Cycle Accounting Analysis on Intel® Core™2 Processors
By Dr. David Levinthal PhD.
Intel Corp.

Introduction

In the process of analyzing an applications' performance with the intention of improving it, a detailed accounting for how the cpu cycles are used is one of the most powerful techniques available. Traditional retirement centric analyses have difficulty doing this when the target architecture has Out of Order (OOO) execution because the entire objective of OOO execution is to continue executing instructions during the period that retirement is blocked.

In order to develop a cycle use based methodology, it is necessary to be acquainted with the basic mechanisms of OOO execution.
An extremely simplified block diagram is shown in Figure 1 for this purpose.
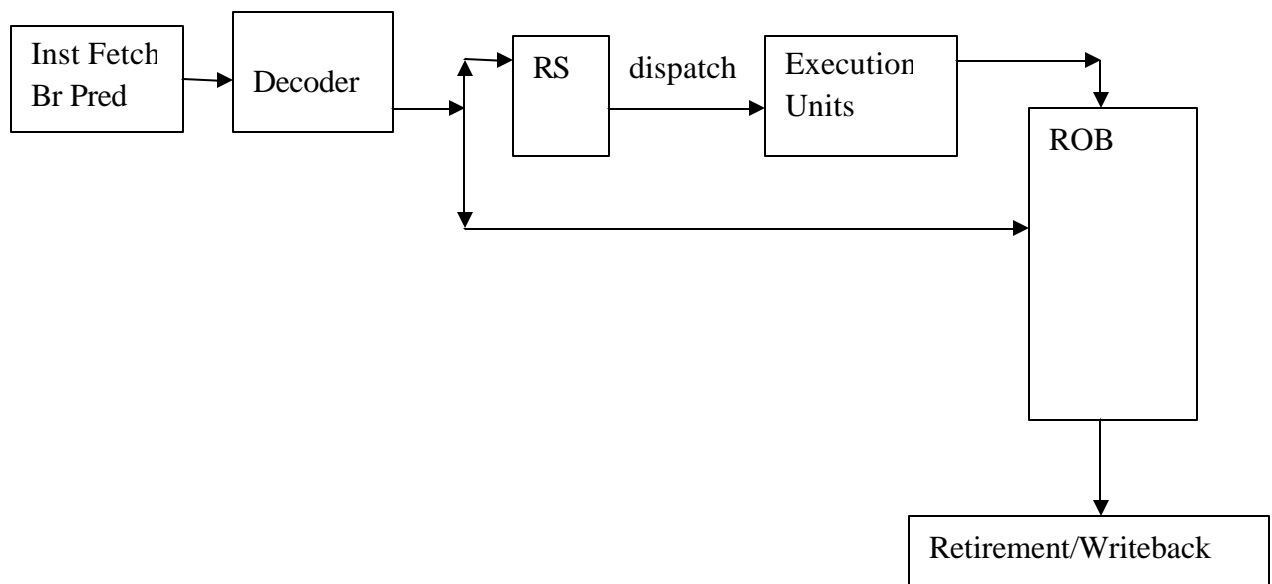

Figure 1

After instructions are decoded into the executable micro operations (uops) they are issued downstream if there are adequate resources. This would include (among other requirements)

1) space in the Reservation Station (RS), where the uops wait until their inputs are available
2) space in the Reorder Buffer, where the uops wait until they can be retired
3) sufficient load and store buffers in the case of memory related uops (loads and stores)

Retirement and write back of state to visible registers is only done for instructions and uops that are on the correct execution path. Instructions and uops of incorrectly predicted paths are flushed upon identification of the misprediction and the correct paths are then

processed. Retirement of the correct execution path instructions can proceed when two conditions are satisfied
1) all the uops associated with the instruction to be retired have completed, allowing the retirement of the entire instruction
2) all older instructions and their uops of correctly predicted paths have retired

The mechanics of following these requirements ensures that the visible state is always consistent with in-order execution of the instructions.

The "magic" of this design is that if the oldest instruction is blocked, for example waiting for the arrival of data from memory, younger independent instructions whose inputs are available can be dispatched to the execution units and warehoused in the ROB upon completion. They will then retire when all the older work has completed.

The difficulty, from the performance analysis perspective, is that this system results in a burst type of flow in the retirement, where cycles of no retirement are followed by streams of maximal retirement flow. Thus what happens on an individual cycle at retirement is not terribly informative and one must use time averages and ratios to acquire a sense of what is happening. The problem that arises is that when ratios are used as the performance metric, the process of optimizing an application will inevitably change both the numerator and denominator, obscuring the progress being made. Consider the standard retirement metric, cycles per instruction (retired) or CPI in the case of optimizing a loop. A high value of CPI is considered a sign of poor performance while a low value is considered a sign of good performance. If during the optimization process the loop is vectorized (compiled to generate SSE instructions) then the number of instructions retired will drop significantly (2X or greater). The change in the cycles consumed is unlikely to decrease at the same rate as this optimization will not change the things like last level cache misses which will cause progress to stop. The net result is that vectorizing a loop is virtually guaranteed to make CPI increase. In fact a technique for minimizing CPI could be to maximize the number of instructions retired!

Rather than focusing on metrics made of ratios of cpu activity events, one can simply focus on minimizing the cycles consumed to accomplish the desired work. A cycle accounting simplifies this by allowing the developer to minimize the individual components of cycle use accounting components. The principal metric is then just cycles and the process of optimizing an application always lowers the principal metric.

Execution Centric Analysis

An out of order execution engine (figure 1) optimizes the instruction flow at the point where the instructions are fed to the execution units. The surrounding buffers accumulate the unissued and finished instructions and reconnect the flow to the originally programmed order. Instruction issue is therefore the point in the pipeline from which this methodology will start the cycle accounting. Such an accounting might also be started from retirement or at any other point in the instruction flow, but as the point of instruction issue is optimized by the hardware to maintain the highest level of activity, it also provides the clearest understanding of the execution.

For the Intel® Core™2 processor methodology outlined here, execution measurements from several different points will ultimately be incorporated. This results in a construction that while not rigorously correct, appears to work as a reasonable approximation.

The description will focus on the flow of the micro-ops (uops) that the execution units actually execute, rather than the x86 instructions generated by the compiler. The role of the pipeline's front end can be thought of as retrieving the required x86 instructions from the memory subsystem, translating them into the uops actually executed and buffering them for transfer to the execution stages. This discussion will not cover that activity in any detail.

The out of order execution, for the purposes of this discussion, can be thought of as working roughly as follows: The uops sit in the Reservation Station (RS) until their inputs are available and proper execution resource free. At that point they are dispatched to the appropriate port to be executed on the execution unit needed.. A priority scheme is in place for the case that more than one uop is candidate for execution. On any cycle up to 6 uops can be dispatched, as there are 6 dispatch ports. Upon completion of a uop the ROB is updated with the uop result and fault information. When all the uops associated with an instruction have completed, the set (i.e. the instruction) is marked ready for retirement. A given instruction can only retire when all older instructions have already retired, restoring the execution order. At most 4 uops can be retired per cycle.

These cycles can be decomposed into two exclusive sets

Total Cycles = Cycles issuing uops + Cycles not issuing uops

The component of cycles where no uops are issued may be thought of as execution stalls.
The event CPU_CLK_UNHALTED.CORE counts the core cpu cycles in the unhalted state. If you are sampling on an executing application or counting in user mode only (to avoid counting cycles during the null process that is "active" during the halted state) then

Total Cycles ~ CPU_CLK_UNHALTED.CORE

The component of the cpu cycles issuing uops can be artificially further decomposed into a component for retiring uops (Retired) and for non retiring uops (Non_Retired). Of course retiring and non retiring uops may well share cycles, but this allows a cycle decomposition into productive and non productive components. In this approximation these components are proportional to the number of uops in each group.
Thus the sum could be written as:

CPU_CLK_UNHALTED.CORE = Retired + Non_Retired + Stalls (1)

Such a decomposition aligns with existing performance events and each of the terms can be evaluated. Equation 1 will be the master cycle accounting equation. The objective of optimizing an existing program is to minimize this sum, by whatever means are practical.
The event RS_UOPS_DISPATCHED counts the number of uops dispatched from the RS on every cycle. The performance event counters in the Performance Monitoring Unit (PMU) can be programmed to accumulate the total. Alternatively it can compare the value on each cycle to a reference value and increment the counter by one if the value is

>= or < the reference value. In this mode cycles are counted. If the reference value is set to 1 (CMASK=1) and the comparison is set to "less than" (INV = 1), then the cycles where no uops are dispatched are counted. In this discussion, this is defined to be the stalled cycles. Conversely if the reference is set to 1 (C=1) and the comparison is left in default mode (INV = 0, >=, default), then the number of cycles dispatching uops is counted. The sum of these two is the total cycles; since on any given cycle you are either dispatching uops or you are not. For the purposes of analyzing the performance of an application we can ignore the difference between total cycles and CPU_CLK_UNHALTED.CORE, as long as the events for the null process associated with the halted state can be isolated. This is most easily accomplished by using event sampling with the VTune™ Performance Analyzer and considering only the samples associated with the process being analyzed. Thus

$$CPU\_CLK\_UNHALTED.CORE \sim RS\_UOPS\_DISPATCHED:C=1 +$$
$$RS\_UOPS\_DISPATCHED:C=1:I=1$$

And the last component of the master cycle accounting equation is

$$Stalls = RS\_UOPS\_DISPATCHED:C=1:I=1 \quad (2)$$

To simplify the PMU programming the VTune™ Performance Analyzer has predefined

$$RS\_UOPS\_DSPATCHED.CYCLES\_NONE = RS\_UOPS\_DISPATCHED:C=1:I=1$$

RS_UOPS_DISPATCHED clearly counts both uops that will ultimately be retired and uops that are speculatively executed but ultimately not retired. The event UOPS_RETIRED.ANY counts the uops which are retired. The event UOPS_RETIRED.FUSED counts the number of those that represent the fusions of two executed uops. Therefore
$$retired\_uops\_executed = UOPS\_RETIRED.ANY + UOPS\_RETIRED.FUSED$$
is the total number of uops that are executed in the production of useful work. Similarly
$$RS\_UOPS\_DISPATCHED -$$
$$(UOPS\_RETIRED.ANY + UOPS\_RETIRED.FUSED)$$
$$RS\_UOPS\_DISPATCHED - retired\_uops\_executed$$
is the total number of executed but non retired uops, representing non-productive work.
If the average rate of issuing uops is the number of dispatched uops divided by the cycles dispatching uops
$$uop\_rate = RS\_UOPS\_DISPATCHED/RS\_UOPS\_DISPATCHED:C=1$$
Then the cycles devoted to issuing non retired uops, the second term in the master equation, can be approximated as

$$Non\_retired = (RS\_UOPS\_DISPATCHED - retired\_uops\_executed)/rate \quad (3)$$

and with CPU_CLOCK_UNHALTED.CORE, all three components of the master cycle accounting equation can be determined. Again, it should be pointed out that this does not

represent a rigorous cycle decomposition, but rather a division of the cycles into useful sets for the purposes of performance analysis.

An overall description of the optimization process might then be given as a three part process.

1) Minimizing the "Retired" component by minimizing the instructions generated by the compiler by vectorization and other techniques.
2) Minimizing the "Stalls" by removing memory access and other bottlenecks.
3) Minimizing the "Non-retired" component by reducing the branch mispredictions

Decomposition of Stalls

The decomposition of the stall cycles is accomplished through another level of approximation. If the assumption that the cycle penalties for each performance impacting event occur sequentially, then the total loss of cycles available for useful work is simply the number of events, N_i, times the average penalty for each type of event, P_i.

$$\text{Lost\_cycles} = \mathbf{S} \; P_i * N_i$$

This only accounts for the performance impacting events that can be counted with a PMU event. Ultimately there will be several sources of stalls that cannot be counted, however their total contribution can be estimated by the difference of

$$\text{Unaccounted} = \text{Stalls} - \text{Lost\_cycles}$$
$$= \text{RS\_UOPS\_DISPATCHED.CYCLES\_NONE} - \mathbf{S} \; P_i * N_i$$

The unaccounted component can become negative as the sequential penalty model is overly simple and usually over counts the contributions of the individual architectural issues. It is meant to represent the components that were either not counted due to lack of performance events or simply neglected during the data collection.

There are several components which cannot be counted reliably on the Intel® Core™2 processor. These fall into three main classes: stalls due to instruction starvation, stalls due to dependent chains of multi cycle instructions (other than divide) and stalls related to FSB saturation. This last term is only an issue on extremely high bandwidth applications and typically requires heavy use of either the HW prefetchers or SW prefetch intrinsics.

Measuring FSB saturation is straightforward. This can be done in terms of 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 cachelines transferred:

Cacheline_Bandwidth ~
64*BUS_TRANS_BURST.BOTH_CORES.ALL_AGENTS*freq
/CPU_CLK_UNHALTED.CORE

Due to the large number of factors that effect the maximum achievable bandwidth on a given system it is always best to first measure the actual limit on a given system by using a simple triad test case.

If the above metrics are used, it will turn out that the limits are the essentially the same whether the compiler generates SSE streaming stores or regular stores that result in reads for ownership.

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
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 $lines Instr
BUS_TRANS_RFO	Whole $lines Read For Ownership
BUS_TRANS_WRB	Whole $line Write Backs (modified $lines)

The stalls due to chains of long latency instructions can be estimated with static analysis of the instructions and basic block execution counts.

Instruction starvation can be seen to some degree by an anti correlation with the event RESOURCE_STALLS.RS_FULL.

Branch Mispredictions and Speculative Execution

In a decomposition of execution inefficiency, the treatment of mispredicted branching and incorrect speculation needs to be specifically addressed. In the simplified OOO engine described in figure 1, branches are "executed" by the front end, and the prediction is checked by the jump execution unit in the OOO machine. This is started by the branch prediction, upstream of the decoding, selecting the expected instruction stream, which is then decoded. In the stream of uops through the OOO engine all that needs to be determined is that the branch was correctly predicted. In the case of a fused compare and branch operation, the compare must also be executed and also goes through the RS. This style of execution results in three components of unused work by the processor in the case of a branch misprediction.

1) Some number of the mispredicted uops are routed through the execution units. This component is included in what has been called the Non-Retired component identified in equation 3.
2) The ROB and RS must then be purged of these incorrect uops. These stall cycles can be approximated with the event RESOURCE_STALLS.BR_MISS_CLEAR and can be easily included in the stall decomposition as the units of the event are cycles
3) Finally, there may be some number of cycles where nothing is dispatched as the pipeline waits for the correct uops to arrive at the front end, be decoded and pushed through the RS. There are no events that allow a direct estimate of this instruction starvation component.

Estimating Penalties for Countable Performance Issues

It is essential to know the penalties associated with the countable events in order to estimate their relative impacts. A description of the memory access related stalls will illustrate the technique and highlight the use of several very useful performance events. The memory subsystem of the Intel® Core™2 processor has a 2-level cache. The precise

events for counting the line misses due to demands (loads) are MEM_LOAD_RETIRED.L1D_LINE_MISS and MEM_LOAD_RETIRED.L2_LINE_MISS respectively. These events are precise in that they use the Precise Event Based Sampling (PEBS) collection to determine the precise IP of the load that triggered the event when the data is collected in sampling mode (discussed later in this paper). The 2-level DTLB system has separate entries for both large and small pages and performance events capable of counting a variety of actions associated with their use. For this discussion only the L1 DTLB misses to small pages will be considered.

As an example of how penalties might be evaluated, consider the class of memory access penalties. To measure memory access latencies, a pointer chasing linked list access is very effective at isolating the memory subsystem and ensuring the OOO engine does not effectively prefetch data by speculatively executing loads well in advance. Such a linked data list might be constructed with a loop like:

```
arr = buf;
LinesCnt=1;
while(LinesCnt < Numaccess)
{
        *arr = ((INT_PTR)arr + buf_offset);
        arr = (INT_PTR*)((INT_PTR)arr + buf_offset);
        LinesCnt++;
}
*arr = (INT_PTR)buf; // make it a loop
```

Then a simple loop, like the following, chasing the pointer list a fixed number of ITERATIONS, can be timed

```
void do_read(INT_PTR count, INT_PTR* a){

      //zero out COUNTER and arrayPOSITION
            xor eax, eax
            mov rdx, a
LOOP1:
      //quit if loop has run allotted number of times
            cmp eax, ITERATIONS
            jge STOP
      //increment iteration counter
            inc eax
      //load value in a[arrayPOSITION] into arrayPOSITION
            mov rdx, [rdx]
      //goto LOOP
            jmp LOOP1
```

A similar loop with the load removed and replaced by a nop can be used to measure a baseline time. The difference in times for the two asm loops, divided by the number of iterations is defined as the penalty. The value of LinesCnt determines where in the memory hierarchy the data will reside (L1, L2, or main memory) and the stride will determine how the DTLB subsystem is invoked. Thus the penalties for both the memory access latency and the DTLB subsystem can be extracted with a single test. The penalties for other architectural pitfalls discussed in this document can be evaluated with similar kinds of micro benchmarks.

It should be noted that the penalty values evaluated in this way are **only approximations**. They are the typical values observed for a specific set of tests. In a real application the penalties may vary due to precise details of the application and the platform (chipset, DIMMS etc), may overlap, or even be completely hidden by the combination of the OOO engine and the compiler. Any values shown here are strictly what was measured with the particular tests used, were evaluated on the particular platforms available for the testing and are for illustrative purposes only.

Example issues and penalties

Issue	Performance Counter	Approximate Penalty
L2 Hit	Mem_load_retired.L1_line_miss-mem_load_retired.L2_line_miss	~12
L2 Miss	Mem_load_retired.L2_line_miss	~165 desktop/~300 server
L1 DTLB Miss	Mem_load_retired.Dtlb_miss	~10
store to unknown addr precedes load	Load_Blocks.Sta	~5
store forwarding 4 bytes from middle of 8	Load_Blocks.Overlap_store	~6
store to known address precedes load offset by N*4096	Load_Blocks.Overlap_store	~6
load from 2 cachelines (not in L1D)	Load_Blocks.UNTIL_RETIRE	~20
Length Changing Prefix (16 bit imm	ILD_STALLS	6

The DTLB penalty can be more accurately evaluated (instead of merely assigning 10 cycles) by using the event PAGE_WALKS.CYCLES to incorporate cases where complex page walks cause an increased penalty. This event counts the duration of the page walks required to resolve the DTLB miss, and can be sampled to localize the cycles in IP. The approximation of

Penalty ~ MEM_LOAD_RETIRED.DTLB_MISS * 4 + PAGE_WALKS.CYCLES

has been observed to work fairly well

The length changing prefix penalty occurs in the decoder, so its impact is weakened by having uop buffer (the RS) between where it occurs and the execution. In addition if the Intel® compilers are used, this almost never occurs.

These approximate penalties are likely to over count, particularly for the events with smaller penalties. It is easier for the compiler and OOO engine to obscure their relatively small effects. To identify real performance problems it is best to use sampling and correlate the stalled cycles, as measured by RS_UOPS_DISPATHED.CYCLES_NONE, to these events in the VTune Analyzer's source or asm view spread sheets. If there are no stalled cycle counts in the vicinity of the event counts, then modifying the code or compilation to reduce the event counts is unlikely to change the performance, as the hardware has already hidden their impact. This is the critical point of having a stall cycle measurement and a cycle accounting methodology. It allows a developer to identify where their optimization efforts are likely to be fruitful and more importantly to identify where they would waste their time and effort.

Analyzing an Application

The virtue of the above discussion is that it immediately leads to a systematic approach to selecting which events to monitor and how to interpret the data. This can be broken down into a series of options, where the decision of what data to collect is based on knowledge of some simple properties of the application and the experience of the analyst.

When collecting sampling data with a performance analysis tool, like the VTune™ Analyzer, one of the critical inputs is what is known as the "Sample After Value" or SAV. This determines the frequency with which an interrupt is generated, allowing the tool to sample the locations in the application (instruction pointer or IP) associated with the event occurrences. The counter is programmed to the overflow value minus the SAV and after SAV events an interrupt is raised. A driver then collects the required data and reprograms the counter to the overflow value minus the SAV. This is repeated throughout the applications execution or until the desired analysis time is reached. The VTune Analyzer can easily sample at a rate of a KHz/event, so for Intel® Core™2 processors the event CPU_CLK_UNHALTED.CORE can be 2 million. If the other events selected are assigned SAV values equal to the SAV value of cycle event divide by the penalty, then the user will only see a significant number of samples if the event is actually causing a significant performance impact. Lowering the SAV value due to insignificant samples only results in determining how irrelevant that event is to performance, and an excessive amount of disk space being wasted. The other virtue is that the ratio of the event sample counts to the cycle sample counts is the ratio of the penalty cycles to the total.

A simple overview analysis can be performed with just 4 events, the big 4.

BIG 4	SAV
CPU_CLK_UNHALTED.CORE	2,000,000
RS_UOPS_DISPATCHED.CYCLES_NONE	2,000,000
BUS_TRANS_ANY.SELF	100,000
MEM_LOAD_RETIRED.L2_LINE_MISS	10,000

Thus the SAV for MEM_LOAD_RETIRED.L2_LINE_MISS is set to the value for CPU_CLK_UNHALTED.CORE/200 (10,000) as a penalty of 200 cycles for an L2 miss is about right. This event selection measures cycles, stalls, bandwidth and long latency memory accesses, delivering the most information for the fewest events. It will take two data collections as it requires 3 events that must be programmed into the two general counters.

Applications can for the most part be divided into two classes, data layout dominated and data value dominated . The first class can be thought of as loop dominated applications that walk through a fixed data layout, typified by HPC, bandwidth dominated applications. The second set is more typified by pointer chasing, latency dominated applications typified by data base transaction processing. These can also be thought of as being loop vs. branch dominated.

A more profound analysis could start with the following list:

FIRST PASS EVENTS	Sample After Value
CPU_CLK_UNHALTED.CORE	2,000,000
RS_UOPS_DISPATCHED.CYCLES_NONE	2,000,000
UOPS_RETIRED.ANY	2,000,000
UOPS_RETIRED.FUSED	2,000,000

RS_UOPS_DISPATCHED	2,000,000
MEM_LOAD_RETIRED.L2_LINE_MISS	10,000
INST_RETIRED.ANY_P	2,000,000

Where the last event INST_RETIRED.ANY_P is the instructions retired event that uses the Precise Event Based Sampling (PEBS) mechanism. This event will not produce a uniform distribution of samples over a basic block, but the average value of this event over a basic block tends to be a better measure of the basic block execution in loops. This allows an estimate of the average loop tripcount for example, which is extremely important knowledge for loop optimization.

These events should be supplemented with events allowing a bandwidth or bus traffic measurement for loop dominated applications. For example:

Loops	Sample After Value
BUS_TRANS_ANY.SELF	100,000
BUS_TRANS_ANY.ALL_AGENTS	100,000

Or of the cycles lost flushing the pipeline after branch mispredictions in the case of branch dominated applications:

Branch Dominated	Sample After Value
RESOURCE_STALLS.BR_MISS_CLEAR	2,000,000

If more details are desired the following events might prove useful:

SECOND LEVEL EVENTS	Sample After Value
MEM_LOAD_RETIRED.DTLB_MISS	20,000
MEM_LOAD_RETIRED.L1D_LINE_MISS	200,000
BR_CND_EXEC, BR_CND_MISSP_EXEC	2,000,000
BR_CALL_EXEC	200,000
BR_CALL_MISSP_EXEC	200,000
ILD_STALLS	200,000
LOAD_BLOCK.STORE_OVERLAP	200,000

The SAV value for MEM_LOAD_RETIRED.DTLB_MISS is a factor of ten smaller than would be indicated by the 10-cycle penalty indicated earlier in this paper. In practice this event occurs quite rarely, allowing the lower SAV and higher accuracy. Knowing the locations of data page changes can sometimes help a developer reconfigure data layout to improve memory access performance.

For floating point intensive codes, it may be of interest to identify the component of the stalls associated with divides, square roots and software assists for handling denormals and such. The stalled cycles that occur while the divide or sqrt unit is executing are measured by the event IDLE_DURING_DIV. One might argue that the cycles spent issuing the uops for floating point software assists should be added to stalls and subtracted from the uop issuing component. This is actually a relatively easy correction to generate.

The count of such software assists can be evaluated with the event FP_ASSIST, which has a penalty of approximately 200 cycles.

Divides, Sqrts and FP SW Assists	Sample After Value
IDLE_DURING_DIV	2,000,000
FP_ASSIST	10,000

These tables of events should serve as a reasonable guide for identifying the most common performance issues in applications. The optimizations that result will almost always prove to be architecture independent. Thus the Intel® Core™2 processor should be considered the analysis platform of choice for any application targeted for the x86 instruction set.

General Comments on Code Optimization

Starting with the basic equations, 1-3, evaluating the three terms of the cycle accounting is straightforward. The tables just discussed can guide the developer in the data collection with the VTune Analyzer. The process of optimizing an application is minimizing all three terms.

Minimizing the instruction count in the critical sections of the code is one of the easier optimizations. Using the Intel® compiler at high optimization levels will do this for you through the generation of SSE 3 simd instructions. For this to work in an optimal fashion it is important to have data 16-byte aligned and to ensure that unrolling loops by 2 (doubles) or 4 (floats) is straightforward for the optimization logic of the compiler.

Another typical strategy is to avoid recalculating quantities and group calculations around the temporal availability of data. Again it is possible to assist the compiler by avoiding complex data access and tricky algorithms, which really just succeeds in confusing the compiler and any other code developers forced to read and/or support the source.

Minimizing stalls is dominated by optimizing memory access in the vast majority of cases. The largest problem is the use of linked lists. Large numbers of disjoint memory allocations connected by linked pointers defeat the HW prefetchers, and further can cause them to retrieve data that will not be used. This in turn may evict data that could have been reused. Allocating large blocks and navigating them in fixed patterns both enables the HW prefetchers and minimizes the instruction counts required for the address calculations.

Another typical issue is clustering data elements together by address that are not used simultaneously in the code. The data layout and data consumption should be tightly coupled. It is far better to have multiple parallel structures defined by data usage as this leads to more complete cacheline consumption.

Both of the optimizations discussed above will lead to reducing DTLB misses as well.

Prefetching data is required to overcome the latency to main memory. In the case of indirectly addressed arrays (of structures, or single elements) it is possible to use the software prefetch instruction to good effect. With the Intel compiler this can be invoked with the mm_prefetch intrinsic from either C/C++ or Fortran. For a simple gather/copy there is no gain, but as long as there is just a reasonable amount of work/gather, it can be extremely effective.

Understanding this example has a general value for loop dominated applications. The reason the SW prefetch does not help for the simple gather copy

For(i=0; i<len; i++)a[i] = b[addr[i]];

is that the OOO engine will automatically unroll the loop and execute all the loads first. This happens by the decoder filling up the RS and then the loads execute as their inputs can all be evaluated in parallel. This effectively acts like a prefetch.

  For a loop using indirectly accessed data that actually does some work, the finite size of the RS restricts the degree of unrolling that the OOO engine can create, and the "effective" prefetching accomplished by RS unrolling is no longer accomplished.
To illustrate this point consider a loop that indirectly (gather) accesses an array of structures that contain an array of FP data that looks like:

```
for(i=0; i<len; i++){
*a1 += a[adr[i]].data[0]*a[adr[i]].data[1];
}
```

The amount of work per gather can be varied in such a synthetic test to go from the 2 FP ops /iteration in the loop above to 16 FP ops/iteration with the following:

```
for(i=0; i<len; i++){
*a1 += a[adr[i]].data[0]*a[adr[i]].data[1]
       + a[adr[i]].data[2]*a[adr[i]].data[3]
       + a[adr[i]].data[4]*a[adr[i]].data[5]
       + a[adr[i]].data[6]*a[adr[i]].data[7]
       + a[adr[i]].data[0]*a[adr[i]].data[7]
       + a[adr[i]].data[2]*a[adr[i]].data[1]
       + a[adr[i]].data[4]*a[adr[i]].data[3]
       + a[adr[i]].data[6]*a[adr[i]].data[5];
}
```

In all such cases a version invoking a SW prefetch can easily be encoded, to use the Intel compiler supported intrinsic _mm_prefetch as:

```
for(i=0; i<len; i++){
  _mm_prefetch(&a[adr[i+pref]].data[0],1);
        *a1 += a[adr[i]].data[0]*a[adr[i]].data[1]
               + a[adr[i]].data[2]*a[adr[i]].data[3];
}
```

Where the variable "pref" is a prefetch distance in iteration counts which can be varied to allow an estimate of an optimal prefetch distance. The results are shown below.

	no prefetch	pref = 8	pref = 16	pref = 32	pref = 64	pref = 96
2 fp ops	34.5	34.9	34.2	37.2	38.7	38.9
4 fp ops	44.5	34.5	33.6	38	42.2	41.4
8 fp ops	74.8	34.8	34.1	38.7	42.7	41.7
16 fp ops	108.9	34.6	34	42.2	50.9	45.6

  The realization that the RS achieves an automatic loop unrolling immediately highlights the virtue of simplifying individual inner loops by loop distribution to the largest degree possible. This both allows the OOO engine to achieve a greater degree of Instruction Level Parallelism (ILP) and simplifies the effort required of the compiler to generate SSE 3 simd instructions. This may require "blocking" the inner loops to keep temporary variables (that became arrays through the loop splitting) in cache along with data values used in separate parts of the chained calculation.

To illustrate this idea consider another synthetic benchmark that creates a nearly endless chain of dependent FP operations. Such a test could be written as

```
for(i=0; i<8192 - 1; i++){
   for(j=0;j<1000;j++){
        var1 = bb[i][j] + 5.*cc[i][j];
        var2 = 5.*bb[i+1][j] + 2.*dd[i+1][j];
        var3 = var2 * cc[i][j] + var1* dd[i][j];
        var4 = (var3+var1)*dd[i][j] + bb[i][j];

        var1 = (var4 + var2)*cc[i][j] + 5.*bb[i][j];
        var2 = (var3 + var1)*dd[i+1][j] + 2.*cc[i][j];
        var3 = (var1 + var2)*dd[i][j] - 5.*bb[i+1][j];
        var4 = (var3 + var1)*bb[i][j] - 2.*cc[i][j];

        var1 = (var4 + var2)*bb[i][j] + 5.*cc[i][j];
        var2 = (var3 + var1)*bb[i+1][j] + 2.*cc[i][j];
        var3 = (var1 + var2)*cc[i][j] - 5.*dd[i+1][j];
        var4 = (var3 + var1)*bb[i][j] - 2.*cc[i][j];
```

continually reproducing the last 8 lines as many times as desired. Such a loop of chained operations can be easily distributed by the use of 4 temporary arrays to hold the values of the variables var1-var4 as:

```
for(i=0; i<8192 - 1; i++){
   for(j=0;j<1000;j++){
        var1 = bb[i][j] + 5.*cc[i][j];
        var2 = 5.*bb[i+1][j] + 2.*dd[i+1][j];
        var3 = var2 * cc[i][j] + var1* dd[i][j];
        var4 = (var3+var1)*dd[i][j] + bb[i][j];
        var1 = (var4 + var2)*cc[i][j] + 5.*bb[i][j];
        var2 = (var3 + var1)*dd[i+1][j] + 2.*cc[i][j];
        var3 = (var1 + var2)*dd[i][j] - 5.*bb[i+1][j];
        var4 = (var3 + var1)*bb[i][j] - 2.*cc[i][j];
        var6[j] = var1;
        var7[j] = var2;
        var8[j] = var3;
        var9[j] = var4;
        }
   for(j=0;j<1000;j++){
        var1 = var6[j];
        var2 = var7[j];
        var3 = var8[j];
        var4 = var9[j];

        var1 = (var4 + var2)*bb[i][j] + 5.*cc[i][j];
        var2 = (var3 + var1)*bb[i+1][j] + 2.*cc[i][j];
        var3 = (var1 + var2)*cc[i][j] - 5.*dd[i+1][j];
        var4 = (var3 + var1)*bb[i][j] - 2.*cc[i][j];
```

and so on.
Using the events RS_UOPS_DISPATCHED and CPU_CLK_UNHALTED.CORE and correcting for any compiler loop unrolling it is straight forward to measure the total cycles for one iteration of the original work vs. the number of uops dispatched on average by the distributed work. The result

cycles/iter vs uops/loop
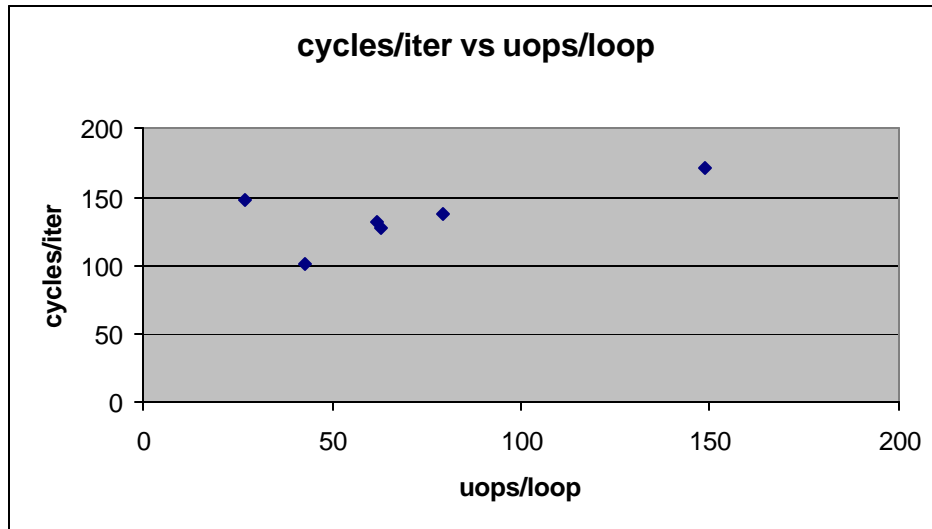
shows a very clear minimum when the number of uops in the inner loop is slightly larger than the RS, meaning that two iterations of the dependent chains of FP operations could be interleaved by the RS.
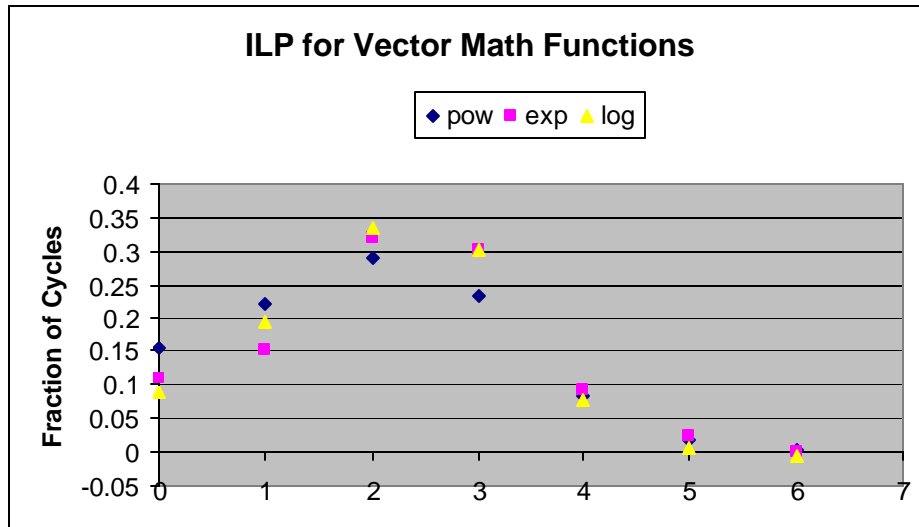
Instruction Level Parallelism

The ability to compare an events value to a reference value on every cycle allows the measurement of the instruction level parallelism (ILP) that an application or even a single loop achieves. If data is collected for all possible values of cmask, the differences of sequential cmask values is the frequency that unique value was seen. Thus for RS_UOPS_DISPATCHED, which can have a value between 0 and 6 one could collect data for:

RS_UOPS_DISPATCHED:cmask=1:inv=1
RS_UOPS_DISPATCHED:cmask=2:inv=1
RS_UOPS_DISPATCHED:cmask=3:inv=1
RS_UOPS_DISPATCHED:cmask=4:inv=1
RS_UOPS_DISPATCHED:cmask=5:inv=1
RS_UOPS_DISPATCHED:cmask=6:inv=1
RS_UOPS_DISPATCHED:cmask=7:inv=1

And by subtracting the sequential values of the data collected for simple loops evaluating the basic math functions evaluating an array of inputs like:

For(i=0;i<len;i++)a[i] = exp(b[i]);

You can produce the distribution of the ILP achieved

ILP for Vector Math Functions

◆ pow ■ exp ▲ log

Fraction of Cycles (y-axis: -0.05 to 0.4)
(x-axis: 0 to 7)

Thus a strategy for optimization would be to minimize the area under a non-normalized form of such a curve by shifting the curve to the right, and achieving greater parallel use of the processors execution units.

Conclusion

The use of cycle accounting allows a systematic minimization of an applications' consumption of processor cycles. Intel® Core™2 processors are the first x86 processors to provide support for this style of performance analysis and application optimization. It should consequently be a developer's first choice for a development platform.

Appendix

A few more useful events

EVENT	Precise	Description
CPU_CLK_UNHALTED		
INST_RETIRED.ANY_P	P	
INST_RETIRED.LOADS		
INST_RETIRED.STORES		
BUS_TRANS_ANY		all bus transactions
BUS_TRANS_MEM		bus trans to memory
BUS_TRANS_BURST		whole $lines to mem
BUS_TRANS_BRD		whole line reads from mem
BUS_TRANS_WB		writebacks (no NT writes)
BUS_TRANS_RFO		$lines in for RFO (no HW pref)
BUS_DRDY_CLOCKS.ALL_AGENTS		all busy bus cycles
BUS_DRDY_CLOCKS.THIS_AGENT		all busy bus cycles due to writes
MEM_LOAD_RETIRED.L2_LINE_MISS	P	L2 demand misses
SSE_PRE_MISS.T1		SW prefetch to L1 inst
SSE_PRE_MISS.T2		SW prefetch to L2 inst
SSE_PRE_MISS.STORES		Non Temporal Stores executed
L2_LINES_IN.SELF.DEMAND		L2$lines in for rfo, load, sw prefetch
L2_LINES_IN.SELF.PREFETCH		L2$lines in for hw prefetch
L2_LINES_OUT.SELF.DEMAND		demanded L2$Lines evicted

| L2_LINES_OUT.SELF.PREFETCH | | HW prefetch L2$lines evicted |