



Technical Note
Robert Müller-Albrecht
Developer Products Division

Optimized for the Intel® Atom™ Processor with Intel's Compiler



Technical Note: Optimized for the Intel® Atom™ Processor with Intel's Compiler

Introduction	3
Overview	3
Intel® Atom™ Processor Specific Optimization	4
In-Order Scheduler	4
Memory Access Address Generation.....	6
Architecture specific instructions and instruction flow optimizations	6
Movbe – Bi-Endian Support	6
SSE3 – Vectorization / Automatic Parallelization using SIMD Instructions	7
Interprocedural Optimizations	8
Profile-guided Optimization	10
Conclusions	12
References.....	12
Where to find it.....	13

Introduction

This technical note provides a quick overview over some of the key ways in which the Intel® C++ Compiler can be used to maximize the application performance you can achieve on an Intel® Atom™ Processor based platform. For the Intel® C++ Compiler 10.1 and 11.x the option settings and techniques described below are identical whether you employ them on a Microsoft Windows* or Linux* based software stack.

The Intel® C++ Compiler is a highly optimizing compiler for Intel® architecture and compatible processor technologies. It can be installed into an existing Microsoft Visual Studio* build environment or into an existing GNU* GCC installation and used to get additional performance out of your most performance sensitive applications.

Overview

The Intel® Atom™ Processor is a new generation of low-power IA-32 based Intel® processors. Their unique design makes it recommendable to optimize your applications specifically for performance on the Intel® Atom™ Processor. Only then will you be able to take full advantage of the power savings and the full execution performance of this micro architecture.

The most obvious difference to other Intel® processors is that the Intel® Atom™ Processor has an in-order instruction scheduler. This implies that instructions are fed into the instruction pipeline in exactly the order as they are fed to it by the binary code of your application. No instruction re-ordering is done at the processor level. As a result the processor is considerably more sensitive to instruction latencies and dependency stalls caused by poor instruction scheduling by the compiler of your choice.

Furthermore you may want you compiler to be more conservative when it comes to picking specific microcode instructions or atomic instructions depending on the memory access latency or risk for dependency stalls a specific instruction brings with it.

Lastly the simplified math instruction handling on the Intel® Atom™ processor makes for additional power savings, but it also means that the compiler has to give extra attention to the specific code generated so as to not impact execution speed of your code.

In this technical note we will briefly go over the various optimizations employed by the Intel® C++ Compiler to specifically target the Intel® Atom processor. Beyond that we will however also briefly introduce the principles of interprocedural optimization and profile-guided optimization, which both can be very useful for optimizing an application targeting the Intel® Atom™ Processor as well.

Intel® Atom™ Processor Specific Optimization

The compiler optimizations specifically targeting the Intel® Atom™ Processor can be grouped into those related to the in-order instruction scheduler and thus minimizing dependency stalls caused by instruction latencies, those taking advantage of new or preferable instructions added to the instruction set and lastly those who take advantage of some of the advanced features like SSE3 instructions and bi-endianness support the Intel® Atom™ Processor shares with some other Intel® processors. Taking advantage of these features is triggered by using the

`-xL (Linux*) or /QxL (Windows*)`

and since the Intel® C++ Compiler 11.x also the

`-xSSE3_ATOM (Linux*) or /QxSSE3_ATOM (Windows*)`

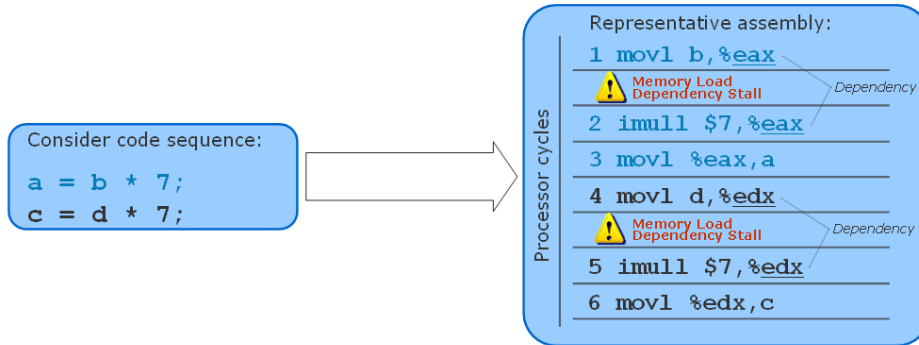
optimization switch.

In-Order Scheduler

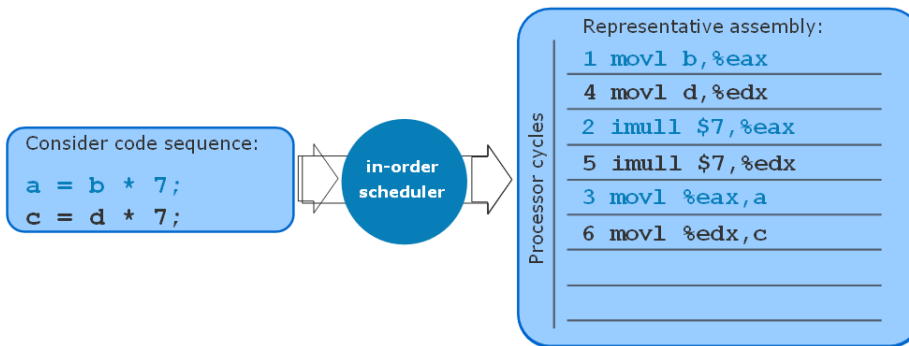
Instructions are fed into the instruction pipeline in exactly the order as they are fed to it by the binary code of your application. No instruction re-ordering is done at the processor level. As a result the processor is considerably more sensitive to instruction latencies and dependency stalls caused by poor instruction scheduling. On other Intel® architecture based processors an out-of-order scheduler unit inside the processor corrects instruction scheduling issues that could affect performance and automatically determines the best order in which to process and execute instructions. On the Intel® Atom™ Processor this additional ordering unit has been removed to allow for decreased power consumption. Although this has great benefits in terms of processor footprint, heat generation and power consumption, it also means that the compiler thus now has the job to ensure best possible instruction ordering.

Technical Note: Optimized for the Intel® Atom™ Processor with Intel's Compiler

A simplified example would be a basic set of multiplications that if coded in the wrong order could cause a severe memory access dependency stall.



By seeing this and interlacing the two multiplications the delay due to memory access dependencies can be reduced drastically



These are the kinds of decisions that the in-order scheduler in the compiler needs to make over and over again to come up with the best possible instruction ordering. The compiler internally models the instruction pipeline of the Intel® Atom™ Processor and re-orders the assembly instructions it generates for the best possible fit into the actual instruction pipeline of the processor.

Technical Note: Optimized for the Intel® Atom™ Processor with Intel's Compiler

Memory Access Address Generation

Following the same idea of minimizing latencies and thus speeding up execution flow the compiler will generate code that is best suited to speed up the process of generating and handing over the address for memory accesses. It does this by using LEA assembly instructions instead of ADD instructions whenever possible, thus minimizing memory accesses during the address generation phase and minimizing the need to incur memory access latencies, since memory access on most any platform tend to be among the most cycle consuming actions in the execution flow.



These address generation optimizations are being employed in a late optimization pass so as many remaining latencies as possible can be caught and affected by this approach.

Architecture specific instructions and instruction flow optimizations

To minimize dependency stalls it is further a good idea to take advantage of the availability of microcode or atomic instructions depending on which fits better into the instruction pipeline and helps to avoid latencies. Furthermore the Intel® Atom™ Processor supports byte-wise division and multiplication. Using byte-wise division and multiplication instructions where the full width of an integer operation is not needed allows for parallelization and using registers more efficiently.

Movbe – Bi-Endian Support

The Intel® Atom™ Processor supports the MOVBE instruction which allows swapping the high and low bits of a long value during a move. This is generally very useful for supporting embedded environments with the Intel® Atom™ processor where the processor needs to interact with peripherals of different endianness as you frequently have it especially in network and storage applications. The MOVBE instruction can however also be used for simply arithmetic transformations and this is something that the Intel® C++ Compiler can take advantage of. The `-xSSE3_ATOM (/QxSSE3_ATOM)` compiler option in the Intel® C++ Compiler from version 11.0 on automatically generates this MOVBE instruction whenever it is useful. To explicitly turn off the generation of this instruction you can use the `-minstruction=[no]movbe (/minstruction=[no]movbe)` option switch.

Technical Note: Optimized for the Intel® Atom™ Processor with Intel's Compiler

SSE3 – Vectorization / Automatic Parallelization using SIMD Instructions

When using the `-xSSE3_ATOM (/QxSSE3_ATOM)` compiler option the Intel® C++ Compiler will automatically use SSE3 instruction for Single Instruction Multiple Data execution of loop structures, thus parallelizing execution of such loops considerably improving execution flow. This vectorization optimization can be very powerful especially for multimedia applications and application with large data streams associated with graphics, gaming or encryption. You can find further details on this in the **The Software Vectorization Handbook** - *Applying Multimedia Extensions for Maximum Performance* [\[1\]](#)

Interprocedural Optimizations

Interprocedural Optimization (IPO) allows the compiler to analyze your code to determine where you can benefit from specific optimizations. In many cases, the optimizations that can be applied are related to the specific architectures.

The compiler might apply the following optimizations:

- inlining
- constant propagation
- mod/ref analysis
- alias analysis
- forward substitution
- routine key-attribute propagation
- address-taken analysis
- partial dead call elimination
- symbol table data promotion
- common block variable coalescing
- dead function elimination
- unreferenced variable removal
- whole program analysis
- array dimension padding
- common block splitting
- stack frame alignment
- structure splitting and field reordering
- formal parameter alignment analysis
- indirect call conversion
- specialization
- Passing arguments in registers to optimize calls and register usage

Technical Note: Optimized for the Intel® Atom™ Processor with Intel's Compiler

IPO is an automatic, multi-step process: compilation and linking; however, IPO supports two compilation models: single-file compilation and multi-file compilation.

Single-file compilation, which uses the `-ip` (Linux* OS) or `/Qip` (Windows* OS) option, results in one, real object file for each source file being compiled. During single-file compilation the compiler performs inline function expansion for calls to procedures defined within the current source file.

The compiler performs some single-file interprocedural optimization at the default optimization level: `-O2` (Linux*) or `/O2` (Windows*); additionally some the compiler performs some inlining for the `-O1` (Linux*) or `/O1` (Windows*) optimization level, like inlining functions marked with inlining directives.

Multi-file compilation, which uses the `-ipo` (Linux) or `/Qipo` (Windows) option, results in one or more mock object files rather than normal object files. (See the *Compilation* section below for information about mock object files.) Additionally, the compiler collects information from the individual source files that make up the program. Using this information, the compiler performs optimizations across functions and procedures in different source files. Inlining is the most powerful optimization supported by IPO.

Compilation

As each source file is compiled with IPO, the compiler stores an intermediate representation (IR) of the source code in a mock object file, which includes summary information used for optimization. The mock object files contain the IR, instead of the normal object code. Mock object files can be ten times larger, and in some cases more, than the size of normal object files.

During the IPO compilation phase only the mock object files are visible. The Intel compiler does not expose the real object files during IPO unless you also specify the `-ipo-c` (Linux*) or `/Qipo-c` (Windows*) option.

Linkage

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 mock objects must be linked with the Intel compiler or by using the Intel linking tools. The compiler calls the linkers indirectly by using aliases (or wrappers) for the native linkers, so you must modify make files to account for the different linking tool names.



Caution

Linking the mock object files with `ld` (Linux*) or `link.exe` (Windows) will cause linkage errors. You must use the Intel linking tools to link mock object files.

During the compilation process, the compiler first analyzes the summary information and then produces mock object files for source files of the application.

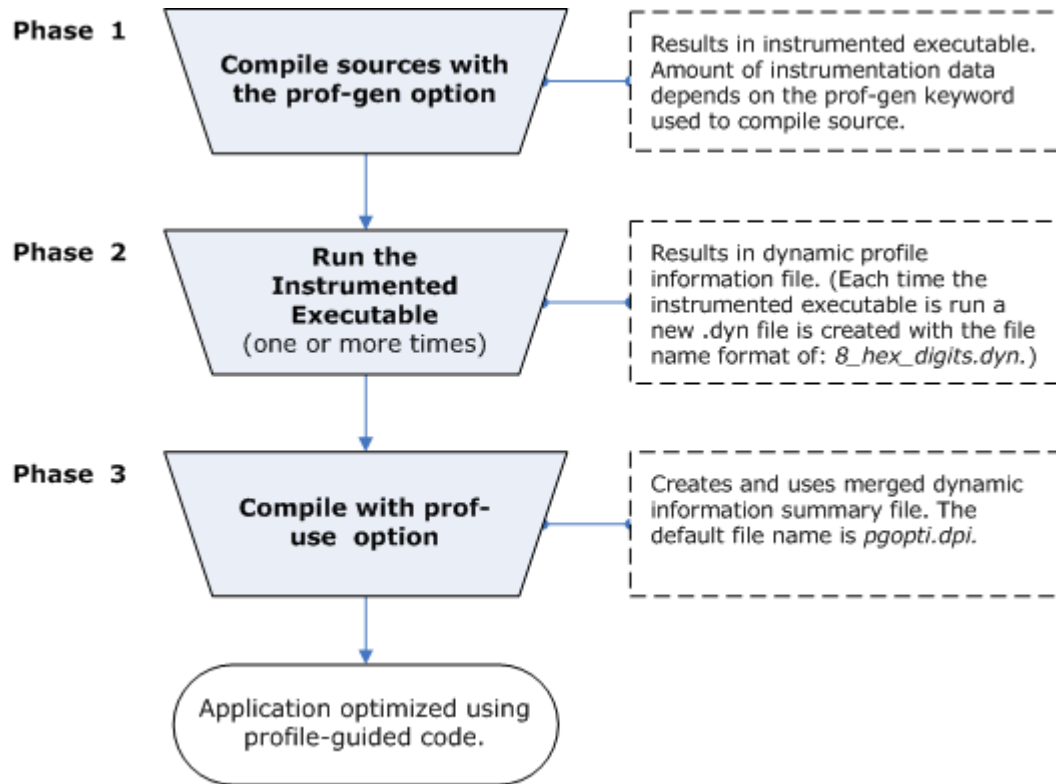
Profile-guided Optimization

Profile-Guided Optimization consists of three phases (or steps):

Generating instrumented code by compiling with the `-prof-gen` (Linux* OS) or `/Qprof-gen` (Windows* OS) option when creating the instrumented executable.

Running the instrumented executable, which produces dynamic-information (`.dyn`) files.

Compiling the application using the profile information using the `-prof-use` (Linux*) or `/Qprof-use` (Windows*) option.



Below are the central compiler options needed for the phases of profile-guided optimization:

Linux*	Windows*	Effect
<code>-prof-gen</code>	<code>/Qprof-gen</code>	Instruments a program for profiling to get the execution counts of each basic block. The option is used in phase 1 (instrumenting the code) to instruct the compiler to produce instrumented code for your object files in preparation for instrumented execution. By default, each instrumented execution creates one dynamic-information (<code>.dyn</code>) file for each executable and (on Windows OS) one for each DLL invoked by the application. You can specify keywords, such as <code>-prof-gen=default</code> (Linux*) or <code>/Qprof-</code>

gen:default (Windows).

The keywords control the amount of source information gathered during phase 2 (run the instrumented executable). The prof-gen keywords are:

Specify default (or omit the keyword) to request profiling information for use with the prof-use option and optimization when the instrumented application is run (phase 2).

Specify srcpos or globdata to request additional profiling information for the code coverage and test prioritization tools when the instrumented application is run (phase 2). The phase 1 compilation creates an spi file.

Specify globdata to request additional profiling information for data ordering optimization when the instrumented application is run (phase 2). The phase 1 compilation creates an spi file.

If you are performing a parallel make, this option will not affect it.

-prof-use

/Qprof-use

Instructs the compiler to produce a profile-optimized executable and merges available dynamic-information (dyn) files into a pgopti.dpi file.

The dynamic-information files are produced in phase 2 when you run the instrumented

When you compile with prof-use, all dynamic information and summary information files should be in the same directory (current directory or the directory specified by the prof-dir option). If you need to use certain profmerge options not available with compiler options (such as specifying multiple directories), use the profmerge tool. For example, you can use profmerge to create a new summary dpi file before you compile with the prof-use option to create the optimized application.

Technical Note: Optimized for the Intel® Atom™ Processor with Intel's Compiler

Conclusions

The Intel® C++ Compiler offers a set of advanced optimization techniques that can be employed to get the best performance out of the Intel® Atom™ Processor. Combining advanced optimization techniques like interprocedural optimization and profile-guided optimization with the Intel® Atom™ Processor specific optimizations triggered with the `-xSSE3_ATOM` (Linux*) or `/QxSSE3_ATOM` (Windows*) it is possible to achieve an additional performance boost over applying purely target hardware agnostic optimization techniques.

Using the Intel® C++ Compiler will enable not only getting great power savings out of the Intel® Atom™ Processor, but also getting the performance and end-user experience out of your application that is expected from Intel's products.

References

[1] The Software Vectorization Handbook

Applying Multimedia Extensions for Maximum Performance

by Aart J.C. Bik

http://www.intel.com/intelpress/sum_vmmx.htm

Technical Note: Optimized for the Intel® Atom™ Processor with Intel's Compiler

Where to find it

The Intel® Debugging Extensions are a part of the Intel® C++ Express Solution Set . It can be purchased, or an evaluation copy can be downloaded, from the Intel® Software Development Products Web pages (<http://www.intel.com/software/products/>). Customers have access to product updates and product support through the following Web pages:

- Intel Support and Downloads:

<http://www.intel.com/support/>

- Intel Software Development Products Support:

<http://www.intel.com/software/products/support/>

- Intel Software Development Products Self Help:

<http://www.intel.com/support/performance/tools/index.htm>

- Intel® Software Network Discussion Forums:

<http://softwareforums.intel.com/ids>

- Intel Premier Support:

<http://premier.intel.com/>

Technical Note: Optimized for the Intel® Atom™ Processor with Intel's Compiler

For product and purchase information visit:

www.intel.com/software/products

Intel, the Intel logo, Intel. Leap ahead. and Intel. Leap ahead. logo, Pentium, Intel Core, and Itanium 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.

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.

Copyright © 2009, Intel Corporation. All Rights Reserved.

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101