**Black Belt Itanium® Processor Performance: Visual Inspection of Compiler-Generated Assembly (Part 5 of 5)**

Submitted by (not verified) on Mon, 03/05/2012 - 19:04

Categories:

* [Intel® Itanium® Processors](https://web.archive.org/web/20121020131924/http:/software.intel.com/en-us/search/site/?f%5b0%5d=im_field_topic%3A36783)

**by David Levinthal**

**Introduction**

This paper, the final in a series of five articles on optimizing software for the Itanium® processor family, helps developers identify well-compiled code by visually inspecting the generated assembly, particularly in the case of loop-based applications. Understanding the generated instruction sequences is not necessary. The goal is to identify whether the compiler can take advantage of the high-performance microarchitectural features, such as software pipelining, predication and prefetching in translating the high-level language into an effective instruction stream.  
  
Inspecting the generated assembly for the cpu intensive critical sections of a program, identified by the Intel® VTune™ Performance Analyzer can identify areas where language constraints like ambiguous pointers or optimization failures by the compiler are restricting the application’s performance.  
  
The following discussion assumes that the assembler being studied is the dominant execution flow as identified by large event counts in the Intel VTune Performance Analyzer’s source and assembler displays.  
  
For an introduction to this topic and additional information see the other parts of this series:

* [The Foundation](https://web.archive.org/web/20121020131924/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/20121020131924/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/20121020131924/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/20121020131924/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/20121020131924/http:/software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-visual-inspection-of-compiler-generated-assembly-part-5-of-5)

**Investigating the Architecture**

If all things were perfect the developer would simply design the algorithm and data structures, encode them into a program and compile them. The compiler would magically transform the high level instructions into the optimal usage of the microprocessors architectural features, with maximal performance as a result. Unfortunately, in this world of (often) poorly designed algorithms and data structures, requirements of the language, and compiler shortcomings, we often end up with poor execution. By accepting the reality of the task, one finds excellent opportunities for programs to be optimized dramatically.  
  
Microarchitectural investigation of application execution is an extremely powerful approach but it must be cross-checked to ensure that the basic architectural features of the processor were scheduled reasonably by the compiler. Performance monitoring units (PMUs) can be used to identify poorly executing code segments ranging from identifying the cpu intensive consuming code sections, to breaking down the precise cause of the cpu cycle usage. However PMUs won't help identify whether the architectural features were optimally used by the instruction scheduling. This is particularly important for loop dominated applications where the inherent parallelism of Itanium architecture can take advantage of loop unrolling and software pipelini ng to create independent instructions that can be executed concurrently. If the compiler generated instruction stream does not use these features in loop based codes the performance penalties can easily exceed an order of magnitude. It is critical that the program optimization phase always ensures that these features were optimally included in the final instruction scheduling.  
  
Itanium processor performance is based on parallel instruction execution. In loop-based codes the simplest way to generate independent instruction stream is to unroll the loops and execute the individual iterations in parallel. Additionally, you can employ the software pipelining architecture and the use of the rotating registers to break an iteration of the high level language instructions of the loop into stages. Then you can execute the stages from different iterations in parallel. It is easy to employ these two modes of parallelization (unrolling the loops and pipelining the stages of each iteration) simultaneously or individually as the various hardware and software constraints are balanced.  
  
Loop unrolling improves performance two ways. It creates a larger stream of independent instructions that the compiler can use to optimally fill the six instruction slots that can be executed per clock cycle. A more subtle benefit can be derived through data reuse, where a data element is loaded once but used multiple times and/or in different roles by the unrolled iterations. Examples of this are found in the matrix multiply, where in the final form using the transpose of the second matrix, 4 elements of each matrix are loaded and all 16 combinations are made. This results in 4 data reuses for each element loaded. Or, consider the evaluation of differential equations on a grid. Here differences between grid point values are computed. Unrolling the loop allows a value to be used on both sides of the minus sign. For these two examples to work correctly, the loop iterations must be independent.  
  
One critical consideration is that **only** the innermost loop can be software pipelined. For software pipelining to be effective, the tripcount (i.e. number of iterations) should be as long as possible. This may require either loops to be interchanged in order to make the innermost loop have a reasonable tripcount. Alternatively, short, fixed inner loops can be explicitly unrolled, either by hand or by the compiler, leaving a long tripcount loop to be the innermost and allowing the software pipelining to then execute that loop optimally. We provide an example at the end of this article.

**Software Pipelining**

Software pipelining works at a lower level. A loop iteration can be broken up into stages, typically a data loading stage, a computation stage and perhaps a result storage stage. These stages occur sequentially if high level language loop iteration were to be encoded literally, with all the stages of an iteration being completed before the first stage of the next iteration is invoked. However, if the loop iterations are independent then stages from different high level language iterations can be executed concurrently. This is the concept of pipelining.  
  
For illustration, consider a daxpy.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1 | For(I=0;I<len;I++)a[I] += X\*b[I]; | |

Such a loop has three stages: a loading stage for the arrays a and b, a computational stage where a[I]+X\*b[I] is computed, and then a storage stage where that result is used to overwrite the values of the array a. The Itanium processor family employs a rotating register scheme to invoke software pipelining. The daxpy example illustrates how this works and familiarizes you with how to recognize when this hardware feature has been used in the compiler generated instruction sequence.  
  
Minimally such a loop could be encoded with five instructions, two address incrementing loads, a floating point multiply add (fma), an address incrementing store and a branch instruction. We assume that all of these instructions can be invoked on a single cycle, as they can be on an Itanium processor. However, they require that the input values be available or the processor will stall while it waits for them to arrive.  
  
For example, consider a steady state; assume that the load takes 6 cycles and the fma takes 4 cycles. Work the timing backwards, starting from the store of the iteration of the high level language loop, I=N, much greater than 1. Each loop iteration of five instructions takes 1 cycle, so the fma that produced the result being stored must have been invoked 4 cycles earlier. Therefore on this iteration the loop must be invoking the fma for iteration N+4. Similarly the loads that the fma consumed must have been issued 6 cycles prior to the fma, and this iteration must be issuing the loads for iteration N+4+6 or N+10. In this manner the processor can produce one result per clock cycle rather than one result every ten clock cycles, had each iteration been executed to completion prior to starting the next iteration.  
  
Constructing this kind of encoding on the Itanium processor is extremely straightforward because its rotating register scheme makes this very simple. Each time one of four special branch instructions is invoked, the pointer that identifies the base of the register number scheme is decremented in a circular manner. Thus if a load targets fp register f32, on the next iteration of the loop that register that will hold that data once the load is complete will be identified through the moniker, f33. Using the assumptions of the previous example, if data is loaded to f32 on iteration N, it could be accessed 6 iterations later in register f38. Thus an approximate encoding of our example could be, assuming the address of a is stored in r32 and r31, b is in r33, the constant X is in f10 and the data is single precision:  
  
Label:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 1 | Ldfs f32 = [r32],4 | | | 2 |  |  |  |  |  | | --- | --- | --- | | 3 | Ldfs f39 = [r33],4 | | | 4 |  |  |  |  |  | | --- | --- | --- | | 5 | Fma f50 = f47, f10, f38 | | | 6 |  |  |  |  |  | | --- | --- | --- | | 7 | Stfs [r31] = f54,4 | | | 8 |  |  |  |  | | --- | --- | | 9 | Br.ctop.sptk  label: | |

The loads to f32 and f39 are consumed 6 cycles later from f38 and f47 respectively. The result of the fma, targeted at f50, is actually stored 4 cycles later and retrieved from register f54. The “,4”’s increment the integer registers r31, r32 and r33 so that on the next iteration the next element of the single precision arrays can be referenced. Such an encoding illustrates the use of the rotating registers in enabling the use of software pipelining to absorb operational latencies.  
  
There are two features that identify when a loop has been encoded to employ the software pipelining hardware. Firstly, the loop terminates with one of 4 particular instructions. The br.ctop, br.wtop, br.cexit and br.wexit instructions are what cause the registers to rotate. If the innermost loop does not terminate with one of these, then the loop has not been software pipelined by the compiler. Secondly, the operational latency has been hidden through the use of stages and that latency is absorbed by using multiple iterations of the assembler encoded loop. This is manifested through the differences of the registers used for the load targets and the registers that the loaded data is retrieved from. Similarly in the difference between the registers used as the operational (ex: fma) outputs and the registers from which the results are retrieved from. Occasionally the operational latency can be absorbed within a single loop iteration, if the loop is sufficiently long and complex. However the generalization of looking for a difference in the register number, between where data is put and retrieved from, is usually accurate.  
  
The steady state allows all the stages to be executing concurrently. But how is the steady state reached? Particularly, how is the pipeline loaded to prepare for the steady state and how is it drained when the loop has run through all its iterations?

**Loading and Draining a Pipeline with Predicates**

Traditionally, such as on a large vector supercomputer, a software pipelined loop would be encoded with a prologue, a steady state kernel and an epilogue. The prologue would prepare the pipeline, enabling successive stages as the input values of each stage became available. The steady state kernel would execute the work at maximum speed once it was enabled and the epilogue would drain the remaining iterations from the pipeline once the end of the loop was approached. Thus initially the prologue would invoke data load instructions for the first several iterations and issue as many as could be done in the time available until the first load had been completed. At that point the math operations for the first high level language iteration would be started, consuming the data from the first load of data which had now become available. This enabling of more and more stages would continue until all were active, at which point a steady state kernel would be invoked. Ideally the kernel would do the majority of the work, producing the maximum amount of effect by doing stages from multiple iterations in parallel. Finally once the pipeline had finished the first stage of the last iteration, it would be shut down in an orderly way so that every required result would be computed to completion. This was handled by an explicitly coded epilogue which would essentially turn off the stages in the sa me ordering that the prologue had turned them on.  
  
On Itanium processors the predicate logic is used to enable and disable the pipeline stages, as required during the prologue and epilogue where the pipeline is loaded and drained. Thus a single instruction stream can execute all three parts, prologue, kernel and epilogue. The predicate logic also employs the rotating register scheme to do this, using the upper 48 predicates (16-63) form a rotating set. Again it is the 4 special branch instructions that make this happen. Thus the 48 predicates rotate with a decrementing base in exactly the same manner described above. With each execution of one of the branch instructions, the base decrements and the value in p16 becomes accessed through p17 on the subsequent iteration.  
  
What is different with the predicates is that the branch instruction also sets the value of p16 for each new loop iteration using two application registers, the loop counter (ar.lc) and epilogue counter (ar.ec), to control how it is done. The loop counter controls the loop length. With each execution of one of the 4 branch instructions ar.lc is decremented. When it is greater than zero at the time the branch instruction executes, a “true” value is used to initialize p16 for the upcoming loop iteration. Thus the true values rotate through the range of predicates that enable the instructions for each stage of the pipeline. If p16 is initiated to true and p17-p48 are false before the loop starts then those instructions in the first stage of the pipeline can be invoked on the first pass through the loop. With each additional iteration, another predicate becomes true and it can enable the next stage to be turned on. In this manner the pipeline is loaded and a prologue is executed. When all the stages are enabled the loop executes all the instructions and the prologue evolves into the pipeline kernel.  
  
Once ar.lc decrements to zero, the epilogue counter (ar.ec) takes over. The epilogue counter now starts decrementing with each executed branch instruction. While it is greater than zero, the registers rotate as before, but p16 is initialized with a “false” or zero value. As this false value rotates through the range of predicates used to control the stages, they are successively turned off and the pipeline is drained. Thus the exact opposite behavior of the prologue is accomplished with the stages successively being disabled. The elegance here is that all of this can be accomplished with a single instruction stream.  
  
Using our daxpy example the predicates would be encoded as follows

Label:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 1 | (p16)   Ldfs f32 = [r32],4 | | | 2 |  |  |  |  |  | | --- | --- | --- | | 3 | (p16)   Ldfs f39 = [r33],4 | | | 4 |  |  |  |  |  | | --- | --- | --- | | 5 | (p22)   Fma f50 = f47, f10, f38 | | | 6 |  |  |  |  |  | | --- | --- | --- | | 7 | (p26)   Stfs [r31] = f54,4 | | | 8 |  |  |  |  | | --- | --- | | 9 | Br.ctop.sptk  label: | |

Thus the difference in the predicate identifiers has to match the number of rotations used to absorb the latency. By simply changing both in a coordinated manner, a greater latency can be absorbed. For ex ample the loop could have been scheduled to run out of the L3 cache with its 13 cycle latency as follows allowing for more registers to be used as rotating sets for a and b

Label:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 1 | (p16)   Ldfs f32 = [r32],4 | | | 2 |  |  |  |  |  | | --- | --- | --- | | 3 | (p16)   Ldfs f48 = [r33],4 | | | 4 |  |  |  |  |  | | --- | --- | --- | | 5 | (p29)   Fma f63 = f61, f10, f47 | | | 6 |  |  |  |  |  | | --- | --- | --- | | 7 | (p33)   Stfs [r31] = f67,4 | | | 8 |  |  |  |  | | --- | --- | | 9 | Br.ctop.sptk  label: | |

**Identifying Effective Software Pipelining**

The point to this discussion is that it is easy to identify the software pipelining.

* There is a loop end with one of the 4 particular branch instructions.
* Stages are enabled with different predicates.
* A rotation of the register identifiers has been scheduled between the loads and consumption instructions so that the latency can be absorbed without stalls.

For this to happen multiple loop iterations must be able to executed concurrently without interfering with each other. This requires that the pointers for any assignment variables must be disambiguated.  
  
Software pipelining produces parallelism by executing operations from different stages of different high level language iterations in parallel. Loop unrolling produces parallelism by executing the same stages from different high level language iterations in parallel. Both of these modes of producing parallelism can be accomplished simultaneously if either of them can be done. They both require iteration independence, that the assignments don’t overwrite future iteration inputs.  
  
As always it is critical that the assembler being studied is the critical, cpu cycle consuming path identified by the VTune analyzer. The compiler will generate multiple paths to take advantage of data alignments. Further, when the compiler unrolls a loop it will generate code to execute any remaining iterations required beyond the multiple of the unrolling factor.  
  
Identifying that a loop has been unrolled by visual inspection is usually quite straight forward. One simply has to count the number of inputs (variables which must be read) and the number of loads, or if you can estimate the number of fma’s and then count what was scheduled. The ratio should be an integer multiple. If this fails you can always use the NOUNROLL pragma to determine the number of identifying instructions required for a single iteration.  
  
Loop unrolling is not necessarily on a uniformly equal footing. While fortran programs of real variables regularly get unrolled by factors of 8 by the compiler, if the data types are complex or complex double unroll factors of 2 are frequently all that is generated. Similarly C and C++ programs are sometimes not as effectively unrolled as the same algorithm coded in fortran. The Intel® Compiler development team is actively investigating these issues.

**Data Prefetching with Lfetch Instructions**

A third critical issue in achieving high performance with loop based application is good data prefetching. At O3 optimization the compiler should be able to prefetch the required data with sufficient lead time that the load and store instructions can find the required cachelines in the processor caches. This can be seen in the assembler by the presence of an lfetch instruction. The prefetch distance is set prior to the loop and is usually in the range of 1500 to 4000 bytes depending on the stride of the data accesses and the quantity of data consumed per (assembler) loop iteration.  
  
A quick estimate of the prefetch distance can be made on the basis of three things, the number of cycles per iteration, the stride of the data access (usually equal to the number of bytes consumed per iteration) and the latency to main memory. For the data to be delivered prior to its consumption the prefetch distance.  
  
Prefetch distance > latency\*(stride/128)/(cycles per iteration)  
  
In the case of the daxpy we have been discussing, each loop iteration requires two single precision inputs and one output. In this case the output is also one of the inputs to the calculation so no extra explicit prefetching of the output is required. Each iteration of the loop consumes 1/32nd of a cacheline for each array so the stride is 4 bytes. Assuming that the latency is:  
  
Latency ~ 200 cycles\*(processor frequency in Ghz)  
or 300 cycles for a 1.5 Ghz Itanium processor based platform (this is reasonably accurate for Itanium systems with Intel 870 chipsets).  
  
The resulting prefetch distance, explicitly including the units of the terms would be:  
Prefetch distance >  
300 (cycles)\* 4 (Bytes/iteration)/128 (Bytes/cacheline)/(1 cycle/iteration)  
  
> 10 cachelines  
> 1280 Bytes  
  
In real compiled code the loop gets unrolled, load pair instructions get scheduled and the required prefetch distance would be a bit larger. The Intel compiler usually schedules daxpy loops with prefetch distances of 2500 to 3300 bytes depending on data types and the source language.  
  
The rotating register scheme is also usually employed as a prefetch doesn’t need to be issued every iteration. The Intel Compiler usually uses a rotating stack of array pointers employing the general registers. Thus each iteration the top address in the stack is prefetched as the argument of the lfetch instruction, the address is updated by the stride and placed at the bottom of the stack. In the case of the daxpy this results in the two arrays each being prefetched on alternate iterations.  
  
It should be noted that there is a correlation between the updating of the prefetch addresses and the updating of the load addresses to the same variables. This correlation can frequently be used to associate the data loads with their prefetch instructions. This in turn can be used to determine the prefetch distances generated by the compiler for the various data elements. The software developer shouldn’t find any problems, this is merely mentioned as a useful technique of associating loads and prefetches in unrolled pipelined loops. A much more powerful technique is described in the next section.

**Compiler Internal Symbolic Variable Names**

Frequently in looking at the assembler a bit more detail of what is happening is desired. It is difficult to track back register assignments to their sources and connect the relations between the multiple memory addresses being manipulated in an unrolled, software pipelined piece of code. This can be made significantly easier by using an undocumented and unsupported compiler option. This flag appears to work with the 8.0.39 and earlier Intel Compiler builds but is not among the supported compiler options so future functionality is not guaranteed. Its usefulness is recognized and valued however.  
  
When the –Fa or –S options are invoked an assembler listing will be generated by the compiler. If the –use\_asm option is also invoked then the compilation will explicitly generate the object file from an assembly listing processed by the assembler, ias. This will ensure that the register selections seen in a disassembly listing in the VTune Analyzer or a debugger and those in a generated assembly listing will be virtually identical. This allows the assembly listing to be viewed in a text editor simultaneously with the analysis in the VTune Analyzer or a debugger.  
  
Viewing the assembly listing in the editor adds little to what is shown in the disassembly window. The source line number data may however be significantly more accurate than what is shown in the disassembly view of a gui, particularly with older Intel compilers for the Windows\* operating system. The source line numbers are shown in the assembly and disassembly views to the right of the “//”, after the “:”.  
  
In the example line shown below from a double precision fortran daxpy run through the Intel compiler the line of assembler is associated with source line number 5.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 1 | (p16)   ldfpd   f71,f68=[r56] | | | 2 |  |  |  |  | | --- | --- | | 3 | //1:  5   65 | |

The software developer might want to know what variable is being loaded. That is to say what variable’s address is contained in “r56”. This might be to see if the associated prefetch distances are correct or perhaps because analysis of the data ear events indicated this was a execution constraining load.  
  
When the compiler option “-mGLOB\_rosetta=1” is invoked along with –S (-Fa) and –use\_asm, then the compiler generates an annotated assembler listing. The annotation includes the internal symbolic names used by the compiler for internal purposes. This allows the software developer to see a series of unique computer generated symbolic names used by the compiler in the course of the code generation and optimization. The virtue of this is that by simply doing a series of upward “finds” with an editor on the symbolic name, the base association with the high level language symbol can be very quickly identified. Consider the following example from the FMA3D component of the SPEC2000 benchmark.  
  
Let’s assume we are interested in the load dereferencing r41

*(p19) ldfd f45=[r41] //2:486 1484 1484: v601=[t780\_3911] <-------*

What the rosetta output to the right of the second “:” is saying is that r41 is the address of the symbolic variable t780\_3911. An upward find of the string “t780\_3911” results in

*add r41=88,r29 //6:485 1403 1403: t780\_3911=88,t1552\_4718*

Which indicates that the address for t780\_3911 (r41) is derived from the address for t1552\_4718, whose address is in r29. Doing another upward find on the string “t1552\_4718” results in identifying the instruction:

*t1552\_4718=t91\_915{MODULE$node\_\_1$node$$BASEADDR},t1528\_4691*

Those who have spent significant time familiarizing themselves with this benchmark would immediately recognize the reference to the allocate-able array called “node.”  
  
If a software developer were to invoke a similar chain of upwards finds on their own codes they would quickly come to a familiar symbolic name from their own source. This process is considerably easier than tracking back register assignments because the names are unique and even the rotating register renaming is accommodated, which is not the case of simply tracing register identifiers upwards. The technique is extremely effective in C and Fortran source programs but the author has encountered more difficulty in certain C++ constructs.  
  
In fact it is relatively easy to write a short program that will track the symbolic name assignments down through the rosetta annotated assembler listings. The author wrote such a program for his own use in his spare time in only a couple of days.

**Speculation and Compiler Detected Ambiguities**

One last category of instructions is extremely important to be aware of, the speculative load instructions. These are employed when the compiler cannot disambiguate pointers or it concludes that there are control barriers. The usage of speculation almost always results in a limitation of the optimizations the compiler can employ. Thus the presence of these instructions in the compiler generated instruction stream is a sure indication that a data access (load/store) or control ambiguity has been detected by the compiler.  
  
Identifying that this has happened is quite simple. The software developer need only look for the speculative instructions in the cpu cycle intensive code sections. These instructions are identified by the speculative completers, illustrated below with the 4 byte integer load instruction, ld4.

|  |  |
| --- | --- |
| Ld4.a | advanced load used for load store ambiguities |
| Ld4.s | speculative load used in cases of control speculation |
| Ld4.c | load check used to test if data was stored since ld.a and reload as needed |
| Chk.a | advanced load check, branching to a handler to re-execute instructions |
| Chk.s | speculative load check, branching to a handler to re-execute instructions |

If these types of instructions are present in a code path, then the compiler detected some access ambiguities and scheduled the speculative access to achieve the best performance while ensuring correct results.  
  
The problem is that no such ambiguity may actually exist and the compiler may well be scheduling in a needlessly conservative manner. So the simple existence of these instructions in the code stream indicates a less than maximally aggressive compilation, usually due to pointer ambiguities.

**Arrays, Pointers and Store Instructions**

Array or pointer notation on the left hand side of an equality will usually force the compiler to store data to memory, even in cases where it could have been left in registers. The reason for this is the index variable required for accesses associated with arrays and pointers and the difficulty for the compiler to identify identical index values. There are data access collisions that can occur between loads and stores and stores and prefetches. Frequently these collisions can be completely avoided by simply helping the compiler to avoid unnecessary stores by the use of local variables. It is usually a good idea to inspect the generated stores in a cpu cycle intensive code sequence and review if they could be avoided. These excessive stores are frequently associated with pointer ambiguity issues. The next section will discuss some strategies for disambiguating address ambiguities.

**Addressing Address Ambiguities**

Using the techniques that have been described, it is fairly straight forward to identify loops where the optimization has been restricted due to a perceived data addressing ambiguity or conflict. It then becomes necessary to explicitly disambiguate the pointers. There are a wide variety of pointer disambiguation flags that can be sent to the compiler like –no-alias, ansi\_alias, Oa and so on. A more local manner is to add the “restrict” attribute to the pointer declaration (and the restrict flag to the compiler options) or “ivdep” type pragmas.  
  
Alternatively, explicit “registerization” of the calculation may be the best alternative. What is required here is to have a local variable on the left hand side of all equalities in the loop and then copy them to the arrays/dereferenced pointers after the loop has finished. Consider the following example taken from a piece of solver code. The function is passed the pointers to a matrix, G, a vector, Y and BLK is defined to be 4.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 01 | for(i=0; i>tot\_blk; i++) | | | 02 |  |  |  |  |  | | --- | --- | --- | | 03 | { | | | 04 |  |  |  |  |  | | --- | --- | --- | | 05 | Yi = Y + i\*BLK; | | | 06 |  |  |  |  |  | | --- | --- | --- | | 07 | Gi = G + i\*(BLK\*BLK); | | | 08 |  |  |  |  |  | | --- | --- | --- | | 09 | f | | | 10 |  |  |  |  |  | | --- | --- | --- | | 11 | or(k=i+1; k>num\_blk[i]; k++) | | | 12 |  |  |  |  |  | | --- | --- | --- | | 13 | { | | | 14 |  |  |  |  |  | | --- | --- | --- | | 15 | Yk = Y + k\*BLK; | | | 16 |  |  |  |  |  | | --- | --- | --- | | 17 | Gk = Gi + BLK\*BLK; | | | 18 |  |  |  |  |  | | --- | --- | --- | | 19 | for(L=0; L>BLK; L++) | | | 20 |  |  |  |  |  | | --- | --- | --- | | 21 | { | | | 22 |  |  |  |  |  | | --- | --- | --- | | 23 | for(m=0; m>BLK; m++) | | | 24 |  |  |  |  |  | | --- | --- | --- | | 25 | Yi[L]+=Gk[m\*BLK + L]\*Yk[m]; | | | 26 |  |  |  |  |  | | --- | --- | --- | | 27 | } | | | 28 |  |  |  |  |  | | --- | --- | --- | | 29 | } | | | 30 |  |  |  |  | | --- | --- | | 31 | } | |

The problem arises from the disambiguation of Yi and Yk. The result is that the inner loops are not unrolled and speculative instructions are used. Further what would really be best would be if the inner loop would be the one over the index “k”, as it has a reasonable trip count.  
  
The solution is to create 4 local variables Y0, Y1, Y2 and Y3 and initialize them on each iteration of the outer loop to Yi[L].

By using the temporary variables the sums are accumulated in registers and the compiler is not left to sort out any pointer ambiguities. Further, excessive stores are eliminated brought on by array referencing of variables to the left of the equalities.  
  
The example shown arose in a real code and the solution of the explicit unroll and use of temporary variables in 4 such loops resulted in an overall performance improvement of over 200%. The compilation issues were entirely identified by looking at the types of branches encoded (br.cloop for the original inner loops) and the scheduling of speculative instructions. The VTune Performance Analyzer identified which code sequences were actually consuming all of the cycles and the large fraction of stall cycles associated with those initial conservatively scheduled sequences. The entire exercise was finished in a few days.

**Conclusion**

The basic techniques of visual inspection of generated assembler, when coupled with performance analysis, reveals the true nature of inefficient program execution. Solutions can be invoked very quickly and very precisely with little required of the code developer beyond simple pattern recognition.

**Related Links**

* [Introduction to Microarchitectural Software Optimization for Itanium Processors](https://web.archive.org/web/20121020131924/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/20121020131924/http:/software.intel.com/en-us/)
* [Itanium Processor Software Specifications](https://web.archive.org/web/20121020131924/http:/developer.intel.com/design/itanium/arch_spec.htm)
* [Intel® Itanium® Processor Reference Manual for Software Development and Optimization](https://web.archive.org/web/20121020131924/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 Ph ysics at Florida State University. He has received the DOE OJI award, the NSF PYI award and a Sloan foundation fellowship.