Analyzing and Resolving multi-core non scaling on Intel® Core™2 processors

Dr. David Levinthal PhD.

Introduction

Extending an application's ability to effectively use as many processors as are available is critical in the multi-core future of modern computing. Among the limitations that can be encountered are those related to sharing and dividing the finite throughput and capacity of the platform and the processors. Judicious use of the Intel® Core™2 processor performance events can be applied to identify the precise limitations that are restricting an application's ability to scale out to the full compliment of available cores.

Defining Non Scaling

In order to study non scaling in parallel execution one must first define what is meant by the term. The scaling that is desired is that the wall clock elapsed execution time should decrease linearly as the number of cores being used by the application is increased. This is limited by the parallel execution form of Amdahl's law, which states that a program's execution time can only be decreased by the time used by the part of the code being optimized. So the first source of non scaling to be investigated would be the degree of parallel execution that is achieved.

The Intel® Core™2 processor performance events can provide a useful metric for this purpose. Each core has its own Performance Monitoring Unit (PMU) and the event CPU_CLK_UNHALTED.CORE counts the number of unhalted cycles on each core. If the VTune™ Performance Analyzer is used to sample on this event, then the number of unhalted cycles, summed over all the cores that were executing the process of interest can be easily measured. This yields the total number of effective cycles that the application consumed. Call this "effective_cycles".

A useful feature in the PMU is the ability to compare the value of an event to a reference value (cmask) on every cycle and determine if it is greater than or equal to the reference (inv=0) or less than the reference (inv=1). By doing this, the PMU will count the cycles when the programmed condition is satisfied. This only works for events counted with the general purpose counters. The fixed counters, for events CPU_CLK_UNHALTED.CORE, INST_RETIRED.ANY and CPU_CLK_UNHALTED.REF, do not make the comparisons. If the event, CPU_CLK_UNHALTED.CORE_P (the general counter version of the unhalted cycle event), is compared to a value of 2 and the "less than" comparison is applied (inv=1), the counter will count all cycles. If this is summed over all processes and then divided by the number of cores in the system the result will be the total time. Call this "total_cycles". For this to work it is essential that the speed stepping be disabled in the bios and OS, or the idle cores may be running at a lower frequency and thus distort the measurement of the total elapsed time.

The ratio of effective_cycles/total_cycles will equal the number of cores used for perfect scaling and will equal 1 for completely serial code. This is independent of the fraction of the available cores used in the execution. This technique can be defeated if spin-wait loops are coded to consume cpu cycles of course, so correctly coded spin loops are required. For a more detailed analysis of the overlap efficiency, the use of the Intel® Thread Profiler would be highly recommended.

Deviations from ideal scaling can be due to the code structure and synchronization logic, but it can also be due to over commitment of shared resources. The objective of this paper is the systematic identification of such limitations and their resolution.

The Limited Resources

It is first necessary to construct a list of the resources that could encounter use contention in parallel execution. These would include:

1) Total Bandwidth throughput
    a) Memory bandwidth throughput
    b) Socket to socket cache coherency bandwidth throughput
2) Disk subsystem access throughput
3) Shared cache capacity
4) DTLB capacity
5) Individual cacheline access
    a) Shared elements in a cacheline
    b) Shared lines without shared elements, aka false sharing

The last component of the list has somewhat different issues associated with it, as the synchronization of threads and processes relies on locked cacheline access. The difference is most clearly illustrated by the point that faster/larger hardware capacity might not overcome the non scaling impact, as would be the case for the other elements of the list.

For these limitations there are two basic scenarios with the scaling out of an application that will be discussed, the splitting of a fixed amount of work between the cores and a linear increase in the amount of work that is done by all the cores. These two scenarios have different meanings when the term “non-scaling” is used and can therefore create different signatures that should be looked for.

Bandwidth

To understand the performance impact of the bus traffic it is necessary to determine both the actual bus traffic (total and its components) and the true limit of the platform being used during the analysis. A platforms’ bandwidth limit can be effected by a large variety of issues. These can include the configuration of the hardware prefetchers, the number, type and slot position of the dimms, the front side bus frequency, the processor frequency, the cache coherency snooping requirements. Consequently there is no one metric value that can be declared to be the bandwidth limitation of an Intel® Core™2 processor based platform. The only reasonable way to determine the bandwidth limitation of an individual platform is to measure it with a synthetic bandwidth benchmark. For this purpose a single, long tripcount, triad loop is almost certainly the best choice. As the multi-core limit will likely be different than the single core limit, the triad benchmark should support parallel execution either at a threaded or process level.

A system bandwidth limit effects execution differently than most other performance impacts. The impact of most performance issues grow linearly with their associated metric, as in the case of last level cache misses where the performance impact in cycles is usually estimated as the event count times the latency. In the case of bandwidth, it is more like a barrier that does not impact performance, unless the application has an appetite that exceeds the platforms capacity. This means that the impact acts almost like a step function rather than the linearly increasing impact observed with most other performance events. One result of this is that the memory access latency increases in a non-linear fashion as the bandwidth limit is approached. This can be observed by measuring the bus latency as the number of simultaneously running triads is increased. The bus latency can be measured with the performance events in counting mode (as opposed to sampling) with the ratio

Bus Latency = BUS_REQUEST_OUSTANDING.SELF/
(BUS_TRANS_BRD.SELF -
BUS_TRANS_IFETCH.SELF)

In many cases the ifetch component will be insignificant, particularly in bandwidth limited situations and can thus be ignored.

There are many sources that contribute to the total bus traffic on a system. The Intel® Core™2 processor performance events enable a variety of techniques to measure the total usage and many of the individual components. Measuring FSB saturation is straightforward. This can be expressed as the fraction of bus cycles used for data transfer:

BUS_DRDY_CLOCKS.ALL_AGENTS/CPU_CLK_UNHALTED.BUS

or by directly counting the number of bytes associated with the cachelines transferred:

Cacheline_Bandwidth (bytes/sec) ~

64*BUS_TRANS_BURST.ALL_AGENTS*core_freq/CPU_CLK_UNHALTED.CORE

A convenient metric is simply cachelines/cycle, as the appetite of N parallel versions of an application/thread is just N times the value for one. So if the platform limit has been determined in these units, the likelihood of saturation occurring during parallel execution can be easily estimated from the measurements taken during a single threaded performance analysis.

The events BUS_TRANS_* can be used in a hierarchical manner to break down the contributions to the front side bus utilization. This is outlined in the following table and is explained in greater detail in the VTune™ Performance Analyzer online help facility.

EVENT	Explanation
BUS_TRANS_ANY	All bus transactions: Mem, IO, Def, Partial
BUS_TRANS_MEM	Whole $lines, Partials and Inval
BUS_TRANS_BURST	Whole $lines: Brd, RFO, WB, Write Combines
BUS_TRANS_BRD	Whole $line reads: Data, Ifetch
BUS_TRANS_IFETCH	Whole Instruction $lines
BUS_TRANS_RFO	Whole $lines Read For Ownership
BUS_TRANS_WRB	Whole $line Write Backs (modified $lines)

There are a series of standard techniques for reducing bandwidth requirements that may be applicable in the case of exceeding the bandwidth limits of the platform. These would include:

1) use all bytes of all cachelines accessed while they are in cache
    a. order the elements of structures in decreasing size to avoid compiler generated padding
    b. define structures by actual usage, not object oriented or topical connection
    c. traverse leading dimensions in the innermost of nested loops

d. cacheline align components of structures when possible
e. place structure components that are used together adjacent to each other in structures

2) Use non temporal stores for long tripcount assignment loops
3) do not prefetch cachelines that are not actually accessed
4) merge cpu limited loops with bandwidth limited loops wherever possible
   This is usually easier when the loop tripcounts match, but can also be done when they don't
5) try to accomplish a data blocking to maximize use of data while it is in cache

The above is not meant to be an exhaustive list, merely an obvious one. The last of these is clearly easier said than done to anybody who has actually tried.

Cacheline Consumption Efficiency

The components 1.a and 1.b highlight one of the most common problems, using only a fraction of the address space in a cacheline. Not only will this increase the bandwidth requirement but it also lowers the effectiveness of the on-die cache memories. If only a fraction of each cacheline is actually used while it is resident in cache, the application has effectively decreased the size of the cache. The component 1.a is referring to the practice of compilers to align the elements in structures to the alignment defined by their size. Thus if the first element of a structure is a "char", followed by a 4 byte "int", the compiler will inject three bytes of padding in between the two, to align the int to a 4 byte boundary.

The issue 1.b might seem self explanatory, but is the one of the most common sources of excessive bandwidth consumption. There are a variety of ways to identify such a problem. In an application dominated by loops a simple, rough way to estimate the effect is to measure the number of times the dominant loops are executed and how many bytes of data they consume per iteration by static analysis of the assembler. The product is an upper limit on the total number of bytes consumed by the loops. Measuring the total number of cachelines transferred on the bus by the loops and multiplying by 64 gives the total number of bytes transferred on the bus. Dividing the total bytes consumed (from the assembly analysis) by the total bytes transferred gives a rough measure of the usage fraction of the consumed bandwidth.

To measure the execution count, average the count of the event INST_RETIRED.ANY_P over the instructions of the loop. This is the basic block execution count. This event does not display a uniform distribution over a basic block; however the total count will be correct over a "large enough" region. The total value for an application is correct, but long latency instructions will collect many more samples than the instructions immediately before or after within the same block. In a loop with a dominant flow through several basic blocks the event should be averaged over all the basic blocks of that dominant flow, assuming the user can demonstrate that in fact they are all executed an equal number of times by static analysis of the assembler. This can be done with the sampling data displayed by the VTune™ Analyzer. Simply highlight all the instructions in the loop and the sum of the samples will be displayed. Count the number of instructions in the highlighted region. Multiply the sum of INST_RETIRED.ANY_P by the Sample After Value (SAV) and divide by the number of instructions. This will be the total execution count.

This technique can also yield the average inner loop tripcount by dividing the execution count of the inner loop by the execution count of the basic blocks before or after the loop. This will become very inaccurate if the inner tripcount is much greater than 100, but is quite accurate for small values of the inner loop tripcount. However, while the technique is inaccurate for high tripcounts, simply knowing the tripcount is very large is usually useful to know.

A simple inspection of the loads and stores can yield an estimate of the total number of bytes consumed per iteration. This of course risks over counting due to identical pointers, particularly when considering multiple loops.

The total number of cachelines consumed is measured with the event BUS_TRANS_BURST.SELF.

Another paper, with a detailed discussion of data access analysis, is in preparation and should be available in the near future.

In cases where only a small fraction of the data transferred over the bus is actually consumed during the high execution loops, there are several options to improve bus usage and thereby both single and parallel performance. The most obvious is to break up the data organization by actual usage, organizing the data in parallel arrays or linked lists of structures. A common situation is where only a subset of each structures' data is heavily consumed many times, while most of it is used only occasionally. In a large application, completely reorganizing the basic data layout may prove extremely difficult. In such a case, the heavily used data can be copied into a convenient, packed array of structures containing only the heavily used elements. This will ensure that the heavily executed loops will only require the minimum amount of bandwidth for their execution. A more aggressive optimization would include "unrolling" the data layout by a factor of 2 if there are floating point double precision data or by 4 if the data is all int's and floats. This will allow extremely effective use of the Intel® Streaming SIMD Extensions 3 (SSE3) instruction set, as no pack instructions will be required. It might even be worth adding a few dummy elements at the end to ensure an even multiple of 2 or 4 array elements.

It may simply be better to use structures of arrays rather than arrays of structures if serial access through an array list is the dominant access pattern as this ensures that all bytes of the cachelines will be used.

Non-Temporal Stores

Compilers will avoid generating non-temporal streaming stores in cases where the loop tripcounts for data assignment loops are unknown. For example in the simple triad function shown below, the value of len cannot be known at compilation time in general.

```
double TRIAD(int len, double a1, double * a, double *b, double*c, double*d){
        int i;
        for(i=0; i<len; i++)a[i] = b[i] + a1*d[i];
        return;
}
```

Most compilers will use a regular store or an SSE packed store instruction for the array "a". This will result in a read for ownership of the cacheline, thereby doubling the bandwidth required for this array. If in fact the array is not actually needed right away or is too large to remain cache resident, this factor of two can be reduced. In the triad example above, if the tripcount "len" times 8, is larger than the LLC size divided by 3, then the first cachelines for the array “a” will have been evicted by the time the loop finishes. To ensure that the Intel® compiler generates the optimal encoding for a long tripcount, one has to ensure that the arrays are 16 byte aligned and then modify the loop by adding two pragmas immediately before the loop.

```
#pragma vector aligned
#pragma vector nontemporal
        for(i=0; i<len; i++)a[i] = b[i] + a1*d[i];
```

Of particular importance also, is excessive cacheline prefetching. Poorly encoded software prefetches can aggravate bandwidth limits. Be sure to avoid prefetching cachelines that will not be read. The prefetch distance also needs to be sufficiently large to actually bring the line into the cache prior to its use. Load instructions that reference cachelines that are still being sw prefetched will increment the event LOAD_HIT_PRE. Thus this is quite easy to identify.

There are 4 hardware prefetchers in Intel® Core™2 processors. Two work with the L2 cache in a manner similar to what was seen in Pentium® 4 processors. There are 2 additional hardware prefetchers working with the L1 data cache, a streaming prefetcher and a prefetcher that looks for striding patterns associated with a particular Instruction Pointer (IP). Typically, systems will have the two L2 cache prefetchers enabled and the L1D IP prefetcher enabled but not the L1D streaming prefetch. It can be useful for analysis purposes to disable the hardware prefetchers. This is usually done in the bios setup, under the advanced folder and then under a “cpu options” type selection. Keywords like “prefetch” and “adjacency” are usually the selections that need to be changed. **Contact with the system vendor or manufacturer for the specifics of any platform before attempting this. Incorrectly modifying bios settings from those supplied by the manufacturer can result in rendering the system completely unusable and may void the warranty.** On some Linux* systems, there are drivers in the OS that can change these settings without having to resort to manipulating the bios settings.

The performance events can be used to determine which hardware prefetchers are enabled. The event L2_LD.SELF.PREFETCH counts the cachelines brought into the L2 cache due to the L2 hardware prefetchers. The event L1D_PREFETCH.REQUESTS will identify if the L1 data cache prefetchers are enabled.

To measure the number of unread cachelines that the hardware prefetchers are requesting, run the program with the HW prefetchers enabled and again with the prefetchers disabled. Measure the total number of cachelines that are transferred with the event BUS_TRANS_BURST.SELF. If there is a significant difference between the two collections, use the VTune™ Analyzer to drill down to identify the region of source where this is occurring. Some frequent causes are

1) nested loops with short inner tripcounts and large strides in the outer loops
2) nested loops with opposing loop directions (outer loop forward with large stride, inner loop backward with small stride)

The hardware prefetching will be driven by the inner loop typically, so in the first case the prefetched lines will be beyond the end of the accesses defined by the inner loop, but will not be used by the next iteration of the outer loop. In the second case the prefetcher may try to prefetch lines that have already been used and will probably not prefetch anything. This situation is very common in iterative solvers and in most cases can be remedied by simply changing the direction of the inner loop and any required pointer definitions. Both cases are likely to have many cachelines requested by the load and store instructions. This can be identified with the MEM_LOAD_RETIRED.L2_LINE_MISS, which only counts loads, and the L2_LD.SELF.DEMAND, which counts both loads and stores to cacheable lines, but also includes the requests due the L1 prefetchers.

Data blocking is a standard solution for bandwidth limitations, usually offered without any advice on how it might actually be accomplished. It is easier said than done. However, there is an intrinsic relationship between data decomposition and data blocking and this can be applied in some cases. A data decomposition can be designed to create a two level hierarchy. The outer decomposition level making one data set per core, and equating to one per thread or process, depending on the decomposition, openmp/explicit threading or MPI for example. The data set per core might then follow exactly the same decomposition strategy as used at the higher level, subdividing the data set further, creating a set of "virtual nodes", whose data set size is determined by the last level cache size. This general strategy should be considered particularly in the early algorithm design phases as it is likely to greatly enhance scaling capabilities.

Disk Subsystem Oversubscription

Oversubscription of the shared disk subsystem means that the fraction of time spent accessing the disk subsystem grows at a greater rate, with the number of cores used, than the rest of the application. This is easily identified in the module view of the VTune™ Performance Analyzer by simply comparing the fraction of time spent in the disk driver as the number of used cores is increased. For either of the scaling scenarios (fixed work or linearly increasing work) this fraction should be constant. Of course, this is only of interest if the (increase in) time spent doing disk access is significant.

Non constant behavior means that the accesses are interfering with each other. This can happen for example due to the different accesses forcing erratic movements in the disk heads and defeating the streaming access which is most efficient.

In cases where it is not constant it is almost always best to serialize the disk access in time and keep the other cores busy doing something else. Thus a misalignment of the phases of the individual threads or processes can be used to avoid saturating the disk access.

Shared Cache Capacity

Non scaling behavior due to oversubscription of a shared cache means that the cache sharing is resulting in a greater amount of cacheline eviction and subsequent cacheline reloading. On Intel® Core™2 processors a Last Level Cache (LLC) line miss is counted by the event L2_LINES_IN.SELF.ANY This event counts LLC misses due to loads, stores, instruction fetching, software and hardware prefetches. For the two scenarios being discussed the signatures differ but only in an obvious manner. If the total work is fixed, then non-contentious parallel execution would result in the total count of the LLC line misses over the execution period being constant as the number of cores used is increased. If the work scales with the number of cores then the total count per core should remain constant as the number of used cores is increased. In either case it is the total count and not the rate that is indicative of the nature of the issue.

There are several issues that can cause these counts to not follow the patterns discussed above. A few are

1) True and false sharing of cachelines causing extra cacheline traffic
2) The working set size distributions' compatibility to the LLC size is degraded with multi-core execution, i.e. the parallel processes evict each others' lines

The first case is easily identified as (true or false) cacheline sharing with the use of the EXT_SNOOPS.ALL_AGENTS.HITM. This event counts the number of times an LLC miss identifies the required line being in another cache and in a modified state. This situation will be discussed at length in the last section of this paper.

The second issue is the main point of this section. The standard example might be an application that performs a large matrix multiply during its execution and that has data blocked the matrix multiply to use more than half (most?) of the capacity of the LLC. The parallelization might result in each thread or process multiplying smaller total matrixes due to the data decomposition, however, if the data blocking is not modified, two simultaneous matrix multiplies would require twice the LLC space that one would and thus more than is available in the LLC. This would then result in cache thrashing with the two threads or processes competing for space and evicting each others cache lines. Such a situation would result in a significant increase in the total number of cachelines read (L2_LINES_IN.SELF.ANY) over the entire program execution, with the change not being related to hitm cacheline access.

Another technique that can be used to help reveal details about how the performance degradation from ideal scaling is due to excessive evictions is to measure the working set size distribution of the application in the large and small number of node cases. This can be done with a PIN tool that measures the Cache Stack Distance (CSD) during the two executions. This technique creates a stack of cachelines, adding each new cacheline access to the top and counting down to the stack until the previous entry is found and removed. The distribution of these cache stack depths is the distribution of the working set size of the application. The relative locations of significant peaks or "shoulders" in the distribution with respect to each cores' share of the LLC will illuminate the degree to which the individual threads or processes are evicting each others cachelines prematurely. Thus if two threads or processes are to run simultaneously in a shared LLC, without interfering with each other, the working sets sizes would have to fit within half of the LLC cache size

If such a mutual eviction is occurring, it implies that a degree of data blocking was achieved when only one thread or process had sole use of the shared cache. Solving a problem of mutual thread/process eviction requires that the achieved data blocking must also scale with the usage, so that the multiple threads/processes coexist within their shares of the limited cache size.

Of course a data decomposition run on many cores and systems (MPI?) might result in the each cores' problem fitting into cache completely, leading to a large drop in cacheline bus traffic, so that possibility should be kept in mind.

DTLB Capacity

The Intel® Core™2 processor has a dedicated 2 level dtlb system for each core. Consequently an application with a "reasonable" data decomposition should use a proportionally smaller number of dtlb entries in our first scenario and a constant number in our second. If the rate of dtlb misses actually increases, this would imply that the data decomposition has increased the total number of pages accessed while the total amount of working set size decreased or remained constant. Clearly this would indicate that the data decomposition page alignment can be substantially improved. This can almost only occur in a threaded application, as a process parallel decomposition creates non overlapping virtual address spaces. Such a situation could arise in a threaded/shared memory application through the use of multi dimensional arrays if the data decomposition was over the leading rather than trailing dimensions. Identifying such a situation can easily be accomplished by counting MEM_LOAD_RETIRED.DTLB_MISS (loads only and precise) or DTLB_MISSES.ANY (loads and stores, not precise). If there is an issue, then sampling on the same event using a large and small number of cores and locating the IP region where the non scaling is manifesting itself in the VTune™ Analyzer displays (module, hotspot and then source) will identify where in the program the problem is occurring. At that point inspection of the source should make the issue relatively easy to isolate.

An alternative follow up analysis might include making a virtual address access histogram. If the access regions are not contiguous and interleaving, then there will clearly be a lot of problems.

Individual Cacheline Access

Data is transported and accessed in cachelines of contiguous pieces of 64 bytes on Intel® Core™2 processors. Shared memory parallel execution can result in contention for access to individual cachelines. Thus whenever two threads, working in the same address space, both use ranges within one of these 64 byte cachelines there can be contention for access, with the execution of the threads that are not the lines' current owner becoming blocked. This problem cannot occur for process parallel execution as separate processes use separate address spaces.

Cacheline sharing does not have to cause an access contention issue. The four state (MESI) cacheline protocol allows for a coherent use of data in a multi-core, multi-socket platform. A line that is only read can be shared and the cacheline access protocol supports this by allowing multiple copies of the cacheline to coexist in the multiple cores. Under these conditions the multiple copies of the cacheline would be in what is called a Shared state (S). Once one of the copies is modified the cacheline's state is changed to Modified (M). That change of state is propagated to the other cores, whose copies are changed to the Invalid state(I). A cacheline can be put in an Exclusive state(E) by applying a lock prefix to the memory access instruction or by using an xchg instruction and so on. This ensures exclusive access of the line. This is the underlying mechanism of the mutex lock synchronization techniques used to coordinate threaded activity.

Cacheline access in shared memory execution divides into two sets of two options. If the thread access ranges within the line overlap, this is true sharing. If the ranges within the line do not overlap, this is false sharing. A second independent but related distinction is defined by whether the access uses a lock on the cacheline or not. Contention for unlocked, shared lines will frequently result in unstable runtimes, performance event counts and even results. This is due to the contention resolution being dependent on the detailed timings of the contentious accesses. If the runtimes are unstable but the results are stable this is likely due to unlocked false sharing. If the results and runtimes are unstable then this is a clear sign of unlocked true shared accesses, frequently this is the result of a race condition. The use of the Intel® Thread Checker can prove extremely effective at identifying the race conditions and the associated access conditions.

Locked sharing is the basis of thread synchronization. Contention for a locked variable or cacheline will serialize access. This will lower the fraction of execution that can occur simultaneously and lead to non scaling performance.

The performance events allow the measurement of the traffic of these contested lines. Contested cacheline access can cause an increase in load driven (demand) last level cache misses and even hardware prefetched cachelines. When a load induces a last level cache miss, the execution may stall for the full latency required to retrieve the cacheline. When a store to the variable is attempted a read for ownership must be executed to have the line in an exclusive state. The impact of parallel execution on bandwidth and the non scaling execution that can result has already been discussed. That is related but distinct from what is discussed here.

There are a spectrum of events that are useful in identifying these kinds of problematic cacheline access. The following is a short list of some of the more obvious ones.

1) EXT_SNOOP.ALL_AGENTS.HITM
2) MEM_LOAD_RETIRED.L2_LINE_MISS
3) MEM_LOAD_RETIRED.L2_MISS
4) BUS_TRANS_BURST.SELF
5) BUS_TRANS_RFO.SELF
6) BUS_HITM_DRV
7) L2_LD.SELF.E_STATE
8) L2_LD.SELF.I_STATE
9) L2_LD.SELF.S_STATE

The EXT_SNOOP event counts the number of times an LLC miss triggered a "hit-modified" (hitm) response from another core over the bus. The BUS_HITM_DRV is the "reciprocal" event counting how many times a particular core supplied such a response. The MEM_LOAD RETIRED events are precise, identifying the exact loads that cause the miss in L2. Such a miss will only cause one transaction on the bus during the period that the cacheline is not available. Thus the L2_LINE_MISS version counts the number of lines and the L2_MISS version counts the total number of misses. The difference is indicative of the number of multiple accesses to a given line that occur between when the first one occurs, inducing the bus request and when the line arrives from the bus.

The case of unlocked false sharing presents some complexities to the problem, particularly, that the L1D IP hardware prefetcher is likely to identify the localized cache miss pattern and attempt to prefetch the cachelines. This is particularly true to false sharing, occurring in loops, when the loops are being simultaneously executed by multiple threads. The difficulty that this causes is that the precise event MEM_LOAD_RETIRED.L2_LINE_MISS may not increment as the LLC misses become attributed to the IP prefetcher. These LLC misses will be included in the L2_LD.SELF.I_STATE. In the case of false sharing the MEM_LOAD_RETIRED.L2_MISS can therefore be useful in localizing a false sharing access problem in a program. By comparing the number of events when there is only one thread and when multiple threads are run, the sudden appearance of large numbers of these events in conjunction with many EXT_SNOOP events in the VTune™ Analyzer source display spreadsheets can be very helpful in identifying where the problem is occurring.

Perhaps the simplest example we might consider is the parallel accumulation of the values of the elements of an array. In such a case each thread would sum its share of the array elements. Such a function might look a bit like:

```
    int sum(int* data,  int* sum, int size, int tid)
    {
        int i;
        for(i=0; i<size; i++)*sum += data[i]*data[i];
        return *sum;
    }
```

If the pointer “sum” is declared as a simple array with the index being the thread ID, then we might be in danger of the threads fighting over the cacheline containing the array elements. Cleary, the compiler could accumulate the array values in a register (optimizing away “sum”) and avoid the problem, which the Intel compiler does. However not all compilers will do this. If the function is declared as

```
    int sum(int* data,  volatile int* sum, int size, int tid)
```

then the compiler is not allowed to registerize the sum and the battle for the cacheline is guaranteed. The performance events indicated above will light up like roman candles.

As a second example we might consider a very simple triply nested loop of array rearrangements admirably suitable for this discussion.

```
#define MAXTHR 4
#define ITERS 1000
#define SIZE 1000
int aa[MAXTHR][SIZE];
volatile int i[MAXTHR], j[MAXTHR], k[MAXTHR], n[MAXTHR], tmp[MAXTHR];
int sort(int *a, int size, int tid) //a = aa[tid][0]
{
    n[tid] = 0;
    for (k[tid]=0; k[tid] < ITERS/2; k[tid]++){
        for (i[tid] = 0; i[tid] < size-1; i[tid]++){
            for (j[tid] = i[tid]+1; j[tid] < size; j[tid]++){
                if (a[i[tid]] > a[j[tid]]){
                    tmp[tid] = a[i[tid]];
                    a[i[tid]] = a[j[tid]];
                    a[j[tid]] = tmp[tid];
                    n[tid]++;
        }   }   }
        for (i[tid] = 0; i[tid] < size-1; i[tid]++){
            for (j[tid] = i[tid]+1; j[tid] < size; j[tid]++){
                if (a[i[tid]] < a[j[tid]]){
                    tmp[tid] = a[i[tid]];
                    a[i[tid]] = a[j[tid]];
```

```
                                        a[j[tid]] = tmp[tid];
                                        n[tid]++;
                    }         }         }
          }
          return n[tid];
}
```

It is clear from inspection that the loop index arrays i,j,k and the tmp array will likely exist in a single cacheline that the multiple threads will then fight over for every iteration of every loop. They have been declared globally as volatile to ensure that the compiler does not do the obvious thing and keep the variables only in registers, which would be absolutely allowed within language standards otherwise. The language standards permit great latitudes to a compiler in data access optimizations. Without something like a volatile declaration a compiler would not have to consider parallel execution requirements in any manner for the above function. The current Intel® compiler (10.0) for Intel® 64 architecture will build a binary completely free of false sharing at O3 optimization, if the volatile declaration is removed from the above function. None of the local loop and temporary variables are ever committed to memory, existing only in registers. Large compiler dependencies are not untypical of unlocked accesses to shared cachelines. When one considers optimizations like inlining, function splitting and so on, the variability in shared unlocked cacheline access becomes almost limitless.

When data is collected in the VTune™ Analyzer one observes a number of relationships between the event counts. The count of EXT_SNOOP.ALL_AGENTS_HITM is approximately equal to BUS_HITM_DRV. The issues of the run to run timing and event count variations are observed. MEM_LOAD_RETIRED.L2_MISS is much greater than MEM_LOAD_RETIRED.L2_LINE_MISS. Further the event count of MEM_LOAD_RETIRED.L2_LINE_MISS is much less than the count of cachelines transferred on the bus by BUS_TRANS_BURST.SELF. The largest contributions to the whole cacheline bus transactions are the reads for ownership, measured with BUS_TRANS_RFO.SELF. The RFO's account for the bulk of the cachelines transferred, measured with BUS_TRANS_BURST.SELF. The balance is accounted for in the shared reads measured by BUS_TRANS_BRD.SELF. The L2_LD.SELF events can be useful in clarifying the cacheline states, which can be particularly useful if the hardware prefetchers are contributing a significant fraction of the transfers.

Consequently, a reasonable approach to identifying cacheline access contention would be to collect data on a single threaded run to establish a baseline and then collect data on a multi-threaded run. It is probably best to use MEM_LOAD_RETIRED.L2_MISS to identify where the false sharing is occurring by comparing the counts and locations of this event in the source view VTune™ Analyzer spread sheet. When this is done in conjunction with the EXT_SNOOPS.ALL_AGENTS.HITM a reasonably clear picture usually appears.

Identifying locked access conflicts is also rather straight forward. The events L2_LOCK.SELF.E_STATE increments whenever locked accesses (ex xchg instruction) are used to create a mutex lock. If the lock has been modified then L2_LOCK.SELF.M_STATE will also increment. Again the IP prefetcher may result in the event MEM_LOAD_RETIRE.L2_LINE_MISS not being effective at identifying locked accesses. In such a case using MEM_LOAD_RETIRED.L2_MISS and looking for the conjunction with the event L2_LD.LOCK.E_STATE in the VTune™ Analyzer source view creates a clear picture of what is happening. Again the EXT_SNOOP.HITM.ALL_AGENTS will also increment in these cases.

Access contention for locked cachelines is frequently due to the use of synchronization APIs. Simple sampling analysis of the type being discussed is likely to only reveal that the application is spending an inordinate fraction of time in the synchronization wait loop. What is needed is the caller of the API, to see if a coding change can reduce the serial execution forced by the locked variables. The use of the VTune™ Analyzer's call graph utility or the Intel® Thread Profiler can quickly identify the callers of these serialization bottlenecks.

While cacheline contention is restricted to shared memory parallelization models, excessive serial execution due to overuse of synchronous MPI operations can cause similar performance limitations. Synchronous message passing, MPI_Wait, and MPI global operations (MPI_Allreduce for example) can have exactly the same kind of stifling effect on scaling as the effects just discussed. The Intel® Trace Analyzer and Collector are designed to identify exactly these and other kinds of MPI usage problems. The use of this tool is essential in achieving the parallel MPI execution required to effectively use large clusters of Intel® Core™2 processors.

CONCLUSION

The Intel® Core™2 processor has a performance event hierarchy that is very effective for analyzing software execution performance limitations. Many causes of multi-core non-scaling can be quickly and easily identified using the performance events with the Intel® VTune™ Performance Analyzer. More complex threading synchronization issues should consider user the Intel® Thread Profiler. For MPI scaling issues, the Intel® Trace Analyzer and Collector would be recommended.