Code size optimization using the Intel® C/C++ Compiler

Sergey Maslov
Kittur Ganesh
Software Services Group

Code size optimization is a key factor, especially critical in embedded systems requiring code size reduction at the cost of application speed! Application developed for an embedded system is generally tuned for a particular processor with a finite memory size and hence memory is the main cost component of an embedded product. Directly impacting the memory requirement in an embedded system is the code size of the application, as reduced code size means lesser memory usage and lower cost of the product. In addition, with code size optimized you can add more functionality; improve code quality and therefore reliability as well. It is therefore natural for developers, especially those developing embedded software to optimize their application to achieve a proper trade-off between code size and performance.

The Intel® C/C++ compiler provides many optimization techniques for maximizing application performance. Additionally, it also provides code optimization options and methods to prioritize code size over performance or a balance thereof, depending on the code characteristics of the application.

Table 1 below lists all relevant optimization methods (and corresponding compiler switches) to use for code size optimization using the Intel® C/C++ Compiler. For compatibility, operability or for performance reasons, some compiler options are not applied by the compiler by default, while others when applied may produce varying results (good, bad or neutral) on code size reduction depending on the characteristics of the target code. As such it is recommended that you verify if the optimization method chosen effectively reduces the size of your application binaries while maintaining acceptable performance.

Note: Please consult the product user documentation for more detailed information on the option description, behavior, syntax, and applicable target platforms.

Table 1: Compiler options and techniques for code size optimization

Optimization techniques	
Favor code size reduction over speed	Optimize exception data handling
Minimize code size over speed	Avoid references to compiler specific libraries
Eliminate unused code and data not used during execution	Strip symbols from binaries
Disable inline expansion for all intrinsic functions	Disable auto vectorization
Link Intel-provided libraries dynamically	Avoid unnecessary 16 byte alignment
Use inter-procedural optimization (IPO)	Disable loop unrolling
Disable argument passing in registers than on stack	Disable all loop optimizations
Disable or decrease inlining by compiler	

Code size reduction Methods

1) Favor code size reduction over speed:

Enables optimizations that produce smaller code size than O2

Compiler Option:

-Os (Linux*/OS X)
/Os (Windows*)

Pros: Code size is smaller than that generated with -O2

Cons: Loss of small speed benefit

2) Minimize code size over speed:

Implies –Os and enables optimization for speed while disabling even more optimizations known to increase code size.

Compiler Option:

-O1 (Linux*/OS X)
/O1 (Windows*)

Pros: May improve performance of large (code) applications.

Cons: May lose some speed benefit.

3) Eliminate unused code and data not used during execution:

This technique is called function-level linking which eliminates unused functions/data at link time.

Compiler Option:

-fdata-sections, –ffunction-sections, –Wl,--gc-sections (Linux*/OS X)
/Gy /Qoption,link,/OPT:REF (Windows*)

Pros: Only the code used remains in the executable and all else is stripped out.

Cons: Object code may become slightly larger; requires linker support and may increase link time.

4) Disable inline expansion for all intrinsic functions:

Intrinsic functions may be expanded inline or faster implementation linked in. Disabling this behavior may improve code size reduction.

Compiler Option:

-fno-builtin (Linux*/OS X)
-Oi- (Windows*)

Pros: Both code size of object files and libraries are reduced.

Cons: Various other performance optimizations disabled; slower library code may be used.

5) Link Intel-provided libraries dynamically:

Avoid libraries linked statically by default as doing so can increase code size.

Compiler Option:

-shared-intel (Linux*/OS X)
None (Windows*)

Pros: Performance not affected significantly. All libraries are shared and available for use.

Cons: Dependent libraries will have to be redistributed with the application for it to work. Not beneficial if you need to build/redistribute a single executable.

6) Use inter-procedural optimization (IPO):

Using IPO may reduce code size as it enables dead code elimination and suppresses generation of code for functions always inlined or proved never to be called during execution.

Compiler Option:

-ipo (Linux*/OS X)
/Qipo (Windoows*)

Pros: Depending on the code characteristics, may reduce executable size and improve performance.

Cons: Binary size can increase depending on code/application. For vendors wishing to ship object files as their final product, this method is not recommended.

7) Disable argument passing in registers than on stack:

Disabling prevents compiler from creating extra function entry points which can increase code size. This option is available only on 32-bit architecture.

Compiler Option:

-opt-args-in-regs=none (Linux*/OS X)
/Q-opt-args-in-regs=none (Windows*)

Pros: Reduced code size is possible

Cons: Code size reduction may be small compared to loss of performance

8) Disable or decrease inlining by compiler:

Inlining yields better performing code and also removes function call overhead but increases code size which can be substantial. Disabling inlining over performance reduces code size and is a tradeoff. An alternative to completely disabling inlining is to use the inline-factor value.

Compiler Option:

-fno-inline (Linux*/OS X)
/Ob0 (Windows*)
Note: To reduce inlining use: /Qinline-factor=n (where 0 <= n < 100)

Pros: Disabling or reducing inlining optimization help reduce code size.

Cons: Performance is likely to be sacrificed.

9) Optimize exception data handling:

Compiler creates special sections to unwind/catch an exception. Disabling may reduce code size. Note: -fno-exceptions must not be used for programs throwing exceptions or require std. C++ handling of classes with destructors; -fno-asynchronous-unwind-tables must not be used for programs throwing non C++ asynchronous exceptions.

Compiler Option:

-fno-exceptions, -fno-asynchronous-unwind-tables (Linux*/OS X),
None (Windows*)

Pros: May shrink the binary size by 15% depending on target platform.

Cons: May change program behavior.

10) Avoid references to compiler specific libraries:

Intel specific libraries improve application performance but may increase binary size, depending on code characteristics.

Compiler Option:

-ffreestanding, -gnudefaultlibs (Linux*/OS X)
/Qfreestanding (Windows*)

Pros: Compiler will not assume presence of compiler specific libraries. Also, the –gnudefaultlibs option additionally improves upon –ffreestanding option.

Cons: May sacrifice performance if code in those libraries is in hotspots domain.

11) Strip symbols from binaries:

Omit debugging and symbol information from the executable without sacrificing operability.

Compiler Option:

-Wl,--strip-all (Linux*/OS X)
None (Windows*)

Pros: Size of binaries noticeably reduced.

Cons: Debugging of stripped application is impractical.

12) Disable auto vectorization:

Compiler explores opportunities for using SIMD (SSE/AVX) instructions for performance improvement (auto vectorization). This may involve loop transformations, memory alias check, data alignment checks (leading to generation of multiple versions of the loop body) and thereby increase code size. Disabling this option may help in reducing code size at the cost of performance. This method can increase binary size depending on code characteristics.

Compiler Option:

-no-vec- (Linux*/OS X)
/Qvec- (Windows*)

Pros: Compile time improved significantly.

Cons: Performance of otherwise vectorized loops may take a significant hit. If performance is a factor, you can use this option selectively suppressing vectorization except for performance critical code paths. Since the compiler option –no-vec applies for the source as a whole, selective suppressing of the vectorization can done using #pragma novector at the loop level.

13) Avoid unnecessary 16 byte alignment:

On 32-bit architecture compiler maintains 16 byte alignment which may require additional instructions to adjust the stack on function entries where no stack adjustment would otherwise be needed. This can impact code size, especially in code containing a number of small routines. Use this option only if:

1) Your code doesn't call any other object/library that may be built without this option and may rely on the stack aligned to 16B.
   OR
2) Your code is targeted for architectures that do not have support for SSE instructions and does not need 16B alignment for correctness reasons.

This method can potentially cause correctness issues when linking with other objects/libraries that aren't built with this option, and should be used in context that is well understood

Compiler Option:

-falign-stack=assume-4-byte (Linux*/OS X 32 bit)
None (Windows*)

Pros: Code size may be reduced as extra instructions are not needed to maintain 16 bit alignment when not needed. Performance can improve in some cases due to reduction in instruction count.

Cons: Can cause incompatibility when linked with other objects/libraries that rely on the stack being 16 bit aligned across calls. This option can sometimes increase binary size depending on code characteristics

14) Disable loop unrolling:

Unrolling a loop increases loop size proportionally to the unroll factor. Disabling (or limiting) this optimization may help reducing code size at the expense of performance. This option is already the default at –Os/-O1.

Compiler Option:

-unroll=0 (Linux*/OS X)
/Qunroll:0 (Windows*)

Pros: Code size is reduced. Also, the compiler option "#pragma unroll" can be used to selectively enable unrolling in specific loops rather than enabling at file level too.

Cons: Performance of otherwise unrolled loops may noticeably degrade as it would limit other possible loop optimizations.

15) Disable all loop optimizations:

Loop optimizations often use code size costly techniques and disabling reduces code size at the expense of performance. Binary size may increase due to code characteristics and –vec- is implied.

Compiler Option:

-hlo0 (Linux*/OS X)
/Qhlo0 (Windows*)

Pros: Code size is reduced.

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

*Other names and brands may be claimed as the property of others.
The linked sites are not under the control of Intel and Intel is not responsible for the content of any linked site or any link contained in a linked site. Intel reserves the right to terminate any link or linking program at any time. Intel does not endorse companies or products to which it links and reserves the right to note as such on its web pages. If you decide to access any of the third party sites linked to this Site, you do this entirely at your own risk.
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 applications. Intel may make changes to specifications and product descriptions at any time, without notice.