**Black Belt Itanium® Processor Performance: Use of Constrained Event Collection (Part 4 of 5)**

Submitted by [David Levinthal...](https://web.archive.org/web/20121020132428/http:/software.intel.com/en-us/user/335758) on Mon, 03/05/2012 - 19:04

Categories:

* [Intel® Itanium® Processors](https://web.archive.org/web/20121020132428/http:/software.intel.com/en-us/search/site/?f%5b0%5d=im_field_topic%3A36783)

**by David Levinthal**

**Introduction**

This paper, fourth in a series of five articles on optimizing software for the Itanium® processor family, focuses on the sources of execution inefficiency and how they can be converted into optimization opportunities. For an introduction to this topic and additional information see the other parts of this series:

* [The Foundation](https://web.archive.org/web/20121020132428/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/20121020132428/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/20121020132428/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/20121020132428/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/20121020132428/http:/software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-visual-inspection-of-compiler-generated-assembly-part-5-of-5)

**Constraining Performance Counters with Operation Codes**

The performance monitoring units (PMUs) of a microprocessor count the occurrences of architecturally defined events. In general these events can be caused by a wide variety of originating instructions. A unique feature of the Itanium processor family is that the PMUs can be constrained to increment only when the architectural event is instigated by a particular instruction or class of instructions. With the aggressive scheduling and reordering of instructions by highly optimizing compilers it can be difficult to untangle the association of the events with the operations and data structures. By constraining the PMU’s by operation code (opcode matching) this can be clarified greatly and an even more precise understanding of the interaction of the programs’ execution with the microarchitecture can be achieved.  
  
The analysis of the event frequencies during the execution of a program provide insight into execution efficiency of the program. We learn how effectively the program interacts with and takes advantage of the microarchitectural features. In practical terms what is usually learned are the sources of execution inefficiency and how they can be converted into optimization opportunities. To accomplish this, the clearest understanding of the interaction of the program, its algorithm and data structures, and the microarchitecture must be established. To this end there are unique features in the Itanium processor families’ PMUs that allow an unprecedented level of detail into these interactions. In this discussion, constraining the PMUs to only count events when they are instigated by particular opcodes will be explored.  
  
The simplest use of opcode matching is to constrain the event IA64\_INSTRUCTIONS\_RETIRED-THIS? by the opcode or class of opcodes that might be of interest. This counts how many of a particular class of instructions were executed. For example this event can be constrained to count fma (fnma) and xma (xnma) multiply add instructions to count the number of mathematical operations executed.  
  
The most obvious use of opcode matching comes from understanding L3 cache misses. All instructions that request data can cause an L3 cache miss. Those misse s caused by read accesses can be counted with the event L3\_Reads.Data\_Reads.Miss. These include integer and floating point loads of various sizes and types as well as the prefetch operations invoked by the lfetch instructions. At O3 optimization, the compiler will aggressively prefetch data and it becomes impossible to determine if the L3 cache misses are due to loads or if they were due to lfetch instructions introduced by the compiler. The first case is bad as they would result in the core pipeline stalling while waiting for the data to be delivered. The second case is good as it would indicate that the compiler had correctly prefetched the data to ensure that the pipeline would not stall.  
  
The Intel® VTune™ Performance Analyzer is used to collect the performance data and correlate the events with the source instructions. This process is accomplished by setting the desired counter to a “sample after” value and each time the selected event occurs the value is decremented. When the counter reaches zero a hardware exception is thrown which is handled by the VTune analyzer data collection driver. The driver records the data in the PMUs, records the exception instruction pointer (exception IP), the time stamp counter and other data and resets the counter to the sample after value. While this process is quite precise, for the majority of events there is an uncertainty in the reconstruction of the address of the exact instruction (the true IP) that actually caused the event to occur. For example an L3 cache miss occurs a significant number of cycles after the originating load, store or prefetch. When this is coupled with the rest of the timing uncertainties, the originating instruction can only really be localized to one of the compiler’s basic blocks.  
  
To further complicate matters, the compiler will order the instruction sequences to maximize parallelism and hide latencies. The result of this is that the sequential ordering of the assembly instructions can have little correlation to the ordering of the parent high level language code. In fact, instructions from many different high level language instructions may be scheduled to execute on the same cycle.  
  
When these effects are taken together, it becomes completely impossible to untangle the true sources of cache misses or any other memory hierarchy event, even when the analysis is done in disassembly view. The solution to this problem was incorporated into the design of the PMUs with the inclusion of the opcode matching feature. The PMUs need simply be programmed to only increment when the events are due to the instructions of interest. For example, the L3 cache miss event can be programmed to only increment if it is due to a floating point load.  
  
It should be noted that not all events can be constrained by opcodes. While the cycle accounting events and the bus events cannot be constrained, a sufficiently large number can be to provide a great deal of useful insights. Details can be found in the [Intel® Itanium® Processor Reference Manual for Software Development and Optimization](https://web.archive.org/web/20121020132428/http:/www.intel.com/design/itanium2/manuals/251110.htm) (hereafter referred to as the SOG) and the parts of that document that are reproduced in the VTune Performance Analyzer's online help facility.

**Operations Encoding**

The inherent parallelism of the Itanium processor family starts with the encoding of the operations. Instructions for the Itanium processors are encoded into 41 bit segments or slots. Three instruction slots are packed into a 128-bit bundle. The remaining 5 bits, the template, are used to determine which functional units are to be used for each of the 3 instructions and to determine the location within the bundle of any instruction group boundaries. The instruction group boundaries create a one cycle barrier to the invocation of subsequent instructions. The 41 bits used for the instruction encoding contains the opcode information, which data registers will be used, the values of immediate constants and which predicates will disable the operation. When long immediate values are required (ex. 64 bit constants), the instruction will be encoded into two 41 bit slots. The encoding of the instructions can be found in Chapter 4 of the third volume of the Intel Itanium Architecture Software Developer’s Manual. These 41 bit encodings form the basis of the user’s ability to opcode constrain the PMUs.  
  
The exact programming of the PMUs to opcode constrain the event counting is explained in section 10.3.4 of the SOG. Programming the PMUs involves defining a 41 bit opcode field, a 41 bit mask to select which of the 41 opcode bits to ignore and a 4 bit functional unit selection (MFBI). In general what is desired is not a unique operation and register usage, like single precision floating point loads targeting f27 and dereferencing r22 with P0 acting as the implied predicate. Ldfs f27 = [r22] Rather a class of operations like all floating point loads is more useful. So only a few of the 41 bits need to be constrained.  
  
Because this process works through an opcode bit matching process not all arbitrary sets of instructions can be encoded into a single opcode string and mask. For example, selecting all integer loads with no other kinds of opcodes appears impossible. Here we discuss the use of those predefined constraints, which are being used experimentally by the Intel VTune Performance Analyzer development team at the time of this writing. The mechanics of the values programmed into the PMUs is not of wide interest but the application of the technique to isolate particular performance inefficiencies certainly is. The list of opcode constraints discussed include those that the Intel compiler development team and the author thought useful, and could be encoded into the constraints of the opcode matching bit fields.  
  
This discussion focuses on opcode matching for memory hierarchy events, instructions used for reciprocals and square roots and opcodes associated with integer to floating point conversions. Matching types of branch instructions will not be discussed.

**Memory Access Events and Opcode Matching**

As has been discussed, memory hierarchy events can be instigated by loads, stores and prefetch instructions, so being able to untangle these is of great importance. The memory access opcode classes that are currently being investigated are:

* Floating Point Loads (including pair loads)
* Floating Point Stores
* Lfetch (ie prefetch)
* Integer Memory Ops (integer load, store, compare exchange, fetch add , getf)
* Integer Stores
* 1 and 2 Byte Integer Stores
* Semaphore and getf (compare exchange, fetch add, getf)

The most obvious use of opcode constraining events is to match the L3\_READS.DATA\_READS.MISS event in order to determine the number of load instructions associated with loads. For floating point loads these can be counted directly. For integer loads it is necessary to opcode match this event with Integer Memory Ops and Semaphore and getf and take the difference. This more complicated procedure is required as the opcode filter to just count integer loads doesn’t appear to exist. There is no correction needed for integer stores as this event isn’t incremented by store instructions, the DATA\_WRITES event is incremented by stores. The number of stalled cycles due to accessing these elements of data from memory can then be estimated with the upper limit of multiplying the opcode tagged count by 200 cycles \*(frequency/1 Ghz), on an 870 chipset based system.  
  
It is also useful to determine the number of misses caused by prefetch instructions to understand the prefetch frequency. This can be done by either taking the L3\_READ.DATA\_READ.MISS event with and without opcode matching and taking a difference or explicitly opcode tagging the event with the lfetch opcode match.  
  
One thing that should be realized is that if the data is not prefetched 200\*(frequency/1GHZ) cycles in advance, the loads will become secondary misses. As the required cache lines have already been requested by the lfetch instructions the loads then get treated like any other secondary miss and recirculate until the full cache lines have arrived. Consequently the secondary miss penalties (14-35 cycles) will be added to the remaining latency of the cache line delivery.  
  
General store instructions are not as frequently used for opcode tagging as there are explicit “write” events which are not incremented by prefetch instructions. They can however be used to distinguish between true stores and semaphore instructions.  
  
Short store instructions (1 and 2 byte length) are very useful however as they are invoked when unaligned data is copied or accessed. The compiler must generate a run time detected path to use these instructions when unaligned data is accessed. If the longer 4 or 8 byte instructions are invoked a hardware exception is invoked which must be handled by the OS. Consequently opcode tagging IA64\_INSTRUCTIONS\_RETIRED on the 1 and 2 byte stores is an extremely good way of identifying poorly aligned data accesses.

**Estimating FP Scoreboard Stalls**

There are a few opcode classes that appear to be useful in the identification and localization of time consuming floating point instruction code sequences. The classes of these types being investigated by the VTune analyzer team are:

* Reciprocal Approximations (frcpa and frsqrta)
* Multiply Adds (fma, xma and fsel)
* Setf and getf

In order to understand their use some background on stall cycle accounting and the details of how floating point computations are done is required.  
  
On Itanium Processors there are events which count the stalled pipeline cycles due to waiting for data delivery. BE\_EXE\_BUBBLE.ALL cou nts all such cycles. These are then divided into two components, BE\_EXE\_BUBBLE.GRALL which counts stalled cycles waiting for integer data and BE\_EXE\_BUBBLE.FRALL which counts stalled cycles waiting for floating point data. These two subevents are not prioritized and hence can in principle double count stalled cycles if the pipeline is simultaneously waiting on integer and floating point data.  
  
The stalled cycles due to waiting for integer data can be further decomposed with the event BE\_EXE\_BUBBLE.GRGR which counts stalled pipeline cycles due to waiting for data from a long latency integer instruction. These are the so called MM instructions, like the variable shift (shl, shr) instructions. The cycles counted by the GRGR subevent are a subset of the GRALL counts. The difference of GRALL-GRGR approximates the memory access stalls due to integer data. It is an approximation, as it is possible to be simultaneously waiting for data from memory and from an MM instruction. Further the cycles counted by GRGR can also overlap cycles counted by FRALL, waiting for integer and FP data simultaneously.  
  
What is missing in this mix is an event that counts stalled pipeline cycles due to waiting for data from floating point instructions, a be\_exe\_bubble.frfr type event. Such stalls naturally arise as all the floating point instructions have 4 cycle latencies. A chain of two fma instructions where the output of the first is used as the input of the second will result in up to 3 stall cycles. There must be at least one instruction group boundary (;;) between the two fma instructions to avoid a scheduling or dependency violation.  
  
Such chains most commonly occur due to divisions and square roots, which are computed in software through Newton-Raphson approximations. These take advantage of the “infinite accuracy” of the intermediate result in the combined multiply add operation and are seeded with reciprocal approximation instructions (frcpa and frsqrta). Algebraically the computation (for the reciprocal) works as follows:

*y = (1 + ? )/b provided by frcpa instr d = - ? = 1 - y\*b fnma 1/b = y/ (1 + ? ) which is what we want = y \* (1 – ? + ?2 – ?3 + ?4 – ?5…) = y \* (1 + d + d2 + d3 + d4 + d5 …)*

The square root is computed through a similar Taylor expansion. As the chains of floating point operations that result from the divides and square roots tend to dominate the so-called scoreboard dependency stall cycles, a good approximation can be computed by counting the number of frcpa and frsqrta instructions. This is done by opcode matching IA64\_INSTRUCTIONS\_RETIRED by these opcodes, or the class that counts both of them and then multiplying by the number of cycles associated with the computations of these Taylor series.  
  
In order to estimate the fp scoreboard stalls a penalty must be established. To do this a simple loop like:

*for(i=0; i*

or more generally

*for(i=0; i*

where “func” is a reciprocal, division, square root or reciprocal square root can be used. Such a loop is simply timed and then the time for the “copy” loop

*for(i=0; i*

is subtracted as a baseline and the difference is divided by len. The results are actually slightly underestimated as the copy should really take only one cycle but one bundle loops are too small to run at full speed so a better estimate is to subtract one cycle for the baseline. The results for double precision data are summarized in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Cycles/Iteration | Baseline Subtracted | "Better Estimate" |
| Copy | 1.7 | 0 |  |
| Recip | 7.1 | 5.4 | 6.1 |
| Div | 7.1 | 5.4 | 6.1 |
| Sqrt | 14.6 | 12.9 | 13.6 |
| Recip\_sqrt | 15.1 | 13.4 | 14.1 |

In summary, penalties of 6 cycles for divisions and reciprocals and 14 cycles for square roots and reciprocal square roots are probably reasonably good.  
  
These Taylor series for a division can also be computed with slightly different optimizations with no loss of accuracy. It can be computed to minimize latency for a single computation or to maximize throughput in the case of divisions done in a loop. The Intel® Compiler has an option that will change the generated code sequence from the default value of the latency encoding to the throughput encoding. This would change the 7.1 cycle measurement seen above to a result of 5.1 cycles. The flag is unsupported and may cease to function in future releases but in the 7.1 and 8.0 compilers can be used. The option, -mP3OPT\_ecg\_thruput\_for\_div=T, will change which encoding is used and can result in a significant performance improvement in some applications. What is of particular note is that the accuracy of these software operations can be easily tuned to any desired level and a simple and well defined trade off between accuracy and computational latency can be established. Thus single precision IEEE results are faster than double precisions IEEE results and are in turn faster than double extended IEEE results.  
  
IEEE accuracy requires the result be accurate to better than ½ of one “unit in the last place”, or ULP. This means the result must be good to ½ of the least significant bit in the mantissa. For double precision results this corresponds to one part in 253. If the algorithm has been encoded such that a slightly less accurate result will be adequate it is possible improve performance by tuning this accuracy.  
  
The Intel Compiler can encode the divisions, reciprocals, square roots and reciprocal square roots such that the results will be good to better than 1 ULP when the Non-IEEE flags are invoked. This corresponds to only a tiny loss in accuracy which can frequently be accepted. The result is around a factor in two faster in the number of cycles required to produce the result of these long latency computations. More importantly the accuracy is uniform unlike most non-IEEE computations done with hardware. These are again unsupported flags but they do work in the 7.1 and 8.0 compilers. They are not guaranteed to work in future compilers but they can prove useful in certain applications that can accept an extremely small, uniform decrease (1/2 ULP) in accuracy. They are:

*-mP3OPT\_ecg\_non\_ieee\_rcpsqrt => 1.0/sqrt(x) frsqrta -mP3OPT\_ecg\_non\_ieee\_rcp => 1.0/x frcpa -mP3OPT\_ecg\_non\_ieee\_sqrt => sqrt(x) frsqrta -mP3OPT\_ecg\_non\_ieee\_div => y/x frcpa*

Using these flags on the double precision loops discussed above yields the timing results summarized in the following table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Function | Cycles/Iteration | "Better Estimate" | Non-IEEE | Performance Gain |
| Copy | 1.7 |  |  |  |
| Recip | 7.1 | 6.1 | 4.1 | 3 |
| Div | 7.1 | 6.1 | 4.1 | 2.8 |
| Sqrt | 14.6 | 13.6 | 7.6 | 7 |
| Recip\_sqrt | 15.1 | 14.1 | 8.6 | 6.5 |

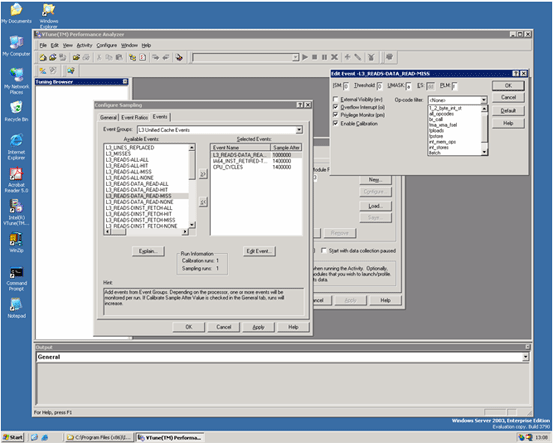
To summarize the Non-IEEE flags will reduce the floating point scoreboard stalls by a factor of 2. So the performance gain of an application can easily be estimated by simply counting the reciprocal approximations with opcode matching and using the appropriate cycle gains.

**setf and getf instructions**

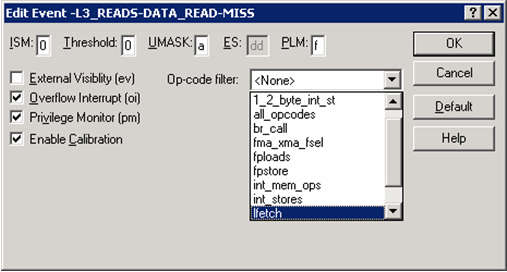
The final opcode matching usage to be discussed will concern the setf and getf instructions. These tend to show up particularly in the use of mod instructions where both arguments are variables. Mod instructions with constant arguments can frequently be computed with shift instructions but general mod instructions require a conversion to floating point format so the divisions and multiply add instructions can be used. These result in quite time consuming calculations. By using the VTune analyzer to locate functions and code sequences with many cycles and many setf and getf instructions being executed, the costly mod instructions can be located. In other words collect data on CPU\_CYCLES, BACK\_END\_BUBBLE.ALL and IA64\_INSTRUCTIONS\_RETIRED opcode matched with the getf/setf class. Once the critical mod instructions have been identified the user should investigate if variable mods are really required or if bit wise “and” instructions could be used instead. For example the calculation that uses a mod with 8 can usually be recoded to instead use an “and” with 7 requiring a zero result.

**Prototype VTune Analyzer Interface**

The complexity of determining the exact bit strings required to construct useful opcode matching classes and the subsequent programming of PMC8, PMC13 and PMC15 is quite complicated. While perhaps educational and ennobling for the soul, it is unreasonable to expect that an application developer with a far more well grounded outlook would be willing to invest such an effort. To this end a more user friendly interface to the opcode matching capability of the Itanium processor is desirable.  
  
To this end what is being considered is to allow the user to add events, to those predefined in the VTune Performance Analyzer, by using the “edit event” interface to create opcode matched events that will be used during the project. The selection of opcode matching classes will be predefined and can be selected through a pull down menu interface. The screen captures shown below give an impression of how such an interface might appear to the user.  
  
In the example shown below the user is editing the L3\_READS.DATA\_READS.MISS event to opcode match it with the lfetch opcode tag.



A close up of the edit event dialog window would show:



**Conclusion**

Opcode matching is a unique performance monitoring capability in the Itanium Processor Family. It enables the user to create new classes of monitoring events that can be used to investigate optimization opportunities beyond those envisioned by the static list of events from which they are created. We discussed examples of how the optimization opportunities can be addressed through advanced use of the Intel Compilers and code and/or data modification.

**Related Links**

* [Introduction to Microarchitectural Software Optimization for Itanium Processors](https://web.archive.org/web/20121020132428/http:/software.intel.com/sites/default/files/m/d/4/1/d/8/219348_software_optimization.pdf) (PDF 466KB)
* [Intel® C/C++ Compiler Users Guide](https://web.archive.org/web/20121020132428/http:/software.intel.com/en-us/)
* [Itanium Processor Software Specifications](https://web.archive.org/web/20121020132428/http:/developer.intel.com/design/itanium/arch_spec.htm)
* [Intel® Itanium® Processor Reference Manual for Software Development and Optimization](https://web.archive.org/web/20121020132428/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.