Itanium® 2 Processor Software Optimization: Introduction

By David Levinthal

**Introduction**

First in a series of five articles on optimizing software for the Itanium® 2 processor family.

1. The Foundation
2. Performance Monitoring Capabilities
3. Data Blocking and Multi Level Cache Usage
4. Use of Constrained Event Collection
5. Visual Inspection of Compiler-Generated Assembly

The software development cycle requires a code optimization phase once basic functionality has been established. Describing the effective execution of this phase using Intel® Software Development Tools is the objective of this document. It is intended to serve as an introductory guide for software developers and application engineers to optimally use their time in extracting the best performance from Intel® Itanium® processors.

**Assumptions**

The process starts with the assumption that the algorithm has been reasonably planned, designed and executed. It can however assist even in the determination that a new design might be required.

- It is assumed that the program compiles, at least at the Od level and runs to completion with correct results. Some statements about debugging code will be made but that is not the focus of this document.

- It is also assumed that the platform being used for the performance analysis is not constrained by inadequate hardware. For example, there must be sufficient physical memory such that page faults are not dominating the applications' performance. Further, that the memory is configured to deliver data at the maximum rates attainable by the chipset, i.e. that it is properly interleaved and is capable of the highest transfer rates. A third hardware limitation may arise from slow peripheral devices (disk drives, video, network, CD-Rom). All such limitations should be addressed before embarking on a program of software optimization as they will mask and mislead the software performance analysis. The Intel® VTune™ Performance Analyzer has a Counter Monitor mode that is extremely useful in monitoring the rates of page faults and can assist in ensuring that you start with a correctly configured system.

Memory bandwidth can be measured with a standard benchmark like "stream" and compared with maximum results attained on similar systems to ensure that the system has optimal DIMMs and that they are correctly interleaved.
It is further assumed that the performance of the application is not limited by the performance of 3rd party libraries and system calls to the OS. This should be established first to ensure that the result of the optimization effort will result in a real reduction in the "wall clock" execution time.

**Using Intel® Tools to Aid Optimization**

**Intel® Compilers**

The Intel® Compilers have many options to enable the developer to improve the performance of their applications. These options are thoroughly described in the various compiler users guides (by language) distributed with the compiler or found in the self-help area on the Intel® Software Development Tools website (<Link: www.intel.com/software/products>). Please refer to those documents if any of the compiler options discussed here are not familiar. The basic optimization levels in order of the aggressiveness of the optimizations attempted are:

- Od (debug/minimal),
- O2 (default),
- O1 (aggressive optimization constrained by executable size, may be best suited for server and branch dominated applications) and
- O3 (maximum optimization, recommended for loop dominated applications and cases where constraints of O1 can be lifted).

In addition to these optimizations that act on single modules, there are additional compiler options to cause:

- function inlining within a single module (IP),
- inlining between modules and whole program optimizations (IPO) and
- profile guided feedback (PGO) which uses data collected during a training execution of the application to better optimize a second pass optimization.

At times in the code development cycle the developer has the computer explain exactly what it is doing. Most commonly this occurs in the process of debugging an application, but is also extremely important during the optimization phase. Therefore from the compiler’s perspective these two processes are very similar as they rely on the compiler providing additional information to the tools (debugger or performance analyzer) to navigate through the execution of the application.
Frequently, application optimization results in the execution order of the machine instructions diverging quite substantially from the order of the source code from which they were derived. If load instructions are invoked earlier in the execution stream, the processor will have to wait less time for that data delivery when the math instructions are invoked. If the data can be requested sufficiently in advance, the math instructions can execute without any waiting. Due to the complex instruction ordering that results from the optimization by the compiler it can be extremely difficult to debug an optimized piece of code. It is critical to ensure that the algorithm produces correct results at low optimization where the machine code instruction sequence is closely aligned with the source code sequence that the developer knows. This same order scrambling can make performance analysis extremely complex as the performance monitoring events (ex: a cache miss event) are in response to the machine instructions and their order, not the order that might be expected from the source sequence.

Simply invoking the most aggressive optimizations on all the modules of an application may not succeed: the compiler may fail, by failing to produce an object file, or failing at runtime, or producing "incorrect" results. Such an attempt may also expose race conditions and inadequately constrained calculations in the program that were unseen at safer, more sequential optimization levels. A more conservative approach may achieve the goal of reaching the desired performance level more quickly.

**Intel VTune Performance Analyzer**

Amdahl's law on optimization always wins: you can at most increase performance in proportion to the number of consumed CPU cycles being used by the part of the code you are optimizing. This immediately indicates the method of a first pass optimization; optimize only those functions and modules that consume a significant fraction of the total CPU execution time. A performance analyzer must be used hand in glove with the aggressive compiler optimization options to ensure the most direct route to a well-optimized application.

The Intel VTune Performance Analyzer performs a rich variety of data collection and display options that allow a development team to quickly focus on those aspects of the application that will yield performance improvements. Two of the very useful modes of the VTune Analyzer, the Call Graph and Event Based Sampling modes, are very closely connected with the IP/IPO and O1/O3 compiler optimization flags and can be used to guide their use.

The first step in optimizing the application is identifying those functions and subroutines that consume significant numbers of CPU cycles. On Itanium processor-based systems, collecting the number of cycles where the pipeline is stalled, as accumulated in the performance event Back_End_Bubble.all, also reveals the most
obvious optimization opportunities. Many of these stalled cycles can be recovered by aggressive optimization that will result in better data prefetching, parallelization and reordering of instructions. These are all strategies that the Intel Compilers employ to improve performance and in doing so, reduce the numbers of stalled cycles.

The VTune analyzer allows the developer to drill down into the target system and see where CPU cycles are spent in a tree of increasing complexity. The tree starts by displaying all the applications, DLL's/Shared objects, and drivers running on the machine. Presumably the developer's application should appear among the most significant CPU cycle consumers. By double clicking with the mouse on a desired application the contributing functions and subroutines will be revealed if the application, DLL/Shared Object or driver has been compiled to include debug symbolic data (/Zi or -g). These in turn can be further decomposed to show the CPU usage by source line or disassembly if the source files are available to the analyzer.

This process immediately reveals the dominant functions and subroutines. These are the obvious modules to compile with the more aggressive optimizations. By viewing the data collected at the source code level, a second important distinction can be determined. Applications dominated by complex branching, frequent with enterprise-class server applications, need to minimize the application size to make the best use of the instruction caches. These applications usually are best optimized with the O1 compiler optimization level which has been tuned for this purpose in the Intel Compilers. Generally, applications dominated by loops do not suffer from these limitations, and can be optimized without excessive consideration of executable size. For these kinds of applications the O3 optimization will invoke the most aggressive optimization strategies that the Intel Compilers widely recommend.

A VTune Analyzer Call Graph analysis will reveal the "hot" code paths - the sets of functions and subroutines are most actively invoked. In a large application letting the compiler search for all possible inlining opportunities may result in the build time becoming unwieldy, so the call graph data allows the user to identify the most critical call stack chains and leaf functions in their application. These functions are natural targets to profit from inlining. Because the compiler’s analysis of interprocedural inlining options grows exponentially with the number of modules it is asked to consider, at this early stage of the optimization process you should limit the modules it considers to those in the most active call stacks/branching structures. Again the use of the Event Based Sampling source view data needs to be considered. Because inlining increases the executable size, applications dominated by branches inlining should only be used in the most intensely used call stacks. With looping applications, inlining can be much more freely invoked.

The lists of functions and subroutines identified by the VTune analyzer with the two methods just described can be further optimized with the profile guided feedback options of the Intel Compilers. In this two-pass process, the
The compiler instruments the chosen modules to collect performance data during their execution on a training data set. This data helps provide better insight into the best optimization strategies, branching structures, executable layout, and other techniques that this detailed information enables. It is important that the training data set represents the spectrum of data sets the code is to consume, or the profile-based optimizations may result in less optimal performance on some data sets.

By following the process described above the developer can identify the dominant optimization opportunities and the simplest strategy of using the compiler to resolve them. At this point the power of the VTune analyzer and the wealth of information it can uncover with the performance monitoring hardware available on Intel® processors can be applied to the applications' performance. While application profiling with the CPU cycle counter reveals where time is being spent, profiling with the full spectrum of the performance monitoring events will reveal why cycles are being spent there. Often, if cycles are being consumed ineffectively the exact cause can be identified.

On 32-bit Intel architectures, the VTune Analyzer Tuning Assistant Wizard can automatically collect the relevant data and advise on optimization. On Itanium processor-based systems the developer performs the data collection. To assist in this process the VTune analyzer contains two detailed documents discussing the Itanium® 2 Processor architecture, performance monitoring and the interpretation of the performance monitoring data. These documents are distributed as PDF files in the top directory of the directory tree during installation of the VTune analyzer.

**Programming Language Impact**

The Itanium architecture-based processors support an explicitly parallel instruction syntax that allows for parallel execution of independent instructions. The large register sets further enhance this. These aspects achieve their greatest impact through the use of the software pipelining and the rotating register schemes which the hardware supports. These hardware features allow the compiler to schedule many iterations of loops simultaneously, if the iterations of the loops are independent of each other. Thus the stores from one iteration are scheduled concurrently with the math from the next, as well as latency of the data loads from yet another. This creates an enormous number of independent instructions that can be executed in parallel, thus benefiting from the full power of this architecture. It allows the load instructions for a particularly high-level language iteration to be scheduled far in advance of the consuming math operations. The memory access latency can be completely absorbed by the intervening instructions from other high-level language loop iterations.

These hardware capabilities result in one of the fundamental differences in how programming languages impact performance on Itanium processor-based systems. Fortran assumes that all arrays and all function arguments
are distinct and do not address identical areas of memory. This means that the compiler never concerns itself with pointer disambiguation unless the user explicitly declares that data spaces are aliased to each other with an "equivalence" declaration. The situation is exactly the opposite in C and C++ where it is assumed as part of the language standard that pointers alias to each other, and the compiler must assume that the data areas that pointers reference will overlap in some manner.

Consider the DAXPY loop where the arrays A and B are passed as arguments to the function:

For C or C++

For(I=0; I<len; I++)A[I] = A[I] + Y*B[I];

And in Fortran

Do I = 1,len
    A(i) = A(i) + Y * B(i)
Enddo

In C/C++, the arrays A and B must be assumed to overlap by default thus that A[I] may be identical to B[I+1] and thus the loop cannot be unrolled and the two iterations executed independently. In FORTRAN however, because the arrays are assumed to be non-overlapping, the loop can be unrolled and the independent iterations can be executed simultaneously with the software pipelining hardware. If the arrays in fact do overlap and an incorrect result is calculated, this is a programming error by the developer. The Intel® FORTRAN Compiler unrolls such a DAXPY 8 fold, producing code that is limited by the bandwidth of data delivery. All C/C++ compilers must schedule each iteration of the loop to completion (loads, math and stores) prior to starting the next iteration in order to ensure correct results. A small degree of software pipelining is possible with speculative loads but the depth of the pipeline is very limited as is the resulting performance.

On Itanium processors alias ambiguation is one of the dominant sources of sub-optimal performance as it completely disables the large register set, the software pipelining and the parallel execution that those features enable. This situation can be quickly identified with the VTune analyzer as it results in a disproportionate number of stalled pipeline cycles that can be counted with the Back_End_Bubble.ALL performance event.

The Intel® C/C++ compilers support a wide variety compiler options, pragmas and language extensions that allow the developer to tell the compiler that the pointers can be treated as disambigous, that they never
overlap. The options cover the range of indicating that all pointers can be assumed independent (/Oa, -fno_alias), to only those indicated with the "restrict" keyword, to even the level of the pointers within a single loop. These allow the compiler to aggressively schedule the instructions, unroll loops as appropriate and take advantage of the full power of Itanium architecture.

**Software Pipelining, Rotating Registers and Pointer Disambiguation**

To comprehend the critical importance of the pointer disambiguation on Itanium processors, it is essential to understand the role of loop independence to achieving high performance. To illustrate this, we will use a few small loops to explain the best way the Itanium architecture can be used to hide the latency of data access and the functional units. The Fibonacci series illustrates the rotating register mechanism used by software pipelining.

The Fibonacci series is

\[ \text{fib}(i) = \text{fib}(i-1) + \text{fib}(i-2) \]

and the assembler to do this is simple.

 Mov r33 = 1  (put the value 1 in r33..fib(0) )
 Mov r34 = 1  (put the value 1 in r33..fib(1) )

Loop:  (simply a line statement label)
 Add r32 = r33, r34  (r32 = r33 + r34..compute fib(i)=fib(i-1)+fib(i-2) )
 St [r10] = r33, 4  (store r33 at the address in r10 then increment the address by 4 bytes)
 Br.ctop.sptk Loop

Each time the br.ctop instruction is executed, the base pointer, which identifies which physical register the symbol "r32" identifies, is decremented. Thus, what was accessed by "r32" is now found in "r33", what was in "r33" is now addressed through "r34". On each successive iteration through the loop, the data of the series is shifted from r32 to r33 and r33 goes to r34 enabling the next element to be calculated with exactly the same expression. In particular the add and the store instruction can be executed on the same cycle since these instructions are independent and the add that created the value in r33 occurred on the previous iteration thereby hiding the functional unit’s latency.

Next consider the loop that sums the elements of an array.
For(I=0; I<len; I++) sum+= a[I];

This is encoded as

Loop:

(p16) ldfs f32 = [r10],4  (single precision FP load)

(p22) fma f40 = f38,f1,f42  (floating point multiply add, f1 = 1.0 so f40 = f38 + f42)

br.ctop.sptk Loop:

Assume the loop starts with all the FP registers initialized to zero, r10 contains the address of "a" and only p16 is set to true. The ldfs instruction uses the array address stored in r10 and the value is loaded to f32 and the address is incremented by 4 bytes. On starting the loop p16 is true. All other predicates (like p22) are false. For the first 6 iterations of the loop, only the load instructions are executed. After 6 iterations, a[0] is accessed by looking in f38, which is f32 after having been rotated 6 times. The predicate p16 has rotated over to p22 so that both the ldfs and the fma can now be invoked. This is very convenient because it takes 6 cycles for FP data to be loaded from the L2 cache. The net result is we have hidden the L2 cache latency by using the rotating registers to software pipeline the loads, using 6 iterations of rotation to hide the latency. In fact we are also hiding the 4 cycles required for the fma, as we are constructing 4 partial sums and rotating them through f39 through f42, moving the new value to f38 with each fma.

Finally consider the DAXPY loop, which clarifies the importance of pointer disambiguation.

For(I=0; I<len; I++)a[I]+= X*b[I];

Encoded as follows. (only the FP registers rotate here and assume X is in f28)

Mov r28 = r32  (copy base address of a into r28)

Loop:

(p16)   ldfs f32 = [r32],4  (load a)

(p16)   ldfs f40 = [r33],4  (load b)

(p22)   fma f50 = f46,f28,f38  (f50 = f46*f28 + f38 = b*X + a)

(p26)   stfs [r28] = f54,4  (store the new value of a back and increment the address for the next element)

br.ctop.sptk Loop:

Here we used the rotating register and the software pipelining to absorb the 6 cycle latency of each of the two FP loads and the 4 cycles of the fma. The loop can be executed in a single clock cycle so we get a result per clock cycle once the first ten iterations have been completed. This only works if all the elements of a can be
processed independently, as we have 10 results in flight at any given time, all at various levels of completion. This is ONLY possible if the compiler knows that the addresses for the arrays a and b are independent. If in fact \( b[I+1] \) were actually \( a[0] \) then we would have to complete the calculation of the new value of \( a[0] \) before starting on the calculation of \( a[1] \). In such a case, we could not use the software pipelining hardware, nor the rotating registers. Instead of getting a result every clock cycle we would get a result every ten clock cycles. The function would take ten times longer! The compiler MUST schedule a completely sequential calculation if it is not told that \( a \) and \( b \) are distinct, so it is critical to disambiguate pointers.

This particular example also illustrates another effect caused by ambiguous pointers. With the large register files available on Itanium processors, most loops will run significantly faster if they are unrolled as well as pipelined. Automatic unrolling of loops by the compiler, however, also requires that the pointers used in the loop are disambiguated. If not, the compiler cannot unroll the loop any more than it could pipeline it. Unrolling allows two main benefits: it creates a larger list of independent instructions that the compiler can schedule in creative ways, making greater use of the parallel execution that the architecture supports, and the load-pair instructions can be invoked which can deliver two (successive) pieces of data with a single instruction.

Unrolling the DAXPY by two could be encoded as follows:

\[
\text{Mov } r28 = r32 \quad \text{(copy base address of a into r28)}
\]
\[
\text{Add } r29 = r28, 4 \quad \text{(save } a[1])
\]

Loop:
\[
\text{(p16) } \quad \text{ldfps } f32, f39 = [r32],8 \quad \text{(load } a[0] \text{ and } a[1])}
\]
\[
\text{(p16) } \quad \text{ldfps } f47, f54 = [r33],8 \quad \text{(load } b[0] \text{ and } b[1])}
\]
\[
\text{(p22) } \quad \text{fma } f62 = f53,f28,f38 \quad \text{(} f62 = f53*a[0] + f38 = b[I]*X + a[I])
\]
\[
\text{(p22) } \quad \text{fma } f67 = f60,f28,f45 \quad \text{(} f67 = f60*a[0] + f45 = b[I+1]*X + a[I+1])
\]
\[
\text{(p26) } \quad \text{stfs } [r28] = f66,8 \quad \text{(store the new value of } a[I] \text{ increment address by 8)}
\]
\[
\text{(p26) } \quad \text{stfs } [r28] = f71,8 \quad \text{(store the new value of } a[I+1] \text{ increment address by 8)}
\]
\[
\text{br.ctop.sptk Loop:}
\]

The loads, fma's and stores can be issued in one cycle. Asymptotically the DAXPY can deliver 2 results per cycle. If the pointers \( a \) and \( b \) are not resolved by the developer and it has to be executed sequentially, there would only be one result every ten cycles, a factor of twenty times slower!
Note: There are some cases where the compiler can disambiguate pointers but it requires whole program optimizations where every use of the pointers and every call stack can be analyzed by the compiler. Even in such cases a few key words/compiler options can make that a lot easier and gives the compiler a better starting point.

These issues affect all compiler architectures, so there are always gains if they are addressed. On Itanium processors the issue is critical, as it removes opportunities of using the software pipelining, large register files and parallelism inherent in the architecture.

**Conclusion**

Many optimizations will automatically be enabled with aggressive compiler flags. However, with a small amount of performance analysis, architectural insight and targeted code modification, the software developer can greatly improve the resulting application performance. Understanding and resolving pointer ambiguation is important on all architectures and will always improve performance, but on Itanium architecture-based processors the payoff can be enormous, and help release the full power of this new architecture.

*Introduction to Microarchitectural Software Optimization for Itanium 2 Processors.*

*Intel® C/C++ Compiler Users Guide*

*Itanium Processor Software Specifications*