



Taking Advantage of Usability Features of the Intel® C++ Compiler 8.1 for Linux*

**by Robert Chesebrough, Technical Consulting Engineer
Max Domeika, Technical Consulting Engineer**

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation Intel on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

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

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

Copyright © 2005 Intel Corporation

Introduction

Software developers compile their applications with the Intel® Compilers primarily for performance reasons. The Intel Compilers employ state-of-the-art compilation technology to enable high application performance on Intel® Architecture. The Intel Compilers offer several advanced optimizations and technologies including Profile Guided Optimization, Interprocedural Optimization, Vectorization¹ and OpenMP*². In addition to performance, developers are affected by a number of secondary attributes in the use of the compiler. For example, compiler compatibility impacts development engineers porting code compiled by one compiler to the Intel compiler because the precise definition of the source language recognized by the two compilers may be slightly different. In this case, developers may be motivated by the performance benefits of the Intel Compilers, but usability features such as compiler compatibility will determine if the port is successful. Other examples of these secondary attributes include compile time, and code size.

The Intel® C++ Compiler 8.1 for Linux* has made great strides in improving in these areas. This paper discusses usability features available in the Intel C++ compiler for improving the secondary attributes of compatibility, compile time, and code size.

Compatibility

Compatibility features reduce the cost of moving from one compiler to another compiler. The types of compatibility features that aid in this reduction are categorized ISO compatibility and strict compiler compatibility.

ISO Compatibility

Two widely used languages for embedded software development are C and C++. Each of these languages has been standardized by various organizations. For example, ISO C conformance is the degree to which a compiler adheres to the ISO C standard. Currently, two ISO C standards are common place in the market: C89³ and C99⁴. C89 is currently the de facto standard for C applications; however use of C99 is increasing. The ISO C++⁵ standard was approved in 1998. The C++ language is derived from C89 and is largely compatible with it; differences between the languages are summarized in Appendix C of the ISO C++ standard. Very few C++ compilers offer full 100% conformance to the C++ standard due to a number of somewhat esoteric features such as exception specifications and the export template. Write code that complies with the ISO standard and use a compiler that enforces ISO conformance to help prevent code from being “locked into” a particular compiler.

The Intel compiler has a unique challenge in maintaining compatibility with other available compilers and functioning with these other compiler’s header files and libraries while also maintaining compatibility with the language standard. In order to accommodate these

needs, the compiler in its default mode is compatible with these other compilers. For example, on Linux*, the Intel C++ compiler is interoperable with the GNU g++ compiler. The Intel compiler also provides options to conform more closely to the ISO standard. The -Wcheck option diagnoses problems in your code that may lead to difficult to debug runtime issues. These options are summarized as follows:

Compilation Mode	Description
Default	Compatibility with entrenched compiler
-ansi	Compatibility with entrenched compiler's definition of ISO conformance
-strict_ansi	Intel C++ Compiler definition of ISO conformance
-Wcheck	Diagnostics for problems that may manifest themselves as difficult to debug runtime issues

G++ Compatibility

The Intel C++ compiler 8.1 for Linux is interoperable⁶ with GNU* g++ 3.2, 3.3, & 3.4, by default. The Intel C++ compiler detects the version of g++ accessible in your search path (specified by the PATH environment variable). In addition, the option, -gcc-version=<Version string> can be used to specify compatibility with the following versions:

Version string	Description
320	G++ 3.2 compatibility
330	G++ 3.3 compatibility
340	G++ 3.4 compatibility

For more information on gcc/g++ compatibility see the Compatibility Whitepaper⁷.

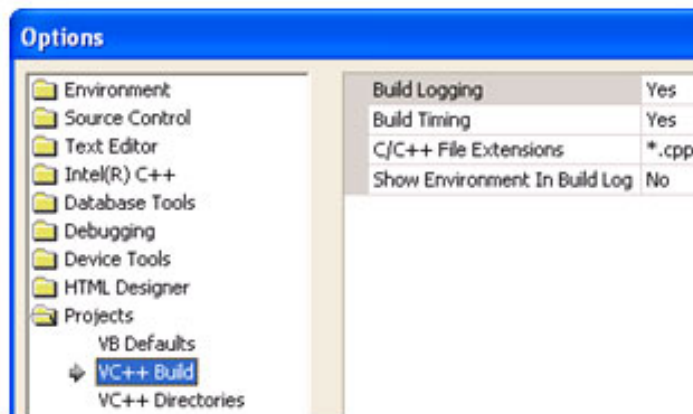
Applications compiled by the Intel C++ Compiler 8.1 for Linux can now be linked with the C++ run-time provided by g++, which includes the standard C++ header files, library, and language support (libstdC++). The Intel Compiler still allows use of the legacy Dinkumware library and headers (libcprts) and libcx and libunwind C++ language support provided with the Intel Compiler (by specifying the cxxlib-icc option).

The two principle advantages of linking with libstdC++ are as follows:

- Greater compatibility to 3rd party libraries and shared objects – The Intel compiler is now interoperable with 3rd party libraries and shared objects which have been compiled by g++ and shipped as binaries.
- Code size - Compatibility with g++ reduces the foot print of user applications shipped to end user Linux systems because the application now does not require the additional shipping of the Dinkumware library. Instead, the application can take advantage of g++ libraries, which are typically already installed on the system.

Compile Time

Compile time is defined as the amount of time required for the compiler to complete compilation of your application. Compile time is affected by many factors such as the size and complexity of the source code, optimization level used, and the host machine speed. On Linux systems, the time command is used to measure compile time. On Windows*, the Visual Studio .NET IDE* provides an option for displaying compile time. The option is specified by setting the following menu option to Yes: Tools/Options/Projects/VC++ Build/Build Timing. Figure 1 displays the timing option:



Also available on Windows is a tool for timing⁸. For best results while timing, keep other activities on the machine down to a minimum.

The Intel Compiler also provides facilities to further analyze compile time. The option, `-mGLOB_tapi`, displays compile time by compilation phase. Example output of compiling with `-mGLOB_tapi` is provided below.

```
TAPI; execution summary for main (covers 20895.0 million clock
ticks)
```

```
TAPI; TOTAL TIME      PHASE TIME

TAPI;  mclocks  %tot  %parent  name [count]

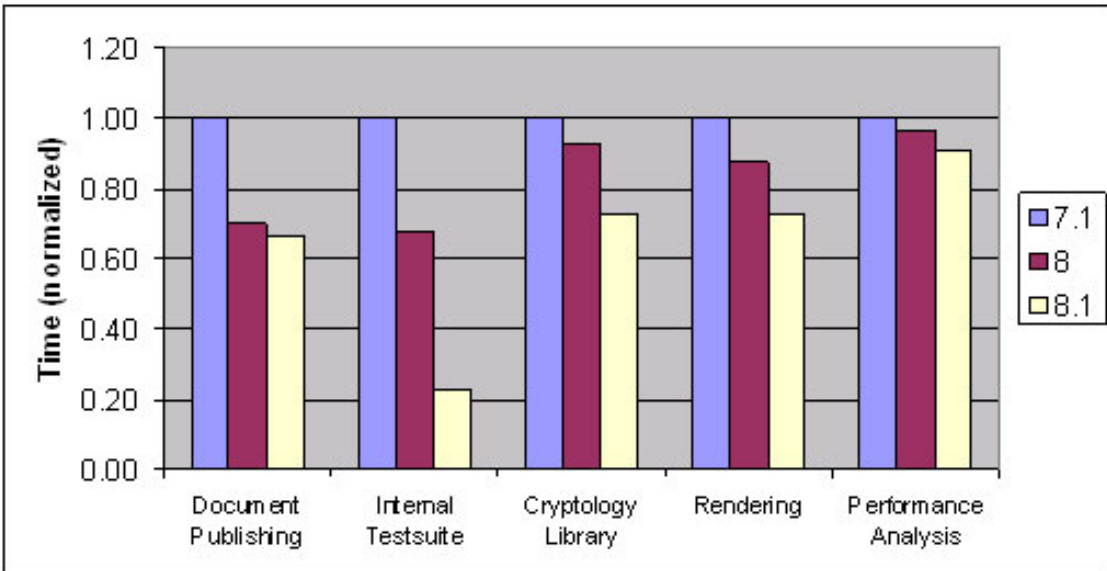
TAPI;  -----  -----

TAPI:   20895.0  100  100      main  [1]

TAPI:   18421.0  88   . 88      Phase2RoutineEnd  [404]
```

8.1 Compile Time Improvements

One of the design goals for the 8.0 and 8.1 compilers was to reduce compile time. Early during the planning phase for the 8.0 compiler, the design team acquired test cases from a number of sources that exhibited unusually long compile times. The team analyzed compilation of these files and found a number of opportunities for improvement. Algorithmic improvements were made to several compiler optimization phases. Figure 2 displays the normalized times for compiling the acquired test cases with the Intel C++ Compiler for Linux, version 7.1, version 8.0, and version 8.1. For example, the compile time using 8.1 on the internal test suite test cases is 23% of the compile time using 7.1.



Precompiled Header Files (PCH)

Precompiled Header Files (PCH) improve compile times in instances where a subset of common header files are included by several source files in your application project. PCH is essentially a memory dump of the compiler after processing a set of headers files that form the precompiled header file set. PCH improves compile time by eliminating recompilation of the same header files by different source files. PCH is made possible by the observation that many source files include the same headers files at the beginning of the source file. PCH has been available for the Intel C++ Compiler for Windows for several releases; for information regarding PCH on Windows, please read the users guide. The options for using PCH with the Intel C++ Compiler for Linux are described in the following table:

PCH Option	Description
-pch	Enable automatic precompiled header file creation/usage
-create_pch	Create precompiled header file
-use_pch	Use precompiled header file
-pch_dir	Name precompiled header directory

The sample code shows a method of arranging header files to take advantage of PCH. First, the common set of header files are included from a single header file that is included by the other files in the project. In the example below, global.h contains the include directives of the common set of header files and is included from the source files, file1.cpp and file2.cpp. The file, file2.cpp also includes another header file, vector. Use the option, -pch, to enable the compiler to create a PCH file during the first compilation and use the PCH file for the subsequent compilations. During compiles with automatic PCH, the compiler parses the include files and attempts to match the sequence of header files with an already created PCH file. If an existing PCH file can be used, the PCH header is loaded and compilation continues. Otherwise, the compiler creates a new PCH file for the list of included files. Pragma hdrstop is used to tell the compiler to attempt to match the set of include files in the source file above the pragma with existing PCH files. In the case of file2.cpp, the use of pragma hdrstop allows the use of the same PCH file used by file1.cpp.

The creation and use of too many unique PCH files can actually slow compilation down in some cases.

```

/* global.h begin */

#include <iostream>

#include <iomanip>

#include <fstream>

#include <cstdlib>

/* end of global.h */

/* file1.cpp begin */

#include "global.h"

```

The following table shows compile time reductions in the sample code below and two applications. These tests were run on a 2.8 Ghz Pentium® 4 system with 512 Megabytes RAM and Red Hat Linux.

Application at -O2	Compile Time Reduction (Version 8.1 Build 20040921Z)
Povray	50%
EON	30%

Parallel Build

A second feature supported by the compiler to reduce compilation times is the use of a parallel build. The Intel Compilers are thread-safe and as such can be used by parallel build tools such as make (with the -j option). The following table shows the compile time reduction building Povray* and EON* on a dual processor system with the make -j 2 command. These tests were run on a dual 3.2 Ghz Pentium 4 system with 512 Megabytes RAM and Red Hat Linux.

Application at -O2	Compile Time Reduction (Version 8.1 Build 20040921Z)
Povray	35%
EON	45%

Code Size

Code size can describe a number of different attributes about your application such as number of instructions in the hot code path, size of the object files, or the size of the executable, and all requisite libraries and shared objects.

An earlier section discussed the benefit of interoperability and the ability of the Intel C++ Compiler 8.1 for Linux to reduce the footprint of a final distributable application by taking advantage of gcc C++ libraries already available on typical Linux systems. Some applications may exhibit an improvement in executable size when using the gcc C++ runtime libraries. For example, the following table shows a reduction of 39% when linking with the gcc C++ libraries instead of the Dinkumware C++ libraries shipped as part of the Intel C++ Compiler and invoked using the `cxxlib_icc` switch.

Executable size for EON compiled with <code>cxxlib_gcc</code> and <code>cxxlib_icc</code>			
Application	Executable Size (bytes) <code>icc -cxxlib-icc</code>	Executable Size (bytes) <code>icc -cxxlib-gcc</code>	Percentage reduction
EON	1,347,185	969,378	39%

To obtain details regarding the hot code path, a developer might use tools such as Intel® VTune Analyzer or the Intel Code Coverage tool. The hot code path is defined as the frequently executed sections of your application. Once these areas are identified they can be targeted for compilation with Interprocedural Optimizations or Profile Guided Feedback to reduce the size of the hot code path and improve instruction efficiency.

Object file size is generally not an important concern. Typically, the sum of the size of the object files will exceed the size of the final executable and so is not a true measure of code size. In addition, different compilers will generate object files with different sizes compiling the same source code. For example, the Intel compiler places symbol debug information directly into an object file, whereas the Microsoft compiler places debug information into a proprietary PDB* file. Object files are also often artificially bloated due to "vague linkage". Vague Linkage refers to a handful of C++ constructs, such as `type_info` objects or template instantiations, which require space in the object file but are not clearly tied to a single translation unit. Most recent linkers on both Linux and Microsoft Windows, cause duplicate copies of these constructs to be discarded at link time. The bottom line is that unless object files are taking up too much disk space on your development system, object file size is more of a matter of perception than a true measure of code size.

Executable size can be important to many developers who wish to distribute their applications with a minimum of media distribution cost or perhaps minimal download time. Reducing executable size may make the difference in having to ship one CD or two or be the difference in an end user deciding to download your product over a 28.8 modem or not.

This paper focuses mainly on how to decrease executable size. In order to measure executable size, executing “ls -al” will provide the number of bytes in the executable. Alternatively, you could invoke the “size” command which will specify both the text and data sizes of the executable. On Windows, File Explorer can display the executable size or alternatively specify the “dumpbin/summary” command which outputs the text and data sizes of the executable. The text segment sizes listed indicate the size of “raw” code while the data sizes listed express the size of your “global” data contained in the executable.

There are two often overlooked compiler switches that can help reduce the size of an executable if a developer has a prior knowledge of the application. One of these switches is “no_cpprt”, which tells the compiler and linker not to include the standard C++ runtime library for those application which are inherently “C” and not C++ applications. The second of these switches is the “fno-exceptions”, which allows normal C++ use but turns off exception handling. This avoids the creation of EH tables which can have a noticeable effect of the size of the executable.

Lets take a look at the effect of no_cpprt on a simple minded “C” based matrix multiplication code, which has a “C” file extension, which follows:

```
// Simple minded matrix multiply

#include <stdio.h>

#include <time.h>

#define NUM 1024

static double a[NUM][NUM], b[NUM][NUM], c[NUM][NUM];

//routine to initialize an array with data

void init_arr(double row, double col, double off, double a[][NUM])

{

    int i,j;

    for (i=0; i< NUM;i++) {

        for (j=0; j<NUM;j++) {

            a[i][j] = row*i+col*j+off;

        }

    }

}
```

```
    }  
}  
  
void multiply_d(double a[][NUM], double b[][NUM], double c[][NUM])  
{  
    int i,j,k;  
    double temp;  
    for(i=0;i<NUM;i++) {  
        for(k=0;k<NUM;k++) {  
            for(j=0;j<NUM;j++) {  
                c[i][j] = c[i][j] + a[i][k] * b[k][j];  
            }  
        }  
    }  
}
```

```
main()  
{  
    clock_t start, stop;  
    int num;  
  
    // initialize the arrays with data  
    init_arr(3,-2,1,a);
```

```

init_arr(-2,1,3,b);

//start timing the matrix multiply code

printf("NUM:%d\n",NUM);

start = clock();

multiply_d(a,b,c);

stop = clock();

}

```

Compiling the above application with the Intel C++ Compiler 8.1 for Linux (build 20040921Z) yields executable reduction of differing amounts as indicated by the following table:

Reduction in executable size for "C" code using "no_cpprt" switch			
Matrix multiply	Itanium(R) Compiler	EM64T Compiler	IA-32 Compiler
	Size in bytes	Size in bytes	Size in bytes
icc	10,756	8,094	9,094
icc -no_cpprt	9180 (14.6% smaller)	6514 (19.5% smaller)	8206 (9.7% smaller)



The second option that can garner nice reduction in executable size is the "fno-exceptions" option. In order to use this option, the developer needs to have a prior knowledge that he will not be using C++ exception handling throughout the application. For the matrix multiplication example, fno-exceptions had little effect, but on the EON benchmark it gave dramatic executable size reductions as demonstrated in the following table

Application	Executable Size (bytes) default compilation (icc)	Executable Size (bytes) compilation (icc - fno-exceptions)	Percentage reduction
EON	969,378	706,843	37%

Conclusion

The Intel Compilers provide a number of features that can be used to obtain maximum application performance; in addition the compiler also provides features that help developers obtain those performance gains more easily - improving areas such as compatibility, compile time, code size, diagnostics, and language features. This paper demonstrated a number of features aimed at providing improvement in these secondary attributes of the compiler.

About the Authors

	<p>Robert Chesebrough is a senior technical consulting engineer with the compiler marketing and technical support group in Intel's Software Products Division. He has been in this position for over four years. He has developed key compiler components for Intel® Software College and is also an instructor for the Intel Software College. He is the main point of contact to the compiler team for several high profile customer engagements. He authored the "Intel® Compiler Black-Belt Users Guide to Undocumented Switches". Robert earned a BS in Physics from the University of New Mexico.</p>
	<p>Max Domeika is a staff software engineer in the Software Products Division at Intel, creating software tools targeting the Intel Architecture market. Over the past 8 years, Max has held several positions at Intel in compiler development which include project lead for the C++ front end and developer on the optimizer and the IA32 code generator. Max currently provides technical consulting for a variety of products targeting Embedded Intel Architecture. Max also provides software tools training serving as an instructor with the Intel Software College. Max earned a BS in Computer Science from the University of Puget Sound and a MS in Computer Science from Clemson University.</p>

Related Links

1. Bik, Aart, Girkar, Milind, Grey, Paul, & Tian, Xinmin, "Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems," Intel Technology Journal, 1st Quarter 2001, http://developer.intel.com/technology/itj/q12001/articles/art_6.htm.
2. OpenMP, <http://www.openmp.org/>.
3. ISO/IEC 9899:1990, Programming Languages C.
4. ISO/IEC 9899:1999, Programming Languages C.
5. ISO/IEC 14882, Standard for the C++ Language.
6. Goodman, Joe, "Interoperability & C++ Compilers," C/C++ Users Journal, March 2004, pg. 44-47.
7. Intel® Compilers for Linux*: Compatibility with GNU Compilers, at <http://www.intel.com/software/products/compilers/techttopics/LinuxCompilersCompatibility.htm>
8. <http://www.microsoft.com/windows2000/techinfo/reskit/tools/existing/timethis-o.asp>

