortran Language Reference*.

Improving Run-time Efficiency

Use these source-coding guidelines to improve run-time performance. The amount of improvement in run-time performance is related to the number of times a statement is executed. For example, improving an arithmetic expression executed within a loop many times has the potential to improve performance, more than improving a similar expression executed once outside a loop.

Avoid Small Integer and Small Logical Data Items

Avoid using integer or logical data less than 32 bits. Accessing a 16-bit (or 8-bit) data type can make data access less efficient, especially on Itanium®-based systems.

To minimize data storage and memory cache misses with arrays, use 32-bit data rather than 64-bit data, unless you require the greater numeric range of 8-byte integers or the greater range and precision of double precision floating-point numbers.

Avoid Mixed Data Type Arithmetic Expressions

Avoid mixing integer and floating-point (`REAL`) data in the same computation. Expressing all numbers in a floating-point arithmetic expression (assignment statement) as floating-point values eliminates the need to convert data between fixed and floating-point

formats. Expressing all numbers in an integer arithmetic expression as integer values also achieves this. This improves run-time performance.

For example, assuming that `I` and `J` are both `INTEGER` variables, expressing a constant number (2.) as an integer value (2) eliminates the need to convert the data. The following examples demonstrate inefficient and efficient code.

Example: Inefficient Code

```
INTEGER I, J
I = J / 2.
```

Example: Efficient Code

```
INTEGER I, J
I = J / 2
```

You can use different sizes of the same general data type in an expression with minimal or no effect on run-time performance. For example, using `REAL`, `DOUBLE PRECISION`, and `COMPLEX` floating-point numbers in the same floating-point arithmetic expression has minimal or no effect on run-time performance. However, this practice of mixing different sizes of the same general data type in an expression can lead to unexpected results due to operations being performed in a lower precision than desired.

Use Efficient Data Types

In cases where more than one data type can be used for a variable, consider selecting the data types based on the following hierarchy, listed from most to least efficient:

- Integer (see above example)
- Single-precision real, expressed explicitly as `REAL`, `REAL (KIND=4)`, or `REAL*4`
- Double-precision real, expressed explicitly as `DOUBLE PRECISION`, `REAL (KIND=8)`, or `REAL*8`
- Extended-precision real, expressed explicitly as `REAL (KIND=16)` or `REAL*16`

However, keep in mind that in an arithmetic expression, you should avoid mixing integer and floating-point (`REAL`) data (see example in the previous subsection).

Avoid Using Slow Arithmetic Operators

Before you modify source code to avoid slow arithmetic operators, be aware that optimizations convert many slow arithmetic operators to faster arithmetic operators. For example, the compiler optimizes the expression `H=J**2` to be `H=J*J`.

Consider also whether replacing a slow arithmetic operator with a faster arithmetic operator will change the accuracy of the results or impact the maintainability (readability) of the source code.

Replacing slow arithmetic operators with faster ones should be reserved for critical code areas. The following list shows the Intel Fortran arithmetic operators, from fastest to slowest:

1. Addition (+), Subtraction (-), and Floating-point multiplication (*)
2. Integer multiplication (*)
3. Division (/)
4. Exponentiation (**)

Avoid Using EQUIVALENCE Statements

Avoid using `EQUIVALENCE` statements. `EQUIVALENCE` statements can:

- Force unaligned data or cause data to span natural boundaries.
- Prevent certain optimizations, including:
 - Global data analysis under certain conditions; see the `-O2` (Linux*) or `/O2` (Windows*) option in Setting Optimizations.
 - Implied-`DO` loop collapsing when the control variable is contained in an `EQUIVALENCE` statement

Use Statement Functions and Internal Subprograms

Whenever the Intel compiler has access to the use and definition of a subprogram during compilation, it may choose to inline the subprogram. Using statement functions and internal subprograms maximizes the number of subprogram references that will be inlined, especially when multiple source files are compiled together at optimization level `-O3` (Linux) or `/O3` (Windows).

For more information, see Efficient Compilation.

Code DO Loops for Efficiency

Minimize the arithmetic operations and other operations in a `DO` loop whenever possible. Moving unnecessary operations outside the loop will improve performance (for example, when the intermediate nonvarying values within the loop are not needed).

For more Information on loop optimizations, see *Pipelining for Itanium®-based Applications and Loop Unrolling*; for information on coding Intel statements, see the *Intel® Fortran Language Reference*.

Using Intrinsics for Itanium®-based Systems

Intel® Fortran supports all standard Fortran intrinsic procedures and provides Intel-specific intrinsic procedures to extend the language functionality. These intrinsic procedures are provided in the following libraries:

Platform	Library
Linux*	libintrins.a
Windows*	libintrins.lib

For more details on these intrinsic procedures, see the *Intel® Fortran Language Reference*.

This topic provides examples of the Intel-extended intrinsics that are helpful in developing efficient applications.

CACHESIZE Intrinsic (Itanium®-based Compiler)

The intrinsic `CACHESIZE(n)` is used only with the Intel® Itanium®-based compiler. `CACHESIZE(n)` returns the size in kilobytes of the cache at level *n*; one (1) represents the first level cache. Zero (0) is returned for a nonexistent cache level.

Use this intrinsic in any situation you would like to tailor algorithms for the cache hierarchy on the target processor. For example, an application may query the cache size and use the result to select block sizes in algorithms that operate on matrices.

Example

```
subroutine foo (level)
integer level
if (cachesize(level) > threshold) then
  call big_bar()
else
  call small_bar()
end if
end subroutine
```

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
- Optimizing the Compilation Process
- 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, 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"> • Enables global optimization. • Disables intrinsic recognition and intrinsics inlining. • On Itanium®-based systems, it disables software pipelining, loop unrolling, and global code scheduling. <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"> • -O1 compiler option
/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"> Enables optimizations for speed, including global code scheduling, software pipelining, predication, speculation, and data prefetch. <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"> -O2 compiler option
/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"> 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. -xP and -axP are the only valid values on Mac OS* systems. <p>Itanium®-based systems:</p> <ul style="list-style-type: none"> Enables optimizations for technical computing applications (loop-intensive code): loop optimizations and data prefetch.

		<p>For more information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-O3</code> compiler option
<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"> • Linux: <code>-ipo</code>, <code>-O3</code>, and <code>-static</code> • Windows: <code>/O3</code> and <code>/Qipo</code> <p>IA-32 and Intel® EM64T systems:</p> <ul style="list-style-type: none"> • Linux: <code>-ipo</code>, <code>-O3</code>, <code>-no-prec-div</code>, <code>-static</code>, and <code>-xP</code> • Windows: <code>/O3</code>, <code>/Qipo</code>, <code>/Qprec-div-</code>, and <code>/QxP</code> <p>Note</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 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"> • <code>-fast</code> compiler option

Restricting Optimization

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

Windows	Linux	Effect
/Od	-O0	<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"> Sets option <code>-fp</code> and option <code>-fmath-errno</code>. <p>Windows:</p> <ul style="list-style-type: none"> 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) <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <code>-O0</code> compiler option
/Zi, /Z7	-g	<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"> <code>-g</code> compiler option
/fltconsistency	-fltconsistency	<p>Enables improved floating-point consistency, but it might slightly reduce execution speed. The option limits floating-point optimizations and maintains declared precision.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <code>-fltconsistency</code> compiler option
No equivalent	<code>-fmath-errno</code> , <code>-fno-math-errno</code>	Instructs the compiler to assume that the program tests <code>errno</code> after calls to math library functions.

		<ul style="list-style-type: none"> • <code>-fmath-errno</code> compiler option
--	--	---

For more information on ways to restrict optimization, see [Using Qoption Specifiers](#).

Diagnostic Options

Windows	Linux	Effect
<code>/Qsox</code>	<code>-sox</code>	<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"> • <code>-sox</code> compiler option

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
<code>/G{5 6 7}</code>	<code>-mtune=<target cpu></code>	<p>IA-32 and Intel® EM64T processors.</p> <p>See Options for IA-32 Processors in Targeting a Processor.</p> <p>Note</p> <p>Mac OS*: This option is not supported.</p>
<code>/G{1 2}</code>	<code>-mtune=<target cpu></code>	<p>Itanium® processors.</p> <p>See Options for Itanium® processors in Targeting a Processor.</p>
<code>/Qx{K W N B P T}</code> <code>/Qax{K W N B P T}</code>	<code>-x{K W N B P T}</code> <code>-ax{K W N B P T}</code>	<p>Target applications to run on the specific processor-based systems and still gear your code to any processors that took advantage of</p>

		<p>the following options:</p> <ul style="list-style-type: none"> Linux: <code>-x{K W N}</code> or <code>-ax{K W N}</code> Windows: <code>/Qx{N}</code> or <code>/Qax{N}</code> <p>See Processor-specific Optimization and Automatic Processor-specific Optimization.</p>
--	--	--

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>	Intel® Pentium® and Pentium® with MMX™ technology processor
/G6	<code>-mtune=pentiumpro</code>	Intel® Pentium® Pro, Pentium® II and Pentium® III processors
/G7	<code>-mtune=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

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.f` 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>ifort -mtune=pentium4 prog.f90</code>
Windows	<code>ifort /G7 prog.f90</code>

Options for Itanium® Processors

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

Windows*	Linux*	Optimizes applications for...
/G1	-tpp1 -mtune=itanium	Intel® Itanium® processors
/G2	-tpp2 -mtune=itanium2	Default. Intel® Itanium® 2 processors
/G2-p9000	-mtune=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` 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>ifort -mtune=itanium2 prog.f</code>
Windows	<code>ifort /G2 prog.f</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>ifort -xN prog.f90</code>
Windows	<code>ifort /QxN prog.f90</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>ifort -axNB prog.f90</code>
Windows	<code>ifort /QaxNB prog.f90</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>ifort -xP prog.f90 -o foo.exe</code>
Windows	<code>ifort /QxP prog.f90 /exe:foo.exe</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.

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, the flags can be set manually by calling the following Intel Fortran intrinsic:

Example

```
RESULT = FOR_SET_FPE (FOR_M_ABRUPT_UND)
```

Optimizing the Compilation Process Overview

This section describes the Intel® compiler options that can optimize the compilation process. By default, the compiler converts source code directly to an executable file. Appropriate options enable you not only to control the process and obtain desired output file produced by the compiler, but also make the compilation itself more efficient.

A group of options monitors the outcome of Intel compiler-generated code without interfering with the way your program runs. These options control some computation aspects, such as allocating the stack memory, setting or modifying variable settings, and defining the use of some registers.

The options in this section provide you with the following capabilities of efficient compilation:

- Compiling efficiently
- Using default compiler optimizations
- Automatic allocation of variables and stacks
- Aligning data
- Converting little-endian to big-endian
- Symbol visibility attribute options

Efficient Compilation

Efficient compilation contributes to performance improvement. Before you analyze your program for performance improvement, and improve program performance, you should think of efficient compilation itself.

Based on the analysis of your application, you can decide which compiler optimizations and command-line options can improve the run-time performance of your application.

Efficient Compilation Techniques

The efficient compilation techniques can be used during the earlier stages and later stages of program development. During the earlier stages of program development, you can use incremental compilation without optimization. For example:

Platform	Examples
Linux*	<pre>ifort -c -g -O0 sub2.f90 ifort -c -g -O0 sub3.f90 ifort -o main -g -O0 main.f90 sub2.o sub3.o</pre>
Windows*	<pre>ifort /c /Zi /Od sub2.f90 ifort /c /Zi /Od sub3.f90 ifort /exe:main.exe /Zi /Od main.f90 sub2.obj sub3.obj</pre>

The above commands turn off all compiler default optimizations, for example, `-O2` (Linux) or `/O2` (Windows), with `-O0` (Linux) or `/Od` (Windows). You can use the `-g` (Linux) or `/Zi` or `/debug:full` (Windows) option to generate symbolic debugging information and line numbers in the object code for all routines in the program for use by a source-level debugger. The `main` file created in the third command above contains symbolic debugging information as well.

During the later stages of program development, you should specify multiple source files together and use an optimization level of at least `-O2` (Linux) or `/O2` (Windows) to allow more optimizations to occur. For instance, the following command compiles all three source files together using the default level of optimization:

Platform	Examples
Linux	<code>ifort -o main main.f90 sub2.f90 sub3.f90</code>
Windows	<code>ifort /exe:main.exe main.f90 sub2.f90 sub3.f90</code>

Compiling multiple source files lets the compiler examine more code for possible optimizations, which results in:

- Inlining more procedures
- More complete data flow analysis
- Reducing the number of external references to be resolved during linking

For very large programs, compiling all source files together may not be practical. In such instances, consider compiling source files containing related routines together using multiple `ifort` commands, rather than compiling source files individually.

Options That Increase Run-time Performance

The table below lists the options that can directly improve run-time performance. Most of these options do not affect the accuracy of the results, while others improve run-time performance but can change some numeric results. The Intel compiler performs some optimizations by default unless you turn them off by corresponding command-line options.

Additional optimizations can be enabled or disabled using command options.

Windows	Linux	Description
<code>/align: keyword</code>	<code>-align keyword</code>	Analyzes and reorders memory layout for variables and arrays. Controls whether padding bytes are added between data items within common blocks, derived-type data, and record structures to make the data items naturally aligned.
No equivalent	<code>-pg</code>	Requests profiling information, which you can use to identify those parts of your program where improving

		source code efficiency would most likely improve run-time performance. After you modify the appropriate source code, recompile the program and test the run-time performance.
/Ob2 + /Ot	No equivalent	Inlines procedures that will improve run-time performance with a likely significant increase in program size.
/Ob2 + /Os	No equivalent	Inlines procedures that will improve run-time performance without a significant increase in program size.
/Qax	-ax	Optimizes application performance for specific processors. Regardless of which argument you choose, the application is optimized to use all the benefits of that processor with the resulting binary file capable of being run on any Intel IA-32 processor. On Mac OS* systems, P is the only valid value for this option.
/Qopenmp	-openmp	Enables the parallelizer to generate multithreaded code based on the OpenMP* directives.
/Qparallel	-parallel	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.
/Qunroll[n] /unroll:n	-unrolln	Specifies the number of times a loop is unrolled (<i>n</i>) when specified with optimization level -O3 (Linux) or /O3 (Windows) or higher.

Options That Decrease Run-time Performance

The table below lists options that can slow down run-time performance. Some applications that require floating-point exception handling or rounding might need to use the `-fpen` (Linux) or `/fpen` (Windows) option. Other applications might need to use the `-assume dummy_aliases` or `-vms` (Linux) or `/assume:dummy_aliases` or `/Qvms` (Windows) options for compatibility reasons.

Options that can slow down the run-time performance are primarily for troubleshooting or debugging purposes. The following table lists the options that can slow down run-time performance.

Windows	Linux	Description
/assume: <i>dummy_aliases</i>	-assume <i>dummy_aliases</i>	Forces the compiler to assume that dummy (formal) arguments to procedures share memory locations with other dummy arguments or with variables shared through use association, host association, or common block use. These program semantics slow performance, so you should specify <i>dummy_aliases</i> only for the called subprograms that depend on such

		aliases.
/c	-c	If you use this option when compiling multiple source files, also specify <code>/object:outputfile</code> to compile many source files together into one object file.
/check:bounds	-check bounds	Generates extra code for array bounds checking at run time.
/fpe:3	-fpe3	Using this option enables certain types of floating-point exception handling, which can be expensive.
/Qsave	-save	Forces the local variables to retain their values from the last invocation terminated. This may change the output of your program for floating-point values as it forces operations to be carried out in memory rather than in registers, which in turn causes more frequent rounding of your results.
/Qvms	-vms	Controls certain VMS-related run-time defaults, including alignment. If you specify this option, you may need to also specify the <code>-align records</code> (Linux) or <code>/align:records</code> (Windows) option (for the ALIGN option) to obtain optimal run-time performance.
/debug:full	-g	<p>Generate extra symbol table information in the object file. Specifying these options also reduces the default level of optimization to no optimization.</p> <p>Note</p> <p>These options only slow your program when no optimization level is specified. For example, specifying <code>-g -O2</code> (Linux) or <code>/debug:full /O2</code> (Windows) allows the code runs close to the speed that would result if the option were not specified.</p>

Stacks: Automatic Allocation and Checking

The options in this group enable you to control the computation of stacks and variables in the compiler generated code.

Automatic Allocation of Variables

-automatic (Linux*) and /automatic (Windows*)

These options specify that locally declared variables are allocated to the run-time stack rather than static storage. If variables defined in a procedure do not have the `SAVE` or `ALLOCATABLE` attribute, they are allocated to the stack. It does not affect variables that appear in an `EQUIVALENCE` or `SAVE` statement, or those that are in `COMMON`.

`-automatic` (Linux) or `/automatic` (Windows) may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across routine calls should appear in a `SAVE` statement.

Note

Linux: If you specify `-recursive` or `-openmp`, the default is `-auto`.

Windows: The Windows NT* system imposes a performance penalty for addressing a stack frame that is too large. This penalty may be incurred with `/Qauto` because arrays are allocated on the stack along with scalars.

-auto-scalar (Linux*) or /Qauto-scalar (Windows*)

These options cause allocation of local scalar variables of intrinsic type `INTEGER`, `REAL`, `COMPLEX`, or `LOGICAL` to the stack. This option does not affect variables that appear in an `EQUIVALENCE` or `SAVE` statement, or those that are in `COMMON`.

The `-auto-scalar` (Linux) or `/Qauto-scalar` (Windows) option may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across subroutine calls should appear in a `SAVE` statement. This option is similar to `-auto` (Linux) and `/Qauto` (Windows) which causes all local variables to be allocated on the stack. The difference is that `-auto-scalar` (Linux) or `/Qauto-scalar` (Windows) allocates only scalar variables of the stated above intrinsic types to the stack.

Note

Windows: Windows NT* imposes a performance penalty for addressing a stack frame that is too large. This penalty may be incurred with `/Qauto` because arrays are allocated on the stack along with scalars. However, with `-auto-scalar` (Linux) or `/Qauto-scalar` (Windows), you would have to have more than 32K bytes of local scalar variables before you incurred the performance penalty.

`-auto-scalar` (Linux) or `/Qauto-scalar` (Windows) enables the compiler to make better choices about which variables should be kept in registers during program execution.

-save, -zero[-] (Linux*) or /Qsave, /Qzero[-] (Windows*)

The `-save` (Linux) or `/Qsave` (Windows) option is opposite of `-auto` (Linux) or `/Qauto` (Windows). The `-save` (Linux) or `/Qsave` (Windows) option saves all variables in static allocation except local variables within a recursive routine.

If a routine is invoked more than once, this option forces the local variables to retain their values from the last invocation. The `-save` (Linux) or `/Qsave` (Windows) option ensures that the final results on the exit of the routine is saved on memory and can be reused at

the next occurrence of that routine. This may cause some performance degradation as it causes more frequent rounding of the results.

Note

Linux: `-save` is the same as `-noauto`.

Windows: `/Qsave` is the same as `/save`, and `/noautomatic`.

When the compiler optimizes the code, the results are stored in registers.

The `-zero[-]` (Linux) or `/Qzero[-]` (Windows) option initializes to zero all local scalar variables of intrinsic type `INTEGER`, `REAL`, `COMPLEX`, or `LOGICAL`, which are saved and not initialized yet. Used in conjunction with `SAVE`.

Summary

There are three options for allocating variables: `-save` (Linux) or `/Qsave` (Windows), `-auto` (Linux) or `/Qauto` (Windows) and `-auto-scalar` (Linux) or `/Qauto-scalar` (Windows). Only one of these three can be specified.

The correlation among them is as follows:

- `-save` (Linux) or `/Qsave` (Windows) disables `-auto` (Linux) or `/Qauto` (Windows), sets `-noauto` (Linux) or `/noautomatic` (Windows), and allocates all variables not marked `AUTOMATIC` to static memory.
- `-auto` (Linux) or `/Qauto` (Windows) disables `-save` (Linux) or `/Qsave` (Windows), sets `-nosave` (Linux) or `/automatic` (Windows) and allocates all variables, scalars and arrays of all types, not marked `SAVE` to the stack.
- `-auto-scalar` (Linux) or `/Qauto-scalar` (Windows) makes local scalars of intrinsic types `INTEGER`, `REAL`, `COMPLEX`, and `LOGICAL` automatic. Additionally, this is the default. There is no `-noauto-scalar` (Linux) or `/Qauto-scalar-` (Windows); however, `-recursive` or `-openmp` (Linux) or `/recursive` or `/Openmp` (Windows) disables `-auto-scalar` (Linux) or `/Qauto-scalar` (Windows) and makes `-auto` (Linux) or `/Qauto` (Windows) the default.

Checking the Floating-point Stack State

When an application calls a function that returns a floating-point value, the returned floating-point value is supposed to be on the top of the floating-point stack. If the return value is not used, the compiler must pop the value off of the floating-point stack in order to keep the floating-point stack in correct state.

If the application calls a function, either without defining or incorrectly defining the function's prototype, the compiler does not know whether the function must return a floating-point value, and the return value is not popped off of the floating-point stack if it is not used. This can cause the floating-point stack to overflow.

The overflow of the stack results in two undesirable situations:

- A NaN value gets involved in the floating-point calculations
- The program results become unpredictable; the point where the program starts making errors can be arbitrarily far away from the point of the actual error.

For IA-32 and Intel® EM64T systems, the `-fpstkchk` (Linux) or `/Qfpstkchk` (Windows) option checks whether a program makes a correct call to a function that should return a floating-point value. If an incorrect call is detected, the option places a code that marks the incorrect call in the program. The `-fpstkchk` (Linux) or `/Qfpstkchk` (Windows) option marks the incorrect call and makes it easy to find the error.

Note

This option causes significant code generation after every function/subroutine call to insure a proper state of a floating-point stack and slows down compilation. It is meant only as a debugging aid for finding floating point stack underflow/overflow problems, which can be otherwise hard to find.

Checking and Setting Space

The following options perform checking and setting space for stacks (these options are supported on Windows only):

- The `/Ge` option enables stack-checking for all functions.
- The `/Gsn` option checks by default the stack space allocated for functions with more than 4KB.
- The `/Fn` option sets the stack reserve amount for the program. The `/Fn` option passes `/stack:n` to the linker.

Aliases

The `-common-args` (Linux) or `/Qcommon-args` (Windows) option assumes that the by-reference subprogram arguments may have aliases of one another.

It is recommended that you use the `-assume dummy_aliases` (Linux) or `/assume:dummy_aliases` (Windows) option instead of the `common-args` option.

For more information about using the preferred option, see the following topic:

- `-assume` compiler option

Preventing CRAY* Pointer Aliasing

Option `-safe-cray-ptr` (Linux) or `/Qsafe-cray-ptr` (Windows) specifies that the CRAY* pointers do not alias with other variables. Consider the following example.

Example
<pre>pointer (pb, b) pb = getstorage()</pre>

```
do i = 1, n
  b(i) = a(i) + 1
enddo
```

When the option is not specified, the compiler assumes that `b` and `a` are aliased. To prevent such an assumption, specify this option, and the compiler will treat `b(i)` and `a(i)` as independent of each other.

However, if the variables are intended to be aliased with CRAY pointers, using the `-safe-cray-ptr` (Linux) or `/Qsafe-cray-ptr` (Windows) option produces incorrect result. For the code example below, the option should not be used.

Example

```
pb = loc(a(2))
do i=1, n
  b(i) = a(i) +1
enddo
```

Cross Platform

The `-ansi-alias` (Linux) or `/Qansi-alias` (Windows) option enables or disables the function of the compiler to assume that the program adheres to the ANSI Fortran type aliasability rules.

For example, an object of type real cannot be accessed as an integer. You should see the ANSI standard for the complete set of rules.

The option directs the compiler to assume the following:

- Arrays are not accessed out of arrays' bounds.
- Pointers are not cast to non-pointer types and vice-versa.
- References to objects of two different scalar types cannot alias. For example, an object of type integer cannot alias with an object of type real or an object of type real cannot alias with an object of type double precision.

If your program satisfies the above conditions, setting this option will help the compiler better optimize the program. However, if your program may not satisfy one of the above conditions, the option must be disabled, as it can lead the compiler to generate incorrect code.

For more information, see the following topic:

- `-ansi-alias` compiler option

Little-endian-to-Big-endian Conversion (IA-32)

The Intel® compiler can write unformatted sequential files in big-endian format and can also read files produced in big-endian format by using the little-endian-to-big-endian conversion feature.

Both on IA-32-based processors and on Itanium®-based processors, Intel Fortran handles internal data in little-endian format. The little-endian-to-big-endian conversion feature is intended for Fortran unformatted input/output operations in unformatted sequential files. The feature enables the following:

- Processing of the files developed on processors that accept big-endian data format
- Producing big-endian files for such processors on little-endian systems.

The little-endian-to-big-endian conversion is accomplished by the following operations:

- The `WRITE` operation converts little-endian format to big-endian format.
- The `READ` operation converts big-endian format to little-endian format.

The feature enables the conversion of variables and arrays (or array subscripts) of basic data types. Derived data types are not supported.

Little-to-Big Endian Conversion Environment Variable

In order to use the little-endian-to-big-endian conversion feature, specify the numbers of the units to be used for conversion purposes by setting the `F_UFMTENDIAN` environment variable. Then, the `READ/WRITE` statements that use these unit numbers, will perform relevant conversions. Other `READ/WRITE` statements will work in the usual way.

In the general case, the variable consists of two parts divided by a semicolon. No spaces are allowed inside the `F_UFMTENDIAN` value. The variable has the following syntax:

Example	
<code>F_UFMTENDIAN=MODE</code>	<code>[MODE;] EXCEPTION</code>

where the following conditions are true:

Conditions	
<code>MODE</code>	<code>= big little</code>
<code>EXCEPTION</code>	<code>= big:ULIST little:ULIST ULIST</code>
<code>ULIST</code>	<code>= U ULIST,U</code>
<code>U</code>	<code>= decimal decimal -decimal</code>

and the following conditions apply:

- `MODE` defines current format of data, represented in the files; it can be omitted.

The keyword `little` means that the data have little endian format and will not be converted. This keyword is a default.

The keyword `big` means that the data have big endian format and will be converted. This keyword may be omitted together with the colon.

- `EXCEPTION` is intended to define the list of exclusions for `MODE`; it can be omitted. `EXCEPTION` keyword (`little` or `big`) defines data format in the files that are connected to the units from the `EXCEPTION` list. This value overrides `MODE` value for the units listed.
- Each list member `u` is a simple unit number or a number of units. The number of list members is limited to 64.
`decimal` is a non-negative decimal number less than 2^{32} .

Converted data should have basic data types, or arrays of basic data types. Derived data types are disabled.

Command lines for variable setting with different shells:

Shell	Command Line
Sh	<code>export F_UFMTENDIAN=MODE;EXCEPTION</code>
Csh	<code>setenv F_UFMTENDIAN MODE;EXCEPTION</code>

Note

Environment variable value should be enclosed in quotes if semicolon is present.

Another Possible Environment Variable Setting

The environment variable can also have the following syntax:

Example
<code>F_UFMTENDIAN=u[,u] . . .</code>

Command lines for the variable setting with different shells:

Shell	Command Line
Sh	<code>export F_UFMTENDIAN=u[,u] . . .</code>
Csh	<code>setenv F_UFMTENDIAN u[,u] . . .</code>

See error messages that may be issued during the little-endian to big-endian conversion. They are all fatal. You should contact Intel if such errors occur.

Usage Examples

The following usage examples illustrate the concepts detailed above.

Example 1

```
F_UFMTENDIAN=big
```

All input/output operations perform conversion from big-endian to little-endian on READ and from little-endian to big-endian on WRITE.

Example 2

```
F_UFMTENDIAN="little;big:10,20"
or
F_UFMTENDIAN=big:10,20
or
F_UFMTENDIAN=10,20
```

In this case, only on unit numbers 10 and 20 the input/output operations perform big-little endian conversion.

Example 3

```
F_UFMTENDIAN="big;little:8"
```

In this case, on unit number 8 no conversion operation occurs. On all other units, the input/output operations perform big-little endian conversion.

Example 4

```
4. F_UFMTENDIAN=10-20
```

Define 10, 11, 12...19, 20 units for conversion purposes; on these units, the input/output operations perform big-little endian conversion.

Assume you set `F_UFMTENDIAN=10,100` and run the following program.

Example 5

```
integer*4    cc4
integer*8    cc8
integer*4    c4
integer*8    c8
c4 = 456
c8 = 789

C  prepare a little endian representation of data
open(11,file='lit.tmp',form='unformatted')
write(11) c8
write(11) c4
close(11)

C  prepare a big endian representation of data
open(10,file='big.tmp',form='unformatted')
write(10) c8
write(10) c4
close(10)

C  read big endian data and operate with them on
```



```

C  little endian machine.

open(100,file='big.tmp',form='unformatted')
read(100) cc8
read(100) cc4

C    Any operation with data, which have been read

C
. . .
close(100)
stop
end

```

You can compare `lit.tmp` and `big.tmp` files to see the difference of the byte order in these files.

Linux* Systems Only

On Linux* systems you can use the `od` utility to compare the files.

Example output
<pre> > od -t x4 lit.tmp 00000000 00000008 00000315 00000000 00000008 00000020 00000004 000001c8 00000004 00000034 > od -t x4 big.tmp 00000000 08000000 00000000 15030000 08000000 00000020 04000000 c8010000 04000000 00000034 </pre>

You can see that the byte order is different in these files. If `info` and `od` are installed on your Linux system, enter `info od` at the prompt to get more information about the utility.

Alignment Options

These options control how the Intel® compiler align data items. Refer to the Compiler Options for more information on using these alignment-related options.

Windows*	Linux*	Description
/align:recnbyte	-align recnbyte	Specifies the alignment constraint for structures on n-byte boundaries. For more information, see the following topic: <ul style="list-style-type: none"> -align compiler option
/align:keyword, /Qpad	-align keyword, -pad	These front-end options changes alignment of variables in a common block.

		For more information, see the following topic: <ul style="list-style-type: none"> • <code>-align</code> compiler option
<code>/Qsfaalign</code> IA-32 Only	No equivalent	This option aligns stack for functions.

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

```
ifort -fvisibility=protected a.f
```

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

Examples

<ul style="list-style-type: none">-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.

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.

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>ifort -ipo -c a.f b.f c.f</code>
Windows*	<code>ifort /Qipo /c a.f b.f c.f</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>ifort -oapp a.o b.o c.o</code>
Windows	<code>ifort /exe:app 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 `ifort` 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>ifort -ipo -oapp a.f b.f c.f</code>
Windows	<code>ifort /Qipo /Feapp a.f b.f c.f</code>

The `ifort` 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 `/exe` (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"> <code>-ipo</code> compiler option
<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"> <code>-ipo-separate</code> compiler option

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>ifort a.o b.o c.o -ipo-separate -ipo-c</code>
Windows	<code>ifort 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>ifort a.o b.o c.o -ipo-separate -ipo-c -o appl.o</code>
Windows	<code>ifort 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 `/exe` (Windows) option to specify a different name. For example:

Platform	Example Command
Linux	<code>ifort -ipo-c -ofilename a.f b.f c.f</code>
Windows	<code>ifort /Qipo-c /exe:filename a.f b.f c.f</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 `/exe` (Windows) option to specify a different name. For example:

Platform	Example Command
Linux	<code>ifort -ipo-S -ofilename a.f b.f c.f</code>
Windows	<code>ifort /Qipo-S /exe:filename a.f b.f c.f</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 `/exe` (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
<p>Invokes the compiler to perform IPO if objects containing <code>IR</code> are found. Invokes <code>GCC ld</code> to link the application.</p> <p>The command-line syntax for <code>xild</code> is the same as that of the GCC linker:</p> <pre>xild [<options>] <LINK_commandline></pre> <p>where:</p> <ul style="list-style-type: none"> <code><options></code> (optional) may include any GCC linker options or options supported only by <code>xild</code>. <code><LINK_commandline></code> is the linker command line containing a set of valid arguments to the <code>ld</code>. <p>To create <code>app</code> using IPO, use the option <code>-ofile</code> as shown in the following example:</p> <pre>xild -oapp a.o b.o c.o</pre> <p>The linker calls the compiler to perform IPO for objects containing <code>IR</code> and creates a new list of object(s) to be linked. The linker then calls <code>ld</code> to link the object files that are</p>

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
<code>/qipo_fa[{file dir/}]</code>	<code>-qipo_fa[file.s]</code>	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

		<p>place the file.</p> <p>The default listing name is depends on the platform:</p> <ul style="list-style-type: none"> Linux: <code>ipo_out.s</code> Windows: <code>ipo_out.asm</code>
<code>/qipo_fo[{file dir/}]</code>	<code>-qipo_fo[file.o]</code>	<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"> Linux: <code>ipo_out.o</code> Windows: <code>ipo_out.obj</code>
<code>/qipo_fas</code>	No equivalent	Add source lines to assembly listing.
<code>/qipo_fac</code>	<code>-ipo-fcode-asm</code>	Adds code bytes to the assembly listing.
<code>/qipo_facs</code>	No equivalent	Add code bytes and source lines to assembly listing.
No equivalent	<code>-ipo-fsource-asm</code>	Adds high-level source code to the assembly listing.
<code>/qv</code>	No equivalent	Display version information.
No equivalent	<code>-ipo-fverbose-asm</code> , <code>-ipo-fnoverbose-asm</code>	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`. This happens transparently when the same `ifort xild` (Linux*) or `xilink` (Windows*) invocation is used for both the link-time compilation and the final link.

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.

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,f,-ip_ninl_max_stats=n</code>
Windows	<code>/Qoption,f,-ip_ninl_max_stats=n</code>

where n is a new value.

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 and Preemption (Linux*)

Function preemption means the code that implements that function at run-time is replaced by different code. When a function is preempted, the new version of this function is executed rather than the old version. Preemption can be used to replace an erroneous or inferior version of a function with a correct or improved version.

The compiler assumes that when `-ip` (Linux) or `/Qip` (Windows) is specified, any function with symbol visibility attribute `EXTERN` can be preempted, and cannot be inlined; however, by default the compiler assigns the visibility attribute `PROTECTED`, which prevents preemption and permits inlining.

See [Symbol Visibility Attribute Options](#).

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 Fortran (`f`) 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"> • <code>-Ob</code> compiler option
<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. For more information, see the following topic: <ul style="list-style-type: none"> • <code>-ip-no-inlining</code> compiler option
<code>/Qip-no-pinlining</code>	<code>-ip-no-pinlining</code>	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

		<p>specified.</p> <p>Note</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"> • <code>-ip-no-pinlining</code> compiler option
<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>Note</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"> • <code>-inline-debug-info</code> compiler option

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
<code>/Qinline-forceinline</code>	<code>-inline-forceinline</code>	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

		<p>presence of directives, like <code>DEC\$ ATTRIBUTES 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"> • <code>-inline-forceinline</code> compiler option
<code>/Qinline-min-size</code>	<code>-inline-min-size</code>	<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"> • <code>-inline-min-size</code> compiler option
<code>/Qinline-max-size</code>	<code>-inline-max-size</code>	<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"> • <code>-inline-max-size</code> compiler option
<code>/Qinline-max-total-size</code>	<code>-inline-max-total-size</code>	<p>Limits the expanded size of inlined functions.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-inline-max-total-size</code> compiler option
<code>/Qinline-max-per-routine</code>	<code>-inline-max-per-routine</code>	<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"> • <code>-inline-max-per-routine</code> compiler option
<code>/Qinline-max-per-compile</code>	<code>-inline-max-per-compile</code>	<p>Defines the number of times a called functions can be 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</p>

		<p>compilation unit.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none">• <code>-inline-max-per-compile</code> compiler option
--	--	--

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.

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.

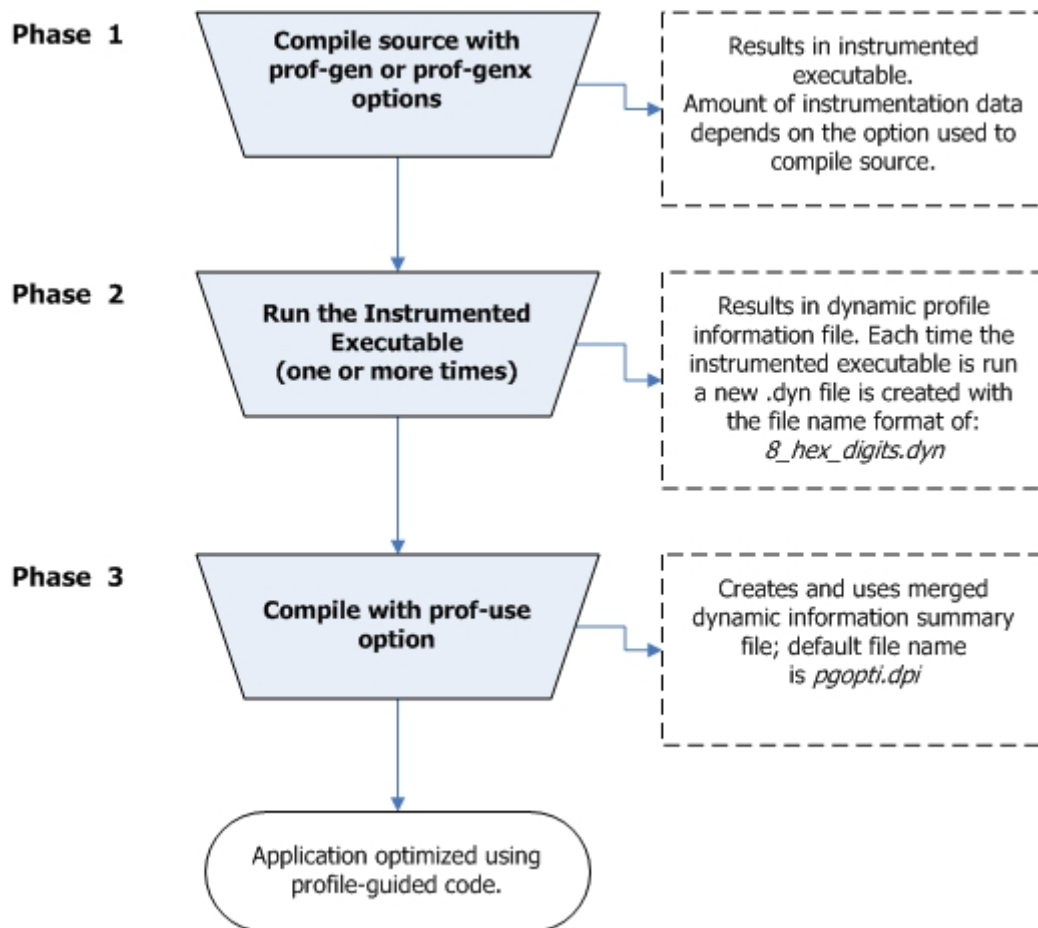
Using PGO often enables the compiler to make better decisions about function inlining, thereby increasing the effectiveness of IPO.

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.

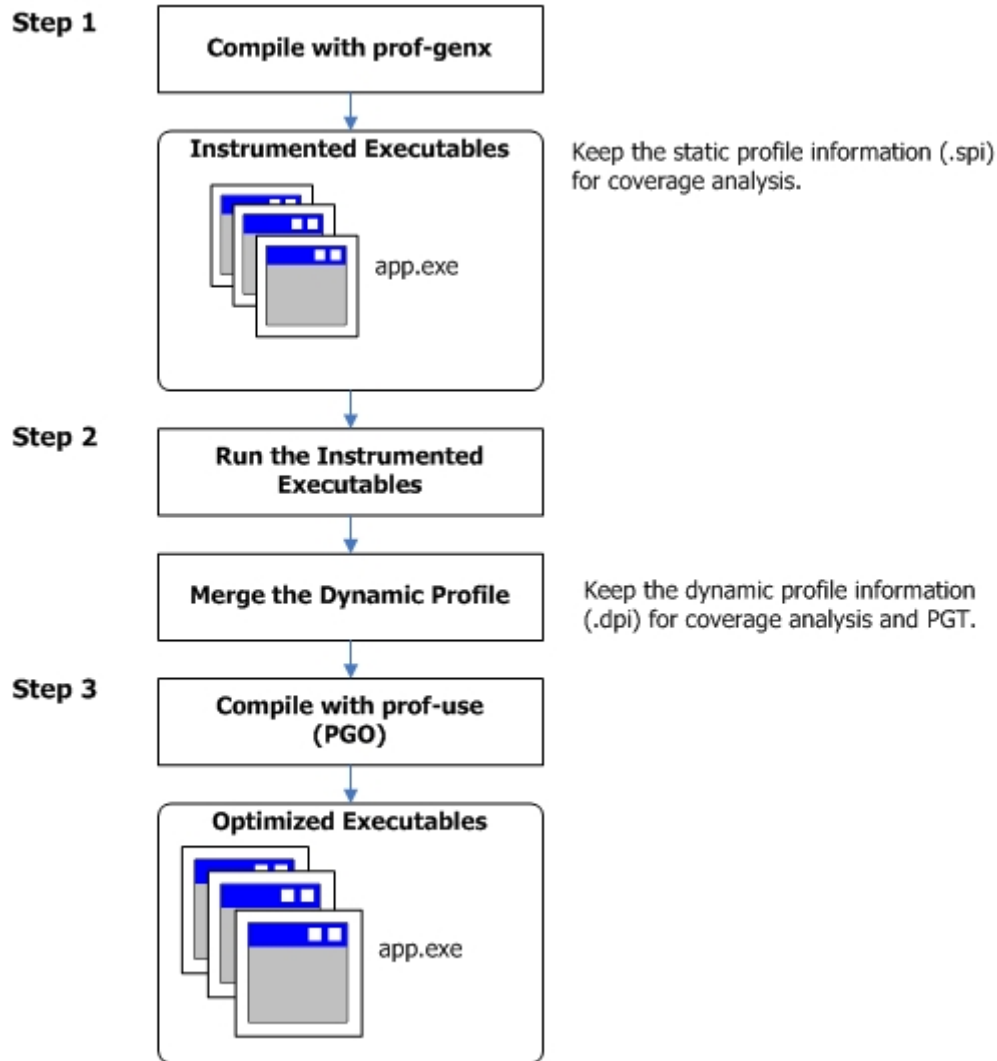
A key factor in deciding whether or not to use PGO lies in knowing what sections of your code are the most heavily used. If the data set provided to your program is very consistent and it displays similar behavior on every execution, then PGO can probably help optimize your program execution.

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.

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>ifort -prof-gen -prof-dir/usr/profdata -c a1.f a2.f a3.f</code> <code>ifort -oal a1.o a2.o a3.o</code>
Windows	<code>ifort /Qprof-gen /Qprof-dir:c:\profdata /c a1.f a2.f a3.f</code> <code>ifort 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>a1</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>ifort -prof-use -prof-dir/usr/profdata -ipo a1.f a2.f a3.f</code>
Windows	<code>ifort /Qprof-use /Qprof-dirc:\profdata /Qipo a1.f a2.f a3.f</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
<code>/Qprof-gen</code>	<code>-prof-gen</code>	Instruments the program for profiling to get the execution count of each basic block. The option is used in phase 1 of

/Qprof-genx	-prof-genx	<p>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 <i>x</i> qualifier, the option gathers extra source information, which enables using specific tools, like the proforder 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"> • -prof-gen and -prof-genx compiler options
/Qprof-use	-prof-use	<p>Instruct the compiler to produce a profile-optimized executable and merges available dynamic-information (.dyn) files into a pgopti.dpi file. This option is used in phase 3 of PGO (generating a profile-optimized executable).</p> <p>Note</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 pgopti.dpi file.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> • -prof-use compiler option
/Qprof-format-32	-prof-format-32	<p>Using 32-bit Counters: 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 in the .dyn and .dpi files.</p> <p>Note</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 .dyn and .dpi files is incompatible with the format used in the current compilation, the compiler issues the following message:</p> <p>Error: xxx.dyn has old or incompatible file</p>

		<p>format - delete file and redo instrumentation compilation/execution.</p> <p>The 64-bit format for counters and pointers in .dyn and .dpi 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"> • -prof-format-32 compiler option
/Qfnsplit-	-fnsplit-	<p>Disables function splitting. Function splitting is enabled by -prof-use (Linux*) or /Qprof-use (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"> • 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. • 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. <p>Note</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>IA-32 systems:</p> <ul style="list-style-type: none"> • The option completely disables function splitting, placing all the code in one section. <p>Itanium®-based systems:</p> <ul style="list-style-type: none"> • 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.

		<p>See an example of using PGO.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-fnsplit</code> compiler option
--	--	---

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
/Qprof-dir	-prof-dir	<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"> • <code>-prof-dir</code> compiler option
/Qprof-file	-prof-file	<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"> • <code>-prof-file</code> compiler option
/Qprof-gen-sampling	-prof-gen-sampling	<p>IA-32 Only. Prepares application executables for hardware profiling (sampling) and causes the compiler to generate source code mapping information.</p> <p>Note</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"> • <code>-prof-gen-sampling</code> compiler option
/Qssp	-ssp	<p>IA-32 Only. Enables Software-based Speculative Pre-computation (SSP) optimization.</p> <p>Note</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"> • <code>-ssp</code> compiler option
--	--	--

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 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 -ipo (/Qipo)	Function Ordering with proforder
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 Fortran program that consists of the following files:

- file1.f
- file2.f

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>ifort -o myprog -prof-genx -prof-dir ./profdata file1.f file2.f</code>
Windows	<code>ifort /exe:myprog /Qprof-genx /Qprof-dir c:\profdata file1.f file2.f</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 make take some time 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>ifort -o myprog -prof-dir ./profdata file1.f file2.f -Xlinker -ORDER:@myprog.txt</code>
Windows	<code>ifort /exe:myprog /Qprof-dir c:\profdata file1.f file2.f /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.

Note

The Profile IGS functions are written in the C language. Fortran applications must call C functions.

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> file, if the <code>.dyn</code> files are newer than an existing <code>pgopti.dpi</code> file. 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.

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

```
void _PGOPTI_Prof_Dump_All(void);
```

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

```
! Selectively collect profile information
! for the portion of the application
! involved in processing input data.
input_data = get_input_data()
do while (input_data)
  call _PGOPTI_Prof_Reset()
  call process_data(input_data)
  call _PGOPTI_Prof_Dump_All()
  input_data = get_input_data()
end do
end program
```

Dumping Profile Data

This discussion provides an example of how to call the C PGO API routines from Fortran.

As part of the instrumented execution phase of Profile-guided Optimization, the instrumented program writes profile data to the dynamic information file (.dyn file).

The file is written after the instrumented program returns normally from `PROGRAM()` or calls the standard exit function. Programs that do not terminate normally, can use the `_PGOPTI_Prof_Dump_All` function. During the instrumentation compilation, using the -

prof-gen (Linux*) or /Qprof-gen (Windows*) option, you can add a call to this function to your program using a strategy similar to the one shown below:

Example

```
interface
  subroutine PGOPTI_Prof_Dump_All()
  !DEC$attributes c,alias:'PGOPTI_Prof_Dump_All'::PGOPTI_Prof_Dump_All
  end subroutine
  subroutine PGOPTI_Prof_Reset()
  !DEC$attributes c,alias:'PGOPTI_Prof_Reset'::PGOPTI_Prof_Reset
  end subroutine
end interface
call PGOPTI_Prof_Dump_All()
```

Caution

You must remove the call or comment it out prior to the feedback compilation with -prof-use (Linux) or /Qprof-use (Windows).

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
<code>void _PGOPTI_Set_Interval_Prof_Dump(int interval);</code>

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.

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 provides 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.
-fcolor <i>color</i>	#ffc000	Specifies the HTML color name for code of the uncovered functions.
-pcolor <i>color</i>	#f0f0f0	Specifies the HTML color name or code of the partially covered code.
-ccolor <i>color</i>	#ffffff	Specifies the HTML color name or code of the covered code.
-ucolor <i>color</i>	#ffffff	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

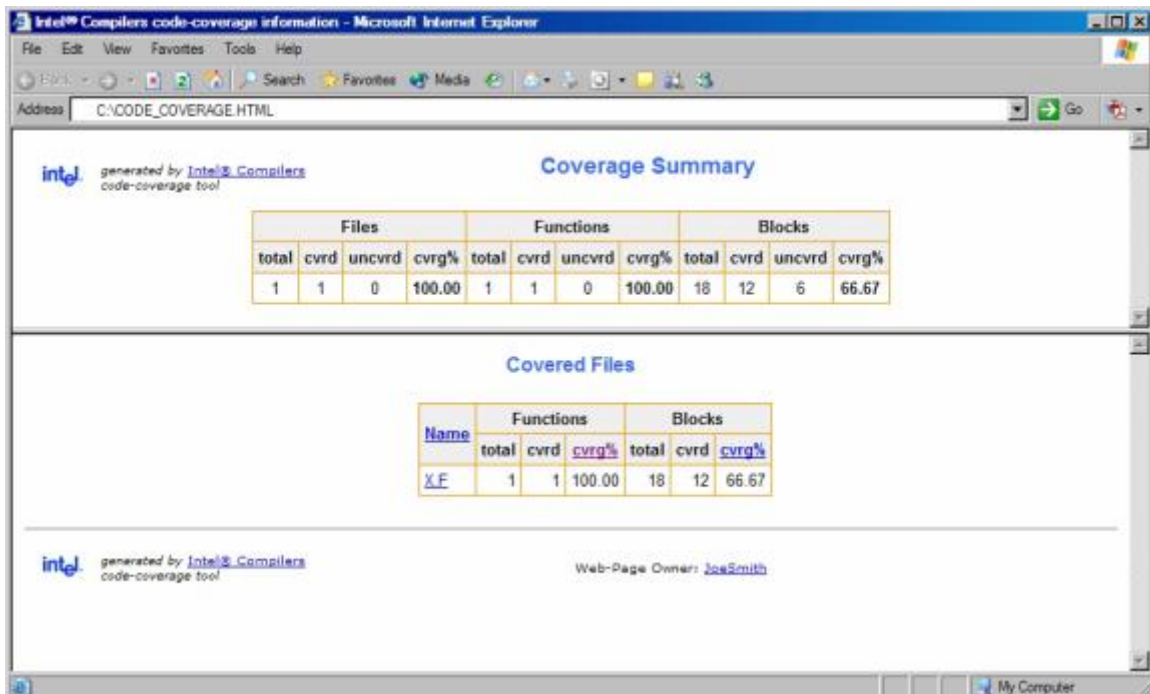
Top Level Coverage

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.

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:

Example
<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 `mod1.f90`, 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.f90`, then only those modules with the name `mod1.f90` 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> $\quad \quad \quad \wedge \quad 10 \quad (1/2)$
12	<code>PRINT N</code> $\quad \quad \quad \wedge \quad 7$

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

```
codecov -prj Project Name -dpi customer.dpi -ref appTests.dpi
```

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.f` and `source2.f`) and files in directories where the name contains the string "source" (like `source/file1.f` and `source2\file2.f`) 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.f` and `source2.f`) and directories where the name contains "source" (like `source/file1.f` and `source2\file2.f`), and exclude any individual files where the name contains "skip" (like `skipthis1.f` and `skipthis2.f`) or directories where the name contains "skip" (like `skipthese1\debug1.f` and `skipthese2\debug2.f`).

Excluding Coverage at the Line and Function Level

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

Sample code

```
print*, "ERROR: n = ", n ! INFEASIBLE
print*, "          n2 = ", n2 ! 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

```
integer n, n2
n = 123
n2 = n*n
if (n2 .lt. 0) then
! /* BINF */
  print*, "ERROR: n = ", n
  print*, "          n2 = ", n2
! // EINF
else if (n2 .eq. 0) then
  print*, "zero: n = ", n, " n2 = ", n2
else
  print*, "positive: n = ", n, " n2 = ", n2
endif
end
```

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

In the above example, we note that function f0, which is defined in file sample.f, 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.F"				
"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:

Example: quick export of block data to XML
<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:

XML-formatted report example
<pre> <PROJECT name = "test1"> <MODULE name = "D:\SAMPLE.F"> <FUNCTION name="f0" freq="2"> ... <BLOCK line="11" col="2"> <INSTANCE id="1" freq="1"> </INSTANCE> </BLOCK> <BLOCK line="12" col="3"> <INSTANCE id="1" freq="2"> </INSTANCE> <INSTANCE id="2" freq="1"> </INSTANCE> </BLOCK> </FUNCTION> </MODULE> </PROJECT> </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.f`. 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 `-txtbcvrgfull` 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:

Test-priorization tool Syntax

<code>tselect -dpi_list file</code>

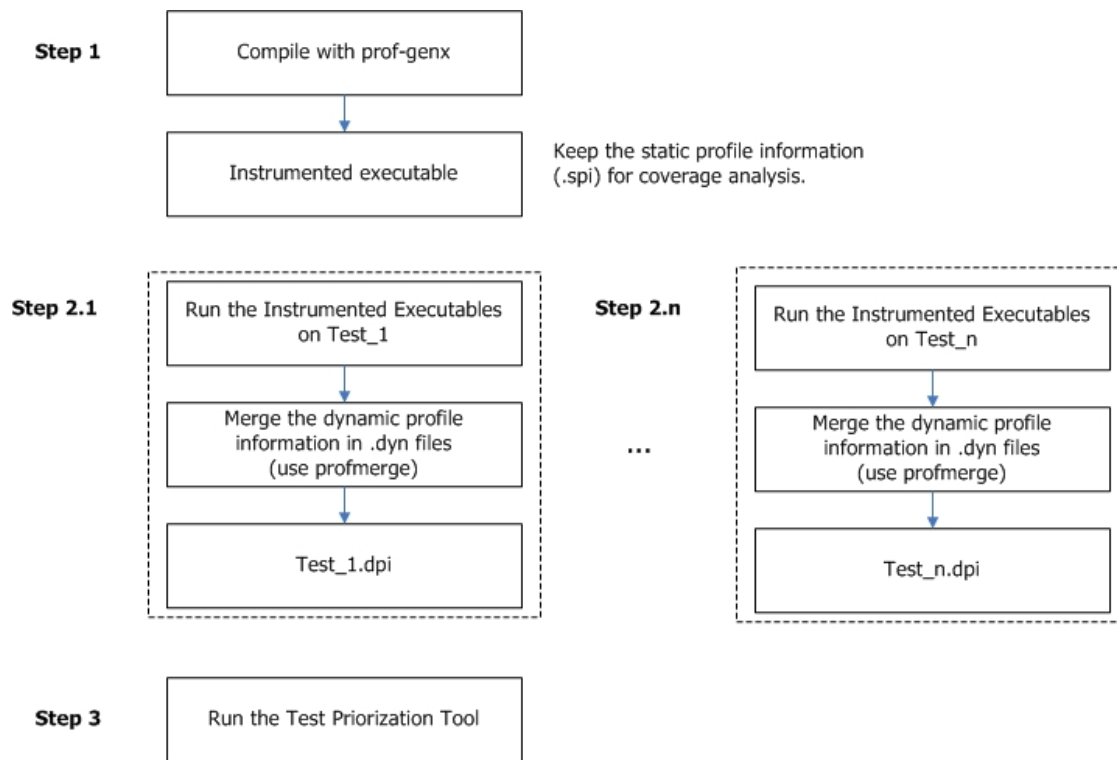
where `-dpi_list` is a required tool option that sets the path to the list file containing the list of the all `.dpi` files. All other commands are optional.

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
<code>-help</code>		Prints options supported by the tool.
<code>-spi file</code>	<code>pgopti.spi</code>	Specifies the file name of the static profile information file (.SPI).

<code>-dpi_list file</code>		Specifies the file name of the file that contains the name of the dynamic profile information (<code>.dpi</code>) files. Each line of the file must contain one <code>.dpi</code> name optionally followed by its execution time. The name must uniquely identify the test.
<code>-prof_dpi dir</code>		Specifies the path name of the output report file.
<code>-o file</code>		Specifies the file name of the output report file.
<code>-comp file</code>		Specifies the file name that contains the list of files of interest.
<code>-cutoff value</code>		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.
<code>-nototal</code>		Instructs the tool to ignore the pre-compute total coverage process.
<code>-mintime</code>		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.
<code>-verbose</code>		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:

Example
<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>ifort -prof-genx myApp.f90</code>
Windows	<code>ifort /Qprof-genx myApp.f90</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:

Example

```
rm prof-dir \*.dyn
```

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

Example

```
myApp < data1
```

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:

Example

```
profmerge -prof_dpi Test1.dpi
```

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:

Example

```
rm prof-dir \*.dyn
```

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:

Example

```
myApp < data2
```

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

Example

```
profmerge -prof_dpi Test2.dpi
```

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:

Example

```
rm prof-dir \*.dyn
```

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:

Example

```
myApp < data3
```

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

```
tselect -dpi list tests list -spi pgopti.spi -cutoff 85.00
```

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

```
profmerge [/nologo] [-prof_dir dir_name]
```

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
-help	List supported options.
-nologo	Disables version information.
-exclude_funcs <i>functions</i>	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.
-prof_dir <i>dir</i>	Specifies the directory from which to read .dyn and .dpi files
-prof_dpi <i>file</i>	Specifies the name of the .dpi file.
-prof_file <i>file</i>	Merges information from file matching: <i>dpi_file_and_dyn_tag</i>
-dump	Displays profile information.
-src_old <i>dir</i> -src new <i>dir</i>	Change the directory path stored within the .dpi.
-a <i>file1.dpi...fileN.dpi</i>	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 routine 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 dir] [-o file]</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 dir</code>	Specifies the directory where the <code>.spi</code> and <code>.dpi</code> file reside.
<code>-prof_file string</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 string.
<code>-prof_dpi file</code>	Specifies the name of the <code>.dpi</code> file.

<code>-prof_spi file</code>	Specifies the name of the .spi file.
<code>-o file</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"> 1. Install Intel® VTune™ Performance Analyzer 3.0 for Linux (or later). 2. Grant utility user to the appropriate access to the VTune™ Performance Analyzer Driver. <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"> 1. Install Intel® VTune™ Performance Analyzer 7.2 (or later). 2. Grant utility user to the appropriate access to the VTune™ Performance Analyzer Driver. <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"> 3. Specify a local disk as the default directory.

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>ifort -oMyApp -O2 -prof-gen-sampling source1.f source2.f</code>
Windows	<code>ifort /FeMyApp.exe /O2 /Qprof-gen-sampling source1.f source2.f</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>ifort -oMyApp -O2 -prof-use source1.f source2.f</code>
Windows	<code>ifort /FeMyApp.exe /O2 /Qprof-use source1.f source2.f</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

```
profrun -command [argument...] application
```

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
-help		Lists the supported tool options.
[sav]	10007	<p>An optional argument supported by the -branch, -dcache, -icache or -event 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 <i>sav</i> 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"> 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. 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.
-branch[:sav]	10007	Collect branch samples. A sample is taken after every interval, as defined by the <i>sav</i>

		<p>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	<p>Collect samples of misses in the Instruction cache. 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>-event:eventname[:sav]</code>	10007	<p>Collect information about valid VTune™ analyzer events; <code>eventname</code> specifies a specific event name. Use this command when <code>-branch</code>, <code>-dcache</code>, or <code>-icache</code> 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>
<code>-tb5:file</code>	<code>pgopti.tb5</code>	<p>Specifies the name of the <code>.tb5</code> 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 <code>profrun</code> completes unless <code>-tb5only</code> is also specified.</p>

		You might consider overriding this behavior and place the <code>.tb5</code> file on a disk with more available space.
<code>-tb5only</code>		Produces only the <code>.tb5</code> file. If this command is not specified the utility will reduce the data into a single <code>.hpi</code> file and delete the <code>.tb5</code> file.
<code>-wait[:time]</code>	600 seconds (10 minutes)	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. Disable the command by specifying a value of 0 (zero).
<code>-hpi:file</code>	<code>pgopti.hpi</code>	Specifies name of the <code>.hpi</code> file containing the profile information for the application. The file resides in the current directory.
<code>-executable:file</code>		Specifies the name of the executable being profiled. By default, this is the first token of the command.
<code>-bufsize:size</code>	65536 KB (64 MB)	Specifies the buffer size used in kilobytes (KB).
<code>-sampint:interval</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.f`, `a2.f` and `a3.f`, 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>ifort -prof-gen a1.f a2.f a3.f -o go</code>
Windows	<code>ifort /Qprof-gen a1.f a2.f a3.f /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>ifort -prof-gen-sampling -prof-use -O3 -ipo a1.f a2.f a3.f -o go</code>
Windows	<code>ifort /Qprof-gen-sampling /Qprof-use /O3 /Qipo a1.f a2.f a3.f /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>ifort -prof-use -ssp -O3 -ipo a1.f a2.f a3.f -o go</code>
Windows	<code>ifort /Qprof-use /Qssp /O3 /Qipo a1.f a2.f a3.f -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 Fortran. 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 `!DEC$ ivdep` to tell the compiler there is no dependency. You may also need the `-ivdep-parallel` (Linux) or `/Qivdep-parallel` (Windows) option to indicate there is no loop carried dependency.
- Use `!DEC$ swp` to enable software pipelining (useful for loop-sided control and unknown loop count).
- Use `!DEC$ loop count(n)` when needed.
- If cray pointers are used, use `-safe-cray-ptr` (Linux) or `/Qsafe-cray-ptr` (Windows) to indicate there is no aliasing.
- Use `!DEC$ distribute point` to split large loops (normally, this is automatically done).
- Check that the prefetch distance is correct. Use `CDEC$ 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 dependences 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` directive 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 `ivdep` directive to ensure there is no loop-carried dependency for the store into `a()`.

Example
<pre>!DEC\$ IVDEP do j=1,n a(b(j)) = a(b(j))+1 enddo</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>/Qunroll n</code> Synonym: <code>/unroll [:n]</code>	<code>-unroll n</code>	Specifies the maximum number of times you want to unroll a loop. The following examples unrolls a loop at most four times: <pre>ifort -unroll4 a.f (Linux)</pre> <pre>ifort /Qunroll4 a.f (Windows)</pre> <p>Note</p> <p>The Itanium® compiler currently recognizes only $n = 0$; 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>/Qunroll 0</code>	<code>-unroll 0</code>	Disables the loop unroller. To disable loop unrolling, specify n as 0. The following examples disables loop unrolling: <pre>ifort -unroll0 a.f (Linux)</pre> <pre>ifort /Qunroll0 a.f (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 coarse-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 .

Example

```
subroutine loop_independent_A (a,b,MAX)
  implicit none
  integer :: j, MAX, a(MAX), b(MAX)
  do j=0, MAX
    a(j) = b(j)
  end do
end subroutine loop_independent_A
subroutine loop_independent_B (a,b,MAX)
  implicit none
  integer :: j, MAX, a(MAX), b(MAX)
  do j=MAX, 0, -1
    a(j) = b(j)
  end do
end subroutine loop_independent_B
```

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

```
subroutine flow_dependence (A,MAX)
  implicit none
  integer :: J,MAX
  real :: A(MAX)
  do j=1, MAX
    A(J)=A(J-1)
  end do
end subroutine flow_dependence
```

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

```
subroutine time stepping loops (a,b,MAX)
  implicit none
  integer :: J,MAX
  real :: a(MAX), b(MAX)
  do j=1, MAX
    a(j) = a(j-1) + b(j)
  end do
end subroutine time stepping loops
```

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

```
subroutine anti dependence (A,MAX)
  implicit none
  integer :: J,MAX
  real :: a(MAX), b(MAX)
  do J=1, MAX
    A(J)=A(J+1)
  end do
end subroutine anti_dependence
```

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

```
subroutine output dependence (A,B,C,MAX)
  implicit none
  integer :: J,MAX
  real :: a(MAX), b(MAX), c(MAX)
  do J=1, MAX
    A(J)=B(J)
```

```

    A(J+1)=C(J)
  end do
end subroutine output_dependence

```

The above example is equivalent to the following lines for the first few iterations:

Sample interactions

```

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

```

subroutine reduction (sum,c,MAX)
  implicit none
  integer :: j,MAX
  real :: sum, c(MAX)
  do J=0, MAX
    sum = sum + c(j)
  end do
end subroutine reduction

```

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.

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). In addition to the `-prefetch` (Linux*) or `/Qprefetch` (Windows*) option, an intrinsic subroutine `mm_prefetch` and compiler directive `prefetch` are also available. See Prefetching Data for more information.

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 `/Od` (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"> • <code>-fp-model</code> compiler option
<code>/fltconsistency</code>	<code>-fltconsistency</code>	<p>Enables improved floating-point consistency and may slightly reduce execution speed. It limits floating-point optimizations and maintains declared</p>

		<p>precision.</p> <p>The option also disables inlining of math library functions.</p> <p>IA-32 and Intel EM64T:</p> <ul style="list-style-type: none"> • In general, the option maintains maximum precision not the declared precision. <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-fltconsistency</code> compiler option
<code>/Qprec</code>	<code>-mp1</code>	<p>Improves floating-point precision; this option has less impact to performance than the <code>-fltconsistency</code> (Linux) or <code>/fltconsistency</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>For more information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-mp1</code> compiler option
<code>/Qftz</code>	<code>-ftz</code>	<p>Flushes denormal results to zero when the application is in gradual underflow mode. Flushing the denormal values to zero with this option may improve overall application performance.</p> <p>Use this option if the denormal values are not critical to application behavior.</p> <p>Architectures:</p> <ul style="list-style-type: none"> • Linux: IA-32, Intel® EM64T, Itanium® • Windows: IA-32, Itanium® • Mac OS*: IA-32 <p>This option affects the result of abrupt underflow by</p>

		<p>setting the floating underflow to zero and allowing the execution to continue.</p> <p>IA-32 and Intel® EM64T:</p> <ul style="list-style-type: none"> • The compiler automatically sets the flush-to-zero mode in the SSE Control Register (MXCSR) when SSE instructions are enabled. • Use this option to flush x87 floating-point values to zero (0). This option can significantly degrade performance in x87 code since the generated code stream must be synchronized after each floating-point instruction to allow the operating system to do the necessary abrupt underflow corrections. There may be performance gains in code targeting SSE and SSE2. There are other considerations when using SSE instructions. Refer to the compiler option topic below for other SSE-specific behaviors. • Use this option on any source where flushing x87 denormal values to zero is desired. <p>Itanium®:</p> <ul style="list-style-type: none"> • 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. • Use this option only on source files containing main; this enables the FTZ mode. The initial thread, and any threads subsequently created by that process, will operate in FTZ mode. • 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. • This option instructs the compiler to treat denormal values used in a computation as zero (0) so no floating invalid exception
--	--	--

		<p>occurs.</p> <p>If this option produces undesirable results of the numerical behavior of your program, you can turn the FTZ mode off by using <code>-no-ftz</code> (Linux) or <code>/Qftz-</code> (Windows) in the command line while still benefiting from the <code>-O3</code> (Linux) or <code>/O3</code> (Windows) optimizations.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-ftz</code> compiler option
<code>/fpe</code>	<code>-fpe</code>	<p>Provides some control over the results of floating-point exception handling at run time for the main program.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-fpe</code> compiler option

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>	<p>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> <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"> • <code>-prec-div</code> compiler option
<code>/Qpc</code>	<code>-pc</code>	<p>Changes the floating point precision control when the <code>main()</code> function is compiled. The program that uses this option must use <code>main()</code> as its entry point, and the file containing <code>main()</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.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-pc</code> compiler option
<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 Fortran 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 Fortran semantics.</p> <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-rcd</code> compiler option
<code>/Qrct</code>	No equivalent	<p>Sets internal FPU rounding control to truncate. Use for rounding control.</p> <p>Use this option to compile the <code>main()</code> routine. The compiler will assume the rounding mode in the floating-point control word is truncate, so it will not change the rounding mode before integer conversions.</p> <p>For detailed information on the FPU control word, refer to <i>IA-32 Intel® Architecture Software Developer's Manual, Volume 1</i>:</p>

		<i>Basic Architecture</i> http://www.intel.com/design/pentium4/manuals/index_new.htm .
/Qfp-port	-fp-port	<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"> Linux: <code>-fp-port</code> is supported on both IA-32 and Intel® EM64T systems. Windows: <code>/Qfp-port</code> is supported on IA-32 only. <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <code>-fp-port</code> compiler option
/Qprec-sqrt	-prec-sqrt	<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"> <code>-prec-sqrt</code> compiler option

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
/QIPF-fma	-IPF-fma	Enables or disables the contraction of floating-point multiply and add/subtract operations into a single operation. Unless

		<p><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"> <code>ifort -fp-model strict -IPF-fma myprog.f</code> (Linux) <code>ifort /fp:strict /QIPF-fma myprog.f</code> (Windows) <p>For more information, see the following topic:</p> <ul style="list-style-type: none"> <code>-IPF-fma</code> compiler option
<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"> <code>-IPF-fp-speculation</code> compiler option
<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"> • <code>-IPF-fp-relaxed</code> compiler option
--	--	--

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 and Intel® EM64T compilers

The Intel® compiler automatically sets flush-to-zero mode in the SSE Control Register (MXCSR) when running on a processor that supports SSE instructions. SSE instructions are enabled by default in the Intel® EM64T compiler. Enable SSE instructions in the IA-32 compiler by using `-xK`, `-xW`, `-xN`, `-xB`, or `-xP` (Linux) or `/QxK`, `/QxW`, `/QxN`, `/QxB`, or `/QxP` (Windows). The MXCSR flush-to-zero setting only affects the behavior of SSE, SSE2, and SSE3 instructions. x87 floating-point instructions are not affected.

When SSE instructions are used, options `-no-ftz` (Linux and Mac OS*) and `/Qftz-` (Windows) are ignored. However, you can enable gradual underflow by calling a function in C that clears the FTZ and DAZ bits in the MXCSR or by using run-time function `for_set_fpe` to clear those bits. Be aware that denormal processing can significantly slow down computation.

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.

Use the `-ftz` (Linux) or `/Qftz` (Windows) option to flush x87 floating-point values to zero. It is necessary to use the option on the source containing PROGRAM and on any source where abrupt underflow is desired.

Note

Windows* Only: The `/Qftz` option is not supported for Intel® EM64T.

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 PROGRAM; the option turns on the Flush-to-Zero (FTZ) mode for the process started by PROGRAM. 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.

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.

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
<code>/Qopt-report</code>	<code>-opt-report</code>	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"> • <code>-opt-report</code> compiler option
<code>/Qopt-report-phase</code>	<code>-opt-report-phase</code>	<p>Specifies the optimization phase to use when generating reports. See Specifying Optimizations to Generate Reports (below) for information about supported phases.</p> <p>For additional information, see the following topic:</p> <ul style="list-style-type: none"> • <code>-opt-report-phase</code> compiler option
<code>/Qopt-report-file</code>	<code>-opt-report-file</code>	<p>Generates an optimization report and directs the report output to the 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"> • <code>-opt-report-file</code> compiler option
<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"> • <code>-opt-report-level</code> compiler option
<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</p>

		topic: <ul style="list-style-type: none"> • <code>-opt-report-routine</code> compiler option
--	--	---

Use syntax similar to the following to generate optimization reports.

Platform	Sample Syntax
Linux	<code>ifort -opt-report -opt-report-phase all myfile.f</code>
Windows	<code>ifort /Qopt-report /Qopt-report-phaseall myfile.f</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 Note 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"> • <code>ilo</code> and <code>pgo</code> • <code>hlo</code>: supported only if <code>-O3</code> (Linux) or <code>/O3</code> (Windows) option is specified. • <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). • <code>all</code>: 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.
Itanium®-based systems only	<ul style="list-style-type: none"> • <code>all</code>: 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 • <code>hlo</code> and <code>ipo</code>: if either a <code>hlo</code> or <code>ipo</code> 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. • <code>ecg</code>

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
<code>ipo_inl</code>	Interprocedural Optimizer, inline expansion of functions
<code>ipo_cp</code>	Interprocedural Optimizer, constant propagation
<code>hlo_unroll</code>	High-level Optimizer, loop unrolling
<code>hlo_prefetch</code>	High-level Optimizer, prefetching
<code>ecg_swp</code>	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>ifort -opt-report -opt-report-phase hlo -O3 a.f b.f</code>
Windows*	<code>ifort /Qopt-report /Qopt-report-phase hlo /O3 a.f b.f</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>ifort -ipo -opt-report -opt-report-phase ipo a.f b.f</code>
Windows*	<code>ifort /Qipo /Qopt-report /Qopt-report-phase ipo a.f b.f</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>ifort -c -opt-report -opt-report-phase ecg_swp swp.f90</code>
Windows*	<code>ifort /c /Qopt-report /Qopt-report-phase ecg_swp swp.f90</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. The sample reports is also shown below.

Example

```
!#define NUM 1024
subroutine multiply d(a,b,c,NUM)
  implicit none
  integer :: i,j,k,NUM
  real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
  NUM=1024
  do i=0,NUM
    do j=0,NUM
      do k=0,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
      end do
    end do
  end do
end subroutine multiply d
```

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 10 in  Z10multiply dPA1024 dS0 S0  in file
SWP report.f90
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 $N * \text{Scheduled II} + \text{number Stages}$ (Where N is 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.</p>
Resource II	Resource II implies what the Initiation Interval should be when considering the number of functional units available.
Recurrence II	<p>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.</p> <p>See Helping the Compiler for more information.</p>
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	<p>The loop-carried memory dependence edges means the compiler avoided WAR (Write After Read) dependency.</p> <p>Loop-carried memory dependence edges can indicate problems with memory aliasing. See Helping the Compiler.</p>

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.
Not enough rotating registers	Indicates that the loop carried values use the rotating registers. Distribute the loop.
Loop too large	Indicates that you should distribute the loop.
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>ifort -xW -vec-report3 matrix1.f > report.txt</code>
Windows	<code>ifort /QxW /Qvec-report3 matrix1.f > report.txt</code>

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
<pre>matmul.f(27) : (col. 9) remark: loop was not vectorized: not inner loop. matmul.f(28) : (col. 11) remark: LOOP WAS VECTORIZED. matmul.f(31) : (col. 9) remark: loop was not vectorized: not inner loop. matmul.f(32) : (col. 11) remark: LOOP WAS VECTORIZED. matmul.f(37) : (col. 10) remark: loop was not vectorized: not inner loop. matmul.f(38) : (col. 12) remark: loop was not vectorized: not inner loop. matmul.f(40) : (col. 14) remark: loop was not vectorized: vectorization possible but seems inefficient. matmul.f(46) : (col. 10) remark: loop was not vectorized: not inner loop. matmul.f(47) : (col. 12) remark: loop was not vectorized: contains unvectorizable statement at line 48.</pre>

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` directive; 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	<code>ifort -xK -vec-report3 file.f</code> <code>ifort -xK -ipo -vec-report3 file.f</code>
Windows	<code>ifort /QxK /Qvec-report3 file.f</code> <code>ifort /QxK /Qipo /Qvec-report3 file.f</code>

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.
- 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

```

subroutine foo(y)
  implicit none
  integer :: i
  real :: y(10)
  do i=2,10
    y(i) = y(i-1)+1
  end do
end subroutine foo
subroutine bar(y)
  implicit none
  integer :: i
  real :: y(10)
  do i=2,10
    y(i) = y(1)+i
  end do
end subroutine bar

```

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

```

function func(n)
  implicit none
  integer :: func, n
  func = n*n-1
end function func
subroutine unsupported_loop_structure(y,n)
  implicit none
  integer :: i,n, func
  real :: y(n)
  do i=0,func(n)
    y(i) = y(i) * 2.0
  end do
end subroutine unsupported_loop_structure

```

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 directive `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

```

subroutine howmany_close(x,y,n)
  implicit none
  integer :: i,n,withinborder

```

```
real :: x(n), y(n), dist
withinborder=0
do i=0,100
  dist=sqrt(x(i)*x(i) + y(i)*y(i))
  if (dist<5) withinborder= withinborder+1
end do
end subroutine howmany_close
```

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
Explicit	Parallelism programmed by the user
OpenMP* (thread-level parallelism) IA-32 and Itanium® architectures	Supported on: <ul style="list-style-type: none"> • IA-32, Intel EM64T, and Itanium-based multiprocessor systems and dual-core processors • Hyper-Threading Technology-enabled systems
Implicit	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"> IA-32, Intel EM64T, and Itanium-based multiprocessor systems and dual-core processors Hyper-Threading Technology-enabled systems
Auto-vectorization (instruction-level parallelism) of inner-most loops; IA-32 and Itanium architectures	Supported on: <ul style="list-style-type: none"> Pentium®, Pentium with MMX™ Technology, Pentium II, Pentium III, and Pentium 4 processors

Parallel Program Development

OpenMP

The Intel® compiler supports the OpenMP* Fortran 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. For example, in the code below, thread-level parallelism can be exploited in the outermost loop, while instruction-level parallelism can be exploited in the innermost loop.

Example	
DO I = 1, 100	! Execute groups of iterations in different threads (TLP)
DO J = 1, 32	! Execute in SIMD style with multimedia extension (ILP)
A(J,I) = A(J,I) + 1	
ENDDO	
ENDDO	

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"> -x
/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"> -ax
/Qvec-report	-vec-report	Controls the diagnostic messages from the vectorizer, see subsection that follows the table. See the following topic in Compiler Options: <ul style="list-style-type: none"> -vec-report

Auto-parallelization: IA-32 and Itanium® architectures

Windows*	Linux*	Description
/Qparallel	-parallel	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. Intel® Itanium®-based systems only: <ul style="list-style-type: none"> Implies <code>-opt-mem-bandwidth1</code> (Linux) or <code>/Qopt-mem-bandwidth1</code> (Windows). See the following topic in Compiler Options:

		<ul style="list-style-type: none"> • <code>-parallel</code>
<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"> • <code>-par-threshold</code>
<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"> • <code>-par-report</code>

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"> • Implies <code>-opt-mem-bandwidth1</code> (Linux) or <code>/Qopt-mem-bandwidth1</code> (Windows). <p>See the following topic in Compiler Options:</p> <ul style="list-style-type: none"> • <code>-openmp</code>
<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"> • <code>-openmp-report</code>
<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"> • <code>-openmp-stubs</code>

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 shows OpenMP directives within the code.

Example

```
!OMP$ PARALLEL PRIVATE(NUM), SHARED (X,A,B,C)
! Defines a parallel region
!OMP$ PARALLEL DO
! Specifies a parallel region that
! implicitly contains a single DO directive
DO I = 1, 1000
  NUM = FOO(B(i), C(I))
  X(I) = BAR(A(I), NUM)
! Assume FOO and BAR have no other effect
ENDDO
```

See examples of the auto-parallelization and auto-vectorization directives in the following topics.

Parallelization with OpenMP* Overview

The Intel® compiler supports the OpenMP* version 2.5 API specification. 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).

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.

In addition, the compiler provides Intel-specific extensions to the OpenMP Fortran 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 Fortran language specifications, see the OpenMP Fortran 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 in the form of the Fortran program comments. 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. See Programming with OpenMP.

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.

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 Fortran 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 OpenMP Fortran API, the `PARALLEL` and `END PARALLEL` directives define 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 `END 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> subroutine F ... !\$OMP parallel... ... call G ... subroutine G ... !\$OMP DO... ... </pre>

The `!$OMP DO` is an orphaned directive because the parallel region it will execute in is not lexically present in `G`.

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 Fortran version 2.5 specifications (<http://www.openmp.org/specs>).

Example 2: Pseudo Code of the Parallel Processing Model

PROGRAM MAIN	! Begin serial execution
...	! Only the master thread executes
!\$OMP PARALLEL	! Begin a Parallel construct, form a team
...	! This is Replicated Code where each team
	! member executes the same code
!\$OMP SECTIONS	! Begin a Worksharing construct
!\$OMP SECTION	! One unit of work
...	!
!\$OMP SECTION	! Another unit of work
...	!
!\$OMP END SECTIONS	! Wait until both units of work complete
...	! More Replicated Code
!\$OMP DO	! Begin a Worksharing construct,
DO	! each iteration is a unit of work
...	! Work is distributed among the team
END DO	!
!\$OMP END DO NOWAIT	! End of Worksharing construct, NOWAIT
	! is specified (threads need not wait
	! until all work is completed before
	! proceeding)
...	! More Replicated Code
!\$OMP END PARALLEL	! End of PARALLEL construct, disband team
	! and continue with serial execution
...	! Possibly more PARALLEL Constructs
END PROGRAM MAIN	! End serial execution

OpenMP* and Hyper-Threading Technology

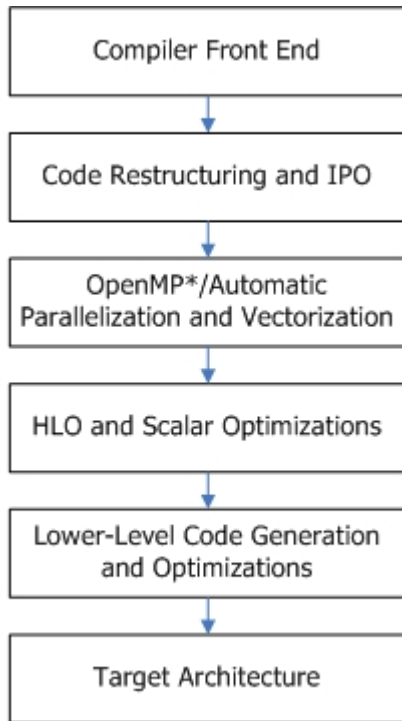
The compiler incorporates many well-known and advanced optimization techniques designed to leverage Intel® processor features for higher performance on IA-32- and Itanium®-based systems.

The Intel® compiler has a common intermediate representation for the supported languages, so that the OpenMP* directive-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 directives 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 directives. 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 directives to describe parallelism to the compiler. Using the supplied directives 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 directives, 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).

Programming with OpenMP*

The Intel® compiler accepts a Fortran program containing OpenMP* directives as input and produces a multithreaded version of the code. When the parallel program begins execution, a single thread exists. This thread is called the master thread. The master thread will continue to process serially until it encounters a parallel region.

Parallel Region

A parallel region is a block of code that must be executed by a team of threads in parallel. In the OpenMP Fortran API, a parallel construct is defined by placing OpenMP directive `PARALLEL` at the beginning and directive `END PARALLEL` at the end of the code segment. Code segments thus bounded can be executed in parallel.

A structured block of code is a collection of one or more executable statements with a single point of entry at the top and a single point of exit at the bottom.

The compiler supports worksharing and synchronization constructs. Each of these constructs consists of one or two specific OpenMP directives and sometimes the enclosed or following structured block of code. For complete definitions of constructs, see the OpenMP Fortran version 2.5 specifications (<http://www.openmp.org/specs>).

At the end of the parallel region, threads wait until all team members have arrived. The team is logically disbanded (but may be reused in the next parallel region), and the master thread continues serial execution until it encounters the next parallel region.

Worksharing Construct

A worksharing construct divides the execution of the enclosed code region among the members of the team created on entering the enclosing parallel region. When the master thread enters a parallel region, a team of threads is formed. Starting from the beginning of the parallel region, code is replicated (executed by all team members) until a worksharing construct is encountered. A worksharing construct divides the execution of the enclosed code among the members of the team that encounter it.

The OpenMP `SECTIONS` or `DO` constructs are defined as worksharing constructs because they distribute the enclosed work among the threads of the current team. A worksharing construct is only distributed if it is encountered during dynamic execution of a parallel region. If the worksharing construct occurs lexically inside of the parallel region, then it is always executed by distributing the work among the team members. If the worksharing construct is not lexically (explicitly) enclosed by a parallel region (that is, it is `orphaned`), then the worksharing construct will be distributed among the team members of the closest dynamically-enclosing parallel region, if one exists. Otherwise, it will be executed serially.

When a thread reaches the end of a worksharing construct, it may wait until all team members within that construct have completed their work. When all of the work defined by the worksharing construct is finished, the team exits the worksharing construct and continues executing the code that follows.

A combined parallel/worksharing construct denotes a parallel region that contains only one worksharing construct.

Parallel Processing Directive Groups

The parallel processing directives include the following groups:

Parallel Region Directives

- `PARALLEL` and `END PARALLEL`

Worksharing Construct Directives

- The `DO` and `END DO` directives specify parallel execution of loop iterations.
- The `SECTIONS` and `END SECTIONS` directives specify parallel execution for arbitrary blocks of sequential code. Each `SECTION` is executed once by a thread in the team.
- The `SINGLE` and `END SINGLE` directives define a section of code where exactly one thread is allowed to execute the code; threads not chosen to execute this section ignore the code.

Combined Parallel/Worksharing Construct Directives

The combined parallel/worksharing constructs provide an abbreviated way to specify a parallel region that contains a single worksharing construct. The combined parallel/worksharing constructs are:

- `PARALLEL DO` and `END PARALLEL DO`
- `PARALLEL SECTIONS` and `END PARALLEL SECTIONS`
- `WORKSHARE` and `PARALLEL WORKSHARE`

Synchronization and MASTER Directives

Synchronization is the interthread communication that ensures the consistency of shared data and coordinates parallel execution among threads. Shared data is consistent within a team of threads when all threads obtain the identical value when the data is accessed. A synchronization construct is used to insure this consistency of the shared data.

The OpenMP synchronization directives are `CRITICAL`, `ORDERED`, `ATOMIC`, `FLUSH`, and `BARRIER`.

Directive	Usage
<code>CRITICAL</code>	Within a parallel region or a worksharing construct only one thread at a time is allowed to execute the code within a <code>CRITICAL</code> construct.
<code>ORDERED</code>	Used in conjunction with a <code>DO</code> or <code>SECTIONS</code> construct to impose a serial order on the execution of a section of code.
<code>ATOMIC</code>	Used to update a memory location in an uninterruptable fashion.
<code>FLUSH</code>	Used to insure that all threads in a team have a consistent view of memory.
<code>BARRIER</code>	Forces all team members to gather at a particular point in code. Each team member that executes a <code>BARRIER</code> waits at the <code>BARRIER</code> until all of the team members have arrived. A <code>BARRIER</code> cannot be used within worksharing or other synchronization constructs due to the potential for deadlock.
<code>MASTER</code>	The directive is used to force execution by the master thread.

See the list of [OpenMP Directives and Clauses](#).

Data Sharing

Data sharing is specified at the start of a parallel region or worksharing construct by using the `SHARED` and `PRIVATE` clauses. All variables in the `SHARED` clause are shared among the members of a team. The application must do the following:

- Synchronize access to these variables. All variables in the `PRIVATE` clause are private to each team member. For the entire parallel region, assuming t team members, there are $t+1$ copies of all the variables in the `PRIVATE` clause: one global copy that is active outside parallel regions and a `PRIVATE` copy for each team member.
- Initialize `PRIVATE` variables at the start of a parallel region, unless the `FIRSTPRIVATE` clause is specified. In this case, the `PRIVATE` copy is initialized from the global copy at the start of the construct at which the `FIRSTPRIVATE` clause is specified.
- Update the global copy of a `PRIVATE` variable at the end of a parallel region. However, the `LASTPRIVATE` clause of a `DO` directive enables updating the global copy from the team member that executed serially the last iteration of the loop.

In addition to `SHARED` and `PRIVATE` variables, individual variables and entire common blocks can be privatized using the `THREADPRIVATE` directive.

Orphaned Directives

OpenMP contains a feature called orphaning that dramatically increases the expressiveness of parallel directives. Orphaning is a situation when directives related to a parallel region are not required to occur lexically within a single program unit.

Directives such as `CRITICAL`, `BARRIER`, `SECTIONS`, `SINGLE`, `MASTER`, and `DO` can occur by themselves in a program unit, dynamically "binding" to the enclosing parallel region at run time.

Orphaned directives enable parallelism to be inserted into existing code with a minimum of code restructuring. Orphaning can also improve performance by enabling a single parallel region to bind with multiple `DO` directives located within called subroutines. Consider the following code segment:

Example

```
subroutine phase1
  integer :: i
  !$OMP DO PRIVATE(i) SHARED(n)
  do i = 1, 200
    call some work(i)
  end do
  !$OMP END DO
end
subroutine phase2
  integer :: j
  !$OMP DO PRIVATE(j) SHARED(n)
  do j = 1, 100
    call more work(j)
  end do
  !$OMP END DO
end
program par
  !$OMP PARALLEL
  call phase1
  call phase2
  !$OMP END PARALLEL
end program par
```

Orphaned Directives Usage Rules

The following orphaned directives usage rules apply:

- An orphaned worksharing construct (`SECTIONS`, `SINGLE`, or `DO`) is executed by a team consisting of one thread, that is, serially.
- Any collective operation (worksharing construct or `BARRIER`) executed inside of a worksharing construct is illegal.
- It is illegal to execute a collective operation (worksharing construct or `BARRIER`) from within a synchronization region (`CRITICAL/ORDERED`).
- The opening and closing directives of a directive pair (for example, `DO` and `END DO`) must occur in a single block of the program.
- Private scoping of a variable can be specified at a worksharing construct. Shared scoping must be specified at the parallel region. For complete details, see the OpenMP Fortran version 2.5 specifications (<http://www.openmp.org/specs>).

Preparing Code for OpenMP Processing

The following are the major stages and steps of preparing your code for using OpenMP. Typically, the first two stages can be done on uniprocessor or multiprocessor systems; later stages are typically done on dual-core processor and multiprocessor systems.

Before Inserting OpenMP Directives

Before inserting any OpenMP parallel directives, verify that your code is safe for parallel execution by doing the following:

- Place local variables on the stack. This is the default behavior of the Intel Compiler when `-openmp` (Linux*) or `/Qopenmp` (Windows*) is specified.
- Use `-automatic` or `-auto-scalar` (Linux) or `/automatic` (Windows) to make the locals automatic. This is the default behavior of the Intel® compiler when `-openmp` (Linux) or `/Qopenmp` (Windows) is specified. Avoid using the `-save` (Linux) or `/save` (Windows) option, which inhibits stack allocation of local variables. By default, automatic local variables become shared across threads, so you may need to add synchronization code to ensure proper access by threads.

Analyze

The analysis includes the following major actions:

1. Profile the program to find out where it spends most of its time. This is the part of the program that benefits most from parallelization efforts. This stage can be accomplished using VTune™ Analyzer or basic PGO options.
2. Wherever the program contains nested loops, choose the outer-most loop, which has very few cross-iteration dependencies.

Restructure

To restructure your program for successful OpenMP implementation, you can perform some or all of the following actions:

- If a chosen loop is able to execute iterations in parallel, introduce a `PARALLEL DO` construct around this loop.
- Try to remove any cross-iteration dependencies by rewriting the algorithm.
- Synchronize the remaining cross-iteration dependencies by placing `CRITICAL` constructs around the uses and assignments to variables involved in the dependencies.
- List the variables that are present in the loop within appropriate `SHARED`, `PRIVATE`, `LASTPRIVATE`, `FIRSTPRIVATE`, or `REDUCTION` clauses.
- List the `DO` index of the parallel loop as `PRIVATE`. This step is optional.
- `COMMON` block elements must not be placed on the `PRIVATE` list if their global scope is to be preserved. The `THREADPRIVATE` directive can be used to privatize to each thread the `COMMON` block containing those variables with global scope.

`THREADPRIVATE` creates a copy of the `COMMON` block for each of the threads in the team.

- Any I/O in the parallel region should be synchronized.
- Identify more parallel loops and restructure them.
- If possible, merge adjacent `PARALLEL DO` constructs into a single parallel region containing multiple `DO` directives to reduce execution overhead.

Tune

The tuning process should include minimizing the sequential code in critical sections and load balancing by using the `SCHEDULE` clause or the `OMP_SCHEDULE` environment variable.

Note

This step is typically performed on dual-core processor and multiprocessor systems.

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>ifort -openmp input_file</code>
Windows	<code>ifort /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).

OpenMP Directive Format and Syntax

OpenMP directives use a specific format and syntax. The following syntax and example help illustrate how to use the directives with your source.

Syntax
<code><prefix> <directive> [<clause>[[,<clause>...]]</code>

where:

- `<prefix>`: Required for OpenMP. For fixed form source input, the prefix is `!$OMP` or `C$OMP`. For free form source input, the prefix is `!$OMP` only.
- `<directive>`: Required must immediately follow the prefix; for example: `!$OMP PARALLEL`
- `[<clause>]`: Must be specified if a directive uses one or more clauses
- `[,<]>`: Commas between the `<clause>`s are optional.

Since OpenMP directives begin with an exclamation point, the directives are interpreted as comments if you omit the `-openmp` (Linux) or `/Qopenmp` ((Windows) option.

The following example demonstrates one way of using an OpenMP* directive to parallelize loops within parallel regions.

Example
<pre>subroutine simple_omp(a, N) integer :: N, a(N) !\$OMP PARALLEL DO do i = 1, N a(i) = i*2 end do end subroutine simple_omp</pre>

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>ifort -openmp -c parallel.f90</code>
Windows	<code>ifort /Qopenmp /c parallel.f90</code>

The compiler might return a message similar to the following:

```
parallel.f90(20) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
```

Syntax for Parallel Regions in the Source Code

The OpenMP constructs defining a parallel region have one of the following syntax forms:

Example	
!\$OMP <directive> <structured block of code> !\$OMP END <directive>	
or	
!\$OMP <directive> <structured block of code>	
or	
!\$OMP <directive>	

where <directive> is the name of a particular OpenMP directive.

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 MASTER, SINGLE, CRITICAL, ORDERED, ATOMIC 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 Fortran 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 DO END PARALLEL DO	<p>A shortcut for a PARALLEL region that contains a single DO directive.</p> <p>Note</p> <p>The OpenMP PARALLEL DO or DO directive must be immediately followed by a DO statement (DO-stmt as defined by R818 of the ANSI Fortran standard). If you place another statement or an OpenMP directive between the PARALLEL DO or DO directive and the DO statement, the compiler issues a syntax error.</p>
PARALLEL SECTIONS END PARALLEL SECTIONS	Provides a shortcut form for specifying a parallel region containing a single SECTIONS construct.
MASTER END MASTER	Identifies a construct that specifies a structured block that is executed by only the master thread of the team.
CRITICAL [lock] END CRITICAL [lock]	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 lock 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 list argument consists of a comma-separated list of variables to be flushed.
ORDERED END ORDERED	The structured block following an ORDERED directive is executed in the order in which iterations would be executed in a sequential loop.
THREADPRIVATE (list)	Makes the named COMMON blocks or variables private to a thread. The list argument consists of a comma-separated list of COMMON blocks or variables.

OpenMP Clauses

Clause	Description
PRIVATE (<i>list</i>)	Declares variables in <i>list</i> to be PRIVATE to each thread in a team.
FIRSTPRIVATE (<i>list</i>)	Same as PRIVATE, but the copy of each variable in the <i>list</i> is initialized using the value of the original variable existing before the construct.
LASTPRIVATE (<i>list</i>)	Same as PRIVATE, but the original variables in <i>list</i> are updated using the values assigned to the corresponding PRIVATE variables in the last iteration in the DO construct loop or the last SECTION construct.
COPYPRIVATE (<i>list</i>)	Uses private variables in <i>list</i> to broadcast values, or pointers to shared objects, from one member of a team to the other members at the end of a single construct.
NOWAIT	Specifies that threads need not wait at the end of worksharing constructs until they have completed execution. The threads may proceed past the end of the worksharing constructs as soon as there is no more work available for them to execute.
SHARED (<i>list</i>)	Shares variables in <i>list</i> among all the threads in a team.
DEFAULT (<i>mode</i>)	Determines the default data-scope attributes of variables not explicitly specified by another clause. Possible values for <i>mode</i> are PRIVATE, SHARED, or NONE.
REDUCTION (<i>{operator intrinsic}:list</i>)	Performs a reduction on variables that appear in <i>list</i> with the operator <i>operator</i> or the intrinsic procedure name <i>intrinsic</i> ; <i>operator</i> is one of the following: +, *, .AND., .OR., .EQV., .NEQV.; <i>intrinsic</i> refers to one of the following: MAX, MIN, IAND, IOR, or IEOR.
ORDERED END ORDERED	Used in conjunction with a DO or SECTIONS construct to impose a serial order on the execution of a section of code. If ORDERED constructs are contained in the dynamic extent of the DO construct, the ordered clause must be present on the DO directive.
IF (<i>scalar_logical_expression</i>)	The enclosed parallel region is executed in parallel only if the <i>scalar_logical_expression</i> evaluates to .TRUE., otherwise the parallel region is serialized.
NUM_THREADS (<i>scalar_integer_expression</i>)	Requests the number of threads specified by <i>scalar_integer_expression</i> for the parallel region.
SCHEDULE (<i>type[, chunk]</i>)	Specifies how iterations of the DO construct are divided among the threads of the team. Possible values for the <i>type</i> argument are STATIC, DYNAMIC, GUIDED, and RUNTIME. The optional <i>chunk</i> argument must be a positive scalar integer expression.
COPYIN (<i>list</i>)	Specifies that the master thread's data values be

	copied to the <code>THREADPRIVATE</code> 's copies of the common blocks or variables specified in <i>list</i> at the beginning of the parallel region.
--	--

Directives and Clauses Cross-reference

See Data Scope Attribute Clauses Overview.

Directive	Use these Clauses
PARALLEL END PARALLEL	<ul style="list-style-type: none"> • COPYIN • DEFAULT • PRIVATE • FIRSTPRIVATE • REDUCTION • SHARED
DO END DO	<ul style="list-style-type: none"> • PRIVATE • FIRSTPRIVATE • LASTPRIVATE • REDUCTION • SCHEDULE
SECTIONS END SECTIONS	<ul style="list-style-type: none"> • PRIVATE • FIRSTPRIVATE • LASTPRIVATE • REDUCTION
SECTION	<ul style="list-style-type: none"> • PRIVATE • FIRSTPRIVATE • LASTPRIVATE • REDUCTION
SINGLE END SINGLE	<ul style="list-style-type: none"> • PRIVATE • FIRSTPRIVATE
PARALLEL DO END PARALLEL DO	<ul style="list-style-type: none"> • COPYIN • DEFAULT • PRIVATE • FIRSTPRIVATE • LASTPRIVATE • REDUCTION • SHARED • SCHEDULE
PARALLEL SECTIONS END PARALLEL SECTIONS	<ul style="list-style-type: none"> • COPYIN • DEFAULT • PRIVATE • FIRSTPRIVATE • LASTPRIVATE

	<ul style="list-style-type: none"> • REDUCTION • SHARED
All others	None

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.

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	<p>Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping.</p> <p>See also the throughput execution mode and the <code>KMP_LIBRARY</code> environment variable. Use the</p>

		optional character suffix s, m, h, or d, to specify seconds, minutes, hours, or days.
KMP_LIBRARY	throughput (execution mode)	Selects the OpenMP run-time library execution mode. The options for the variable value are throughput or turnaround.
KMP_MONITOR_STACKSIZE	max(32k, system minimum thread stack size)	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_lib.h` header file is provided in the `INCLUDE` directory of your compiler installation (for use with the Fortran `INCLUDE` statement).
- The `omp_lib.mod` file is provided in the `INCLUDE` directory (for use with the Fortran `USE` statement).
- The `omp_lib.f` is provided in the `INCLUDE` directory.

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 Fortran version 2.5 specifications (<http://www.openmp.org/specs>).

Execution Environment Routines

Function	Description
SUBROUTINE OMP_SET_NUM_THREADS(<i>num_threads</i>) INTEGER <i>num_threads</i>	Sets the number of threads to use for subsequent parallel regions.
INTEGER FUNCTION OMP_GET_NUM_THREADS()	Returns the number of threads that are being used in the current parallel region.
INTEGER FUNCTION OMP_GET_MAX_THREADS()	Returns the maximum number of threads that are available for parallel execution.
INTEGER FUNCTION OMP_GET_THREAD_NUM()	Determines the unique thread number of the thread currently executing this section of code.
INTEGER FUNCTION OMP_GET_NUM_PROCS()	Determines the number of processors available to the program.
LOGICAL FUNCTION OMP_IN_PARALLEL()	Returns <code>.TRUE.</code> if called within the dynamic extent of a parallel region executing in parallel; otherwise returns <code>.FALSE.</code>
SUBROUTINE OMP_SET_DYNAMIC(<i>dynamic_threads</i>) LOGICAL <i>dynamic_threads</i>	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If <i>dynamic_threads</i> is <code>.TRUE.</code> , dynamic threads are enabled. If <i>dynamic_threads</i> is <code>.FALSE.</code> , dynamic threads are disabled. Dynamic threads are disabled by default.
LOGICAL FUNCTION OMP_GET_DYNAMIC()	Returns <code>.TRUE.</code> if dynamic thread adjustment is enabled, otherwise returns <code>.FALSE.</code>
SUBROUTINE OMP_SET_NESTED(<i>nested</i>) LOGICAL <i>nested</i>	Enables or disables nested parallelism. If <i>nested</i> is <code>.TRUE.</code> , nested parallelism is enabled. If <i>nested</i> is <code>.FALSE.</code> , nested parallelism is disabled. Nested parallelism is disabled by default.
LOGICAL FUNCTION OMP_GET_NESTED()	Returns <code>.TRUE.</code> if nested parallelism is enabled, otherwise returns <code>.FALSE.</code>

Lock Routines

Function	Description
SUBROUTINE OMP_INIT_LOCK(<i>lock</i>) INTEGER (KIND=OMP_LOCK_KIND) :: <i>lock</i>	Initializes the lock associated with <i>lock</i> for use in subsequent calls.
SUBROUTINE OMP_DESTROY_LOCK(<i>lock</i>) INTEGER (KIND=OMP_LOCK_KIND) :: <i>lock</i>	Causes the lock associated with <i>lock</i> to become undefined.
SUBROUTINE OMP_SET_LOCK(<i>lock</i>) INTEGER (KIND=OMP_LOCK_KIND) :: <i>lock</i>	Forces the executing thread to wait until the lock associated with <i>lock</i> is available. The thread is granted ownership of the lock when it becomes available.
SUBROUTINE OMP_UNSET_LOCK(<i>lock</i>)	Releases the executing thread from

INTEGER(KIND=OMP_LOCK_KIND) :: <i>lock</i>	ownership of the lock associated with <i>lock</i> . The behavior is undefined if the executing thread does not own the lock associated with <i>lock</i> .
LOGICAL OMP_TEST_LOCK(<i>lock</i>) INTEGER(KIND=OMP_LOCK_KIND) :: <i>lock</i>	Attempts to set the lock associated with <i>lock</i> . If successful, returns <code>.TRUE.</code> , otherwise returns <code>.FALSE.</code> .
SUBROUTINE OMP_INIT_NEST_LOCK(<i>lock</i>) INTEGER(KIND=OMP_NEST_LOCK_KIND) :: <i>lock</i>	Initializes the nested lock associated with <i>lock</i> for use in the subsequent calls.
SUBROUTINE OMP_DESTROY_NEST_LOCK(<i>lock</i>) INTEGER(KIND=OMP_NEST_LOCK_KIND) :: <i>lock</i>	Causes the nested lock associated with <i>lock</i> to become undefined.
SUBROUTINE OMP_SET_NEST_LOCK(<i>lock</i>) INTEGER(KIND=OMP_NEST_LOCK_KIND) :: <i>lock</i>	Forces the executing thread to wait until the nested lock associated with <i>lock</i> is available. The thread is granted ownership of the nested lock when it becomes available.
SUBROUTINE OMP_UNSET_NEST_LOCK(<i>lock</i>) INTEGER(KIND=OMP_NEST_LOCK_KIND) :: <i>lock</i>	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> .
INTEGER OMP_TEST_NEST_LOCK(<i>lock</i>) INTEGER(KIND=OMP_NEST_LOCK_KIND) :: <i>lock</i>	Attempts to set the nested lock associated with <i>lock</i> . If successful, returns the nesting count, otherwise returns zero.

Timing Routines

Function	Description
DOUBLE-PRECISION FUNCTION OMP_GET_WTIME()	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.
DOUBLE-PRECISION FUNCTION OMP_GET_WTICK()	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
- Getting and setting thread sleep time for the throughput execution mode

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.

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
FUNCTION <code>KMP_GET_STACKSIZE_S()</code> INTEGER(KIND=KMP_SIZE_T_KIND) & <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 via the <code>KMP_SET_STACKSIZE_S</code> routine, prior to the first parallel region or via the <code>KMP_STACKSIZE</code> environment variable.
FUNCTION <code>KMP_GET_STACKSIZE()</code> INTEGER <code>KMP_GET_STACKSIZE</code>	This routine is provided for backwards compatibility only; use <code>KMP_GET_STACKSIZE_S</code> routine for compatibility across different families of Intel processors.
SUBROUTINE <code>KMP_SET_STACKSIZE_S(size)</code> INTEGER (KIND=KMP_SIZE_T_KIND) <i>size</i>	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.
SUBROUTINE KMP_SET_STACKSIZE_S(<i>size</i>) INTEGER <i>size</i>	This routine is provided for backward compatibility only; use KMP_SET_STACKSIZE_S(<i>size</i>) 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
FUNCTION KMP_MALLOC(<i>size</i>) INTEGER(KIND=KMP_POINTER_KIND) KMP_MALLOC INTEGER(KIND=KMP_SIZE_T_KIND) <i>size</i>	Allocate memory block of <i>size</i> bytes from thread-local heap.
FUNCTION KMP_CALLOC(<i>nelem</i> , <i>elsize</i>) INTEGER(KIND=KMP_POINTER_KIND) KMP_CALLOC INTEGER(KIND=KMP_SIZE_T_KIND) <i>nelem</i> INTEGER(KIND=KMP_SIZE_T_KIND) <i>elsize</i>	Allocate array of <i>nelem</i> elements of size <i>elsize</i> from thread-local heap.
FUNCTION KMP_REALLOC(<i>ptr</i> , <i>size</i>) INTEGER(KIND=KMP_POINTER_KIND) KMP_REALLOC INTEGER(KIND=KMP_POINTER_KIND) <i>ptr</i> INTEGER(KIND=KMP_SIZE_T_KIND) <i>size</i>	Reallocate memory block at address <i>ptr</i> and <i>size</i> bytes from thread-local heap.
SUBROUTINE KMP_FREE(<i>ptr</i>) INTEGER (KIND=KMP_POINTER_KIND) <i>ptr</i>	Free memory block at address <i>ptr</i> from thread-local heap. Memory must have been previously allocated with KMP_MALLOC, KMP_CALLOC, or KMP_REALLOC.

In the throughput execution mode, threads wait for new parallel work at the ends of parallel regions, and then sleep, after a specified period of time. This time interval can be set by the KMP_BLOCKTIME environment variable or by the KMP_SET_BLOCKTIME() function.

Thread Sleep Time

Function	Description
FUNCTION KMP_GET_BLOCKTIME() INTEGER KMP_GET_BLOCKTIME	Returns the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping, as set either by the

	KMP_BLOCKTIME environment variable or by <code>kmp_set_blocktime()</code> .
<pre>FUNCTION KMP_SET_BLOCKTIME(<i>msec</i>) INTEGER <i>msec</i></pre>	Sets the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping. In order for <code>kmp_set_blocktime()</code> to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region.

Examples of OpenMP* Usage

The following examples show how to use the OpenMP* features.

See more examples in the OpenMP Fortran version 2.5 specifications (<http://www.openmp.org/specs>).

DO: A Simple Difference Operator

This example shows a simple parallel loop where each iteration contains a different number of instructions. To get good load balancing, dynamic scheduling is used. The `end do` has a `nowait` because there is an implicit `barrier` at the end of the parallel region.

Example
<pre>subroutine do 1 (a,b,n) real a(n,n), b(n,n) !\$OMP PARALLEL !\$OMP& SHARED(A,B,N) !\$OMP& PRIVATE(I,J) !\$OMP DO SCHEDULE(DYNAMIC,1) do i = 2, n do j = 1, i b(j,i) = (a(j,i) + a(j,i-1)) / 2 enddo enddo !\$OMP END DO NOWAIT !\$OMP END PARALLEL end</pre>

DO: Two Difference Operators

This example shows two parallel regions fused to reduce `fork/join` overhead. The first `end do` has a `nowait` because all the data used in the second loop is different than all the data used in the first loop.

Example
<pre>subroutine do 2 (a,b,c,d,m,n) real a(n,n), b(n,n), c(m,m), d(m,m) !\$OMP PARALLEL</pre>

```

!$OMP& SHARED(A,B,C,D,M,N)
!$OMP& PRIVATE(I,J)
!$OMP DO SCHEDULE(DYNAMIC,1)
  do i = 2, n
    do j = 1, i
      b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
    enddo
  enddo
!$OMP END DO NOWAIT
!$OMP DO SCHEDULE(DYNAMIC,1)
  do i = 2, m
    do j = 1, i
      d(j,i) = ( c(j,i) + c(j,i-1) ) / 2
    enddo
  enddo
!$OMP END DO NOWAIT
!$OMP END PARALLEL
end

```

SECTIONS: Two Difference Operators

This example demonstrates the use of the `SECTIONS` directive. The logic is identical to the preceding `DO` example, but uses `SECTIONS` instead of `DO`. Here the speedup is limited to 2 because there are only two units of work whereas in `DO`: Two Difference Operators above there are $n-1 + m-1$ units of work.

Example

```

subroutine sections 1 (a,b,c,d,m,n)
real a(n,n), b(n,n), c(m,m), d(m,m)
!$OMP PARALLEL
!$OMP& SHARED(A,B,C,D,M,N)
!$OMP& PRIVATE(I,J)
!$OMP SECTIONS
!$OMP SECTION
  do i = 2, n
    do j = 1, i
      b(j,i)=( a(j,i) + a(j,i-1) ) / 2
    enddo
  enddo
!$OMP SECTION
  do i = 2, m
    do j = 1, i
      d(j,i)=( c(j,i) + c(j,i-1) ) / 2
    enddo
  enddo
!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL
end

```

SINGLE: Updating a Shared Scalar

This example demonstrates how to use a `SINGLE` construct to update an element of the shared array `a`. The optional `nowait` after the first loop is omitted because it is necessary to wait at the end of the loop before proceeding into the `SINGLE` construct.

Example

```

subroutine sp 1a (a,b,n)
real a(n), b(n)
!$OMP PARALLEL
!$OMP& SHARED(A,B,N)
!$OMP& PRIVATE(I)
!$OMP DO
  do i = 1, n
    a(i) = 1.0 / a(i)
  enddo
!$OMP SINGLE
  a(1) = min( a(1), 1.0 )
!$OMP END SINGLE
!$OMP DO
  do i = 1, n
    b(i) = b(i) / a(i)
  enddo
!$OMP END DO NOWAIT
!$OMP END PARALLEL
end

```

OpenMP Directive Descriptions**Combined Parallel and Worksharing Constructs**

The combined parallel/worksharing constructs provide an abbreviated way to specify a parallel region that contains a single worksharing construct. The combined parallel/worksharing constructs are:

- `PARALLEL DO` and `END PARALLEL DO` directives
- `PARALLEL SECTIONS` and `END PARALLEL SECTIONS` directives
- `WORKSHARE` and `PARALLEL WORKSHARE` directives

For more details on these directives, see OpenMP* Fortran Compiler Directives in the *Intel® Fortran Language Reference*.

PARALLEL DO and END PARALLEL DO

Use the `PARALLEL DO` directive to specify a parallel region that implicitly contains a single `DO` directive. You can specify one or more of the clauses for the `PARALLEL DO` directives.

The following example shows how to parallelize a simple loop. The loop iteration variable is private by default, so it is not necessary to declare it explicitly. The `END PARALLEL DO` directive is optional:

Example

```

subroutine par(a, b, N)
  integer :: i, N, a(N), b(N)
  !$OMP PARALLEL DO
  do i= 1, N
    b(i) = (a(i) + a(i-1)) / 2.0
  enddo
end

```

```

end do
!$OMP END PARALLEL DO
end subroutine par

```

PARALLEL SECTIONS and END PARALLEL SECTIONS

Use the `PARALLEL SECTIONS` directive to specify a parallel region that implicitly contains a single `SECTIONS` directive. You can specify one or more of the clauses for the `PARALLEL SECTIONS` directives.

The last section ends at the `END PARALLEL SECTIONS` directive.

In the following example, subroutines `X_AXIS`, `Y_AXIS`, and `Z_AXIS` can be executed concurrently. The first `SECTION` directive is optional. Note that all `SECTION` directives must appear in the lexical extent of the `PARALLEL SECTIONS/END PARALLEL SECTIONS` construct:

Example

```

!$OMP PARALLEL SECTIONS
!$OMP SECTION
    CALL X_AXIS
!$OMP SECTION
    CALL Y_AXIS
!$OMP SECTION
    CALL Z_AXIS
!$OMP END PARALLEL SECTIONS

```

WORKSHARE and PARALLEL WORKSHARE

Use the `WORKSHARE` directive to divide work within blocks of worksharing statements or constructs into different units. This directive distributes the work of executing the units to threads of the team so each unit is only executed once.

Use the `PARALLEL WORKSHARE` directive to specify parallel regions, in an abbreviated way, that contain a single `WORKSHARE` directive.

When using either directive, be aware that your code cannot branch in to or out of the block defined by these directives.

Parallel Region Directives

The `PARALLEL` and `END PARALLEL` directives define a parallel region as follows:

Example

```

!$OMP PARALLEL
! parallel region
!$OMP END PARALLEL

```

When a thread encounters a parallel region, it creates a team of threads and becomes the master of the team. You can control the number of threads in a team by the use of an environment variable or a run-time library call, or both.

Clauses Used

The `PARALLEL` directive takes an optional comma-separated list of clauses that specify as follows:

- `IF`: whether the statements in the parallel region are executed in parallel by a team of threads or serially by a single thread.
- `PRIVATE`, `FIRSTPRIVATE`, `SHARED`, or `REDUCTION`: variable types
- `DEFAULT`: variable data scope attribute
- `COPYIN`: master thread common block values are copied to `THREADPRIVATE` copies of the common block

Changing the Number of Threads

Once created, the number of threads in the team remains constant for the duration of that parallel region. To explicitly change the number of threads used in the next parallel region, call the `OMP_SET_NUM_THREADS` run-time library routine from a serial portion of the program. This routine overrides any value you may have set using the `OMP_NUM_THREADS` environment variable.

Assuming you have used the `OMP_NUM_THREADS` environment variable to set the number of threads to 6, you can change the number of threads between parallel regions as follows:

Example
<pre>CALL OMP SET NUM_THREADS (3) !\$OMP PARALLEL ... !\$OMP PARALLEL CALL OMP SET NUM_THREADS (4) !\$OMP PARALLEL DO ... !\$OMP END PARALLEL DO</pre>

Setting Units of Work

Use the worksharing directives such as `DO`, `SECTIONS`, and `SINGLE` to divide the statements in the parallel region into units of work and to distribute those units so that each unit is executed by one thread.

In the following example, the `!$OMP DO` and `!$OMP END DO` directives and all the statements enclosed by them comprise the static extent of the parallel region:

Example
<pre>!\$OMP PARALLEL</pre>

```
!$OMP DO
  DO I=1,N
    B(I) = (A(I) + A(I-1))/ 2.0
  END DO
!$OMP END DO
!$OMP END PARALLEL
```

In the following example, the `!$OMP DO` and `!$OMP END DO` directives and all the statements enclosed by them, including all statements contained in the `WORK` subroutine, comprise the dynamic extent of the parallel region:

Example

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
  DO I=1,N
    CALL WORK(I,N)
  END DO
!$OMP END DO
!$OMP END PARALLEL
```

Setting Conditional Parallel Region Execution

When an `IF` clause is present on the `PARALLEL` directive, the enclosed code region is executed in parallel only if the scalar logical expression evaluates to `.TRUE.`. Otherwise, the parallel region is serialized. When there is no `IF` clause, the region is executed in parallel by default.

In the following example, the statements enclosed within the `!$OMP DO` and `!$OMP END DO` directives are executed in parallel only if there are more than three processors available. Otherwise the statements are executed serially:

Example

```
!$OMP PARALLEL IF (OMP_GET_NUM_PROCS() .GT. 3)
!$OMP DO
  DO I=1,N
    Y(I) = SQRT(Z(I))
  END DO
!$OMP END DO
!$OMP END PARALLEL
```

If a thread executing a parallel region encounters another parallel region, it creates a new team and becomes the master of that new team. By default, nested parallel regions are always executed by a team of one thread.

Note

To achieve better performance than sequential execution, a parallel region must contain one or more worksharing constructs so that the team of threads can execute work in parallel. It is the contained worksharing constructs that lead to the performance enhancements offered by parallel processing.

For more details on this directive, see OpenMP* Fortran Compiler Directives in the *Intel® Fortran Language Reference*.

Synchronization Constructs

Synchronization constructs are used to ensure the consistency of shared data and to coordinate parallel execution among threads.

The synchronization constructs are:

- `ATOMIC` directive
- `BARRIER` directive
- `CRITICAL` directive
- `FLUSH` directive
- `MASTER` directive
- `ORDERED` directive

For more details on these constructs, see OpenMP* Fortran Compiler Directives in the *Intel® Fortran Language Reference*.

ATOMIC Directive

Use the `ATOMIC` directive to ensure that a specific memory location is updated atomically instead of exposing the location to the possibility of multiple, simultaneously writing threads.

This directive applies only to the immediately following statement, which must have one of the following forms:

Form	Description
<code>x = x operator expr</code>	<code>x</code> is a scalar variable of intrinsic type
<code>x = expr operator x</code>	<code>expr</code> is a scalar expression that does not reference <code>x</code>
<code>x = intrinsic (x, expr)</code>	<code>intrinsic</code> is either <code>MAX</code> , <code>MIN</code> , <code>IAND</code> , <code>IOR</code> or <code>IEOR</code>
<code>x = intrinsic (expr, x)</code>	<code>operator</code> is either <code>+</code> , <code>*</code> , <code>-</code> , <code>/</code> , <code>.AND.</code> , <code>.OR.</code> , <code>.EQV.</code> , or <code>.NEQV.</code>

This directive permits optimization beyond that of a critical section around the assignment. An implementation can replace all `ATOMIC` directives by enclosing the statement in a critical section. All of these critical sections must use the same unique name.

Only the load and store of `x` are atomic; the evaluation of `expr` is not atomic. To avoid race conditions, all updates of the location in parallel must be protected by using the `ATOMIC` directive, except those that are known to be free of race conditions. The function `intrinsic`, the operator `operator`, and the assignment must be the intrinsic function, operator, and assignment.

This restriction applies to the `ATOMIC` directive: All references to storage location `x` must have the same type parameters.

In the following example, the collection of `Y` locations is updated atomically:

Example
<pre>!\$OMP ATOMIC Y = Y + B(I)</pre>

BARRIER Directive

To synchronize all threads within a parallel region, use the `BARRIER` directive. You can use this directive only within a parallel region defined by using the `PARALLEL` directive. You cannot use the `BARRIER` directive within the `DO`, `PARALLEL DO`, `SECTIONS`, `PARALLEL SECTIONS`, and `SINGLE` directives.

When encountered, each thread waits at the `BARRIER` directive until all threads have reached the directive.

In the following example, the `BARRIER` directive ensures that all threads have executed the first loop and that it is safe to execute the second loop:

Example
<pre>c\$OMP PARALLEL c\$OMP DO PRIVATE(i) DO i = 1, 100 b(i) = i END DO c\$OMP BARRIER c\$OMP DO PRIVATE(i) DO i = 1, 100 a(i) = b(101-i) END DO c\$OMP END PARALLEL</pre>

CRITICAL and END CRITICAL

Use the `CRITICAL` and `END CRITICAL` directives to restrict access to a block of code, referred to as a critical section, to one thread at a time.

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same name.

When a thread enters the critical section, a latch variable is set to closed and all other threads are locked out. When the thread exits the critical section at the `END CRITICAL` directive, the latch variable is set to open, allowing another thread access to the critical section.

If you specify a critical section name in the `CRITICAL` directive, you must specify the same name in the `END CRITICAL` directive. If you do not specify a name for the `CRITICAL` directive, you cannot specify a name for the `END CRITICAL` directive.

All unnamed `CRITICAL` directives map to the same name. Critical section names are global to the program.

The following example includes several `CRITICAL` directives, and illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a critical section. Because there are two independent queues in this example, each queue is protected by `CRITICAL` directives having different names, `X_AXIS` and `Y_AXIS`, respectively:

Example

```
!$OMP PARALLEL DEFAULT(PRIVATE), SHARED(X,Y)
!$OMP CRITICAL(X_AXIS)
CALL DEQUEUE(IX NEXT, X)
!$OMP END CRITICAL(X_AXIS)
CALL WORK(IX NEXT, X)
!$OMP CRITICAL(Y_AXIS)
CALL DEQUEUE(IY NEXT, Y)
!$OMP END CRITICAL(Y_AXIS)
CALL WORK(IY NEXT, Y)
!$OMP END PARALLEL
```

Unnamed critical sections use the global lock from the Pthread package. This allows you to synchronize with other code by using the same lock. Named locks are created and maintained by the compiler and can be significantly more efficient.

FLUSH Directive

Use the `FLUSH` directive to identify a synchronization point at which a consistent view of memory is provided. Thread-visible variables are written back to memory at this point.

To avoid flushing all thread-visible variables at this point, include a list of comma-separated named variables to be flushed. The following example uses the `FLUSH` directive for point-to-point synchronization between thread 0 and thread 1 for the variable `ISYNC`:

Example

```
!$OMP PARALLEL DEFAULT(PRIVATE), SHARED(ISYNC)
IAM = OMP GET THREAD NUM()
ISYNC(IAM) = 0
!$OMP BARRIER
CALL WORK()
! Synchronize With My Neighbor
ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
! Wait Till Neighbor Is Done
DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
END DO
!$OMP END PARALLEL
```

MASTER and END MASTER

Use the `MASTER` and `END MASTER` directives to identify a block of code that is executed only by the master thread.

The other threads of the team skip the code and continue execution. There is no implied barrier at the `END MASTER` directive. In the following example, only the master thread executes the routines `OUTPUT` and `INPUT`:

Example
<pre>!\$OMP PARALLEL DEFAULT(SHARED) CALL WORK(X) !\$OMP MASTER CALL OUTPUT(X) CALL INPUT(Y) !\$OMP END MASTER CALL WORK(Y) !\$OMP END PARALLEL</pre>

ORDERED and END ORDERED

Use the `ORDERED` and `END ORDERED` directives within a `DO` construct to allow work within an ordered section to execute sequentially while allowing work outside the section to execute in parallel.

When you use the `ORDERED` directive, you must also specify the `ORDERED` clause on the `DO` directive.

Only one thread at a time is allowed to enter the ordered section, and then only in the order of loop iterations. In the following example, the code prints out the indexes in sequential order:

Example
<pre>!\$OMP DO ORDERED, SCHEDULE(DYNAMIC) DO I=LB,UB,ST CALL WORK(I) END DO SUBROUTINE WORK(K) !\$OMP ORDERED WRITE(*,*) K !\$OMP END ORDERED</pre>

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
<pre>COMMON /BLK1/ SCRATCH COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD !\$OMP THREADPRIVATE(/BLK1/, /FIELDS/)</pre>

For more details on this directive, see OpenMP* Fortran Compiler Directives in the *Intel® Fortran Language Reference*.

Worksharing Construct Directives

A worksharing construct must be enclosed dynamically within a parallel region if the worksharing directive is to execute in parallel. No new threads are launched and there is no implied barrier on entry to a worksharing construct.

The worksharing constructs are:

- `DO` and `END DO` directives
- `SECTIONS`, `SECTION`, and `END SECTIONS` directives
- `SINGLE` and `END SINGLE` directives

For more details on these directives, see OpenMP* Fortran Compiler Directives in the *Intel® Fortran Language Reference*.

DO and END DO

The `DO` directive specifies that the iterations of the immediately following `DO` loop must be dispatched across the team of threads so that each iteration is executed by a single thread. The loop that follows a `DO` directive cannot be a `DO WHILE` or a `DO` loop that does not have loop control. The iterations of the `DO` loop are dispatched among the existing team of threads.

The `DO` directive optionally lets you:

- Control data scope attributes
- Use the `SCHEDULE` clause to specify schedule type and chunk size (see Specifying Schedule Type and Chunk Size)

Clauses Used

The clauses for `DO` directive specify:

- Whether variables are `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, or `REDUCTION`
- How loop iterations are `SCHEDULED` onto threads
- In addition, the `ORDERED` clause must be specified if the `ORDERED` directive appears in the dynamic extent of the `DO` directive.
- If you do not specify the optional `NOWAIT` clause on the `END DO` directive, threads synchronize at the `END DO` directive. If you specify `NOWAIT`, threads do not synchronize, and threads that finish early proceed directly to the instructions following the `END DO` directive.

Usage Rules

- You cannot use a `GOTO` statement, or any other statement, to transfer control onto or out of the `DO` construct.
- If you specify the optional `END DO` directive, it must appear immediately after the end of the `DO` loop. If you do not specify the `END DO` directive, an `END DO` directive is assumed at the end of the `DO` loop, and threads synchronize at that point.
- The loop iteration variable is private by default, so it is not necessary to declare it explicitly.

SECTIONS, SECTION and END SECTIONS

Use the noniterative worksharing `SECTIONS` directive to divide the enclosed sections of code among the team. Each section is executed just one time by one thread.

Each section should be preceded with a `SECTION` directive, except for the first section, in which the `SECTION` directive is optional. The `SECTION` directive must appear within the lexical extent of the `SECTIONS` and `END SECTIONS` directives.

The last section ends at the `END SECTIONS` directive. When a thread completes its section and there are no undispatched sections, it waits at the `END SECTION` directive unless you specify `NOWAIT`.

The `SECTIONS` directive takes an optional comma-separated list of clauses that specifies which variables are `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, or `REDUCTION`.

The following example shows how to use the `SECTIONS` and `SECTION` directives to execute subroutines `X_AXIS`, `Y_AXIS`, and `Z_AXIS` in parallel. The first `SECTIONS` directive is optional:

Example
<pre>!\$OMP PARALLEL !\$OMP SECTIONS !\$OMP SECTION CALL X_AXIS !\$OMP SECTION</pre>

```
CALL Y AXIS
!$OMP SECTION
CALL Z AXIS
!$OMP END SECTIONS
!$OMP END PARALLEL
```

SINGLE and END SINGLE

Use the `SINGLE` directive when you want just one thread of the team to execute the enclosed block of code.

Threads that are not executing the `SINGLE` directive wait at the `END SINGLE` directive unless you specify `NOWAIT`.

The `SINGLE` directive takes an optional comma-separated list of clauses that specifies which variables are `PRIVATE` or `FIRSTPRIVATE`.

When the `END SINGLE` directive is encountered, an implicit barrier is erected and threads wait until all threads have finished. This can be overridden by using the `NOWAIT` option.

In the following example, the first thread that encounters the `SINGLE` directive executes subroutines `OUTPUT` and `INPUT`:

Example
<pre>!\$OMP PARALLEL DEFAULT(SHARED) CALL WORK(X) !\$OMP BARRIER !\$OMP SINGLE CALL OUTPUT(X) CALL INPUT(Y) !\$OMP END SINGLE CALL WORK(Y) !\$OMP END PARALLEL</pre>

Data Scope Attribute Clauses Overview

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.

Each of the data scope attribute clauses accepts a list, which is a comma-separated list of named variables or named common blocks that are accessible in the scoping unit. When you specify named common blocks, they must appear between slashes (*/name/*).

Not all of the clauses are allowed on all directives, but the directives to which each clause applies are listed in the clause descriptions.

The data scope attribute clauses are:

- COPYIN
- DEFAULT
- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- REDUCTION
- SHARED

COPYIN Clause

Use the `COPYIN` clause on the `PARALLEL`, `PARALLEL DO`, and `PARALLEL SECTIONS` directives to copy the data in the master thread common block to the thread private copies of the common block. The copy occurs at the beginning of the parallel region. The `COPYIN` clause applies only to common blocks that have been declared `THREADPRIVATE`.

You do not have to specify a whole common block to be copied in; you can specify named variables that appear in the `THREADPRIVATE` common block. In the following example, the common blocks `BLK1` and `FIELDS` are specified as thread private, but only one of the variables in common block `FIELDS` is specified to be copied in:

Example
<pre>COMMON /BLK1/ SCRATCH COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD !\$OMP THREADPRIVATE(/BLK1/, /FIELDS/) !\$OMP PARALLEL DEFAULT(PRIVATE),COPYIN(/BLK1/,ZFIELD)</pre>

DEFAULT Clause

Use the `DEFAULT` clause on the `PARALLEL`, `PARALLEL DO`, and `PARALLEL SECTIONS` directives to specify a default data scope attribute for all variables within the lexical extent of a parallel region. Variables in `THREADPRIVATE` common blocks are not affected by this clause. You can specify only one `DEFAULT` clause on a directive. The default data scope attribute can be one of the following:

Attribute	Description
PRIVATE	Makes all named objects in the lexical extent of the parallel region private to a thread. The objects include common block variables, but exclude <code>THREADPRIVATE</code> variables.
SHARED	Makes all named objects in the lexical extent of the parallel region shared among all the threads in the team.
NONE	Declares that there is no implicit default as to whether variables are <code>PRIVATE</code> or <code>SHARED</code> . You must explicitly specify the scope attribute for each variable in the lexical extent of the parallel region.

If you do not specify the `DEFAULT` clause, the default is `DEFAULT (SHARED)`. However, loop control variables are always `PRIVATE` by default.

You can exempt variables from the default data scope attribute by using other scope attribute clauses on the parallel region as shown in the following example:

Example

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) , FIRSTPRIVATE(I) , SHARED(X) ,
!$OMP& SHARED(R) LASTPRIVATE(I)
```

PRIVATE, FIRSTPRIVATE, and LASTPRIVATE Clauses

PRIVATE

Use the `PRIVATE` clause on the `PARALLEL`, `DO`, `SECTIONS`, `SINGLE`, `PARALLEL DO`, and `PARALLEL SECTIONS` directives to declare variables to be private to each thread in the team.

The behavior of variables declared `PRIVATE` is as follows:

- A new object of the same type and size is declared once for each thread in the team, and the new object is no longer storage associated with the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.
- Variables defined as `PRIVATE` are undefined for each thread on entering the construct, and the corresponding shared variable is undefined on exit from a parallel construct.
- Contents, allocation state, and association status of variables defined as `PRIVATE` are undefined when they are referenced outside the lexical extent, but inside the dynamic extent, of the construct unless they are passed as actual arguments to called routines.

In the following example, the values of `I` and `J` are undefined on exit from the parallel region:

Example

```
INTEGER I, J
I = 1
J = 2
!$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
I = 3
J = J + 2
!$OMP END PARALLEL
PRINT *, I, J
```

FIRSTPRIVATE

Use the `FIRSTPRIVATE` clause on the `PARALLEL`, `DO`, `SECTIONS`, `SINGLE`, `PARALLEL DO`, and `PARALLEL SECTIONS` directives to provide a superset of the `PRIVATE` clause functionality.

In addition to the `PRIVATE` clause functionality, private copies of the variables are initialized from the original object existing before the parallel construct.

LASTPRIVATE

Use the `LASTPRIVATE` clause on the `PARALLEL`, `DO`, `SECTIONS`, `SINGLE`, `PARALLEL DO`, and `PARALLEL SECTIONS` directives to provide a superset of the `PRIVATE` clause functionality.

When the `LASTPRIVATE` clause appears on a `DO PARALLEL DO` directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct.

When the `LASTPRIVATE` clause appears on a `SECTIONS` or `PARALLEL SECTIONS` directive, the thread that executes the lexically last section updates the version of the object it had before the construct.

Sub-objects that are not assigned a value by the last iteration of the `DO` loop or the lexically last `SECTION` directive are undefined after the construct.

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. You must list all such variables as arguments to a `LASTPRIVATE` clause so that the values of the variables are the same as when the loop is executed sequentially. As shown in the following example, the value of `I` at the end of the parallel region is equal to `N+1`, as it would be with sequential execution.

Example
<pre>!\$OMP PARALLEL !\$OMP DO LASTPRIVATE(I) DO I=1,N A(I) = B(I) + C(I) END DO !\$OMP END PARALLEL CALL REVERSE(I)</pre>

REDUCTION Clause

Use the `REDUCTION` clause on the `PARALLEL`, `DO`, `SECTIONS`, `PARALLEL DO`, and `PARALLEL SECTIONS` directives to perform a reduction on the specified variables by using an operator or intrinsic as shown:

Example
<pre>REDUCTION (operator or intrinsic :list)</pre>

where:

- *Operator* can be one of the following: `+`, `*`, `-`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.`
- *Intrinsic* can be one of the following: `MAX`, `MIN`, `IAND`, `IOR`, or `IEOR`.

The specified variables must be named variables of intrinsic type and must be `SHARED` in the enclosing context. Deferred-shape and assumed-size arrays are not allowed. A private copy of each specified variable is created for each thread as if you had used the `PRIVATE` clause. The private copy is initialized to a value that depends on the operator or intrinsic as shown in the following table. The actual initialization value is consistent with the data type of the reduction variable.

Operators/Intrinsics and Initialization Values for Reduction Variables

Operators/Intrinsic	Initialization Value
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	Largest representable number
MIN	Smallest representable number
IAND	All bits on
IOR	0
IEOR	0

At the end of the construct to which the reduction applies, the shared variable is updated to reflect the result of combining the original value of the `SHARED` reduction variable with the final value of each of the private copies using the specified operator.

Except for subtraction, all of the reduction operators are associative and the compiler can freely reassociate the computation of the final value. The partial results of a subtraction reduction are added to form the final value.

The value of the shared variable becomes undefined when the first thread reaches the clause containing the reduction, and it remains undefined until the reduction computation is complete. Normally, the computation is complete at the end of the `REDUCTION` construct. However, if you use the `REDUCTION` clause on a construct to which `NOWAIT` is also applied, the shared variable remains undefined until a barrier synchronization has been performed. This ensures that all of the threads have completed the `REDUCTION` clause.

The `REDUCTION` clause is intended to be used on a region or worksharing construct in which the reduction variable is used only in reduction statements having one of the following forms:

Form	Description
<code>x = x operator expr</code>	<code>x</code> is a scalar variable of intrinsic type
<code>x = expr operator x</code>	except for subtraction

<code>x = intrinsic (x, expr)</code>	<i>intrinsic is either MAX, MIN, IAND, IOR, or IEOB</i>
<code>x = intrinsic (expr, x)</code>	<i>operator is either +, *, -, /, .AND., .OR., .EQV., or .NEQV.</i>

Some reductions can be expressed in other forms. For instance, a `MAX` reduction might be expressed as follows:

Example
<code>IF (x .LT. expr) x = expr</code>

Alternatively, the reduction might be hidden inside a subroutine call. Be careful that the operator specified in the `REDUCTION` clause matches the reduction operation.

Any number of reduction clauses can be specified on the directive, but a variable can appear only once in a `REDUCTION` clause for that directive as shown in the following example:

Example
<code>!\$OMP DO REDUCTION(+: A, Y), REDUCTION(.OR.: AM)</code>

The following example shows how to use the `REDUCTION` clause:

Example
<pre>!\$OMP PARALLEL DO DEFAULT(PRIVATE), SHARED(A,B), REDUCTION(+: A,B) DO I=1,N CALL WORK(ALOCAL, BLOCAL) A = A + ALOCAL B = B + BLOCAL END DO !\$OMP END PARALLEL DO</pre>

SHARED Clause

Use the `SHARED` clause on the `PARALLEL`, `PARALLEL DO`, and `PARALLEL SECTIONS` directives to make variables shared among all the threads in a team.

In the following example, the variables `X` and `NPOINTS` are shared among all the threads in the team:

Example
<pre>!\$OMP PARALLEL DEFAULT(PRIVATE), SHARED(X,NPOINTS) IAM = OMP_GET_THREAD_NUM() NP = OMP_GET_NUM_THREADS() IPOINTS = NPOINTS/NP CALL SUBDOMAIN(X, IAM, IPOINTS) !\$OMP END PARALLEL</pre>

Specifying Schedule Type and Chunk Size

The `SCHEDULE` clause of the `DO` or `PARALLEL DO` directive specifies a scheduling algorithm that determines how iterations of the `DO` loop are divided among and dispatched to the threads of the team. The `SCHEDULE` clause applies only to the current `DO` or `PARALLEL DO` directive.

Within the `SCHEDULE` clause, you must specify a *schedule type* and, optionally, a *chunk size*. A *chunk* is a contiguous group of iterations dispatched to a thread. Chunk size must be a scalar integer expression.

The following list describes the schedule types and how the chunk size affects scheduling:

Schedule Type	Description
STATIC	<p>The iterations are divided into pieces having a size specified by chunk. The pieces are statically dispatched to threads in the team in a round-robin manner in the order of thread number.</p> <p>When chunk is not specified, the iterations are first divided into contiguous pieces by dividing the number of iterations by the number of threads in the team. Each piece is then dispatched to a thread before loop execution begins.</p>
DYNAMIC	<p>The iterations are divided into pieces having a size specified by chunk. As each thread finishes its currently dispatched piece of the iteration space, the next piece is dynamically dispatched to the thread. When no chunk is specified, the default is 1.</p>
GUIDED	<p>The chunk size is decreased exponentially with each succeeding dispatch. Chunk specifies the minimum number of iterations to dispatch each time. If there are less than chunk number of iterations remaining, the rest are dispatched. When no chunk is specified, the default is 1.</p>
RUNTIME	<p>The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by using the <code>OMP_SCHEDULE</code> environment variable. When you specify <code>RUNTIME</code>, you cannot specify a chunk size.</p>

The following list shows which schedule type is used, in priority order:

1. The schedule type specified in the `SCHEDULE` clause of the current `DO` or `PARALLEL DO` directive.
2. If the schedule type for the current `DO` or `PARALLEL DO` directive is `RUNTIME`, the default value specified in the `OMP_SCHEDULE` environment variable.
3. The compiler default schedule type of `STATIC`.

The following list shows which chunk size is used, in priority order:

- The chunk size specified in the `SCHEDULE` clause of the current `DO` or `PARALLEL DO` directive.
- For `RUNTIME` schedule type, the value specified in the `OMP_SCHEDULE` environment variable.
- For `DYNAMIC` and `GUIDED` schedule types, the default value 1.
- If the schedule type for the current `DO` or `PARALLEL DO` directive is `STATIC`, the loop iteration space divided by the number of threads in the team.

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

```

subroutine ser(a, b, c)
  integer, dimension(100) :: a, b, c
  do i=1,100
    a(i) = a(i) + b(i) * c(i)
  enddo
end subroutine ser

```

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

```

subroutine par(a, b, c)
  integer, dimension(100) :: a, b, c
  ! Thread 1
  do i=1,50
    a(i) = a(i) + b(i) * c(i)
  enddo
  ! Thread 2
  do i=51,100
    a(i) = a(i) + b(i) * c(i)
  enddo
end subroutine par

```

Programming with Auto-parallelization

The auto-parallelization feature implements some concepts of OpenMP*, such as the worksharing construct (with the `PARALLEL DO` 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.
- Insert the `!DEC$ PARALLEL` directive to disambiguate assumed data dependencies.
- Insert the `!DEC$ NOPARALLEL` directive before loops known to have insufficient work to justify the overhead of sharing among threads.

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

```
subroutine no_dep
  parameter (n=100000000)
  real a, c(n)
  do i = 1, n
    a = 2 * i - 1
    c(i) = sqrt(a)
  enddo
  print*, n, c(1), c(n)
end subroutine no_dep
```

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 `!DIR$ PARALLEL` directive.

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. Fortran 90 programmers can use the `PURE` attribute to assert that subroutines and functions contain no side effects. 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.

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 `!DEC$ parallel:`

Example

```
subroutine add(k, a, b)
  integer :: k
  real :: a(10000), b(10000)
  !$DEC parallel
  do i = 1, 10000
    a(i) = a(i+k) + b(i)
  end do
end subroutine add
```

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.

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>ifort -c -parallel myprog.f</code>
Windows	<code>ifort -c /Qparallel myprog.f</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 Directives

Auto-parallelization uses two specific directives, `!DEC$ PARALLEL` and `!DEC$ NO PARALLEL`.

Auto-parallelization Directives Format and Syntax

The format of an auto-parallelization compiler directive is:

Syntax
<code><prefix> <directive></code>

where the brackets above mean:

- `<xxx>`: the prefix and directive are required
 - For fixed form source input, the prefix is `!DEC$` or `CDEC$`
 - For free form source input, the prefix is `!DEC$` only

The prefix is followed by the directive name; for example:

Syntax
<code>!DEC\$ PARALLEL</code>

Since auto-parallelization directives begin with an exclamation point, the directives take the form of comments if you omit the `-parallel` (Linux) or `/Qparallel` (Windows) option.

The `!DEC$ PARALLEL` directive instructs the compiler to ignore dependencies that it assumes may exist and which would prevent correct parallelization in the immediately following loop. However, if dependencies are proven, they are not ignored.

The `!DEC$ NOPARALLEL` directive disables auto-parallelization for the following loop:

Example
<pre> program main parameter (n=100) integer x(n), a(n) !DEC\$ NOPARALLEL do i=1,n x(i) = i enddo !DEC\$ PARALLEL do i=1,n a(x(i)) = i enddo end </pre>

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

```
subroutine no_par(a, MAX)
  integer :: i, a(MAX)
  do i = 1, MAX
    a(i) = mod((i * 2), i) * 1 + sqrt(3.0)
    a(i) = a(i-1) + i
  end do
end subroutine no_par
```

You can use `-par-report3` (Linux) or `/Qpar-report3` (Windows) by entering a command similar to the following:

Platform	Commands
Linux	<code>ifort -parallel -par-report3 -c diag_prog.f90</code>
Windows	<code>ifort /Qparallel /Qpar-report3 /c diag_prog.f90</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 26
  flow data dependence from line 27 to line 28, due to "A"
  flow data dependence from line 28 to line 28, 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.
- Use the `!DEC$ PARALLEL` directive to eliminate assumed data dependency.

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
- 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 be REAL 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`.

Conversion to/from some types of floats is not valid. 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 be arithmetic or logical operations (again, typically for arrays). Arithmetic operations are limited to such operations as addition, subtraction, `ABS`, `MIN`, and `MAX`. Logical operations include bitwise `AND`, `OR`, and `XOR` operators. 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 valid. The loop body cannot contain any function calls other than the ones described above.

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

```
subroutine dep(data, n)
  real :: data(n)
  integer :: i
  do i = 1, n-1
    data(i) = data(i-1)*0.25 + data(i)*0.5 + data(i+1)*0.25
  end do
end subroutine dep
```

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

```
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 `DO-END DO` and `DO WHILE`, or by using an `IF/GOTO` and a label. 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

```
subroutine vec(a, b, c)
  dimension a(100), b(100), c(100)
  integer i
  i = 1
  do while (i .le. 100)
    a(i) = b(i) * c(i)
    if (a(i) .lt. 0.0) a(i) = 0.0
    i = i + 1
  enddo
end subroutine vec
```

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

```
subroutine no_vec(a, b, c)
  dimension a(100), b(100), c(100)
  integer i
  i = 1
  do while (i .le. 100)
    a(i) = b(i) * c(i)
    ! The next statement allows early
    ! exit from the loop and prevents
    ! vectorization of the loop.
    if (a(i) .lt. 0.0) go to 10
    i = i + 1
  enddo
  10 continue
end subroutine no_vecN
```

END

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

```
subroutine cnt1 (a, b, c, n, lb)
  dimension a(n), b(n), c(n)
  integer n, lb, i, count
  ! Number of iterations is "n - lb + 1"
  count = n
  do while (count .ge. lb)
    a(i) = b(i) * c(i)
    count = count - 1
    i = i + 1
  enddo ! lb is not defined within loop
end
```

The following example demonstrates a different countable loop construct.

Example: Countable Loop

```
! Number of iterations is (n-m+2)/2
subroutine cnt2 (a, b, c, m, n)
  dimension a(n), b(n), c(n)
  integer i, l, m, n
  i = 1;
  do l = m, n, 2
    a(i) = b(i) * c(i)
    i = i + 1
  enddo
end
```

The following examples demonstrates a loop construct that is non-countable due to dependency loop variant count value.

Example: Non-Countable Loop

```
! Number of iterations is dependent on a(i)
subroutine foo (a, b, c)
  dimension a(100), b(100), c(100)
  integer i
  i = 1
  do while (a(i) .gt. 0.0)
    a(i) = b(i) * c(i)
```

```

    i = i + 1
  enddo
end

```

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 = 1
do while (i<=n)
  a(i) = b(i) + c(i) ! Original loop code
  i = i + 1
end do

```

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 = 1
do while (i < (n - mod(n,4)))
! Vector strip-mined loop.
  a(i:i+3) = b(i:i+3) + c(i:i+3)
  i = i + 4
end do
do while (i <= n)
  a(i) = b(i) + c(i)      !Scalar clean-up loop
  i = i + 1
end do

```

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. The two-dimensional array *A* is referenced in the *j* (column) direction and then in the *i* (row) direction (column-major order); array *B* is referenced in the opposite manner (row-major order). Assume the memory layout is in column-major order; therefore, the access strides of array *A* and *B* for the code would be 1 and MAX, respectively. In example 2: BS = block_size; MAX must be evenly divisible by BS.

Consider the following loop example code:

Example: Original loop <pre> REAL A (MAX,MAX) , B (MAX,MAX) DO I =1, MAX DO J = 1, MAX A(I,J) = A(I,J) + B(J,I) ENDDO ENDDO </pre>
--

The arrays could be blocked into smaller chunks so that the total combined size of the two blocked chunks is smaller than the cache size, which can improve data reuse. One possible way of doing this is demonstrated below:

Example: Transformed Loop after blocking <pre> REAL A (MAX,MAX) , B (MAX,MAX) DO I =1, MAX, BS DO J = 1, MAX, BS DO II = I, I+MAX, BS-1 DO JJ = J, J+MAX, BS-1 A(II,JJ) = A(II,JJ) + B(JJ,II) ENDDO ENDDO ENDDO ENDDO </pre>
--

Vectorization Examples

This section contains simple examples of some common issues in vector programming.

Argument Aliasing: A Vector Copy

The loop in the example of a vector copy operation does not vectorize because the compiler cannot prove that `DEST(A(I))` and `DEST(B(I))` are distinct.

Example: Unvectorizable Copy Due to Unproven Distinction

```
SUBROUTINE VEC_COPY(DEST,A,B,LEN)
  DIMENSION DEST(*)
  INTEGER A(*), B(*)
  INTEGER LEN, I
  DO I=1,LEN
    DEST(A(I)) = DEST(B(I))
  END DO
  RETURN
END
```

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).

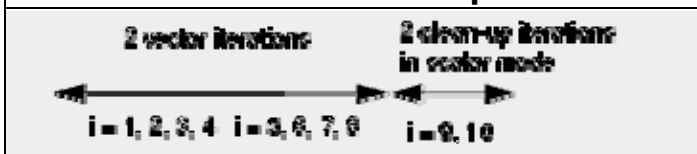
The figure (below) shows the effect of a data cache unit (DCU) split due to misaligned data. The code loads the misaligned data across a 16-byte boundary, which results in an additional memory access causing a six- to twelve-cycle stall. You can avoid the stalls if you know that the data is aligned and you specify to assume alignment

Misaligned Data Crossing 16-Byte Boundary



After vectorization, the loop is executed as shown in figure below

Vector and Scalar Clean-up Iterations



Both the vector iterations `A(1:4) = B(1:4)`; and `A(5:8) = B(5:8)`; can be implemented with aligned moves if both the elements `A(1)` and `B(1)` 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.

Alignment Strategy

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 below (several other strategies are supported as well). If in the loop shown below 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. In this case, the vectorizer applies dynamic loop peeling, a specific Intel® Fortran feature.

Examples of Data Alignment**Example: Original loop**

```
SUBROUTINE SIMLOOP(A)
REAL A(100)      ! alignment of argument A is unknown
DO I = 1, 100
  A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

Example: Aligning Data

```
! The vectorizer applies dynamic loop peeling as follows:
SUBROUTINE SIMLOOP(A)
REAL A(100)
! let P be (A%16) where A is address of A(1)
IF (P .NE. 0) THEN
  P = (16 - P)/4      ! determine run-time peeling factor
  DO I = 1, P
    A(I) = A(I) + 1.0
  ENDDO
ENDIF
! Now this loop starts at a 16-byte boundary, and will be
! vectorized accordingly
DO I = P + 1, 100
  A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

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

```

subroutine matmul_slow(a, b, c)
  integer :: i, j, k
  real :: a(100,100), b(100,100), c(100,100)
  do i = 1, n
    do j = 1, n
      do k = 1, n
        c(i,j) = c(i,j) + a(i,k)*b(k,j);
      end do
    end do
  end do
end subroutine matmul_slow

```

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

```

subroutine matmul_fast(a, b, c)
  integer :: i, j, k
  real :: a(100,100), b(100,100), c(100,100)
  do j = 1, n
    do k = 1, n
      do i = 1, n
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
    enddo
  enddo
end subroutine matmul_fast

```

Interchanging is not always possible because of dependencies, which can lead to different results.

Optimization Support Features Overview

This section describes the Intel® compiler features such as 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.

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

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

Pipelining for Itanium®-based Applications

The `SWP` directive indicates preference for loops to be software pipelined. The directive does not help data dependency, but overrides heuristics based on profile counts or unequal control flow.

The syntax for this directive is shown below:

Syntax

```
!DEC$ SWP
```

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 directive to instruct the compiler to attempt software pipelining.

Example: Using SWP

```
subroutine swp(a, b)
  integer :: i, a(100), b(100)
  !DEC$ SWP
  do i = 1, m
    if (a(i) .eq. 0) then
      b(i) = a(i) + 1
    else
      b(i) = a(i)/c(i)
    endif
  enddo
end subroutine swp
```

For more details on this directive, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

Loop Count and Loop Distribution

Loop Count

The `loop count` directive indicates the loop count is likely to be an integer constant. The syntax for this directive is shown below:

Syntax
<code>!DEC\$ LOOP COUNT (n)</code>

where *n* is an integer value.

The value of loop count affects heuristics used in software pipelining and data prefetch.

Example: Using loop count
<pre>subroutine loop_count(a, b, N) integer :: i, N, b(N), c(N) !DEC\$ LOOP COUNT (1000) ! This should enable software pipelining ! for this loop. do i = 1, 1000 b(i) = a(i) + 1 enddo end subroutine loop_count</pre>

For more details on this directive, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

Loop Distribution

The `distribute point` directive indicates a preference for performing loop distribution. The syntax for this directive is shown below:

Syntax
<code>!DEC\$ DISTRIBUTE POINT</code>

Loop distribution may cause large loops be distributed into smaller ones. This strategy might enable more loops to get software-pipelined.

- If the directive is placed inside a loop, the distribution is performed after the directive and any loop-carried dependency is ignored.
- If the directive is placed before a loop, the compiler will determine where to distribute and data dependency is observed. Multiple distribute directive are supported if they are placed inside the loop.
- When the directives are placed inside the loop, they cannot be put inside an IF statement.

Example: Using distribute point

```

subroutine dist1(a, b, c, d, N)
  integer :: i, N, a(N), b(N), c(N), d(N)
  !DEC$ DISTRIBUTE POINT
  do i = 1, N
    b(i) = a(i) + 1
    c(i) = a(i) + b(i)
    ! Compiler will decide where to distribute.
    ! Data dependency is observed.
    d(i) = c(i) + 1
  enddo
end subroutine dist1

subroutine dist2(a, b, c, d, N)
  integer :: i, N, a(N), b(N), c(N), d(N)
  do i = 1, N
    b(i) = a(i) + 1
    !DEC$ DISTRIBUTE POINT
    ! Distribution will start here, ignoring all
    ! loop-carried dependency.
    c(i) = a(i) + b(i)
    d(i) = c(i) + 1
  enddo
end subroutine dist2

```

Loop Unrolling Support

The `UNROLL [n]` directive tells the compiler how many times to unroll a counted loop.

The general syntax for this directive is shown below:

Syntax

```
!DEC$ UNROLL (n)
```

where n is an integer constant from 0 through 255.

The `UNROLL` directive must precede the `DO` statement for each `DO` 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 directive is supported only when option `-O3` (Linux*) or `/O3` (Windows*) is used. The `UNROLL` directive overrides any setting of loop unrolling from the command line.

Currently, the directive 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

```
subroutine unroll(a, b, c, d)
```

```

integer :: i, b(100), c(100), d(100)
!DEC$ UNROLL(4)
do i = 1, 100
    b(i) = a(i) + 1
    d(i) = c(i) + 1
enddo
end subroutine unroll

```

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

Vectorization Support

The directives discussed in this topic support vectorization.

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.

For example, if the expression `j >= 0` is always true in the code fragment below, the `IVDEP` directive can communicate this information to the compiler. This directive informs the compiler that the conservatively assumed loop-carried flow dependences for values `j < 0` can be safely ignored:

Example

```

!DEC$ IVDEP
do i = 1, 100
    a(i) = a(i+j)
enddo

```

Note

The proven dependences that prevent vectorization are not ignored, only assumed dependences are ignored.

The usage of the directive differs depending on the loop form, see examples below.

Example: Loop 1

```

do i
    = a(*) + 1
    a(*) =
enddo

```

Example: Loop 2

```

do i
    a(*) =
    = a(*) + 1
enddo

```

For loops of the form 1, use old values of `a`, and assume that there is no loop-carried flow dependencies from `DEF` to `USE`.

For loops of the form 2, use new values of `a`, and assume that there is no loop-carried anti-dependencies from `USE` to `DEF`.

In both cases, it is valid to distribute the loop, and there is no loop-carried output dependency.

Example 1

```
!DEC$ IVDEP
do j=1,n
  a(j) = a(j+m) + 1
enddo
```

Example 2

```
!DEC$ IVDEP
do j=1,n
  a(j) = b(j) + 1
  b(j) = a(j+m) + 1
enddo
```

Example 1 ignores the possible backward dependencies and enables the loop to get software pipelined.

Example 2 shows possible forward and backward dependencies involving array `a` in this loop and creating a dependency cycle. With `IVDEP`, the backward dependencies are ignored.

`IVDEP` has options: `IVDEP:LOOP` and `IVDEP:BACK`. The `IVDEP:LOOP` option implies no loop-carried dependencies. The `IVDEP:BACK` option implies no backward dependencies.

The `IVDEP` directive is also used for Itanium®-based applications.

Overriding the Efficiency Heuristics in the Vectorizer

In addition to `IVDEP` directive, there are `VECTOR` directives that can be used to override the efficiency heuristics of the vectorizer:

- `VECTOR ALWAYS`
- `NOVECTOR`
- `VECTOR ALIGNED`
- `VECTOR UNALIGNED`
- `VECTOR NONTEMPORAL`

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 and NOVECTOR Directives

The `VECTOR ALWAYS` directive overrides the efficiency heuristics of the vectorizer, but it only works if the loop can actually be vectorized, that is: use `IVDEP` to ignore assumed dependences.

The `VECTOR ALWAYS` directive can be used to override the default behavior of the compiler in the following situation. Vectorization of non-unit stride references usually does not exhibit any speedup, so the compiler defaults to not vectorizing loops that have a large number of non-unit stride references (compared to the number of unit stride references). The following loop has two references with stride 2. Vectorization would be disabled by default, but the directive overrides this behavior.

Example

```
!DEC$ VECTOR ALWAYS
do i = 1, 100, 2
  a(i) = b(i)
enddo
```

If, on the other hand, avoiding vectorization of a loop is desirable (if vectorization results in a performance regression rather than improvement), the `NOVECTOR` directive can be used in the source text to disable vectorization of a loop. For instance, the compiler vectorizes the following example loop by default. If this behavior is not appropriate, the `NOVECTOR` directive can be used, as shown below.

Example

```
!DEC$ NOVECTOR
do i = 1, 100
  a(i) = b(i) + c(i)
enddo
```

VECTOR ALIGNED/UNALIGNED Directives

Like `VECTOR ALWAYS`, these directives also override the efficiency heuristics. The difference is that the qualifiers `UNALIGNED` and `ALIGNED` instruct the compiler to use, respectively, unaligned and aligned data movement instructions for all array references. This disables all the advanced alignment optimizations of the compiler, such as determining alignment properties from the program context or using dynamic loop peeling to make references aligned.

Note

The directives `VECTOR [ALWAYS|UNALIGNED|ALIGNED]` should be used with care. Overriding the efficiency heuristics of the compiler should only be done if the programmer is absolutely sure the vectorization will improve performance. Furthermore, instructing the compiler to implement all array references with aligned data movement instructions will cause a run-time exception in case some of the access patterns are actually unaligned.

The VECTOR NONTEMPORAL Directive

The `VECTOR NONTEMPORAL` directive results in streaming stores on the Intel® Pentium® 4 processor-based systems. A floating-point type loop together with the generated assembly are shown in the example below. For large n , significant performance improvements result on a Pentium 4 system over a non-streaming implementation.

The following example shows the `VECTOR NONTEMPORAL` directive:

Example

```
subroutine set(a,n)
integer i,n
real a(n)
!DEC$ VECTOR NONTEMPORAL
!DEC$ VECTOR ALIGNED
do i = 1, n
  a(i) = 1
enddo
end
program setit
parameter(n=1024*1204)
real a(n)
integer i
do i = 1, n
  a(i) = 0
enddo
call set(a,n)
do i = 1, n
  if (a(i).ne.1) then
    print *, 'failed nontemp.f', a(i), i
    stop
  endif
enddo
print *, 'passed nontemp.f'
end
```

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the Intel® Fortran Language Reference.

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 does 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.

Directives

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.

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

```
!DEC$ NOPREFETCH c
!DEC$ PREFETCH a
  do i = 1, m
    b(i) = a(c(i)) + 1
  enddo
```

The following example is for Itanium®-based systems only:

Example

```
do j=1,lastrow-firstrow+1
  i = rowstr(j)
  iresidue = mod( rowstr(j+1)-i, 8 )
  sum = 0.d0
  !DEC$ NOPREFETCH a,p,colidx
  do k=i,i+iresidue-1
    sum = sum + a(k)*p(colidx(k))
  enddo
  !DEC$ NOPREFETCH colidx
  !DEC$ PREFETCH a:1:40
  !DEC$ PREFETCH p:1:20
  do k=i+iresidue, rowstr(j+1)-8, 8
    sum = sum + a(k )*p(colidx(k ))
    &      + a(k+1)*p(colidx(k+1)) + a(k+2)*p(colidx(k+2))
    &      + a(k+3)*p(colidx(k+3)) + a(k+4)*p(colidx(k+4))
    &      + a(k+5)*p(colidx(k+5)) + a(k+6)*p(colidx(k+6))
    &      + a(k+7)*p(colidx(k+7))
  enddo
```

```
q(j) = sum  
enddo
```

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

Intrinsics

Before inserting compiler intrinsics, experiment with all other supported compiler options and directives. Compiler intrinsics are less portable and less flexible than either a compiler option or compiler directives.

Directives enable compiler optimizations while intrinsics perform optimizations. As a result, programs with directives are more portable, because the compiler can adapt to different processors, while the programs with intrinsics may have to be rewritten/porting for different processors. This is because intrinsics are closer to assembly programming.

The compiler supports an intrinsic subroutine `mm_prefetch`. In contrast the way the `prefetch` directive enables a data prefetch from memory, the subroutine `mm_prefetch` prefetches data from the specified address on one memory cache line. The `mm_prefetch` subroutine is described in the *Intel® Fortran Language Reference*.

Optimizing Applications Glossary

Term	Definition
A	
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	
branch probability database	The database generated by the branch count profiler. The database contains the number of times each branch is executed.
C	
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.
D	
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.
E	
empty declaration	A semicolon and nothing before it.

F	
frame pointer	A pointer that holds a base address for the current stack and is used to access the stack frame.
G	
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>
H	
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>
I	
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.
L	
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.
P	
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.
R	
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.
S	
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
T	
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.
U	
unreachable code	Instructions that are never executed by the compiler.
unused code	Instructions that produce results that are not used in the program.
V	
variable	An optimization in which the compiler renames instances of a variable

renaming	that refer to distinct entities.
----------	----------------------------------

Index

.

.dpi file..... 110, 125, 132

.dyn file..... 110, 125, 132

.hpi file..... 135

.spi file..... 103, 110, 125

.tb5 file..... 135

/

/Qprof-gen compiler option

 using with SSP 139

/Qprof-gen-sampling compiler option

 using with profrun 135

 using with SSP 139

/Qprof-genx compiler option

 code-coverage tool..... 110

 test-priorization tool 125

/Qprof-use compiler option

 code-coverage tool..... 110

 profmerge utility..... 132

 using with profrun 135

 using with SSP 139

/Qssp compiler option

 using with SSP 139

A

accessing arrays efficiently..... 36

accuracy

 controlling..... 156

advanced PGO options 97, 102

aliases 65, 239

aligning data 22, 29

alignment

 example 239

 options 73

 strategy 73, 239

alignment..... 22, 29, 62

alignment..... 73

alignment..... 78

alignment..... 239

ALLOCATABLE

 arrays as arguments 36

 effects of compiler options on
 allocation 65

allocating temporary arrays 190

analyzing

 effects of multfile IPO 82

 programming 2

analyzing applications

Intel® Debugger	3	alignment in vectorization	239
Intel® Threading Tools	3	efficient compilation of	62
VTune™ Performance Analyzer	3	loop blocking	238
analyzing applications	3	operations in a loop body	234
analyzing applications	5	rules for improving I/O performance	41
analyzing applications	9	using efficiently	36
analyzing hotspots	4	assumed-shape arrays	36
application		ATOMIC	
basic block	110	using	209
code coverage	110	automatic	
OpenMP*	174	allocation of stacks	62, 65
pipelining	243	checking of stacks	65
tests	110	optimization for IA-32 systems	59
visual presentation	110	auto-parallelization	
application characteristics	9	diagnostic	230
application performance	9	enabling	228
application tests	125	overview	222
architectures		programming with	223
coding guidelines for	26	threshold	230
argument aliasing	239	auto-parallelization	174
arithmetic precision		auto-parallelization	222
improving	158	auto-parallelized loops	230
restricting	158	auto-parallelizer	
arrays		controls	174, 230

enabling 174
 auto-parallelizer 174
 auto-parallelizer 222
 auto-vectorization 26, 174
 auto-vectorizer 231
 avoid
 EQUIVALENCE statements 46
 inefficient data types 46
 mixed arithmetic expressions 46
 slow arithmetic operators 46
 small integer data items 46
 unnecessary operations in DO loops
 46

B

BARRIER

using 209
 basic PGO options 99
 big-endian data
 conversion of little-endian data to ... 70
 big-endian data 62
 browsing frames using the coverage tool
 110
 buffers
 UBC system 41
 buffers 41

C

cache size intrinsic 48
 changing number of threads 206
 checking
 floating-point stacks 65
 stacks 65
 chunk size
 specifying 221
 clauses
 in parallel region directives 206
 in worksharing construct directives 213
 summary table of 192
 cleanup of loops 238
 code design considerations
 preparing for OpenMP* programming
 185
 code-coverage tool
 coloring scheme for 110
 dynamic counters in 110
 export data 110
 options 110
 options in 110
 syntax of 110
 visual presentation of 110
 code-coverage tool 110

coding	
for efficiency in Intel Fortran	46
guidelines for Intel Architectures	26
combined parallel and worksharing constructs	205
compilation	
efficient	62
optimizing	62
phase	78
compilation	87
compilation units	91
compiler	
intermediate language files produced by	87
compiler reports	
High-Level Optimization (HLO).....	164
Interprocedural Optimizations (IPO)	165
report generation	160
software pipelining.....	166
vectorization	169
compiler reports	160
computing denormals.....	158
conditional parallel region execution	206
controlling	
auto-parallelizer diagnostics .	174, 230
data scope attributes.....	213
inline expansion	90
rounding	154
COPYIN	
summary of data scope attribute clauses	215
correct usage of countable loop	
countable loop.....	237
correct usage of countable loop	236
correct usage of countable loop	237
COS	
SIMD version of	233
countable loop	
correct usage of	236, 237
counters for dynamic profile	108
CPU time	23
creating	
DPI list.....	125
multifile IPO executable	80, 83
multithreaded applications	26
criteria	
for inline function expansion	87
cross-iteration dependencies.....	185
D	
data alignment	22

data format		dependence of data	235
alignment	73, 239	dequeuing	209
dependence	144, 230, 243	derived-type components	36
options	65	determining parallelization	174
partitioning	223	device-specific blocksize	41
prefetching	143, 244	diagnostic reports	190, 230
scope attribute clauses	215	diagnostics	
sharing	174	auto-parallelizer	174, 230
structure	239	OpenMP*	190
type	46, 174, 231	diagnostics	174
data prefetches	249	diagnostics	232
data scope attribute clauses	215	difference operators	203
data types	29	differential coverage	110
dataflow analysis	174, 222	directives	
DEFAULT		commons	29
summary of data scope attribute		dcommons	29
clauses	215	records	29
using	216	sequence	29
deferred-shape arrays	36	structure	29
denormal exceptions	26	directives	29
denormal numbers		directives for OpenMP*	
flush-to-zero	26	ATOMIC	209
denormal numbers	26	BARRIER	209
denormals	158	CRITICAL	209
denormals-are-zero	26, 60		

DO	213	function splitting	99
END DO.....	213	inlining.....	90
END PARALLEL.....	206	disk I/O	
END PARALLEL DO	205	efficient use of.....	41
END PARALLEL SECTIONS	205	DISTRIBUTE POINT	
END SECTIONS.....	213	using	244
END SINGLE.....	213	division-to-multiplication optimization	154
FLUSH.....	209	DO constructs	
MASTER.....	209	order of.....	36
ORDERED.....	209	DO constructs	236
PARALLEL	206	double-precision	
PARALLEL DO	205	numbers	26, 46, 198
PARALLEL SECTIONS	205	dual-core.....	174
PARALLEL WORKSHARE.....	205	Dual-Core Intel® Itanium® 2 Processor 9000	56
SECTION.....	213	dummy arguments	36, 41
SECTIONS	213	dumping profile information	107, 109
SINGLE	213	dyn files	97, 99, 102, 106, 107, 108
WORKSHARE	205	dynamic counters	110
directives for OpenMP*	205	dynamic information	
directives for OpenMP*	213	directory for files.....	102
directory		dumping profile information.....	107
specifying for dynamic information files.....	102	files.....	97
directory	102	resetting profile counters.....	108
disabling		threads	198

- dynamic information 94
- dynamic information 106
- dynamic-information files 99
- E**
- effects of multifile IPO 82
- efficient
 - compilation 62
 - use of arrays..... 36
 - use of record buffers..... 41
- efficient..... 41
- efficient..... 46
- enabling
 - auto-parallelizer 174
 - implied-DO loop collapsing..... 41
 - inlining 90
 - parallelizer 174
 - PGO options 99
 - SIMD-encodings 238
- END PARALLEL DO
 - using 205
- END PARALLEL DO 205
- endian data 70
- enhancing optimization 9
- enhancing performance 9
- environment variables
 - and OpenMP* extension routines .200
 - for auto-parallelization.....228
 - for little endian conversion 70
 - FORT_BUFFERED41
 - OMP_NUM_THREADS206
 - OMP_SCHEDULE221
 - OpenMP* 197
 - PROF_DUMP_INTERVAL 109
 - routines overriding 198
- EQUIVALENCE
 - effect on run-time efficiency46
- example of
 - auto-parallelization228
 - auto-vectorization246
 - dumping profile information..... 107
 - loop constructs236
 - parallel program development 174
 - using OpenMP*203
 - using profile-guided optimization97
 - vectorization239
- exceptions
 - denormal26
- exclude code

code-coverage tool	110	summary of data scope attribute clauses	215
exclude code	110	using	217
execution environment routines	198	floating-point applications	
execution flow	223	comparisons	158
execution mode	200	floating-point arithmetic	
exit conditions	237	array operations	234
explicit-shape arrays	36	on IA-32 systems	154
F		on Itanium®-based systems	156
files		options for	151
.dpi	99, 110, 125, 132	overview	151
.dyn	99, 102, 106, 107, 108, 110, 125, 132	performance	158
.hpi	135	restricting precision of	158
.spi	110, 125	floating-point arithmetic	46, 50
.tb5	135	floating-point arithmetic	158
formatted	41	flow dependency in loops	146
IR	78	flush-to-zero mode	26
object files	80, 83	formatted files	41
OpenMP* header	198	FORT_BUFFERED environment variable	41
real object	87	FTZ mode	26
source	62, 80, 97	function expansion	91
unformatted	41	function order list	103
FIRSTPRIVATE		function splitting	
in worksharing constructs	213	enabling or disabling	99

G

general compiler directives 245, 249

generating

instrumented code 99

processor-specific code 59

profile-optimized executable 99

profiling information 105

reports 160

guidelines

for auto-parallelization 223

for high performance programming .. 2

for IA-32 architecture 26

for improving run-time efficiency..... 46

for profile-guided optimization 94, 102

for vectorization 231, 232

H

helper thread optimization 139

heuristics

affecting data prefetches 244, 249

affecting software pipelining . 243, 244

for inlining functions 87, 89

high performance 1

high performance programming 2, 94

high-level optimizer 143, 160

HLO

reports 164

HLO 143

hotspots 4, 5

Hyper-Threading Technology

parallel loops 224

thread pools 224

using OpenMP* 182

I

I/O

improving performance 41

list 41

parsing 41

performance 41

IA-32 architecture

applications for 143

dispatch options for 59

guidelines for 26

options for 58, 59

options targeting 55, 56

processors for 58, 59, 174

report generation 160

targeting 56, 58

ILO 160

implied-DO loop	46	program.....	94
improving		integer pointers	
floating-point arithmetic precision .	158	preventing aliasing	65
I/O performance.....	41	Intel® architectures.....	26
run-time performance	46	Intel® compiler-generated code	62
improving.....	216	Intel® Core™ Duo processors.....	56
inefficient		Intel® Core™ Solo processors	56
code.....	46	Intel® Debugger	3
initialization values for reduction		Intel® extension routines	200
variables	218	Intel® Itanium® 2 processors	56
inlining		Intel® Itanium® processors	56
compiler directed	90	Intel® Pentium® 4 processors.....	56
developer directed	91	Intel® Pentium® II processors.....	56
inlining.....	46, 62	Intel® Pentium® III processors.....	56
inlining.....	90	Intel® Pentium® Pro processors	56
inlining.....	91	Intel® Pentium® processors.....	56
inlining.....	94	Intel® Threading Tools	3
instruction-level parallelism.....	174	Intel® Xeon® processors.....	56
instrumentation		Intel®-extended intrinsics	48
compilation	97	intermediate language files (IL)	
repeat	102	implementing with version number..	87
instrumentation.....	107	intermediate language scalar optimizer	
instrumented code		160
execution - run.....	97	intermediate results	
generating.....	99	using memory for	41

- intermediate results 41
- internal subprograms 46
- interprocedural optimizations
 - code layout 103
- interprocedural optimizations 50, 62, 77, 90, 94
- interprocedural optimizations 160
- interval profile dumping
 - initiating 109
- intrinsics 249
- introduction to Optimizing Applications 1
- IPO
 - code layout 86
 - generating multiple IPO object files 81
 - issues 79
 - overview 78
 - performance 79
 - reports 165
- Itanium®-based applications
 - auto-vectorization in 174
 - floating point options..... 151, 156
 - HLO 143
 - options targeting 55
 - pipelining for 243
 - report generation 160
 - targeting 56
 - using intrinsics in..... 48
- IVDEP
 - effect in loop transformations 144
 - effect of compiler option on 145
 - effect when tuning applications 143
- L**
- LASTPRIVATE
 - summary of data scope attribute clauses 215
 - using 217
- libraries
 - libintrins.lib 48
 - OpenMP* run-time routines .. 198, 200
- libraries 23, 41
- libraries 78
- libraries 206
- library routines
 - Intel extension 200
 - OpenMP* run-time routines 198
 - to control number of threads 206
- little-endian-to-big-endian conversion. 70
- lock routines 198
- LOOP COUNT
 - and loop distribution 244

loop interchange.....	14	parallelization	174, 223, 233
loop unrolling		reductions	146
limitations of.....	145	sectioning.....	238
support for	245	transformations ..	14, 50, 62, 143, 144, 238
using the HLO optimizer	143, 160	types vectorized	233
loop unrolling.....	232	unrolling	145, 245
loop unrolling.....	242	using for arrays	36
loops		vectorization	233
anti dependency	146	vectorized.....	233
arrays within	234	loops	234
blocking	238, 252	loops	236
body.....	234	loops	244
collapsing.....	41	M	
constructs	236	maintainability	46
count.....	244, 245	manual transformations	17
data dependency	235	master thread	
dependencies	223, 246	copying data in	216
distribution	143, 144, 244	matrix multiplication	
exit conditions.....	237	example of	241
flow dependency.....	146	memory	
independence	146	allocation.....	200
interchange..	14, 17, 26, 143, 144, 241	dependency	144, 145
manual transformation.....	17	layout	26
output dependency	146	memory aliasing	14

- memory file system 41
- misaligned data 239
- mixing vectorizable types in a loop .. 232
- MMX(TM) 231
- multidimensional arrays
 - using effectively 36
- multifile IPO
 - analyzing the effects of 82
 - creating and using an executable for
..... 80, 83
 - optimization 77
 - overview 81
- multithreaded programs 26, 174, 222
- multithreading 196, 223
- N**
- natural alignment 29
- naturally aligned
 - data 62, 73
 - records 73
 - storage 41
- non-unit memory access 14
- NOPREFETCH
 - using 249
- NOSWP
 - using 243
- O**
- obj files 62, 80
- OMP directives 174, 185, 203
- OpenMP*
 - directives 182, 205, 206, 209, 212,
213
 - environment variables 197
 - Hyper-Threading Technology 182
 - parallel processing thread model .. 179
 - pragmas 182
 - run-time library routines 198
 - support libraries 196
- OpenMP* Fortran directives
 - clauses for 192
 - examples of 203
 - features of 185
 - for synchronization 209
 - Intel extensions for 200
 - programming using 185
 - syntax of 190
- OpenMP* Fortran directives 192
- optimal records to improve performance
..... 41
- optimization
 - analyzing applications 9

application-specific	9	overview of	50, 94
hardware-related	9	parallelization	174
library-related.....	9	PGO methodology	95
methodology	5	profile-guided	94
options	50	SSP	139
OS-related	9	support features for	242
reports	160, 242, 243	optimizations.....	1
strategies	9	optimizations.....	50
system-related	9	optimizer report generation.....	160
targeting processors	55	optimizing	
tuning tools	4	applications	9
optimization support.....	1	helping the compiler	14
optimizations		overview	1
compilation process.....	50	technical applications	9
default level of	62	optimizing	1
floating-point.....	151	optimizing performance	3
for specific processors.....	55	options for efficient compilation	62
helper thread	139	OptReport support	160
high-level language	143	ORDERED	
interprocedural.....	77	example of	209
IPO	78	in DO directives.....	213
multiple IPO	77	overview of OpenMP* directives and clauses	192
options for IA-32	56	overflow	65, 151
options for Itanium® architecture....	56	overriding	

- call to a runtime library routine 198
- loop unrolling 245
- software pipelining..... 243
- the threads number 206
- vectorization 246
- overview
 - of data scope attribute clauses..... 215
 - of optimizing compilation 62
 - of optimizing different application types 50
 - of optimizing for specific processors55
 - of parallelism 174
 - of programming for high performance 2
- overview 1
- P**
- packed structures..... 22
- PARALLEL
 - using SHARED clause in..... 220
- parallel construct..... 185
- PARALLEL DO
 - and synchronization constructs 209
 - example of 205
 - SCHEDULE clause..... 221
 - summary of OpenMP* directives and clauses..... 192
 - using COPYIN clause in216
 - using DEFAULT clause in216
 - using FIRSTPRIVATE clause in ...217
 - using LASTPRIVATE clause in.....217
 - using PRIVATE clause in217
 - using REDUCTION clause in218
 - using SHARED clause in220
- parallel invocations with makefile .83, 99
- PARALLEL OpenMP* directive 192, 209, 216, 217, 218, 220
- parallel processing
 - thread model 179
- parallel processing..... 185
- parallel programming 1, 174
- parallel regions
 - directive defining206
 - directives affecting206
 - library routine affecting.....206
- parallel regions 192
- PARALLEL SECTIONS
 - and synchronization constructs.....209
 - example of205
 - summary of OpenMP* directives... 192
 - using COPYIN clause in216
 - using DEFAULT clause in216

using FIRSTPRIVATE clause in ...	217	_PGOPTI_Set_Interval_Prof_Dump	109
using LASTPRIVATE clause in	217	enable	105
using PRIVATE clause in	217	PGO tools	
using REDUCTION clause in	218	code-coverage tool	110
using SHARED clause in.....	220	helper threads	139
PARALLEL WORKSHARE		profmerge.....	132
using	205	profororder	132
parallelism	174, 178, 198, 222	profrun.....	135
parallelization		software-based precomputation....	139
diagnostic	230	test-prioritization tool	125
parallelization	174, 178, 222, 223	PGO tools	110
parallelization	233	pgopti.dpi file	99, 106
passing		pgopti.spi file.....	95, 110, 125
array arguments efficiently	36	pgouser.h header file	105
options to other tools	89	pipelining	
passing.....	36	affect of LOOP COUNT on	244
passing.....	87	for Itanium®-based applications ...	243
performance analyzer	23, 178	pipelining	174
performance issues with IPO	79	pointer aliasing	65
PGO	94, 95	PREFETCH	
PGO API		options used for	149
_PGOPTI_Prof_Dump_And_Reset	108	using	249
_PGOPTI_Prof_Reset.....	108	prefetches of data	
		optimizations for	149

- prefetches of data 249
- preloading 252
- prioritizing application tests 125
- PRIVATE
 - in DEFAULT clause 216
 - in the DO directive 213
 - relationship to REDUCTION clause
..... 218
 - summary of data scope attribute
clauses 192, 215
 - used in PARALLEL directive 206
 - using 217
- processor
 - achieving optimum performance for 26
 - optimizing for specific 55, 58
 - run-time checks for IA-32 systems . 60
 - targeting 55
- processor-based optimizations 55
- processors 55
- processor-specific runtime checks 60
- PROF_DIR environment variable 106
- PROF_DUMP_INTERVAL environment
variable 106
- PROF_NO_CLOBBER environment
variable 106
- prof-gen compiler option
 - using with SSP 139
- prof-gen-sampling compiler option
 - using with profrun 135
 - using with SSP 139
- prof-genx compiler option
 - code-coverage tool 110
 - test-priorization tool 125
- profile data
 - dumping 107, 109
 - resetting dynamic counters for 108
- profile-guided optimization
 - API support 105
 - dumping profile information 108
 - environment variables 106
 - example of 97
 - interval profile dumping 109
 - methodology 95
 - options 99, 102
 - overview 94
 - resetting dynamic profile counters 108
 - resetting profile information 108
 - support 105
- profile-guided optimization 94
- profile-guided optimization 95

profile-optimized code	99, 105	R
profiling		READ
generating information	105	using for little-to-big endian conversion
specifying a summary	102	70
profmerge		real object files
code-coverage tool	110	compiling with
profmerge	132	87
profrun		RECL
.hpi file	135	specifier for OPEN
.tb5 file	135	41
requirements	135	record buffers
SSP	139	efficient use of
profrun	135	41
-prof-use compiler option		record structures
code-coverage tool	110	and alignment
profmerge utility	132	73
using with profrun	135	REDUCTION
using with SSP	139	in the DO directive
program loops	222	213
programs		summary of data scope attribute clauses
high performance	2	192, 215
interprocedural optimization of	77	used in PARALLEL directive
pseudo code		206
parallel processing model	179	using
		218
		variables
		218
		reductions in loops
		146
		report generation
		160
		report software pipelining (SWP)
		166
		resetting
		dynamic profile counters
		108
		profile information
		108

routines		shared variables	218
Intel extension	200	significand.....	154
OpenMP* run-time.....	198	single-precision real.....	26, 46
timing	198	SMP systems.....	222
run-time checks		software pipelining	
processor-specific.....	60	affect of LOOP COUNT on	244
run-time performance		for Itanium®-based applications ...	243
improving	46	optimization	243
slowing down	62	reports.....	166
S		software pipelining	166
sample of timing	23	software pipelining	174
scalar clean-up iterations	239	source code	46
scalar replacement.....	144	specialized code	26, 59, 174
scalars		specifiers	
allocation of	65	/Qoption compiler option	89
SCHEDULE		-Qoption compiler option	89
DYNAMIC	221	specifiers	89
GUIDED.....	221	specifying symbol visibility	74
RUNTIME	221	SSE	
STATIC.....	221	optimizing	26
using in DO directives.....	213	SSE	231
using to specify types and chunk sizes.....	221	SSE2	231
serial execution	185	SSP	
shared scalars.....	203	profun.....	139

using with /Qprof-gen	139	for optimization.....	242
using with /Qprof-use.....	139	for prefetching	249
using with /Qssp	139	for vectorization.....	246
using with -prof-gen	139	parallel run-time	222
using with -prof-use	139	SWP	
using with -ssp.....	139	SWP reports.....	166
SSP	139	using	243
-ssp compiler option		symbol visibility	
using with SSP	139	specifying	74
stacks	62, 65	symbol visibility	74
statement functions	46	symbol visibility on Linux*	74
statements		symbol visibility on Mac OS*	74
in the loop body	234	synchronization	
strategies for optimization	9	constructs.....	209
Stream_LF records	41	synchronization.....	174
Streaming SIMD Extensions	26, 232, 238	synchronization.....	222
stripmining.....	238	T	
structures		targeting.....	56
record	73	targeting optimizations.....	55
subroutines in the OpenMP* run-time library	198	targeting processors	
support		Itanium®.....	56
for loop unrolling	245	Itanium® 2.....	56
for OpenMP*	196	targeting processors	55
		technical applications	9

- testing applications..... 125
- test-prioritization tool
 - examples 125
 - options 125
 - requirements..... 125
 - usage..... 125
- test-prioritization tool 125
- thread pooling 224
- threads
 - changing the number of..... 206
 - parallel processing model for..... 179
 - thread sleep time 200
 - thread-level parallelism..... 174
- threshold control for auto-parallelization
..... 230
- timeout 87
- timing
 - OpenMP* routines for 198
- tool options
 - code-coverage tool 110
 - profmerge 132
 - proforder 132
 - profrun 135
 - test-prioritization tool 125
- tool options..... 110
- tools
 - code-coverage tool 110
 - strategies 4
 - test-prioritization tool 125
 - tuning 4
- tools 110
- transformations
 - loop 144
 - reordering..... 232
- tuning 4
- types of loop vectorized 233
- U**
 - UBC buffers 41
 - unaligned data 29
 - unbuffered WRITES 41
 - underflow 26, 65, 151
 - unformatted files 41
 - unvectorizable copy 232
 - usage rules 83, 185
 - user functions 90, 91
 - using
 - advanced PGO 102
 - auto-parallelization 174, 222
 - dynamic libraries 198

EQUIVALENCE statements	46	vector copy	239
floating-point conversions.....	26	vector dependencies	246
formatted or unformatted files.....	41	vectorization	
implied-DO loops	41	examples.....	239
intrinsics.....	48	options	231
memory.....	41	options for	174
noniterative worksharing SECTIONS	213	overview	231
OpenMP*	203	patterns	235, 246
profile-guided optimization.....	97	programming guidelines.....	231, 232
slow arithmetic operators.....	46	reports	169
timing for an application.....	23	support for	246
unbuffered WRITES	41	vectorizing	
worksharing	206	loops	94, 236
utilities		vectorizing	232
profmerge	132	VOLATILE	
proforder	132	using for loop collapsing	41
utilities	110	VTune™ Performance Analyzer	
V		profrun.....	135
variables		VTune™ Performance Analyzer	3
automatic.....	65	W	
length of.....	41	worker thread.....	196
loop assigns for	217	WORKSHARE	
PGO environment.....	106	using	205
renaming.....	50	worksharing	

directives 205, 213
worksharing..... 174, 192
worksharing..... 205
worksharing..... 206
worksharing..... 213
worksharing..... 222

X

xild80, 83
xilink.....80, 83

Z

zero denormal values 156