Cycle Accounting and Gooda

Overview

The exercise of optimizing an application is a rapidly rising curve of increasing effort. One starts by finding the hotspots in the application, making sure that the time consuming work is being approached in a reasonable manner. Basically algorithm sanity testing. Next, the time consuming functions should be compiled aggressively to capture those performance gains that the compiler can recognize. At that point the problem difficulty escalates rather dramatically. The developer will need to not only find where the time is spent but what the microprocessor is doing with the time. The objective is to identify performance issues related to inefficient use of the hardware and try to overcome the obstacles. The first part of this, identifying hardware related performance problems can be addressed with performance analysis tools. The second part is attacked with experience.

A performance tool based on the analysis of hardware performance events has two objectives: Collecting the right events and presenting a correct interpretation that identifies what the problem is. Most developers have never had the luxury of the time required to actually learn these things on even one microarchitecture, let alone the rapidly evolving spectrum of processors that continually flood onto the market. Even if the tool is successful, all it has accomplished is to identify what the problem is, where it occurs in the code and its magnitude. If generating correct advice were realistic, compilers would already be using the techniques. Thus the reality is that once the first phases have been dealt with the problem is going to become very difficult.

Performance analysis and SW optimization is about minimizing the cycle count, so in order to present the assorted measurements of incompatibilities of code and data with the microprocessor architecture in a sensible manner all such performance issues must be presented in the same units, cycles. If one considers the spectrum of such issues there might easily be 50 to 100 different possible problems on a given processor. Even so, they can be grouped into roughly a dozen classes that are independent of the microarchitecture. These classes can be illustrated graphically as a cycle accounting tree diagram. Not all classes will be applicable on all processors (ex: instruction collisions in SMT pipelines cannot occur on processors without SMT). Further on the vast majority of processors most of the classes cannot be measured in any sort of reliable and accurate manner and evaluated in terms of cycles. If a critical issue cannot be expressed as a cost in cycles then in reality it has not been measured in a useful manner and the user should find a platform where it can be measured and not be tricked into believing that a misleading measurement is actually useful.

The usual technique used for evaluating the cost of a performance issue is to run a customized kernel that serializes the cost of a single issue. This also validates that the processor has an event that counts the occurrences of the issue. The penalty is then evaluated from the difference of the cycles/iteration with the issue, minus the cycles per iteration for the kernel when it does not experience the issue. Under such circumstances the cost of the issue during the execution of a program can then be evaluated as the product of the penalty times the occurrence count. This is effectively serializing execution of course and thus overcounts the cost as it does not consider temporally overlapping penalties. For some performance bottlenecks it is possible to define an upper limit for the cost of a sum of related issues and thereby allow a correction for overlapping penalties.

Event selection and validation

Since users cannot be expected to know which events on a given architecture are needed to explore the range of possible performance issues that processor might encounter it is a requirement for the analysis tool package to offer a predefined collection that is will cover all issues that can be accurately measured. The underlying data collection package (ex: perf) must support multiplexing of events and support correcting for the unequal time slicing that results. The penalties associated with the events need to also be integrated into the tool. As these penalties can be dependent on the processor frequencies a simple way of adjusting the penalties should be available in the analysis tool package. Perhaps the most critical aspect of these predefined events and penalties is that the tool developers must validate the events they propose using so as to ensure the methodology they are advocating actually works.

A critical part of the event validation process is to make sure the event increments only when the issue in question occurs and for no other reason. The most frequent problem with using hardware performance events is due to the events counting something other than what is causing a performance problem in the software. On systems with SMT in the cores it is also possible to have events on one thread show up on the counters of the other. This effect is worth noting as it is particularly difficult to test. Such false positives lead to software developers wasting their time and becoming discouraged of using hardware based performance analysis techniques.

The Gooda package includes an ever growing collection of kernels. Almost all of the statements in this paper can be confirmed with the current collection. This is left as an exercise for the reader. It should be noted that kernel based event validation and processor reverse engineering is the most effective way to learn HW based performance analysis techniques. Appendix 1 describes in some detail the systematic approach advocated by the author.

Due to the difficulty associated with event validation it is highly advisable if there are multiple ways to make critical measurements. This is by far the easiest way to find an error in an events properties in a complex application. One simply looks for the inconsistencies of multiple independent measurements.

A particularly difficult issue is created when there is a time gap between when the periodic sampling counter has triggered until the interrupt is actually raised. Events that occur in such a shadow period become invisible. The classic example of this is a loop with an odd/even else if block, where one branch does an array element copy and the other takes a square root.

```
        for(i-0; i< len; i++){
                if((i&1) == 0)a[i] = b[i];
                else a[i] = sqrt(b[i]);
                }
```

For such a loop > 99% of all samples of instructions_retired will show up on the sqrt because the execution can progress from anywhere in the loop to the long latency sqrt() in the time between the counter overflow and when the interrupt is actually raised. This distortion of the sample distribution in the profile can cause incorrect diagnoses. For the instruction_retired event this results in extremely incorrect estimates of basic block execution counts and hence loop tripcounts.

Server application characteristics

Enterprise class applications are frequently large C++ based binaries (> 100MB) analyzing very large amounts of data (tens to thousands of GBs). Such codes appear to have a characteristic execution the author calls the “rule of threes”: stalled 1/3 of the time waiting for data cachelines, stalled 1/3 of the time waiting for instruction cachelines and executing 1/3 of the time at 3 instructions per cycle (producing an IPC of 1) and this is running only 1 thread/physical core. If both logical cores are used the stall fraction/thread goes up. It is precisely the high stall fraction that makes such codes very effective consumers of SMT or hyperthreading.

HPC applications are dominated by loops over large arrays of data. Optimization is mostly an exercise in getting the data laid out to allow effective use of simd instructions, getting the HW prefetchers to be effective and the compilers to generate correctly vectorized loops. Ultimately such codes are bandwidth limited.

Basic Performance Metrics and counter capabilities

A hardware based performance monitoring unit (PMU) needs some basic events. The usual set are cycles, instructions retired (non speculative) and uops_retired. In addition a system that allows a cycle by cycle

(>=) or (<) comparison that turns the cycle by cycle count increment into a 0 or 1 value, thereby counting cycles the condition is true. Edge detect and privilege level filtering complete the basic PMU requirements. Interrupt on overflow, enable and disable are required for profiling.

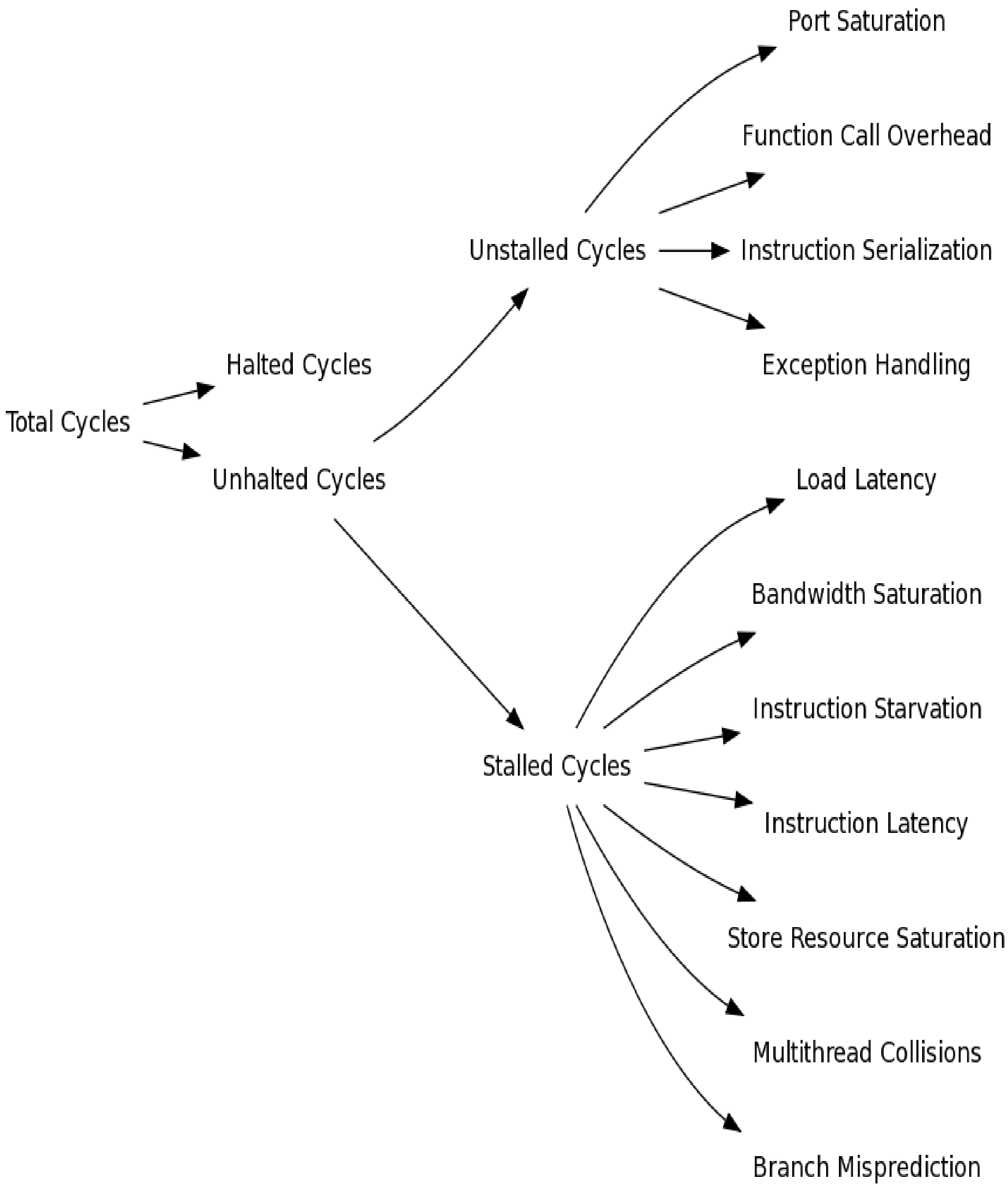
Cycles is not quite as obvious a metric as it might seem. One needs a core pipeline cycle frequency event and a fixed cycle event. Further the cycle events should only count in the unhalted, high power state. The study of concurrent execution needs a broadcast interrupt capability. This is best done from an offcore component of the socket that uses a fixed frequency counter that never turns off. If a bit mask is recorded of the non halted core IDs is recorded, only the unhalted cores need to be interrupted. This allows a concurrency analysis that identifies which cores were idle for lack of work.

Generic Performance Metrics

A generic performance metric based approach can be applied for workload characterizations with counting mode data collection (ex perf stat) or using profile data collected with periodic interrupt based sampling (ex perf record). In counting mode it can be done more accurately since the fluctuations of profiling and the precision of the location of the interrupts do not apply. This also means that differences and ratios of performance events can be used, which are usually hopelessly inaccurate when done at the granularity of instruction pointer values associated with profiled data. The larger the granularity the more reasonable the values extracted from differences and ratios become.

The critical metric is time and decomposition of how time is used ineffectively creates the largest class of important metrics for performance analysis. A generic decomposition of cycle usage can be constructed from a knowledge of the microarchitecture, a validated set of events and knowledge of the penalties associated with those events. While the specific events, penalties and microarchitectural interpretations are different on every processor, high level groupings can be defined that make sense on a large range of processors.

The generic cycle decomposition is best explained by simply showing the diagram, it is self explanatory for the most part.

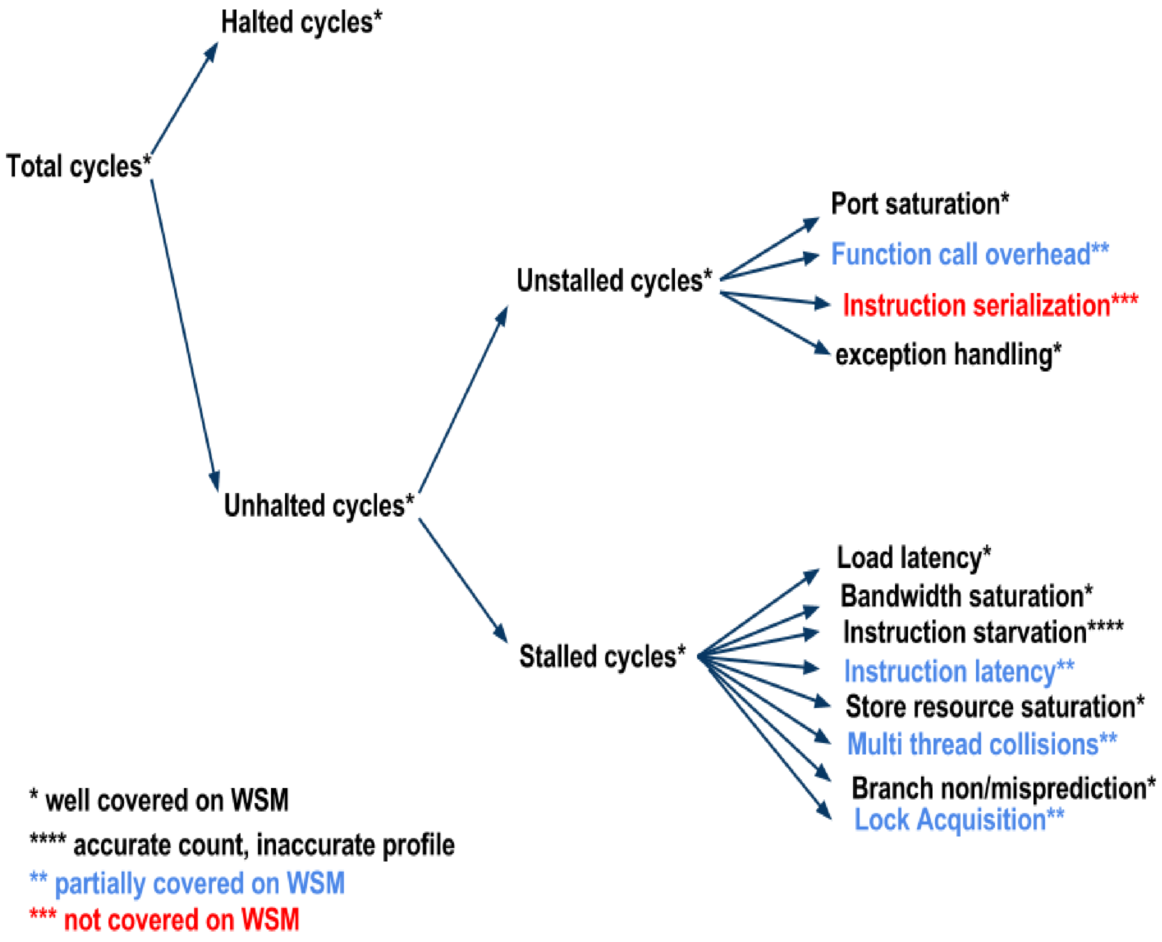


As the generic metrics are constructed as a sum of event counts times penalties, the sum of the generic metrics can be greater than 1 if there are temporally overlapping penalties. In addition branches can have hierarchical relations. For example bandwidth saturation can be caused by purely load driven cacheline requests. Thus the load latency branch can have a significant count when the real problem is bandwidth saturation. In fact because there are many overlapping cacheline requests, the load latency cycle count will exceed the cycle count in such cases. A randomized gather loop to dram will cause exactly such an effect.

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

This particular example can result in the calculated load latency penalty cycles being 4 to 5 times as great as the number of cycles. This illustrates the usefulness of multiple independent measurements. By measuring the cycles with load misses outstanding (offcore_requests_outstanding and l1d_pend_miss on Intel processors) the overcounting is easily identified.

The branches of the graph are computed with whatever techniques are supported by a given microarchitecture. It is up to the performance analysis tool to incorporate whatever is necessary to accomplish the task. The techniques frequently change dramatically from one generation to the next as the coverage of the branches improves and new techniques are developed. The Gooda analyzer does this for those processors that have a reasonable fraction of the branches accurately covered. The currently supported list are Intel WSM, SNB and IVB processors.
The Intel Westmere-EP processor is used as an example as the branch coverage is fairly good as illustrated by the graph shown below. Note however that issue coverage is different for counting mode and profiling in some cases. This is due to having to make measurements with differences and/or ratios which do not work well in profiles.



As an illustration of how a branch is evaluated consider the load latency branch as evaluated on a WSM-EP processor. The penalties considered are the extra cycles that a code must wait for load instructions to deliver data to the functional units. These were determined with a linked list walker that can confound the HW prefetcher. On a dual socket system with 3 cache levels and a Non Uniform Memory Architecture there are a significant number of individual issues with different costs. There are further the costs associated with DTLB translations and those associated with loads blocked from being able to forward previously stored data residing in store buffers. The penalties were evaluated with specific kernels as discussed above and are processor frequency dependent (both core and uncore frequency)

- **6*mem_load_retired:l2_hit**
- **52*mem_load_retired:l3_unshared_hit (should be called l3_hit_no_snoop)**
- **85*(mem_load_retired:other_core_L2_hit_hitm - mem_uncore_retired:local_hitm)**
- **95*mem_uncore_retired:local_hitm**
- **250*mem_uncore_retired:local_dram_and_remote_cache_hit**
- **450*mem_uncore_retired:remote_dram**
- **450*mem_uncore_retired:remote_hitm**
- **250*mem_uncore_retired:other_llc_miss**
- **7*(dtlb_load_misses:stlb_hit + dtlb_load_misses:walk_completed) + dtlb_load_misses:walk_cycles**
- **8*load_block_overlap_store**

The truth is that small load latency penalties can be hidden by the combination of OOO execution and good compilation. So the terms associated with

- **6*mem_load_retired:l2_hit**
- **7*dtlb_load_misses:stlb_hit**

can usually be ignored.

The generic cycle accounting metrics

The objective of the generic cycle accounting metrics is to organize the potential performance limitations an application can encounter into a list of easily understood categories. The high level view of the applications' performance limitations will be architecture independent to a remarkable degree in the experience of the author. The creation of such a list also provides a foil against which the coverage a given processor's event list can be evaluated. The sad reality is that the vast majority of processors reveal themselves as being incapable of reconstructing these metrics with any usable accuracy, particularly for application performance tuning where an accurate (precise) instruction level profile is required.

The high level metrics are reconstructed from a varying number of low level events per metric. The dual requirements of having an event per penalty value and an event per "solution type" defines the concept of event coverage. The events should have well defined penalties so a simple sum of event count times penalty is reasonably well defined. A sum of issues can be profiled as the profile of the sum will be the sum of the profiles. The same cannot be said for metrics constructed from differences or ratios.

In addition different performance problems can have different solutions and the events must allow the user to identify which problems they will need to overcome. Knowing just the cost is not sufficient. To illustrate this consider the following examples:

1) A bandwidth limitation to remote dram can be fixed not just by improving the code layout but by improving the data decomposition. That second solution is not available for bandwidth limitations to local dram. Thus while the high level generic metrics make it easier to comprehend the data, once the code developer starts to fix the performance problems they will need to know what all of the problematic events observed really count.

2) Measuring instruction starvation by counting the cycles the RS/Scheduler is empty is an outstanding method. There are multiple causes for draining the RS, ifetches from L3 and branch mispredictions to name just two. Being able to distinguish these causes would be a great benefit.

3)Cacheline forwarding from one core to another for loads costs ~ 100 cycles. But the nature of the code improvements are wildly different for lines in modified vs unmodified state. The modified line access indicates a threading synchronization issue, while the unmodified access is just a load cache miss.

What follows is a list of the generic metrics and some thoughts on the event requirements needed for accurate measurement.

Halted cycles

Halted cycles measure the time that the OS has nothing to run. This represents a performance problem when the reason for the halted state is all threads waiting for an interrupt. Examples would be disk or network I/O or waiting for a synchronized message (which could be a subset of network I/O).

This would seem to be easily measured but as we now deal with processors dynamically changing frequency in response to power utilization (turbo) it becomes a bit tricky. Perf by default prints out times in nanoseconds. The assorted cycle performance events count in various constant and nonconstant frequencies and usually count unhalted cycles. Many of these can be adjusted on a given system with bios settings such that perf does not actually capture their frequency (ex: Intel’s fixed counter reference clock which has the same frequency as the TSC). Thus it becomes necessary to have a truly fixed frequency event that counts unhalted cycles. Intel does provide such an event which goes under varying names like cpu_clk_unhalted:ref_p (133 Mhz on NHM/WSM) and cpu_clk_thread_unhalted:ref_xclk (100 Mhz on SNB/IVB). Both cases are event code = 3, umask = 1.

If one has such an event then computing the halted cycles is easy as the time of the first sample and the last sample will have time stamps measured in nominal nanoseconds.

Stall Cycles and Uop Flow

An overall measurement of pipeline stalls acts measures an upper limit of the effects from all the stall cycle components. There are three points in an out of order pipeline where stalls and uop flow should be measured: output of the front end, at execution/output of the schedulers and at retirement. It is particularly difficult to define uop flow and stalls at execution due to speculative dispatch of uops as this kind of execution causes a uop to be counted multiple times. This problem was finally solved on Intel’s IVB processors and represents a serious improvement to their performance problem coverage.

Load Latency

In server applications the dominant source of pipeline execution stalls is due to instructions being blocked while waiting for loads to complete. Stores waiting for cachelines mostly do not stall the pipeline (more on this later) except on in order architectures. To evaluate load latency accurately one shadowless, precisely located load retired event per data source/latency is needed. Of these requirements the lack of a shadow is the least important. Further even if the latencies are identical, if the application tuning recipes are different for different well defined classes of load accesses, then the classes need to be distinguished to identify the correct recipe. As load requests that require the delivery of a line in an exclusive modified state from another core tend to have different latencies than for lines in the unmodified state the ability to identify this distinction is absolutely critical. Further the performance “solutions” for dealing with shared modified lines are very different than for unmodified lines. Thus there is a second equally important reason to measure these accesses with independent events.

A basic cache miss event is not very useful as a miss does not have a well defined penalty. To get the data source a difference of miss events must be measured and profiling on differences of counts does not work. A cache miss event that measures the latency and the data source is however a good supplement/alternative to the list below. However, a single event that cannot be filtered to raise the sample purity for the desired problem becomes extremely impractical. The ability to select the long latency misses either through the latency or through the miss level is needed. Longitudinal profiling has to be triggered on uops to create a measurement with an absolute normalization. Server applications usually have load driven fetches to dram at the rate of 1 in 500 to 1 in 2000 uops. Thus three orders of magnitude more data is needed to achieve an equivalent statistical accuracy in the measurement. A longitudinal sampling system might work with a two level sampling period. The first level selects uops and then checks them against a filter. This needs to run at the highest rate possible without adversely effecting performance. Once the filter is passed the event signal can be processed in the usual way with counter overflow generating the actual interrupt.

Filtering directly on the latency can make it difficult to produce an absolutely normalized result. The normalization may be application dependent due to the locality of long latency loads in the code. Consequently, the individual data source events must be used to evaluate the normalization for each application. Thus triggering on a cache miss and measuring a latency from miss detection to data availability might provide a more useful event.

Thus triggering on a cache miss and measuring a latency from miss detection to data availability might provide a more useful event.

If one were to write down a complete list of load access events needed to evaluate latency on a dual socket NUMA system with a 2 level cache per core and a shared L3 cache one might start with the following (any similarity with any existing or planned processors is purely coincidental)

all loads
l1d_hit
secondary_load_miss (aka hit_line_fill_buffer allocated by a load)
secondary_other_miss (aka hit_line_fill_buffer allocated by a store or prefetch)
L2_hit
l3_hit:no_snoop
l3_hit:snoop_miss
l3_hit:snoop_hit (should not include hitm)
l3_hit:snoop_hitm
l3_miss_local:dram
l3_miss_local:non_dram
remote_l3_hit:no_snoop
remote_l3_hit:snoop_miss
remote_l3_hit:snoop_hit (should not include hitm)
remote_l3_hit:snoop_hitm
l3_miss_remote:dram
l3_miss_remote:non_dram

the dtlb component

second_level_dtlb_hit
second_level_dtlb_miss (per page size)
cycles_dtlb_walker_active

blocked store forward

this is rarely an issue for binaries compiled with gcc or icc but a load can block on being unable to forward data from a store buffer (for example when there is a preceding store to overlapping address that is smaller than the size of the load). This is complicated for profiling as it is dependent on 2 instructions. Identifying the blocked load is a good start.

Dealing with temporally overlapping latency penalties

Being able to monitor the offcore request queue (Superqueue) is an extremely powerful technique. It can provide insight on load latency issues, instruction delivery latency and bandwidth saturation if its monitoring is correctly designed. If the offcore requests can be separated into types: loads, stores, instructions, hw prefetch of data and hw prefetch of instructions and all, a surprising number of tricks can be accomplished.

For load latency if one can count the number of entries in the queue that are directly due to loads (including slots originally allocated for HW prefetch but converted to loads before the line arrived), then counting the cycles where there is at least 1 entry measures the total cycles lost to offcore load requests and corrects for temporally overlapping requests. Similarly if it is possible to monitor the number of fill buffers allocated for loads, the total cycles with a L1D miss pending can be monitored as well. These two metrics may not have precise locations but they are extremely useful for identifying temporally overlapping loads and possible event errors.

On the IVB processors the cycle_activity event, through a very clever combination of increment values and umasks supports the measurement of combined conditions of execution stall with load requests in flight. This provides a very good confirmation of how the load driven cacheline fetches are stalling execution.

Another useful event for this category of performance problem is one that identifies when loads request access to a SW prefetched line prior to its actually arriving (Intel's load_hit_pre:sw_prefetch). This indicates when sw prefetches were not issued early enough.

Memory Bandwidth saturation

Memory bandwidth performance problems are exceptionally poorly understood. In almost all cases measuring the bandwidth to dram has little to no informative value for application tuning. The problem is that memory bandwidth is only a performance limitation in cases where the application is running very close to the bandwidth limit. What makes this hard to measure is that the bandwidth limit on a multi core, multi socket, NUMA server is a function of almost everything, including the number of cores simultaneously consuming bandwidth, which dram they access, processor stepping, dram types, sizes and manufacturer, bios version and settings, motherboard revision, etc, etc. What is actually desired is a signal that measures the impact of bandwidth saturation. When a processor is bandwidth limited the cacheline requests accumulate in the assorted queues used to orchestrate cacheline delivery (the superqueue and CBox queues in Intel processors for example). Thus the way to identify if the execution is bandwidth limited is to monitor the queue occupancy and count the cycles that the occupancy is larger than some threshold. For example a threshold of 6 on offcore_requests_outstanding works quite well for WSM processors.

In the sandybridge and later processors from Intel the HW prefetch has been improved where under some conditions the hw prefetched cachelines are only brought as far as L3. In such cases the superqueue does not need to track the request and the fill buffer can be released. This allows single threaded applications to get more simultaneous cacheline requests to dram in flight simultaneously. The lines are then pulled from L3 by the loads being executed out of order from the large scheduler (reservation station). This will cause the load latency events for L3 accesses to fire, increasing the load latency counts, while simultaneously lowering the occupancy of the superqueue. This is still being studied at the time of this writing.

Another way of measuring this on Intel processors is the event L1D_miss_pending which counts the number of fill buffers allocated by "demand reads" (these include l1d HW prefetches as well as loads). This is an example of having two events that count related quantities through different techniques and being able to use them to cross check the measurement.

The problem with just monitoring the offcore requests is that the request pileup might not actually effect performance if only a small fraction of the lines being transferred are actually consumed. A possible solution might be to make a combined event that creates an AND condition between the queue pileup measurement with no uops being executed. This would be analogous to the cycle_activity event on IVB.

The count of cycles which are bandwidth saturated can be well supplemented by the count of cacheline accesses to local and remote dram. This is useful in knowing, as improving the data decomposition to be NUMA friendly can raise the bandwidth limit.

Another useful event for this category is identifying when loads request access to a HW prefetched line prior to its having arrived. This is usually an indication of bandwdith limitations. Intel's load_hit_pre:hw_prefetch is a good example of this.

Instruction Starvation

The high fraction of time spent waiting for instruction cachelines is dominated by instruction fetching from L3 on Intel processors as the latency for such fetches can exceed 50 cycles in server parts. This does not necessarily require branch mispredictions, simply a very large active binary footprint (several MBs). Branch misprediction simply aggravates the issue (more on this later).

At the time of this writing the Intel processors do not have a precise L3 ifetch event making localization of this problem extremely difficult. This is without question the largest gap in the performance issue coverage of current Intel products. This could be resolved many ways. AMD’s longitudinal profiling of front end issues offer one approach. A precise event that captures the address of the ifetch or adding ifetch source bits to the LBR targets could be a solution. Such decisions are best left to the PMU designers and the author awaits their clever solution.

In the meantime using the L2_rqsts:ifetch_hit/miss events is what is used. In this case the miss event really is an L3 hit since the author has not yet encountered execution with a significant rate of instruction line fetches from dram or remote cache. Of course the offcore_response events could also be used.

Instruction fetch events occur for addresses that are not even in the pipeline. Thus interrupts from counter overflow are going to be far away from the offending address. The interrupts will be assigned the address of what is retiring at the time of the counter overflow and can easily be located 100 instructions earlier in the instruction flow.

Starting with the SNB processors the cycles when there are no uops in the Reservation Station (RS/scheduler) can be monitored. This measures the real performance cost of instruction starvation. Unless the lack of flow from the front end significantly drains the scheduler there is likely little cost due to a lack of instruction delivery to the execution units. Thus short interruptions (L2 ifetch hits with 10 cycle latency) have a significant probability of being hidden by the instruction queue, instruction decode queue and RS. Branch mispredictions will also cause instruction delivery interruptions. Being able to sample the durations of the periods the RS is empty would be a great assistance in clarifying the effects. Short durations (2-6 cycles) would be dominated by branch misprediction, while long periods (50 cycles) can only be caused by offcore instruction line requests.

As the the RS empty event is generated near the end of the pipeline, its interrupt location is likely to be the address of the fetched line for long durations stalls.

The event monitoring the occupancy of the offcore request queue can be used here also if the slots allocated for instructions can be counted by themselves. Again counting the cycles with at least one entry for an instruction line measures the total cycles spent waiting for instruction lines from offcore sources.

There are more sources of instruction starvation associated with “turbulent” instruction/uop flow in the the pipelines front end. Some can cause a complete interruption in uop delivery by the front end (ex: length changing prefix issues) but most simply lower the flow rate out of the front end without completely shutting it off. In those cases one would group the effects under an underutilization branch on the unstalled side of the decomposition.

Instruction latency

Long latency instructions that block execution can be an important issue. The usual causes are sqrt and divide instructions, as these are exceptionally long (~ 25 cycles and more) and frequently not pipelined. Having an event that counts the cycles the sqrt/divide unit is active is a very convenient way of measuring this effect.

The author is unaware of any processor with the ability to detect serialized FP operations that can cause such stalls. One merely needs to consider an FP based instruction like

```
   a = b + c + d  + e + f + g + h + i + j ;
```

being evaluated left to right in accordance with standards to see the problem.
This is a gap in current architectures as far as the author knows.

Store resource saturation, long latency RFOs

OOO superscalar pipelines usually have separate store pipelines that allow store instructions to retire without actually completing the writeback to visible memory (ie L1D). This is done by holding the results in store buffers until the writes can complete. On x86 architectures the stored results must appear in order. Thus if a store causes a cacheline to be pulled from dram no other stores can complete until the cacheline arrives and is available in L1D. A stream of stores will pile up in the store buffers. When no more store buffers are available the pipeline will stall.

This is measured on Intel processors with an event called something like resource_stalls:store_buffers and it counts the cycles that there are no store buffers available.

Correlating this with the cycles that there are offcore request queue entries for stores (RFO) can add a lot of insight into the problem and provides a confirmation of the root of the issue.

As the size of the coherency domain is extended the latency of a read for ownership can increase. This can in turn increase the likelihood that all the store buffers will be in use.

Long latency RFOs can have side effects on other performance metrics. For example when all the store buffers are in use the uop flow can be interrupted. This in turn can result in the RS/Scheduler draining all the instructions and resulting in an instruction starvation signature.

Another example is loads blocked by stores, of store-forward blocked. A small store that overlaps in address range with a subsequent larger, overlapping address range load will block the load until the has completed the writeback to cache. For long latency RFOs this can result in a large stall.

Other kinds of store blocking can also occur depending on the architecture, for example interleaved l1d misses and hits, etc.

Prefetchw instructions can in principle be used to alleviate long latency RFOs, but without a precise RFO miss to dram type of event it is nearly impossible to identify which stores should be prefetched.

Multi thread collisions

This covers a variety of effects where threads run on an SMT architecture can interfere with each other in the pipeline flow. It is not meant to describe effects due to the threads evicting each others cachelines as the load latency and bandwidth saturation measurements will include such effects. Specifically in Intel processors this means collisions in the front end when both threads have instructions available for decoding, collisions for a single resource in the pipeline at execution (1 load port on WSM, the sqrt/divide non pipelined units, etc) and at retirement as it appears the two threads alternate the ability to retire instructions.

The author does not know an easy way these can be directly measured and profiling where they occur in the code is even more difficult. This represents a coverage gap. Consequently the gooda analyzer does not attempt to measure and display this component

They can be approximately inferred in counting mode if one assumes the collisions are random. The author does this in his own counting mode spreadsheet analyses and the techniques illustrate the nature of the performance issue. The total collision rate is approximated as simply the product of the probabilities that on a random cycle the resource was required by both threads. That is probably reasonable for execution and retirement collisions but front end collisions are different because the uops can only flow from the front end when the back end has resources. Thus the time window for uop flow is reduced from

cycles to cycle - resource_stalls

As resource stalls are likely to be correlated between the two threads, both threads have to squeeze their uop flow into the same reduced time window. Thus the probability of a collision for two threads executing the same code would be something like

(uops_issued:any:c=1/cycles-resource_stalls:any)**2

where uops_issued:any:c=1 is the number of cycles where uops are issued by the FE for a given thread.

Branch mispredictions

The total cost of a branch misprediction can be divided into a series of phases.

1) Cycles spent pushing the wrong path instructions through the pipeline prior to recognizing the misprediction (ex: consider a conditional branch dependent on data that must be loaded from dram). This is frequently referred to a "wasted work".

2) cycles spent flushing the pipeline (resource_stalls:br_miss_clear on core2)

3) cycles spent fetching the correct cachelines. This is included in the instruction starvation branch so it is not included as a component here

4) cycles to get the new flow of uops to the execution stages (approximately the length of the FE of the pipeline perhaps plus a couple cycles).

Frequently the second item overlaps with the third and fourth so it can in practice be ignored if it is smaller than the fourth. For example on Intel WSM processors the author measured a 6 cycle penalty in a micro kernel that was L1I resident when executed with vs without branch mispredictions.

There are really two classes of branch mispredictions, the classic ones just described where either the direction of a conditional branch or the target of an indirect branch are mispredicted and the case where the number of branches in the "hot" component of the code exceeds the size of the branch history tables of the branch predictor. This second case could be called non predicted branches rather than mispredicted. On Intel processors it is measured with the baclear event and can easily be studied with kernels with very high branch densities. Such studies showed again a 6 cycle penalty on WSM.

The precise location of branch mispredictions is required if they are to be resolved. The misprediction penalties likely relieve some of the problems of shadowing closely spaced mispredictions.

Lock acquisition

Highly contested locks might be treated as a separate category from long latency loads as the performance impact is the serialization of threaded execution of many cores. Identifying the subset of long latency loads associated with these issues can be done with an event that measures the actual latency and not just the data source. It is preferable if the hardware can also filter on the minimum desired latency required to increment the counter as high data purity greatly simplifies the analysis of the data.

Unstalled performance problems

While pipeline/retirement/execution stalls are dramatic interruptions in forward progress, unstalled cycles can still be ineffectively used. The generic metric methodology attempts to classify some of the more obvious examples of this and include them in the Gooda analyzer.

Underutilization

There are a variety of mechanisms that can result in uops trickling through the pipeline, so that while not stalled, the execution flow is greatly reduced. An overall metric that counts the equivalent cycles lost can be constructed as follows for a processor that has a four wide uop path.

(4*uops_retired:any:c=1 - uops_retired.any)/4

Port saturation

If a code stream results in uops being executed on a single port for a dominant fraction of the cycles (~ 80%) then the only way to improve performance is to reduce the uops using that port. An example of this is a system with a single load port and 85% of the cycles have a load being executed. This can be caused either by not using simd instructions or by a series of small independent loops that pass the results of one to be used as the inputs for the next through L1D. An example of the latter might be a fortran 90 vector syntax that gets compiled as a series of loops.

The resolution of the first example is to generate simd instructions, the resolution of the second is to jam the loops together and keep the results in registers.

Call overhead

Modern coding styles result in applications being constructed from a large number of small functions. While this is good for testing, code integration and support it is not so good for performance. Large enterprise codes frequently have a ratio of instructions retired/call of between 30 and 80. FDO compilation will frequently boost this by 50%.

What is frequently seen on x86 processors is that function call with multiple arguments will result in

a bunch of instructions to put the argument values in registers or on a stack

some number of unconditional branches before the execution actually arrives in the target function body (trampolines, 2 to 4 is very common particularly when the binary uses shared objects)

a series of push operations to free up registers for use in the function

after the function has finished its work a series of pop instructions are needed to restore the architectural state of the caller

and finally a return

This can easily result in more than a dozen instructions just to call a function.

Accurately measuring call count frequency is hard due to shadowing effects mentioned earlier. This is also a class of events that must be precise as the location of the execution can have large discontinuities. A rotating buffer of branch source and target pairs that can be filtered on branch type and privilege level is a far more effective way to do this than a single occurrence event even if precise and shadowless. The Intel processors have exactly such a mechanism called the LBR. By filtering on return branches which directly connect source and target no static analysis of disassembly or dynamic branch address modification at runtime is required to deal with the trampolines. Gooda uses this approach to provide a call count graph from HW without any binary instrumentation. Having an event that counts the LBR inserts is particularly useful as then only one event needs to be capable of a shadowless interrupt. It is better not to use PEBS with the LBRs as PEBS introduces a larger shadow than the same event without PEBS enabled.

Instruction serialization

This was already discussed in instruction latency but even for 1 cycle latency instructions the result is a low flow of uops. The same issues apply in that there is a gap in being able to profile on such effects.

Exceptions and microcode

Denormalized FP calculations will result in a large flow of uops to handle the math in accordance with IEEE standards. Similarly some instructions are microcoded and have long latencies created by a large flow of uops (ex: idiv, sincos). Counting the uops from the microcode sequencer or the cycles the microcode sequencer is active if an extremely effective way of identifying where this is happening and the cost.

Control flow

While not a generic cycle metric, measuring control flow of execution can be critical. There was already a discussion of using the LBRs filtered on returns to get call counts. This technique can be used to make a call count graph and display statistics on which functions get heavily called. Gooda has a call graph display and displays the call counts of sources and targets of the hottest functions.

Measuring basic block execution counts in critical for a variety of things like input for feedback directed optimizations in the compiler and estimating average loop tripcounts. By collecting the LBR data with no filtering and sampling on lbr inserts or br_inst_retired:near_return (without PEBS), the LBR will capture a trajectory through the last 15 superblocks. Thus all the instructions between a branch target and the address of the next branch will be executed one time. On Intel processors there appear to be a minute fraction of cases where the next branch will not be in the same function range as the previous target. These measurements must be thrown out of the analysis. They are likely due to OS interrupts. Using this measure the Gooda analyzer is able to correctly determine the basic block execution counts on extremely difficult test cases like a nested loop structure with an inner even/off else if block doing a copy or sqrt. Using instructions_retired to try to do this results basic block execution counts that are off by > 1000X, while the LBR technique is usually good to 0.1%.

Filtering the LBR on calls can in principle allow a call chain analysis to identify the paths that lead to problematic issues. Gooda does not support such an analysis at this time, but the LBR hardware does and starting with the Haswell (HSW) processors there will even be a call stack mode supported in the LBRs.

Function arguments and register value capture on interrupt

In x86_64 integer function arguments are passed in registers if there are 6 or less arguments. By having a precise, shadowless event that triggers on call instructions with register capture enables the function argument values can be evaluated. At the time of this writing the perf utility does not support register capture even though the Intel PEBS mechanism has from the very beginning. A simple display can be constructed that simply shows the minimum, maximum, average and rms of the register values and this can be extremely useful. Beyond give function arguments, this can be used with static analysis of the disassembly to give variations in loop tripcounts, which can be critical for determining the correct optimization approach in a loop.

Variations in execution vs averages

The cycle accounting approach described in this paper only measures an average of the execution but not the variations. In large distributed multi machine execution flows the slowest node will frequently define the latency of the operation. In such cases the variations in execution can be more important than the average. Measuring variations is a great deal more complex. Longitudinal profiling perhaps offers a technique to capture time durations and allow an analysis of their variations. To date the author is unaware of any successful usage of this approach. Adding time intervals to LBRs is a rather obvious idea and might be worth considering.

Cascading performance counters offers a very flexible solution of using an event with a non zero cmask and edge detect as a sample trigger that then starts a coupled counter that accumulates cycle counts.

For example counter 1 overflows on an event with a cmask=1 and edge detect set and invert set, thereby statistically sampling a time period where the event is incrementing by 0. By sampling on the event with the edge detect one ensures that stall periods are all equally probable of having their durations measured. Counter two then gets enabled to start counting cycles on the event programmed to counter one but without the edge detect. Thus the cycle count starts with a small delay relative to the counter that detects the trigger which should be corrected for. This ensures that counter two will count for the entire duration of non zero values. When the cmasked event changes value from 1 to zero the cycle accumulation is stopped and the PMI raised. The cascaded counter value in counter 2 then represents the duration that the cmasked logical test was true. This would allow the measurement of the variations in stall durations for example.

There are no doubt an endless number of good ideas of how to use hardware performance monitoring, hopefully what has been described here can be viewed as a set of minimal requirements.

Address analysis

Analysis of the virtual addresses of load and store operations is something that has been investigated with load latency events and AMDs IBS. The new HSW processors now support address capture on all the memory PEBS events. An obvious use is an analysis of arrays of large structures with the intent of creating parallel arrays of smaller structures tuned to their usage. The usual display is of something like

(Address - array_base_address)%structure_size

to identify what structure elements are accessed in which functions.

Another obvious usage of the precise hitm events with address capture is to identify cachelines that are accessed in non overlapping ranges by multiple threads. This identifies cases of false sharing that actually effect performance in threaded applications.

The addresses of loads and stores that cause page walks can be useful in identifying where to use large pages.

The Gooda analyzer

The analysis of the interrupt records in the perf.data file is done in a 2 phase process. The raw records are processed creating a hierarchical tree of of structures that represents the module memory maps per PID. PIDs with the same binary path are “merged” to all refer to a single principal process, which is defined as the first PID of the set that gets a sample record. Only the principal process has a tree of module structures where the sample data is accumulated by RVA. Module structures are created when the first sample is bound to a particular mmap record. All mmap structures with the same path, for a family of PIDS defined by having the same principal process, will then have pointers to that module. This completes the merging process.

All structures have a sample data array that accumulates sample counts per core per event and the totals per event. The term event here includes software derived counts from LBRs and such. The module structures have a linked list of RVA structures (also accessible through dynamic hash lookup) and these thus represent the sample distributions per instruction pointer.

Once the raw data has been stored into the memory model, the linked lists of RVA structures are sorted into increasing order based on RVA. The modules structure address ranges are then divided into function ranges using a popen call to readelf. The functions are sorted into increasing address order and the RVA sample data is then accumulated into sample structures for each function.

The process and module structures/process are sorted into descending order by total sample count. A global linked list of all functions with samples is created and it is also sorted into descending order based on total samples. These two operations define the process-module tables and the hotspot function tables.

The hottest “N” functions (N = 20 but if there are more than 500 functions in the 95% function profile list then N is set to 200 by default, it can also be set by input argument) are then processed. The address ranges are disassembled. The branch instructions are identified and the list of all branch instructions and targets are used to break the disassembly into basic blocks. Each basic block has a sample structure which accumulates the data from all the instructions in the basic block.The debug information for each asm address is walked to find the two ends of the inlining chain. For each function a principal source file is identified (allowing for the debug information to be imperfect) and every asm line has a principal source file line associated with it. Each source file line has a structure which accumulates the sample data from all the asm lines that point to it. This enables the creation of the annotated source line and asm tables. The basic block information is used to make a control flow graph using Dot. The three displays are linked and reached in the display by double clicking on the function name in the function hotspot table.

Intel Sandybridge and Ivybridge processors have an LBR mechanism that is sufficiently sophisticated to create a call count graph with hardware based data that can be collected with perf record. The LBR is filtered to capture only near return branches, thus 16 samples are collected per interrupt vastly reducing the impact of data collection shadowing. The LBR records can be captured with either the br_inst_retired:near_return or rob_misc_events:lbr_inserts. To minimize shadowing if the near_return event is used, pebs should not be enabled. The LBR data is processed using each sample in two directions to identify the sources and targets of function calls. The data is stored in linked lists of structures anchored in the RVA structures. The data is aggregated per function and sorted by decreasing branch sample count.

The top 10 sources and targets for each function are added to the function spreadsheet as expandable rows. A call count graph is created with Dot, based on the hottest 50 functions and their sources and targets except for the sources of hot leaf functions that have more than 50 sources. There is no point is drowning the graph in calls to memset and memcpy.

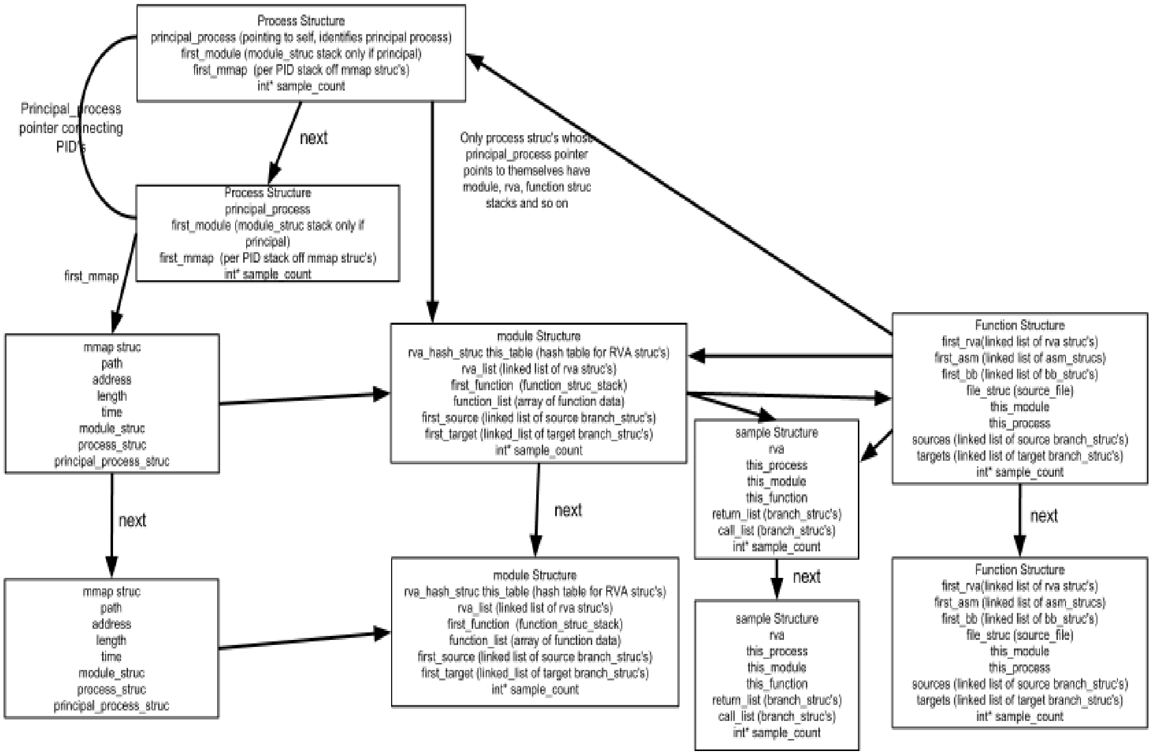
If the LBR is set to run with no filtering it will capture all taken branches. This can be used to extract a measure of basic block execution counts that is much more accurate than by averaging instructions_retired over the basic block. On the test case bb_exec kernel provided with the Gooda package it is also significantly more accurate even than using the inst_retired:prec_dist version of the instruction retired event. There is the further advantage that by capturing the LBR execution trajectory one has essentially sampled several hundred instruction_retired equivalents, providing far better statistical accuracy per interrupt. On Sandybridge and Ivybridge processors the events br_inst_retired:near_taken (preferably without PEBS) or rob_misc_events:lbr_inserts can be used to trigger record capture. The Gooda analyzer evaluates a basic block execution count for the functions that have disassembly analysis performed on them and also creates a software based evaluation of instructions retired that can be used for a future instruction mix analysis.

The tables that are created have control structures to enable the expandable cycle accounting display. Each table has a row containing an encoding defining every columns position in the multi-level cycle accounting tree. There are also rows containing the event names, sampling periods, penalties, multiplexing corrections and whether a column can be displayed in cycles. Thus once the table is created there is no need to go back to the raw data.

The architecture dependent cycle accounting is encoded in a template table that is used to drive the evaluation of derived data columns like the branch and sub-branch cycle count values for each table row. Thus a single set of functions appears to be capable of handling almost any architecture that supports real measurements of the generic cycle accounting metrics.

An approximate diagram of the dominant data structures used in the analyzer is shown in the diagram below:

Greatly Simplified Gooda Analyzer Data Structures


Process Structure
principal_process (pointing to self, identifies principal process)
first_module (module_struc stack only if principal)
first_mmap (per PID stack off mmap struc's)
int* sample_count
Principal_process pointer connecting PID's
next
Only process struc's whose principal_process pointer points to themselves have module, rva, function struc stacks and so on
Process Structure
principal_process
first_module (module_struc stack only if principal)
first_mmap (per PID stack off mmap struc's)
int* sample_count
first_mmap
mmap struc
path
address
length
time
module_struc
process_struc
principal_process_struc
module Structure
rva_hash_struc this_table (hash table for RVA struc's)
rva_list (linked list of rva struc's)
first_function (function_struc_stack)
function_list (array of function data)
first_source (linked list of source branch_struc's)
first_target (linked_list of target branch_struc's)
int* sample_count
Function Structure
first_rva(linked list of rva struc's)
first_asm (linked list of asm_strucs)
first_bb (linked list of bb_struc's)
file_struc (source_file)
this_module
this_process
sources (linked list of source branch_struc's)
targets (linked list of target branch_struc's)
int* sample_count
sample Structure
rva
this_process
this_module
this_function
return_list (branch_struc's)
call_list (branch_struc's)
int* sample_count
next
next
next
next
mmap struc
path
address
length
time
module_struc
process_struc
principal_process_struc
module Structure
rva_hash_struc this_table (hash table for RVA struc's)
rva_list (linked list of rva struc's)
first_function (function_struc_stack)
function_list (array of function data)
first_source (linked list of source branch_struc's)
first_target (linked_list of target branch_struc's)
int* sample_count
sample Structure
rva
this_process
this_module
this_function
return_list (branch_struc's)
call_list (branch_struc's)
int* sample_count
Function Structure
first_rva(linked list of rva struc's)
first_asm (linked list of asm_strucs)
first_bb (linked list of bb_struc's)
file_struc (source_file)
this_module
this_process
sources (linked list of source branch_struc's)
targets (linked list of target branch_struc's)
int* sample_count

Summary:

The author has attempted to describe a generic cycle accounting methodology, a hardware based control flow analysis and the hardware performance event requirements needed to support the methodologies. In addition the use of the methodology as the basis of the Gooda analyzer and some of its inner workings have been discussed.

Appendix 1

Validating performance events

What will be described here is some of the post silicon event validation techniques used by the author. Most kernel test will provide useful validation tests for a far larger number of events than might initially be realized.

For almost all tests the same approach is applied for validating the events on Intel processors:

a huge number of events is placed in a list, for most tests the list should just be all the core events with a careful selection of offcore_response events interspersed in the list (interleave.sh). The gooda-analyzer/kernels/validation directory has such a list of offcore_response events for SNB and IVB processors

The tests are run on four events at a time (to avoid multiplexing corrections using fourby.sh) until all the events in the list have been run through perf stat (counting mode) with a field separator (-X @) used so the data can be opened in a spreadsheet program

Each test is run in all configurations with and without the prefetchers if that variation is important.

1)triad

For a triad one first generates triads with RFOs (no NT stores) using default SSE4.2 and AVX code generation. Versions with NT stores are generated with SSE and AVX encodings. ICC is required to do this as one needs pragmas to force NT store generation.

The three RFO triad binaries are then run with buffers with sizes set to be resident in L1D, L2 and L3. with and without the HW prefetchers enabled (18 runs). The complete set of 5 binaries are then run with buffer sizes large enough for the data to come dram using the initialization to force local or remote dram as the source. Again this is done with the HW prefetchers enabled and disabled. Thus a total of 20 more runs are made. As the runs are relatively quick this can be done on all core events including a large number of the matrix type offcore_response events, between 500 and 1000 events.

The resulting outputs are then merged (merge2.sh) into single spreadsheets based on the memory subsystem level (5 spreadsheets).

Once the spreadsheets are constructed the data is sorted by the event name or by the event code & umask code if perf's raw programmings are used. In the latter case a column must be added for event/umask codes as many events will be programmed by name with only some using perf's raw programming mode.

Looking at the disassembly and capturing the total triad loop tripcount for each configuration will allow the user to compute an expected count for most of the events. In particular the technique allows a great deal of negative testing.

By having the count results for all the events in multi column spreadsheets errors in events are seen extremely quickly (ex: cacheline movement events accumulating billions of counts on a triad resident in L1D running for 2 seconds)

Below some of the particular tests and events are discussed

2) Load latency events

A linked list walker (kernels/mem_latency/walker.c or walker_open.c) will validate many events beyond just load memory access events. These tests can create access patterns of sufficient complexity to overcome the HW prefetchers. Walker.c divides the buffer into N sections and lines point to a different section in a randomized loop over sections. Walker_open randomizes the linked list walk over a contiguous block the blocks must be at least 16 pages (1024 lines) to defeat the prefetcher. This can be used to alleviate the need to disable the prefetchers. If the test is run using a single section or line/segment, then the linked list loop points to the next line and the prefetchers must be disabled. This serves as a good cross check.

By setting the stride to be a non zero multiple of pages (+ 64 bytes) the TLB system can be tested and the tlb events validated. Large pages of 2MB can be used as the buffers are allocated with mmap. This also avoids the 2GB limit inherent in many versions of malloc.

When testing the TLB subsystem a much smaller set of events is used as these are very specific tests.

The test help section explains the options used to drive the various configurations.

3) snoop tests

Multi threaded shared access tests are one of the few that the author does not run on the full set of events due to the large number of runs required in a dual socket system and the very specific nature of the tests. For these tests a buffer size is selected (2 values, L3 resident or dram resident), an initialization core, a “read” core and a “write” core. The write thread can either modify the lines or not, thus testing the hitm vs shared access. The segment size that is manipulated by each thread in succession is set to be L1D, L2 or L3 resident. Thus for a single location for the read thread on a dual socket system there are

(2 buffersize)*(2 initialization cores)*(2 write cores)*(3 segment sizes)*(2 hitm/share) = 48 runs

4) instruction starvation

The easiest way to test instruction starvation is with a test that streams cachelines of instructions that are built of instruction flows that will saturate the FE bandwidth. The linear_gen_full*.c tests are program generators that will make binaries of any desired size using 64 bit immediate mov instructions, which are 10 byte instructions. Building binaries that are resident in L1I, L2, L3 or dram and then running them with the prefetchers disabled makes it possible to measure the instruction delivery latency and validate the instruction starvation metrics (assorted uop flow with c=1:i=1 and rs_empty). What is measured is the bandwidth so the change in cycles/cacheline must be corrected for the overlapping cacheline requests. Thus the offcore_requests_outstanding:ifetch event has to work to extract a latency. The tests can also validate the ifetch and ITLB events, instruction and uop flow events, etc,etc.

5) branch kernels

There are a set of “kernels” providing coverage for a range of branch events. The loop kernels provide some testing but these are far more useful. They are again code generators so binaries of various sizes and complexities are created. These can be used to validate branch taken, mispredicted and baclear events by branch type. The sizes of branch prediction tables has grown sufficiently that making L1I resident baclear tests has become difficult. Once the tests become too big to fit in L1I the penalties can become difficult to evaluate accurately.

6) LBR validation and return stack

the callchain test can be used to test the LBRs and the misprediction bits in the LBR as well as the mispredicted near return event. The call chain just needs to be sufficiently long. The misprediction bits in the LBR targets may need to use the gooda file reader binary to be tested as they cannot be checked by perf stat. This test must be run with perf record and either run through gooda and the displays used to confirm the correct counts or the perf.data file can be dumped to text with the reader program (option of Makefile.gooda).

7)PEBS

In the mem_latency directory there is a walker_pebs.c test which is a linked list walker kernel with a large number of independent xor instructions added. The location of the memory access events can skid a considerable distance from the load if PEBS is not used, so this is a good test of the PEBS mechanism. This test must be run using perf record as the annotated disassembly is what shows that the event locations for PEBS are correct.

7)address capture

The best way the author knows to validate address capture is to also capture the registers. This allows an event dump to be used and analyzed for consistency between the captured address and the address reconstructed from the register values. This test must be run using perf record and then the raw buffer must be inspected to confirm the consistency of the PEBS captured load address with the values of the registers. Thus such a test is blocked until perf supports register capture.