**Black Belt Itanium® Processor Performance: Performance Monitoring Capabilities (Part 2 of 5)**

Submitted by [David Levinthal...](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/user/335758) on Mon, 03/05/2012 - 19:03

Categories:

* [Intel® Itanium® Processors](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/search/site/?f%5b0%5d=im_field_topic%3A36783)

**by David Levinthal**

**Introduction**

This paper, second in a series of five articles on optimizing Itanium® processor family software, focuses on the unprecedented performance monitoring capabilities of the Itanium® processor. The future articles will cover data blocking and multi level caches, the use of opcode matching and data ear events in performance analysis, visual inspection of assembler in the Intel® VTune™ analyzer and a variety of other topics. See the other parts of this series for more information:

* [The Foundation](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-the-foundation-part-1-of-5)
* [Performance Monitoring Capabilities](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-performance-monitoring-capabilities-part-2-of-5)
* [Data Blocking and Multi Level Cache Usage](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-data-blocking-and-multi-level-cache-usage-part-3-of-5)
* [Use of Constrained Event Collection](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-use-of-constrained-event-collection-part-4-of-5)
* [Visual Inspection of Compiler-Generated Assembly](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-visual-inspection-of-compiler-generated-assembly-part-5-of-5)

In this discussion we again make the initial assumptions about the basic program performance:

* The algorithm has been reasonably planned, designed and executed.
* The program compiles, at least at the Od level and runs to completion with correct results.
* The platform being used for the performance analysis is not constrained by inadequate hardware.
* The performance of the application is not limited by the performance of 3rd party libraries and system calls to the OS.

**Performance Through Advanced Architectural Features**

The software developer gains an inside view of program execution that has never before been visible. Taking advantage of these capabilities is best accomplished with a precise methodology.  
  
The Itanium processor provides software developers a platform with a wide variety of opportunities to achieve remarkable software execution performance. Achieving these performance levels will usually require a methodical approach to software optimization that incorporates the advanced architectural features of the Itanium processor family and fully exploits them. This effort is facilitated significantly by the design of the performance-monitoring unit that supports several unique features and allows the developer unprecedented insights into the interaction of the application with the microarchitecture. The development and optimization process is thereby streamlined but requires that the use of the compiler and a performance analyzer be tightly coupled throughout the development process. This document serves as an introduction to this methodology.  
  
The Itanium processor architecture is well described in other documents and it is assumed that the reader is familiar with its features. Reference documents can be found [online](https://web.archive.org/web/20121105222531/http:/developer.intel.com/design/itanium/arch_spec.htm) with particular attention to the [Intel® Itanium® Processor Reference Manual for Software Development and Optimization](https://web.archive.org/web/20121105222531/http:/developer.intel.com/design/itanium2/manuals/251110.htm). Included here is a short discussion of the core pipeline, software pipelining and the use of rotating registers.  
  
As in all microprocessors, the core pipeline of the Itanium processor executes the stream of instructions that make up an application. To do this it must ensure that the instructions and the required data are dispatched to the functional units in a synchronized manner so that the required results can be computed. When the required instructions, data or computing resources are for any reason unavailable the pipeline will stall, and wait until everything required is available. Known as a pipeline stall, this is a standard feature of all microprocessors. Their minimization is often the main objective of software optimization.  
  
The Itanium processor core pipeline is made of two asynchronous parts. The front end collects the stream of instructions from the instruction cache and formats them for consumption by the processor back end. They are staged in an intermediate instruction buffer that allows the two parts of the pipeline to proceed asynchronously. The back end is responsible for the data, instruction and functional unit synchronization and for stalling the instruction execution whenever it is required. Due to this two-stage structure the only stalls that cause any impact to the steady flow of computations are the stalls in the pipelines' back end. Stalls in the front end are only relevant if they are of a duration where the back end drains the instruction buffer and is finally forced to wait until the front end can deliver more instructions for execution. This structure lies at the heart of the optimization methodology and allows a developer to improve the real performance execution in an unprecedented surgical manner.  
  
The performance-monitoring feature known as cycle accounting is significant in the optimization process. The performance-monitoring units (PMUs) count the occurrences of the performance monitoring events. The event, "CPU\_CYCLES" should be self-explanatory and having the PMU count this event will allow the user to determine how many CPU cycles were consumed during program execution. What is unique to the Itanium processor family is that the cycles where the core pipeline is stalled can also be counted, and can even be broken up into architecturally distinct, non-overlapping components. As the different components require different mechanisms to address and remove the sources of the core pipeline stalls, this cycle accounting-based optimization methodology removes all guesswork about what is required to improve the performance of a given algorithm.  
  
The cycle accounting tree starts with the total number of cycles where the core pipeline was stalled for any reason. The performance event, "BACK\_END\_BUBBLE.ALL" occurs whenever the pipeline stalls, thus counting this event accumulates the total number of back end pipeline stalls that occur during the monitoring time period.  
  
Start with a program that executes to completion and gives correct answers when run on Itanium-based architecture. The first step in the optimization process is to identify the functions and subroutines that consume a significant fraction of the CPU cycles used by the execution. The VTune Performance Analyzer does this with the event-based sampling mode by sampling on the performance event "CPU\_CYCLES" on a r egular basis and recording the instruction pointer. As the Itanium processor performance-monitoring unit can collect data on up to four events in a given run, collecting data also on the events "BACK\_END\_BUBBLE.ALL" at the same time can prove useful. The VTune analyzer will therefore sample the two events independently and record the IP at regularly sampled intervals during the accumulation of the two events. Stalled cycles are frequently eliminated simply by raising the optimization level of the compiler, thus functions and subroutines that have significant numbers of stalled cycles can represent easy optimization opportunities.  
  
The VTune analyzer can collect data on any number of performance events, grouping them so the data can be collected in as few data runs as possible. It will display the modules in the application sorted by the number of counts of the event selected, so the developer can quickly make a list of the most significant modules rated by both cpu\_cycles and back\_end\_bubble.all. At this point the application should be rebuilt after changing the build process so that these significant modules are all compiled at O3 (or O1 in the case of branch dominated server applications) with the Intel® Compilers to ensure that the binaries are properly optimized to make detailed studies worthwhile.

**Using Pointers**

At this point the developer may want to investigate the use of ambiguous pointers in these modules. If the functions have distinct pointers passed as arguments or through methods it is important to let the compiler know that there are no ambiguities. In C and C++ the compiler must assume that pointers can be overlapping if there is any possibility of that happening. The only pointers that can be disambiguated are those created locally where the references can be clearly identified to be distinct (example: two successive calls to malloc) and those that are declared to the compiler to be distinct with the "restrict" keyword. Otherwise compiler flags must be used. There are a wide range of pointer disambiguation flags from -Oa or -fno\_alias which declare all pointers in the module to be distinct to various flags applying to ansi types, input arguments only and so on. See the [Intel® C/C++ Compiler Users Guide](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/)for more information on this subject. Also look at the discussion at the end of [Black Belt Itanium® Processor Performance: Foundation (Part 1)](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-the-foundation-part-1-of-5). Before proceeding, be sure that the significant modules are being compiled with more aggressive compiler flags (O3 or O1 if performance is dominated by branches).  
  
If the optimization process is to result in a substantial performance improvement, the sum of the cycles consumed by the “significant modules” should account for at least 50% of the total cpu cycles. As Amdahl's law points out, the total amount of time spent in a section of code being optimized limits the performance improvement that can be achieved in the total program.  
  
This list may also include any modules with a sufficiently large number of stall cycles that a substantial reduction would result in a noticeable performance improvement.

**Pipeline Stalls and Cycle Accounting**

Once you reach a reasonable starting point, you can take advantage of the detailed breakdown of CPU time that can be collected on Itanium systems. This breakdown of the sources of pipeline stalls can be translated into different optimization strategies, depending on the exact nature of the dominant source of stalls. The total number of cycles where the core pipeline back end was stalled for any reason (thus not executing instructions) is accumulated by the event BACK\_END\_BUBBLE.ALL  
  
This event can be decomposed into five (sometimes six) pieces. The decomposition is exact as a given stalled cycle is assigned to one and only one of these sources. This is because the sources are prioritized and the cycle is assigned to the highest priority reason should there be more than one reason for the core pipeline back end to stall on a given cycle. The basic decomposition is:

*BACK\_END\_BUBBLE.ALL = BE\_FLUSH\_BUBBLE.ALL + BE\_L1D\_FPU\_BUBBLE.ALL + BE\_EXE\_BUBBLE.ALL + BE\_RSE\_BUBBLE.ALL + BACK\_END\_BUBBLE.FE*

The five components are due to:  
  
1) BE\_FLUSH\_BUBBLE.ALL: Stall cycles due to flushing the pipeline due to a mispredicted branch or an exception. Note this will not include any cycles consumed by the exception handler, for when the exception handler is running the pipeline is not stalled.  
  
2) BE\_L1D\_FPU\_BUBBLE.ALL: Stall cycles due to either the L1 data cache micropipeline or the floating point unit micropipeline requiring the core pipeline to wait.  
  
3) BE\_EXE\_BUBBLE.ALL: Stall cycles due to the EXE stage. These are mostly due to functional units having to wait because input data has not yet been delivered. This can be due to either latency of the memory sub system or long latency functional units that are being used in a chained fashion.  
  
4) BE\_RSE\_BUBBLE.ALL: Stall cycles required for the Register Stack Engine to function. If a new set of registers must be allocated for local use (typically in the chain of a function call stack) and an insufficient number are still free, then previous register frames (from higher up in the call stack) must be moved out to the backing store to free up the required registers. Similarly when the call stack is being rewound, registers that were pushed out to the backing store must be restored. This occurs automatically and the pipeline is stalled to allow this to happen. This event counts these cycles. And finally  
  
5) BACK\_END\_BUBBLE.FE: This is a "sub event" of BACK\_END\_BUBBLE.ALL. It accumulates the cycles where the pipeline back end is stalled due to a lack of instructions available for processing. This is usually due to a branch misprediction or exception requiring instructions that were not readily available.  
  
The prioritization is based on how far downstream in the pipeline the condition is detected. If the pipeline is going to be flushed (due to a branch misprediction) then a simultaneous stall due to the execution unit waiting for valid data delivery is irrelevant: the instruction requiring the data is not going to be executed, it is in the wrong code path.  
  
A further decomposition is possible as the event BE\_FLUSH\_BUBBLE.ALL can be split exactly into two prioritized sub-components: BE\_FLUSH\_BUBBLE.ALL = BE\_FLUSH\_BUBBLE.XPN + BE\_FLUSH\_BUBBLE.BRU For the analysis and optimization approach discussed here, this deeper distinction will not be used.  
  
You can find a graphical display of the organization of the Itanium processor's performance monitoring events in the "[Introduction to Microarchitectural Software Optimization for Itanium Processors](https://web.archive.org/web/20121105222531/http:/software.intel.com/sites/default/files/m/d/4/1/d/8/219348_software_optimization.pdf)."

**Initial Responses to Back End Pipeline Stalls**

Stalls due to branch mispredictions and exception handling (be\_flush\_bubble) and stalls due to lack of instructions delivered by the front end (back\_end\_bubble.fe) tend to be related. The stalls from lack of instructions are usually due to a mispredicted or unpredicted (exception) branch, as the correctly predicted branches cause the hardware to prefetch the needed instructions with sufficient advance timing to avoid front end stalls. This correlation translates into a common strategy being applied to improve the performance. The compiler needs to change the layout of the binary so that the predictions are correct or that the needed code pieces are adjacent in memory so that their access can make better use of the cache structure of the processor. This affects both the instruction cache misses and a more efficient use of the ITLB recourses as the hot code segments will be gathered to use fewer ITLB entries.  
  
This is done with the profile-guided optimization that can be invoked with the Intel Compilers. A profile-guided compilation is a two-step process. The compiler builds an instrumented binary that can collect data on branches, memory usage, code flow during execution and many other things and then when the developer runs the instrumented binary on a representative data set a data file is created with the results of branches, memory usage, conditional assignments and all of the information the instrumentation can provide to the next stage. The compiler then reads this data file during a second compilation and the data is used to change the layout of the binary and the optimization strategies that the compiler applies.  
  
The classic example of this is an if-else block. All compilers assume that the code is written in the form:

|  |  |  |
| --- | --- | --- |
| 1 | If (dominant flow condition) | |
| 2 | {execute default code} |

|  |  |
| --- | --- |
| 3 | else(less likely flow) |
| 4 | {execute more rare instructions} | |

This results in the compiler organizing the code so that the fall through behavior (i.e. no branches executed) is to invoke the “default” code. Execution of the “rare” code will require a branch to get to the code and then a branch to get back to the main code stream flow.  
  
If this behavior is not in fact the dominant execution flow then the profile guided feedback will automatically result in the binary being laid out as if the logic of the else if were reversed and the “default&r dquo; and “rare” flows were swapped. This results in a reduction of the branch mispredictions but more importantly the code sections are positioned in memory for smoother flow through the instruction cache and fewer stalls arising from a lack of prepared instructions for back end consumption (i.e. back\_end\_bubble.fe).  
  
The Itanium processor family instruction set supports a wide variety of instruction prefetch hints associated with branch instructions. Profile guided feedback will guide the compiler's choice in using these hints to maximize the execution of the branching stream.  
  
Stalls due to the register stack engine arise from the compiler scheduling algorithms that consume too many register resources. This rare occurrence, while significant, can easily be addressed. RSE stalls are usually associated with complicated, intricate loops, recursive algorithms or complex call stack chains that are traversed with great frequency. The OS can compound the impact on the programs' throughput, as the invocation of the RSE engine can cause the OS to swap out the application, increasing the consumed system time. Consequently it may be important to respond to RSE stalls at a lower threshold than other types of backend stalls. The response by the developer to avoid RSE stalls depends on the exact scenario, so this discussion will give a few examples from which you can generalize.  
  
Programs with long intricate loops (meaning many instructions in the loop as opposed to the trip count) can require a very large number of registers to contain all the intermediate results and data addresses. Breaking the loop into several simpler loops, and/or using data blocking to reduce bandwidth pressures, will allow the compiler to schedule the computation using fewer registers. This will remove the need to push registers from further up the call stack out to the backing store in order to have a large number available for allocation.  
  
A related strategy would be to lower the optimization of low CPU cycle usage routines that are in the call stack path. High optimization usually results in more registers being used, the standard trading space for speed. Consequently a lower level of optimization on the intermediate routines in a call stack and the initialization routines may result in a larger number of registers being available in the cpu intensive kernels where they are truly useful.  
  
Profile guided feedback (PGO) and Interprocedural Optimizations (IPO) can also help. Inlining functions will reduce the use of registers because the input/output registers are not required for passing arguments and local registers can be reused. If this backfires, complicating the parent function when the inline code overwhelms the resources, it should be treated as a compiler bug.  
  
The dominant sources of back end pipeline stalls in most applications tend to be related to memory access. Such stalls tend to dominate the cycles collected by the events BE\_EXE\_BUBBLE and BE\_L1D\_FPU\_BUBBLE. These events can be broken down into sub events but the user should be aware that the subevents are not prioritized so double counting can occur. The results can virtually always be untangled reasonably well. The user should consult the [Introduction to Microarchitectural Optimization for Itanium Processors](https://web.archive.org/web/20121105222531/http:/software.intel.com/sites/default/files/m/d/4/1/d/8/219348_software_optimization.pdf) distributed with the VTune analyzer to see a more detailed discussion.  
  
BE\_EXE\_BUBBLE can be decomposed into two dominant sub components, be\_exe\_bubble.grall and be\_exe\_bubble.frall. These are the stalls due to accessing integer and floating-point data respectively. The be\_exe\_bubble.grall can be further decomposed with the be\_exe\_bubble.grgr event which is the stall cycles due to data access stalls associated with long latency integer instructions (variable shift, multi media instructions, etc..not data loads) being chained together with insufficient intervening instructions to absorb the 3 cycle latency for these instructions. This tends to be a very rare situation.  
  
BE\_L1D\_FPU\_BUBBLE monitors stall cycles assigned to the L1 data cache micropipeline and the floating point unit micropipeline which are coupled to the core pipeline's back end. The two micropipelines stall in response to very different situations. So the developer’s response must also be flexible.  
  
The floating point units' micropipeline can stall if certain floating-point exceptions are detected. These tend to be dominated by the use and generation of denormalized floating point values. This in turn is usually associated with single precision floating-point calculations. With the Intel compilers the developer can either suppress the denormalization with a "flush to zero" compilation (-ftz) or by changing the precision of the data types used to doubles or real\*8, etc.  
  
Considerably more complex are L1 data cache micropipeline stalls, which. are due to various congestions in the data deliver paths from either the L1 or L2 data caches. These range from various queues being saturated to the DTLBs needing updates to provide the virtual to physical address translations. There are a large number of subevents that might need to be monitored to figure out the exact cause.  
  
In order to determine the nature of data access stalls, so that they can be worked around, it is usually necessary to use more traditional architectural events (level 2 cache misses, etc.) to decompose the exact source of the execution inefficiency. To do this it is critical to have an estimate of the performance penalty for the occurrences of these events. In most cases these values can be used to estimate the impact, however this should not be done blindly as the processor can do many things in parallel, so the estimate may badly overestimate the impact.  
  
One very useful technique to ensure that the correct events are being addressed is to verify, with the VTune analyzer, that the architectural events are correlated to the same instructions as the pipeline stall cycle events. This can be done by "drilling down" into the application with the GUI. It requires that the application be compiled with symbolic debug information (-Zi or -g) and the source be available. The correlation can be seen in the source view (if the debug information generated by the compiler is adequate) or in the disassembly view if there is any doubt.

**Conclusion**

We discussed an introduction to using the Itanium processor's performance monitoring capabilities for software development and optimization. This subject will be continued in Part 3 of this series, [Data Blocking and Multi Level Cache Usage](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-data-blocking-and-multi-level-cache-usage-part-3-of-5).

**Related Links**

* [Introduction to Microarchitectural Software Optimization for Itanium Processors](https://web.archive.org/web/20121105222531/http:/software.intel.com/sites/default/files/m/d/4/1/d/8/219348_software_optimization.pdf)
* [Intel® C/C++ Compiler Users Guide](https://web.archive.org/web/20121105222531/http:/software.intel.com/en-us/)
* [Itanium Processor Software Specifications](https://web.archive.org/web/20121105222531/http:/developer.intel.com/design/itanium/arch_spec.htm)
* [Intel® Itanium® Processor Reference Manual for Software Development and Optimization](https://web.archive.org/web/20121105222531/http:/developer.intel.com/design/itanium2/manuals/251110.htm)

**About the Author**

David Levinthal is a lead Itanium® Processor software performance support engineer for the VTune™ Performance Analyzer program. He joined Intel in 1995 working for the Supercomputing Systems Division and then the Microprocessor Software Labs. He has been a lead software support engineer on the Itanium Processor Family since 1997. Prior to joining Intel he was a Professor of Physics at Florida State University. He has received the DOE OJI award, the NSF PYI award and a Sloan foundation fellowship.