Building Linux* Kernel with Intel® C++ Compiler for Linux*

White Paper

Feilong Huang

Developer Products Division

Introduction

Intel® C++ Compilers have been in the market for over ten years. More and more software developers are interested in using Intel® C++ compilers to optimize their applications on Windows*, Linux* and Mac OS* X.

As the most essential part of a Linux operating system, the Linux kernel is highly-optimized by kernel developers. Additionally there are many GNU C Language extensions, programming tricks and inline assembly code in it. This makes it challenging for compilers other than GNU C compiler, to compile and optimize the kernel. Building the Linux kernel with the Intel C++ Compiler (icc) is an ongoing project at Intel. The goal is to improve gcc source compatibility with the Intel C++ Compiler and to find opportunities to improve kernel performance.

Red Flag* Software Co., Ltd started to use the Intel C++ Compiler for Linux to compile the Linux kernel in its commercial version of Linux operating system in 2004[1]. While this is not a how-to guide, this document provides extensive hints for the readers to build the Linux kernel using the Intel C++ Compiler.

Compiler Options Compatibility

icc supports most gcc options, but not all of them, therefore, unsupported options must either be replaced with icc equivalent options, or removed. Most of them are not critical and can be removed without breaking the Linux kernel. For example, the following unsupported options can be removed without any risks.

```
-frename-registers
-fno-unit-at-a-time
-msoft-float
-mfloat-abi
-gstabs
```

Ignoring the following options may break the Linux kernel. Replacing them with icc equivalent options is a must.

```
-mno-mmx
-mno-sse2
-mno-sse3
```

Table 1 summarizes some gcc options and support in Intel C++ Compiler 11.1, 12.x and 13.0.[1]

Table 1. Some GCC Options and Support in Intel® C++ Compiler for Linux*

GCC Options	Intel C++ Compiler 11.1	Intel C++ Compiler 12.x	Intel C++ Compiler 13.0
-ffreestanding	Yes	Yes	Yes

[1] URL of Red Flag's announcement in 2004: http://www.redflag-linux.com/news/today/1000000565.html

-fno-asynchronous-unwind-tables	No	No	Yes
-fno-optimize-sibling-calls	No	No	Yes
-fno-strict-overflow	No	Yes	Yes
-fno-unit-at-a-time	No	No	No
-m32	Yes	Yes	Yes
-m64	Yes	Yes	Yes
-maccumulate-outgoing-args	No	No	No
-mfixed-range	Yes[1]	Yes[1]	Yes[1]
-mregparam	Yes	Yes	Yes

[1] Partially implemented.

The Intel C++ Compiler is stricter with syntax checking and reports more warnings than the GNU compiler. As a result, the `-Werror` option may cause the compilation to stop due syntax errors, and therefore should be removed.

To compile the Linux kernel with icc, setting environment variables `HOSTCC` and `CC` to icc is required.

```
make menuconfig
make HOSTCC=icc CC=icc
make modules_install
```

Optionally, user may create a simple wrapper script to remove unsupported compiler options or replace them with equivalent icc options and then invoke icc. In this case, environment variables `HOSTCC` and `CC` need to be set to the name of the wrapper script.

```
make menuconfig
make HOSTCC=<name_of_wrapper> CC=<name_of_wrapper>
make modules_install
```

For an example of wrapper script, see Appendix A.

Source Compatibility

The Intel C++ Compiler does not support a few gcc's builtin functions and IA-64 inline assembly code. Workarounds are needed.

- Inline assembly code

The Intel C++ Compiler supports inline assembly code on IA-32 and Intel 64. IA-64 compilers do not support inline assembly. Instead intrinsics that are C-like functions are recommended. Assembly code on IA-64 needs to be rewritten using corresponding intrinsics. The Intel C++ Compiler documentation includes a mapping of assembly instructions to intrinsics. Most of these changes have been checked into the Linux kernel source tree.

Linux Kernel Source Defects

Some Linux kernel source defects were observed during compilation of the Linux kernel with the Intel C++ Compiler. These defects may have been fixed in the newer Linux kernel already.

- `volatile` attribute

Look at the following code snippet from `include/asm-ia64/spinlock.h`

```
# define _raw_spin_lock(x)                              \
do {                                                    \
      __u32 *ia64_spinlock_ptr = (__u32 *) (x);         \
      __u64 ia64_spinlock_val;                          \
      …                                                 \
      if (unlikely(ia64_spinlock_val)) {                \
            do {                                        \
                  while (*ia64_spinlock_ptr)            \
                        ia64_barrier();                 \
            …                                           \
            } while (ia64_spinlock_val);                \
      }                                                 \
} while (0)
```

In the above code snippet, `ia64_spinlock_ptr` points to a 32-bit volatile data in memory. Without a `volatile` keyword here, the compiler may generate the following asm code (shown in pseudo code) for the while loop when optimization option is turned on:

```
       load ia64_spinlock_ptr, register
label: test register
       jump-if-not-zero label
```

Unfortunately, the above code results in a dead lock of the Linux kernel because the 32-bit data pointed to by `ia64_spinlock_ptr` is not reloaded. The GNU compiler happens to generate the “right” code, which is what kernel developers want:

```
label: load ia64_spinlock_ptr, register
       test register
       jump-if-not-zero label
```

In this case, a “volatile” attribute is needed for the variable `ia64_spinlock_ptr`, to make sure other compilers do not fail.

- `inline` keyword

The `inline` keyword is just a hint to the compiler. Compilers may or may not inline a function declared as inline. For example, in some applications `gettimeofday()` is

done very often like when the kernel is timestamping all transactions. It would help performance if it could be implemented with very low overhead.

One way of obtaining a fast `gettimeofday()` is by writing the current time in a fixed place on a page mapped into the memory of all applications and updating this location on each clock interrupt. These applications could then read this fixed location with a single instruction - no system call required.

There might be other data that the kernel could make available in a read-only way to the process, like perhaps the current process ID. A `vsyscall` is a "system" call that avoids crossing the userspace-kernel boundary.

`vsyscall()` and `do_vgettimeofday()` are in a special page, which can be accessed in user mode.

The Intel C++ Compiler doesn't inline the function `sync_core()`, which is marked as an inline function in `include/asm-x86_64/processor.h`. Thus, the function is compiled as a separate function in the kernel image. `vsyscall()` calls `do_vgettimeofday()` and `do_vgettimeofday()` calls `sync_core()`. The first two functions are called by user applications while `sync_core()` is a kernel function. This will cause a page fault. The following illustrates the call-graph of these 3 functions.

```
      vsyscall( )
          |
do_vgettimeofday( )                  user space
          |                  -------------------------
    sync_core( )                     kernel space
```

gcc inlines `sync_core`. As a result, gcc avoids the page fault. To fix this source defect, adding a `always_inline` attribute to `sync_core()`.

Conclusion

Intel® C++ Compiler is highly compatible with the GNU Compiler. With a small wrapper script and a limited number of temporary source patches, we've successfully compiled Linux kernels 2.4.21, 2.6.9, 2.6.18 and 2.6.32 with the Intel C++ Compilers on IA-32, Intel® 64 and IA-64. Since newer Linux kernels may have fewer source compatibility issues, we recommend that you build newer Linux kernels with the latest Intel C++ Compiler.

Additional Information and Links

Intel® Compilers for Linux: Compatibility with GNU Compilers White Paper*: http://software.intel.com/en-us/articles/intel-c-compiler-for-linux-compatibility-with-the-gnu-compilers/

Appendix A: An Example Wrapper Script

The following is an example of wrapper script.

```
#------------------------------------------------------
# This is a wrapper script for icc 13.0 on Intel 64
#------------------------------------------------------
ARGS=$@
ICCARGS=""

# For loop to change options of icc
for ARG in $@
do
case $ARG in
-Wno-pointer-sign | -Werror )
     ;;

* )
     ICCARGS="$ICCARGS $ARG"
     ;;
esac
done

icc $ICCARGS

exit $?
```

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

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804