Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors

By Dr David Levinthal PhD.
Version 1.0

Introduction

With the introduction of the Intel® Core™ i7 processor and Intel® Xeon™ 5500 processors, mass market computing enters a new era and with it a new need for performance analysis techniques and capabilities. The performance monitoring unit (PMU) of the processor has progressed in step, providing a wide variety of new capabilities to illuminate the code interaction with the architecture.
In this paper I will discuss the basic performance analysis methodology that applies to Intel® Core™ i7 processor and platforms that support Non-Uniform Memory Access (NUMA) using two Intel® Xeon 5500 processors based on the same microarchitecture as Intel® Core™ i7 processor. The events and methodology that referred to Intel® Core™ i7 processor also apply to Intel® Xeon™ 5500 processors which are based on the same microarchitecture as Intel® Core™ i7 processor. Thus statements made only about Intel® core™ i7 processors in this document also apply to the Intel® Xeon™ 5500 processor based systems. This will start with extensions to the basic cycle accounting methodology outlined for Intel® Core™2 processors(1) and also include both the specific NUMA directed capabilities and the large extension to the precise event based sampling (PEBS) .

Software optimization based on performance analysis of large existing applications, in most cases, reduces to optimizing the code generation by the compiler and optimizing the memory access. This paper will focus on this approach. Optimizing the code generation by the compiler requires inspection of the assembler of the time consuming parts of the application and verifying that the compiler generated a reasonable code stream. Optimizing the memory access is a complex issue involving the bandwidth and latency capabilities of the platform, hardware and software prefetching efficiencies and the virtual address layout of the heavily accessed variables. The memory access is where the NUMA nature of the Intel® Core™ i7 processor based platforms becomes an issue.

Performance analysis illuminates how the existing invocation of an algorithm executes. It allows a software developer to improve the performance of that invocation. It does not offer much insight about how to change an algorithm, as that really requires a better understanding of the problem being solved rather than the performance of the existing solution. That being said, the performance gains that can be achieved on a large

existing code base can regularly exceed a factor of 2, (particularly in HPC) which is certainly worth the comparatively small effort required.

Basic Intel® Core™ i7 Processor and Intel® Xeon™ 5500 Processor Architecture and Performance Analysis

Performance analysis on a micro architecture is the experimental investigation of the micro architecture's response to a given instruction and data stream. As such, a reasonable understanding of the micro architecture is required to understand what is actually being measured with the performance events that are available.

This section will cover the basics of the Intel® Core™ i7 processor and Intel® Xeon™ 5500 processor architecture. It is not meant to be complete but merely the briefest of introductions. For more details the reader should consult the Software Developers Programming Optimization Guide. This introduction is broken into sections as

1) Overview
2) Core out of order pipeline
3) Core memory subsystem
4) Uncore overview
5) Last Level Cache and Integrated memory controller
6) Intel® QuickPath Interconnect (Intel QPI)
7) Core and Uncore PMUs

Overview

The Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors are multi core, Intel® Hyper-Threading Technology (HT) enabled designs. Each socket has one to eight cores, which share a last level cache (L3 CACHE), a local integrated memory controller and an Intel® QuickPath Interconnect. Thus a 2 socket platform with quad core sockets might be drawn as:

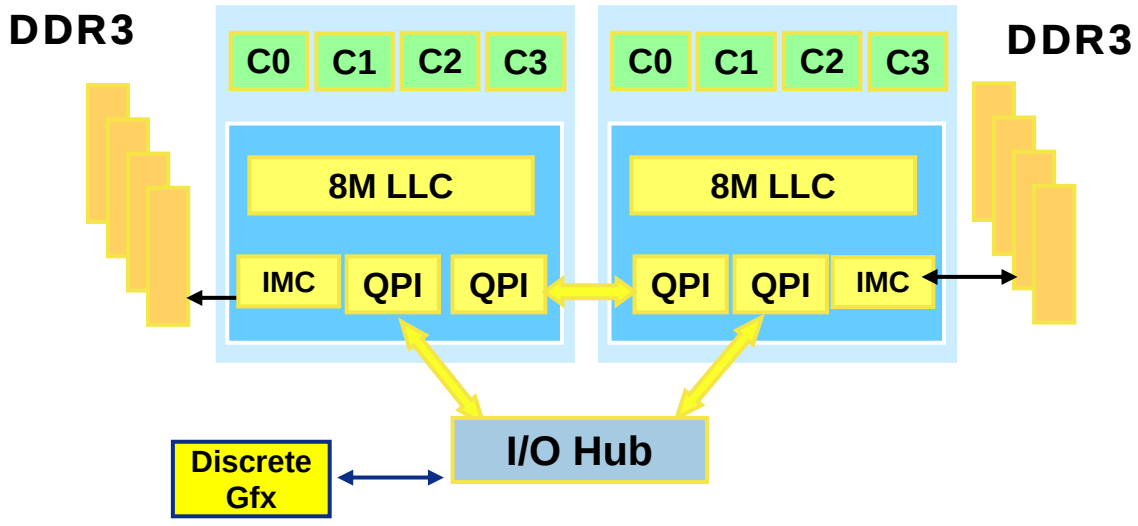


Figure 1

Each core is quite similar to that of the Intel® Core™2 processor. The pipelines are rather similar except that the Intel® Core™ i7 core and pipeline supports Intel® Hyper-Threading Technology (HT), allowing the hardware to interleave instructions of two threads during execution to maximize utilization of the core's resources. The Intel® Hyper-Threading Technology (HT) can be enabled or disabled through a bios setting. Each core has a 32KB data and instruction cache, a 256 KB unified mid-level cache and 2 level DTLB system of 64 and 512 entries. There is a single, 32 entry large page dtlb. The cores in a socket share an inclusive last level cache. The inclusive aspect of this cache is an important issue and will be discussed later. In the usual DP configuration the shared, inclusive last level cache is 8MB and 16 way associative.
The cache coherency protocol messages, between the multiple sockets, are exchanged over the Intel® QuickPath Interconnects. The inclusive L3 CACHE allow this protocol to be extremely fast, with the latency to the L3 CACHE of the adjacent socket being even less than the latency to the local memory.
One of the main virtues of the integrated memory controller is the separation of the cache coherency traffic and the memory access traffic. This enables an enormous increase in memory access bandwidth. This results in a non-uniform memory access (NUMA). The latency to the memory dimms attached to a remote socket is considerably longer than to the local dimms. A second advantage is that the memory control logic can run at processor frequencies and thereby reduce the latency.

The development of a reasonably hierarchical structure and usage of the performance events will require a fairly detailed knowledge of exactly how the components of Intel® Core™ i7 processor execute an application's stream of instructions and delivers the required data. What follows is a minimal introduction to these components.

Core Out of Order Pipeline

The basic analysis methodology starts with an accounting of the cycle usage for execution. The out of order execution can be considered from the perspective of a simple block diagram as shown below:

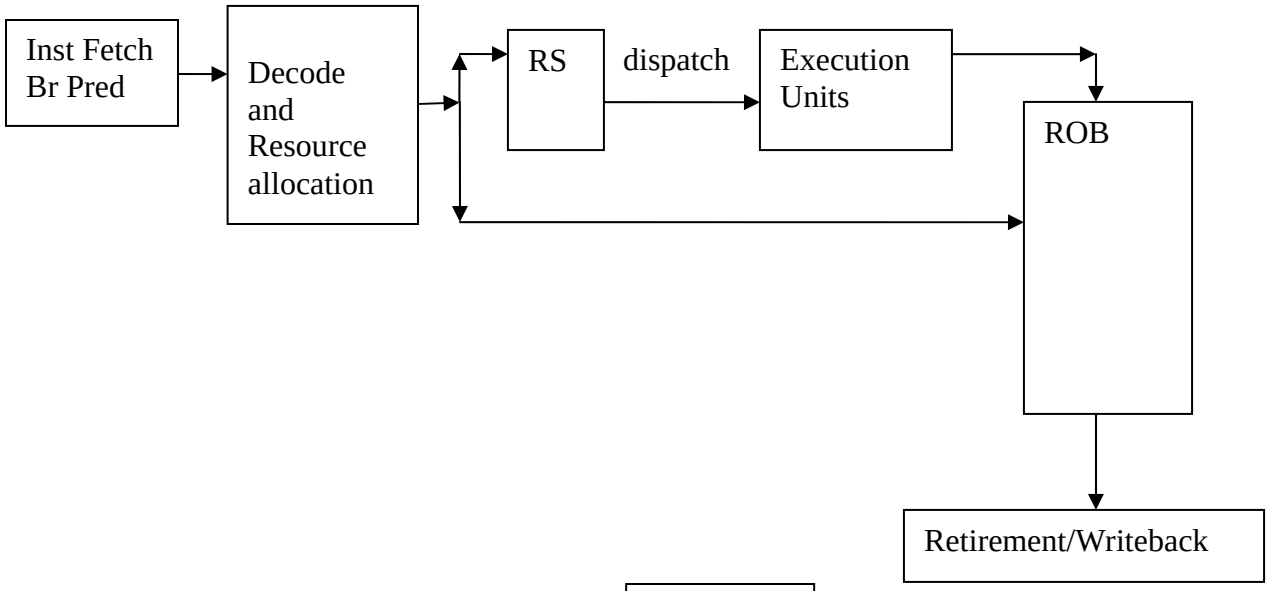


Figure 2

After instructions are decoded into the executable micro operations (uops), they are assigned their required resources. They can only be **issued** to the downstream stages when there are sufficient free 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) The uops associated with the instruction to be retired have completed, allowing the retirement of the entire instruction, or in the case of instructions that generate very large number of uops, enough to fill the retirement window
2) 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 and uops, 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 terms "**issued**", "**dispatched**", "**executed**" and "**retired**" have very precise meanings as to where in this sequence they occur and are used in the event names to help document what is being measured.
In the Intel® Core™ i7 Processor, the reservation station has 36 entries which are shared between the Hyper-threads when that mode (HT) is enabled in the bios, with some entries reserved for each thread to avoid locking. If not, all 36 could be available to the single running thread, making restarting a blocked thread inefficient. There are 128 positions in the reorder buffer, which are again divided if HT is enabled or entirely available to the single thread if HT is not enabled. As on Core™2 processors, the RS dispatches the uops to one of 6 dispatch ports where they are consumed by the execution units. This implies that on any cycle between 0 and 6 uops can be dispatched for execution.

The hardware branch prediction requests the bytes of instructions for the predicted code paths from the 32KB L1 instruction cache at a maximum bandwidth of 16 bytes/cycle. Instructions fetches are always 16 byte aligned, so if a hot code path starts on the 15th byte, the FE will only receive 1 byte on that cycle. This can aggravate instruction bandwidth issues. The instructions are referenced by virtual address and translated to physical address with the help of a 128 entry instruction translation lookaside buffer (ITLB). The x86 instructions are decoded into the processors uops by the pipeline front end. Four instructions can be decoded and issued per cycle.
If the branch prediction hardware mispredicts the execution path, the uops from the incorrect path which are in the instruction pipeline are simply removed where they are, without stalling execution. This reduces the cost of branch mispredictions. Thus the "cost" associated with such mispredictions is only the wasted work associated with any of the incorrect path uops that actually got dispatched and executed and any cycles that are idle while the correct path instructions are located, decoded and inserted into the execution pipeline.

Core Memory Subsystem

In applications working with large data footprints, memory access operations can dominate the application's performance. Consequently a great deal of effort goes into the design and instrumentation of the data delivery subsystem. Data is organized as a contiguous string of bytes and is transferred around the memory subsystem in cachelines of 64 bytes.
Generally, load operations copy contiguous subsets of the cachelines to registers, while store operations copy the contents of registers back into the local copies of the cachelines. SSE streaming stores are an exception as they create local copies of cachelines which are then used to overwrite the versions in memory, thus are slightly different. The local copies of the lines that are accessed in this way are kept in the 32KB L1 data cache. The access latency to this cache is 4 cycles.
While the program references data through virtual addresses, the hardware identifies the cachelines by the physical addresses. The translation between these two mappings is maintained by the operating system in the form of translation tables. These tables list the translations of the standard 4KB aligned address ranges called pages. They also handle any large pages that the application might have allocated. When a translation is used it is

kept in the data translation lookaside buffers (DTLBs) for future reuse, as all load and store operations require such a translation to access the data caches. Programs reference virtual addresses but access the cachelines in the caches through the physical addresses. As mentioned before, there is a multi level TLB system in each core for the 4KB pages. The level 1 caches have TLBs of 64 and 128 entries respectively for the data and instruction caches. There is a shared 512 entry second level TLB. There is a 32 entry DTLB for the large 2/4MB pages should the application allocate and access any large pages. There are 7 large page ITLB entries per HT. When a translation entry cannot be found in the DTLBs the hardware page walker (HPW) works with the OS translation data structures to retrieve the needed translation and updates the DTLBs. The hardware page walker begins its search in the cache for the table entry and then can continue searching in memory if the page containing the entry required is not found.

Cacheline coherency in a multi core multi socket system must be maintained to ensure that the correct values for the data variables can be retrieved. This has traditionally been done through the use of a 4 value state for each copy of each cacheline. The four state (MESI) cacheline protocol allows for a coherent use of data in a multi-core, multi-socket platform. A line that is only read can be shared and the cacheline access protocol supports this by allowing multiple copies of the cacheline to coexist in the multiple cores. Under these conditions, the multiple copies of the cacheline would be in what is called a Shared state (S). A cacheline can be put in an Exclusive state (E) in response to a “read for ownership” (RFO) in order to store a value. All instructions containing a lock prefix will result in a (RFO) since they always result in a write to the cache line. The F0 lock prefix will be present in the opcode or is implied by the xchg and cmpxchg instructions when a memory access is one of the operands. The exclusive state ensures exclusive access of the line. Once one of the copies is modified the cacheline’s state is changed to Modified (M). That change of state is propagated to the other cores, whose copies are changed to the Invalid state (I).

With the introduction of the Intel® QuickPath Interconnect protocol the 4 MESI states are supplemented with a fifth, Forward (F) state, for lines forwarded from on socket to another.

When a cacheline, required by a data access instruction, cannot be found in the L1 data cache it must be retrieved from a higher level and longer latency component of the memory access subsystem. Such a cache miss results in an invalid state being set for the cacheline. This mechanism can be used to count cache misses.

The L1D miss creates an entry in the 16 element superqueue and allocates a line fill buffer. If the line is found in the 256KB mid level cache (MLC, also referred to as L2), it is transferred to the L1 data cache and the data access instruction can be serviced. The load latency from the L2 CACHE is 10 cycles, resulting in a performance penalty of around 6 cycles, the difference of the effective L2 CACHE and L1D latencies. If the line is not found in the L2 CACHE, then it must be retrieved from the uncore.

When all the line fill buffers are in use, the data access operations in the load and store buffers cannot be processed. They are thus queued up in the load and store buffers. When all the load or store buffers are occupied, the front end is inhibited from issuing uops to the RS and OOO engine. This is the same mechanism as used in Core™2 processors to maintain pipeline consistency.

The Intel® Core™ i7 processor has a 4 component hardware prefetcher very similar to that of the Core™ processors. Two components associated with the L2 CACHE and two components associated with the L1 data cache. The 2 components of L2 CACHE hardware prefetcher are similar to those in the Pentium™ 4 and Core™ processors. There is a "streaming" component that looks for multiple accesses in a local address window as a trigger and an "adjacency" component that causes 2 lines to be fetched instead of one with each triggering of the "streaming" component. The L1 data cache prefetcher is similar to the L1 data cache prefetcher familiar from the Core™ processors. It has another "streaming" component (which was usually disabled in the bios' for the Core™ processors) and a "stride" or "IP" component that detected constant stride accesses at individual instruction pointers. The Intel® Core™ i7 processor has various improvements in the details of the hardware pattern identifications used in the prefetchers.

Uncore Memory Subsystem

Overview

The "uncore" is essentially a shared last level cache (L3 CACHE), a memory access chipset (Northbridge) , and a socket interconnection interface integrated into the multi processor package. Cacheline access requests (i.e. L2 Cache misses, uncacheable loads and stores) from the cores are serviced and the multi socket cacheline coherency is maintained with the other sockets and the I/O Hub.
There are five basic configurations of the Intel® Core™ i7 processor uncore.

1. Intel® Xeon™ 550 processor has a 3 channel integrated memory controller (IMC), 2 Intel® QuickPath Interconnects to support up to a DP configuration and an 8 MB L3 CACHE. This is the main focus of this document
2. Intel® Core™ i7 processor-HEDT (High End Desk Top) has a 3 channel IMC, 1 Intel® QuickPath Interconnect to access the chipset and an 8 MB L3 CACHE. This is for UP configurations
3. A quad core mainstream configuration with a 2 channel IMC, integrated PCI-e and an 8MB L3 CACHE
4. A dual core mainstream configuration where the memory access is through an off die chipset to enable support of more memory dimm formats equipped with a 4MB L3 CACHE
5. The 8-core implementation based on the Nehalem microarchitecture will be the MP configuration. This will be described in later documents.

Intel® Xeon™ 5500 Processor

IA block diagram of the Intel® Xeon™ 5500 processor package is shown below:

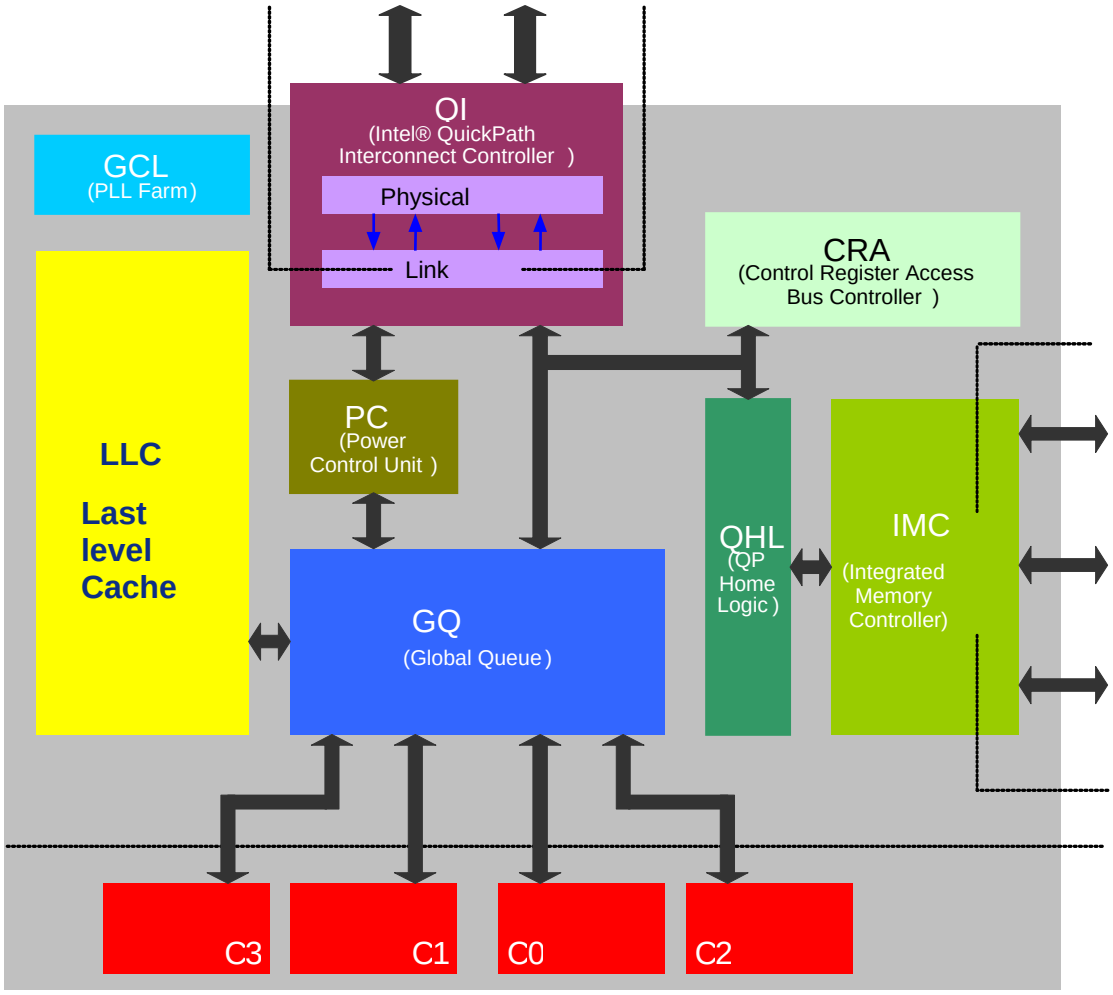


Figure 3

Cacheline requests from the cores or from a remote package or the I/O Hub are handled by the Intel® Xeon™ 5500 processor Uncore's Global Queue (GQ). The GQ contains 3 request queues for this purpose. One for writes with 16 entries and one of 12 entries for off package requests delivered by the Intel® QuickPath Interconnect and one of 32 entries for load requests from the cores.

On receiving a cacheline request from one of the cores, the GQ first checks the Last Level Cache (L3 CACHE) to see if the line is on the package. As the L3 CACHE is inclusive, the answer can be quickly ascertained. If the line is in the L3 CACHE and was owned by the requesting core it can be returned to the core from the L3 CACHE directly. If the line is being used by multiple cores, the GQ will snoop the other cores to see if there is a modified copy. If so the L3 CACHE is updated and the line is sent to the requesting core. In the event of an L3 CACHE miss the GQ must send out requests for the line. Since the cacheline could be in the other package, a request through the Intel® QuickPath Interconnect (Intel QPI) to the remote L3 CACHE must be made. As each Intel® Core™ i7 processor package has its own local integrated memory controller the GQ must identify the "home" location of the requested cacheline from the physical address. If the address identifies home as being on the local package, then the GQ makes a simultaneous request

to the local memory controller, the Integrated memory controller (IMC). If home is identified as belonging to the remote package, the request sent by the QPI will also be used to access the remote IMC.
This process can be viewed in the terms used by the Intel® QuickPath Interconnect protocol. Each socket has a Caching agent (that might be thought of as the GQ plus the L3 CACHE) and a Home agent (the IMC). An L3 CACHE miss results in simultaneous queries for the line from all the Caching Agents and the Home agent (wherever it is). This is shown diagrammatically below for a system with 3 caching agents (2 sockets and an I/O hub) none of whom have the line and a home agent, which ultimately delivers the line to the caching agent C that requested it.

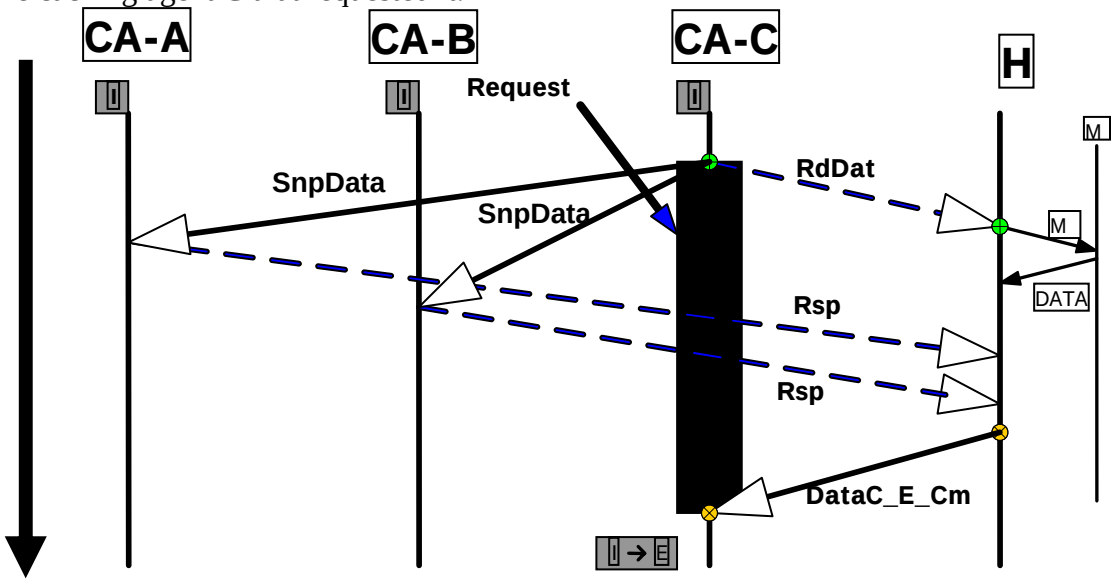


Figure 4

Clearly, the IMC has queues for handling local and remote, read and write requests. These will be discussed at greater length as the events that monitor their use are described.

Core Performance Monitoring Unit (PMU)

Each core has its own PMU. They have 3 fixed counters and 4 general counters for each Hyper-Thread. If HT is disabled in the bios only one set of counters is available. All the core monitoring events count on a per thread basis with one exception that will be discussed. The PMIs are raised on a per logical core or HT basis when HT is enabled. There is a significant expansion of the PEBS events with respect to Intel® Core™2 processors. This will be discussed in detail. The Last Branch Record (LBR) has been expanded to hold 16 source/target pairs for the last 16 taken branch instructions.

Uncore Performance Monitoring Unit (PMU)

The Uncore has its own PMU for monitoring its activity. It consists of 8 general counters and one fixed counter. The fixed counter monitors the uncore frequency, which is different than the core frequency. In order for the uncore PMU to generate an interrupt it must rely on the core PMUs. If an interrupt on overflow is desired, a bit pattern of which core PMUs to signal to raise a PMI must be programmed. As the uncore events have no knowledge of the core, PID or TID that ultimately generated the event, the most reasonable approach to sampling on uncore events requires sending an interrupt signal to all of the core PMUs and generating one PMI per logical core.

Performance Analysis and the Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processor Performance Events: Overview

The objective of performance analysis is of course to improve an applications performance. The process reveals insights into an existing code base's performance limitations. In general it does not tell the user how to improve an algorithm, merely the limitations in the algorithm the user has already chosen. Improving an algorithm usually requires deeper insight into the problem being solved, rather than insight into how the chosen solution performs.

There are dominantly two types of performance limitations identified through analysis. These are sub optimal code generation and sub optimal interaction of the code and the micro architecture. Performance event profiling with software tools like the VTune™ performance analyzer, PTU and such address both issues. A profile of an execution sensitive event like core cycles or instructions (uops) executed identifies which parts of the code are actually being executed and thus dominating the applications performance. One of the primary uses of such tools is as execution sensitive assembly editors. It is the disassembly view of such tools that allow a user to evaluate the code generation and determine if the compilation was optimal, if the high level language encoding results in artificially constraining the compiler's options (false loop dependencies for example), or identifying a compiler's inadequacies. While this is certainly one of the most important aspects of performance analysis, it is not the subject of this paper. The focus here is on the Intel® Core™ i7 processor specifics and identifying performance bottlenecks in the applications interaction with the micro architecture.

The spectrum of performance monitoring events on the Intel® Core™ i7 processor provides unprecedented insights into and application's interaction with the processor micro architecture. The remainder of this paper will be devoted to describing the systematic use of the performance events. It is divided into the following discussions

1) Cycle Accounting and Uop Flow
2) Stall Decomposition Overview
3) Precise Memory Access Events (PEBS)
4) Precise Branch Events (PEBS, LBR)
5) Core Memory Access Events (non-PEBS)
6) Other Core Events (non-PEBS)
7) Front End Issues

8) Uncore Events

Cycle Accounting and Uop Flow

Improving performance starts with identifying where in the application the cycles are spent and identifying how they can be reduced. As Amdahl's law points out, an application can only be sped up by the fraction of cycles that are being used by the section of code being optimized. To accomplish such a cycle count reduction it is critical to know how the cycles are being used. This is both to identify those places where there is nothing to be gained, but more importantly where any effort is most likely to be fruitful. Such a cycle usage decomposition is usually described as cycle accounting. The first step of the decomposition is usually to divide the cycles into two groups, productive and unproductive or "stalled". This is particularly important, as stalled cycles are usually the easiest to recover.

The techniques described here rely on several of the PMU programming options beyond the performance event selection. These include the threshold (:cmask=val), comparison logic (:inv=1), edge detection (:edge=1) and privilege level (:usr=1, :sup=1) filtering being applied to the event counting. When the PMU is programmed to count an event the PerfEvtSel register for one of the 4 programmable counters is programmed with the event code and umask value to select which event should be counted. There are many different programmable conditions under which the event can be counted. These are also controlled by the PerfEvtSel register. When the cmask value is zero the counter is incremented by the value of the event on each cycle. (ex: inst_retired.any can have any value from 0 to 4). If the cmask is non zero then the value on each cycle is compared to the cmask value and thus the cycles for which the comparison condition is true are counted. The condition can be either GE (>=) or LT (<), depending on whether the "inv" bit is zero or not. The "edge detect" can be used to count the changing of the condition determined by the cmask, this is used to count divides and sqrt instructions as will be discussed later. There are more details on the PerfEvtSel register in Appendix II.
The cycle usage is best evaluated with the flow of uops through the pipeline. In the Intel® Core™ i7 processor core there are three particularly valuable places where this can be done. Using the simplified pipeline diagram we identify these three spots as:

1) Output of the decoding and resource allocation (issue)
2) Execution
3) Retirement

On the diagram below we highlight a few of these performance events and where they monitor the uop flow.

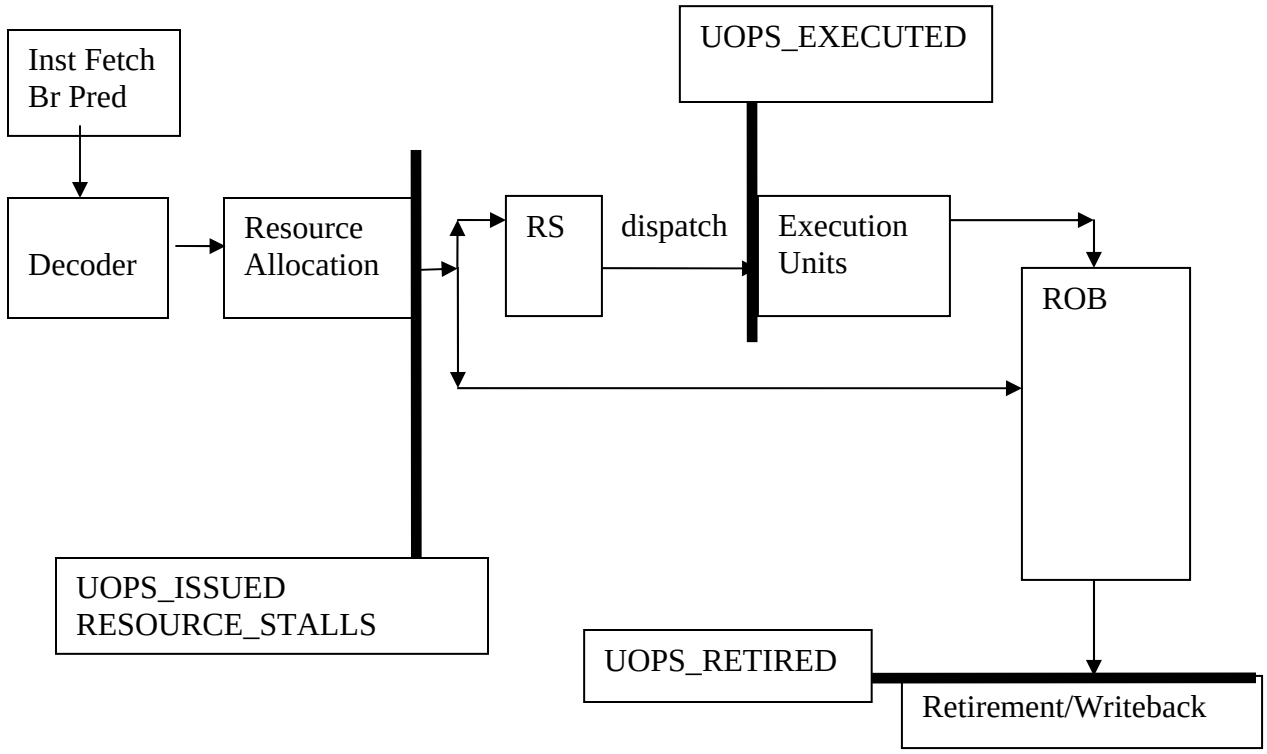


Figure 5

The pipeline has buffers distributed along the uops flow path, for example the RS and ROB. The result is the flow discontinuities (stalls) in one location do not necessarily propagate at all locations. The OOO execution can keep the execution units occupied during cycles where no uops retire, with the completed uops simply being staged in the ROB for future retirement. Similarly the buffering in the RS can similarly keep the execution units occupied during short discontinuities in uops being issued by the pipeline front end. The design optimizes the continuity of the uop flow at the dispatch to the execution units. Thus SW performance optimization should also focus on this objective.

In order to evaluate the efficiency of execution, cycles are divided into those where micro-ops are dispatched to the execution units and those where no micro-ops are dispatched, which are thought of as execution stalls. In the Intel® Core™ i7 processor (as on the Intel® Core™ 2 processors) uops are dispatched to one of six ports. By comparing the total uop count to 1 (cmask=1) and using a “less than” (inv=1) and “greater than or equal to” (inv = 0) comparison, the PMU can divide all cycles into “stalled” and “unstalled” classes. These PMU programmings are predefined enabling the identify:

Total cycles = UOPS_EXECUTED.CORE_STALL_CYCLES +
UOPS_EXECUTED.CORE_ACTIVE_CYCLES

Where UOPC_EXECUTED.CORE_STALL_CYCLES is defined as UOPS_EXECUTED:CMASK=1:INV=1, using the usual shorthand notation.

This expression is in a sense a trivial truism, uops either are, or are not, executed on any given cycle. This technique can be applied to any core event, with any threshold (cmask) value and it will always be true. Any event, with a given cmask threshold value, counts the cycles where the events value is >= to the cmask value (inv=0) , or < the cmask value (inv=1). Thus the sum of the counts for inv =0 and inv=1 for a non-zero cmask will always be the total core cycles, not just the unhalted cycles. This sum value is of course subject to any frequency throttling the core might experience during the counting period. The choice of dividing cycles at execution in this particular manner is driven by the realization that ultimately keeping the execution units occupied is one of the essential objectives of optimization.
Total cycles can be directly measured with CPU_CLK_UNHALTED.TOTAL_CYCLES. This event is derived from CPU_CLK_UNHALTED.THREAD by setting the cmask = 2 and inv = 1, creating a condition that is always true. The difference between these two is the halted cycles. These occur when the OS runs the null process.
The signals used to count the memory access uops executed (ports 2, 3 and 4) are the only core events which cannot be counted on a logical core or HT basis. Thus the total execution stall cycles can only be evaluated on a per core basis. If the HT is disabled this presents no difficulty. There is some added complexity when the HT is enabled however. While the memory ports only count on a per core basis, the ALU ports (0,1,5) count on a per thread basis. The number of cycles where no uops were dispatched on the ALU ports can be evaluated on a per thread basis consequently. This event is called UOPS_EXECUTED.PORT015_STALL_CYCLES. Thus in the case where HT is enabled we have the following inequality

UOPS_EXECUTED.CORE_STALL_CYCLES <=
True execution stalls per thread <=
UOPS_EXECUTED.PORT015_STALL_CYCLES

Of course with HT disabled then
UOPS_EXECUTED.CORE_STALL_CYCLES =
True execution stalls per thread

In addition the uop flow can be measured at issue and retirement on a per thread basis and so can the number of cycles where no uops flow at those points. These events are predefined as UOPS_ISSUED.STALL_CYCLES for measuring stalls in uop issue and UOPS_RETIRED.STALL_CYCLES for measuring stalls in uop retirement, respectively.

The edge detection option in the PMU can be used to count the number of times an event’s value changes, by detecting the rising signal edge. If this is applied to UOPS_EXECUTED.CORE_STALLS_CYCLES as, (UOPS_EXECUTED:CMASK=1:INV=1:EDGE=1), then the PMU will count the number of stalls. This programming is defined as the event UOPS_EXECUTED.CORE_STALL_COUNT. The ratio,

UOPS_EXECUTED.CORE_STALLS_CYCLES/
UOPS_EXECUTED.CORE_STALLS_COUNT

is the average stall duration, and with the use of sampling can be measured reasonably accurately even within a code region like a single loop.

Branch mispredictions, Wasted Work, Misprediction Penalties and UOP Flow

Branch mispredictions can introduce execution inefficiencies that are typically decomposed into three components.

1) Wasted work associated with executing the uops of the incorrectly predicted path
2) Cycles lost when the pipeline is flushed of the incorrect uops
3) Cycles lost while waiting for the correct uops to arrive at the execution units

In the Intel® Core™ i7 processor, there are no execution stalls associated with clearing the pipeline of mispredicted uops (component 2). These uops are simply removed from the pipeline without stalling executions or dispatch. This typically lowers the penalty for mispredicted branches. Further, the penalty associated with instruction starvation (component 3) can be measured for the first time in OOO x86 architectures.

Speculative OOO execution introduces a component of execution inefficiency due to the uops on mispredicted paths being dispatched to the execution units. This represents wasted work as these uops will never be retired as is part of the cost associated with mispredicted branches. It can be found through monitoring the flow of uops through the pipeline. The uop flow can be measured at 3 points in the diagram shown above, going into the RS with the event UOPS_ISSUED, going into the execution units with UOPS_EXECUTED and at retirement with UOPS_RETIRED. The differences of between the upstream measurements and at retirement measure the wasted work associated with these mispredicted uops.

As UOPS_EXECUTED must be measured per core, rather than per logical core/HT, the wasted work per core is evaluated as

Wasted Work = (UOPS_EXECUTED.PORT234_CORE +
UOPS_EXECUTED.PORT015 (for HT1) +
UOPS_EXECUTED.PORT015 (for HT2)) –
(UOPS_RETIRED.ANY(for HT1) +
UOPS_RETIRED.ANY(for HT2))

The events were designed to be used in this manner without corrections for micro or macro fusion. If HT is disabled, the count for the second HT is not needed.
A “per thread” measurement can be made looking at the difference between the uops issued and uops retired as both of these events can be counted per logical core/HT. It over counts slightly, by the mispredicted uops that are eliminated in the RS before they can waste cycles being executed, but this is a small correction.

Wasted Work/thread = (UOPS_ISSUED.ANY + UOPS_ISSUED.FUSED)
– UOPS_RETIRED.ANY

As stated above, there is no interruption in uop dispatch or execution due to flushing the pipeline. Thus the second component of the misprediction penalty is zero.

The third component of the misprediction penalty, instruction starvation, occurs when the instructions associated with the correct path are far away from the core and execution is stalled due to a lack of uops. This can now be explicitly measured at the output of the resource allocation as follows. Using a cmask =1 and inv=1 logic applied to UOPS_ISSUED, we can count the total number of cycles where no uops were issued to the OOO engine.

UOPS_ISSUED.STALL_CYCLES = UOPS_ISSUED.ANY:CMASK=1:INV=1

Since the event RESOURCE_STALLS.ANY counts the number of cycles where uops could not be issued due to a lack of downstream resources (RS or ROB slots, load or store buffers etc), the difference is the cycles no uops are issued because there were none available.

With HT disabled we can identify an instruction starvation condition indicating that the front end was not delivering uops when the execution stage could have accepted them.

Instruction Starvation =

UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY

When HT is enabled, the uop delivery to the RS alternates between the two threads. In an ideal case the above condition would then count 50% of the cycles, as those cycles were delivering uops for the other thread. We can modify the expression by subtracting the cycles that the other thread is having uops issued.

Instruction Starvation =

UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY
-UOPS_ISSUED.ANY:CMASK=1(other thread)

But this will over count as the resource_stall condition could exist on “this” thread while the other thread was issuing uops. An alternative might be

CPU_CLK_UNHALTED.THREAD – UOPS_ISSUED.CORE_CYCLES_ACTIVE-RESOURCE_STALLS.ANY

Where UOPS_ISSUED.CORE_CYCLES_ACTIVE counts the UOPS_ISSUED.ANY event with cmask=1 and allthreads=1, thus counting the cycles either thread issues uops. The problem of course is that if the other thread can always issue uops, it will mask the stalls in the thread that cannot.

The event INST_RETIRED.ANY (instructions retired) is most commonly used to evaluate a cycles/instruction ratio, but the most powerful usage is in evaluating basic block execution counts. All of the instructions in a basic block are retired exactly the same number of times by the very definition of a basic block. As several instructions tend to be retired on each cycle where instructions are retired there tends to be a clustering of the IP values associated with sampling on INST_RETIRED.ANY. This same clustering also occurs for the cpu cycle counting events. The result is that the distribution of samples in a VTune™ Analyzer type disassembly spreadsheet is far from uniform. Frequently there are instructions with no samples at all right next to instructions with

thousands of samples. The solution to this is to average the sample counts over the instructions of the basic block. This will result in yielding the best measurement of the basic block execution count.

Basic Block Execution Count =

$$\Sigma_{\mathbf{inst\_in\_BB}} \text{Samples(inst\_retired)*Sample\_after\_Value} / (\text{Number of inst in BB})$$

When analyzing the execution of loops, the basic block execution counts can be used to get the average tripcount (iteration count) of the loop. For a simple loop with no conditional branches, this ends up being the ratio of the basic block execution count of the loop block to the basic block execution count of the block immediately before and/or after the loop block. Judicious use of averaging over multiple blocks can be used to improve the accuracy. Usually the objective of the analysis is just to determine if the tripcount is large (> 100) or very small (<10), so this rough technique is usually adequate.
There is a fixed counter version of the event and a version that can be programmed into the general counters, which also uses the PEBS (precise event based sampling) mechanism. The PEBS mechanism is armed by the overflow of the counter. There is a short propagation delay between the counter overflow and when PEBS is ready to capture the next event. This shadow makes the use of the precise event inappropriate for basic block execution counting. By far the best mechanism for this is to use the PEBS br_inst_retired.all_branches event and capture the LBRs (Last Branch Records). More will be said of the use of the precise version in the section on precise events.
A final event should be mentioned in regards to stalled execution. Chains of dependent long latency instructions (fmul, fadd, imul, etc) can result in the dispatch being stalled while the outputs of the long latency instructions become available. In general there are no events that assist in counting such stalls with the exception of the divide and sqrt instructions. For these two instructions the event ARITH can be used to count both the occurrences of these instructions and the duration in cycles that they kept their execution units occupied. The event ARITH.CYCLES_DIV_BUSY counts the cycles that either the divide/sqrt execution unit was occupied. (perhaps the events name is thus a bit misleading)

The flow of uops is mostly due to the decoded instructions. There are also uops that can enter the flow due to micro coded exception handling, like those associated with floating point exceptions. Micro code will be covered as part of the Front End discussion.

In summary, a table of these events is shown below, with C indicating the CMASK value, I indicating the INV value, E indicating the EDGE DETECT value and AT indicating the value of the ALLTHREAD bit. For the Edge Detect to work, a non zero cmask value must also be used.
Table 1

Event Name	Definition	Umask	Event	C	I	E	AT
ARITH.CYCLES_DIV_BUSY	Cycles the divider is busy	1	14	0	0	0	0
ARITH.DIV	Divide Operations executed	1		0	0	1	0
ARITH.MUL	Multiply operations executed	2		0	0	0	0
CPU_CLK_UNHALTED.REF	Reference cycles when thread is not halted	0	Fixed Ctr	0	0	0	0

CPU_CLK_UNHALTED.THREAD	Cycles when thread is not halted		0	Fixed Ctr	0	0	0	0
CPU_CLK_UNHALTED.THREAD_P	Cycles when thread is not halted (programmable counter)		0	3C	0	0	0	0
CPU_CLK_UNHALTED.REF_P	Reference cycles when thread is not halted (programmable counter)		1		0	0	0	0
INST_RETIRED.ANY	Instructions retired (fixed counter)		0	Fixed Ctr	0	0	0	0
INST_RETIRED.ANY_P	Instructions retired (programmable counter)		1	C0	0	0	0	0
UOPS_EXECUTED.PORT0	Uops dispatched from port 0		1	B1	0	0	0	0
UOPS_EXECUTED.PORT1	Uops dispatched on port 1		2		0	0	0	0
UOPS_EXECUTED.PORT2_CORE	Uops dispatched on port 2		4		0	0	0	1
UOPS_EXECUTED.PORT3_CORE	Uops dispatched on port 3		8		0	0	0	1
UOPS_EXECUTED.PORT4_CORE	Uops dispatched on port 4		10		0	0	0	1
UOPS_EXECUTED.PORT5	Uops dispatched on port 5		20		0	0	0	0
UOPS_EXECUTED.PORT015	Uops dispatched on ports 0, 1 or 5		40		0	0	0	0
UOPS_EXECUTED.PORT015_STALL_CYCLES	Cycles no Uops dispatched on ports 0, 1 or 5		40		1	1	0	0
UOPS_EXECUTED.PORT234_CORE	Uops dispatched on ports 2, 3 or 4		80		0	0	0	1
UOPS_EXECUTED.CORE_ACTIVE_CYCLES	Cycles no Uops dispatched on any port	3F			1	0	0	1
UOPS_EXECUTED.CORE_STALL_COUNT	Number of times no Uops dispatched on any port	3f			1	1	1	1
UOPS_EXECUTED.CORE_STALL_CYCLES	Cycles no Uops dispatched on any port	3F			1	1	0	1
UOPS_ISSUED.ANY	Uops issued		1	0E	0	0	0	0
UOPS_ISSUED.STALL_CYCLES	Cycles no Uops were issued		1		1	1	0	0
UOPS_ISSUED.FUSED	Fused Uops issued		2		0	0	0	0
UOPS_RETIRED.ACTIVE_CYCLES	Cycles Micro-ops are retiring		1	C2	1	0	0	0
UOPS_RETIRED.ANY	Micro-ops retired		1		0	0	0	0
UOPS_RETIRED.STALL_CYCLES	Cycles Micro-ops are not retiring		1		1	1	0	0
UOPS_RETIRED.RETIRE_SLOTS	Number of retirement slots used		2		0	0	0	0
UOPS_RETIRED.MACRO_FUSED	Number of macro-fused Uops retired		4		0	0	0	0
RESOURCE_STALLS.ANY	Resource related stall cycles		1	A2	0	0	0	0
RESOURCE_STALLS.LOAD	Load buffer stall cycles		2		0	0	0	0
RESOURCE_STALLS.RS_FULL	Reservation Station full stall cycles		4		0	0	0	0
RESOURCE_STALLS.STORE	Store buffer stall cycles		8		0	0	0	0
RESOURCE_STALLS.ROB_FULL	ROB full stall cycles		10		0	0	0	0
RESOURCE_STALLS.FPCW	FPU control word write stall cycles		20		0	0	0	0
RESOURCE_STALLS.MXCSR			40		0	0	0	0
RESOURCE_STALLS.OTHER	Other Resource related stall cycles		80		0	0	0	0

Stall Decomposition Overview

The decomposition of the stall cycles is accomplished through a standard approximation. It is assumed that the penalties occur sequentially for each performance impacting event. Consequently, the total loss of cycles available for useful work is then the number of events, N_i, times the average penalty for each type of event, P_i.

$$\text{Counted\_Stall\_Cycles} = \sum P_i * N_i$$

This only accounts for the performance impacting events that are or 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

$$\text{Unaccounted} = \text{Stalls} - \text{Counted\_Stall\_Cycles}$$
$$= \text{UOPS\_EXECUTED.CORE\_STALL\_CYCLES} - \sum P_i * N_i(\text{both threads})$$

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. As UOPS_EXECUTED.CORE_STALL_CYCLES counts on a per core basis rather than on a per thread basis, the over counting can become severe. In such cases it may be preferable to use the port 0,1,5 uop stalls, as that can be done on a per thread basis.

$$\text{Unaccounted/thread} = \text{Stalls/thread} - \text{Counted\_Stall\_Cycles/thread}$$
$$= \text{UOPS\_EXECUTED. PORT015\_THREADED\_STALL\_CYCLES} - \sum P_i * N_i$$

This unaccounted component is meant to represent the components that were either not counted due to lack of performance events or simply neglected during the data collection.

One can also choose to use the “retirement” point as the basis for stalls. The PEBS UOPS_RETIRED.STALL_CYCLES event has the advantage of being evaluated on a per thread basis and being having the HW capture the IP associated with the retiring uop. This means that the IP distribution will not be effected by STI/CLI deferral of interrupts in critical sections of OS kernels, thus producing a more accurate profile of OS activity.

Measuring Penalties

Decomposing the stalled cycles in this manner should always start by first considering the large penalty events, events with penalties of greater than 10 cycles for example. Short penalty events (P < 5 cycles) can frequently be hidden by the combined actions of the OOO execution and the compiler. Both of these strive to create maximal parallel execution for precisely the purpose of keeping the execution units busy during stalls due to instruction dependencies. The large penalty operations are dominated by memory access and the very long latency instructions for divide and sqrt.

The largest penalty events are associated with load operations that require a cacheline which is not in one of the core’s two data caches. Not only must we count how many occur, but we need to know what penalty to assign. The standard approach to measuring latency is to measure the average number of cycles a request is in a queue.

$$\textbf{Latency} = (\Sigma_{\text{cycles}}\ \textbf{Queue\_entries\_outstanding})/\textbf{Queue\_inserts}$$

However, the penalty associated with each queue insert (ie cachemiss), is the latency divided by the average queue occupancy. This correction is needed to avoid over counting associated with overlapping penalties.

Average Queue Depth = (Σ_{cycles} Queue_entries_outstanding) /Cycles_queue_not_empty

Thus

Penalty = Latency/Average Queue Depth
= Cycles_queue_not_empty/Queue_inserts

An alternative way of thinking about this is to realize that the sum of all the penalties, for an event that occupies a queue for its duration, cannot exceed the time that the queue is not empty.

Cycles_queue_not_empty >= Events * <Penalty>

The equality results in the expression derived in the first part of the discussion.

Neither of these more standard techniques will be used much for this processor. In part due to the wide number of data sources and the large variations in their data delivery latencies. The Precise Event Based Sampling (PEBS) will be the technique of choice The use of the precise latency event, that will be discussed later, provides a more accurate and flexible measurement technique when sampling is used. As each sample records both a load to use latency and a data source, the average latency per data source can be evaluated. Further as the PEBS hardware supports buffering the events without generating a PMI until the buffer is full, it is possible to make such an evaluation quite efficient.

While there are many events that will yield the number of L2 CACHE misses, the associated penalties may average over a wide variety of data sources which actually have individual penalties that vary by an order of magnitude. A more detailed decomposition is needed that just an L2 CACHE miss.

The approximate latencies for the individual data sources that respond to an L2 CACHE miss are shown in table 2. These values are only approximate as they depend on processor frequency and DIMM speed among other things.

Table 2

Data Source	Latency
L3 CACHE hit, line unshared	~40 cycles
L3 CACHE hit, shared line in another core	~65 cycles
L3 CACHE hit, modified in another core	~75 cycles
remote L3 CACHE	~100-300 cycles
Local Dram	~60 ns
Remote Dram	~100 ns

NOTE: THESE VALUES ARE ROUGH APPROXIMATIONS. THEY DEPEND ON CORE AND UNCORE FREQUENCIES, MEMORY SPEEDS, BIOS SETTINGS, NUMBERS OF DIMMS, ETC,ETC..YOUR MILEAGE MAY VARY.

Core Precise Events

Overview

The Precise Event Based Sampling (PEBS) mechanism enables the PMU to capture the architectural state and IP at the completion of the instruction that caused the event. This not only allows the location of the events in the instruction space to be accurate profiled, but by capturing the values of the registers, instruction arguments can be reconstructed in a post processing phase. The Intel® Core™ i7 processor has greatly expanded the numbers and types of precise events.
The mechanism works by using the counter overflow to arm the PEBS data acquisition. Then on the next event, the data is captured and the interrupt is raised.
The captured IP value is sometimes referred to as IP +1, because at the completion of the instruction, the IP value is that of the next instruction.
By their very nature precise events must be “at-retirement” events. For the purposes of this discussion the precise events are divided into Memory Access events, associated with the retirement of loads and stores, and Execution Events, associated with the retirement of all instructions or specific non memory instructions (branches, FP assists, SSE uops)

Precise Memory Access Events

There are two enormously powerful properties common to all precise memory access events:

1) The exact instruction can be identified because the hardware captures the IP of the offending instruction. Of course the captured IP is that of the following instruction but one simply moves the samples up one instruction. This works even when the recorded IP points to the first instruction of a basic block because in such a case the offending instruction has to be the last instruction of the previous basic block, as branch instructions never load or store data.
2) The PEBS buffer contains the values of all 16 general registers, R1-R16, where R1 is also called RAX. When coupled with the disassembly the address of the load or store can be reconstructed and used for data access profiling. The Intel® Performance Tuning Utility does exactly this, providing a wide variety of powerful analysis techniques.

The Intel® Core™ i7 processor precise memory access events mainly focus on loads as those are the events typically responsible for the very long duration execution stalls. They are broken down by the data source, thereby indicating the typical latency and the data locality in the intrinsically NUMA configurations. These precise load events are the only L2 CACHE, L3 CACHE and DRAM access events that only count loads. All others will also include the L1D and/or L2 CACHE hardware prefetch requests. Many will also include RFO requests, both due to stores and to the hardware prefetchers.
All four general counters can be programmed to collect data for precise events.
The ability to reconstruct the virtual addresses of the load and store instructions allows an analysis of the cacheline and page usage efficiency. Even though cachelines and pages are defined by physical address the lower order bits are identical, so the virtual address can be used.

As the PEBS mechanism captures the values of the register at completion of the instruction, the dereferenced address for the following type of load instruction (Intel asm convention) cannot be reconstructed.

MOV RAX, [RAX+const]

This kind of instruction is mostly associated with pointer chasing

mystruc = mystruc->next;

This is a significant shortcoming of this approach to capturing memory instruction addresses.

The basic memory access events are shown in the table below:

Table 3

Event Name	Description	umask	Event
MEM_INST_RETIRED.LOADS	Instructions retired which contains a load	01	0B
MEM_INST_RETIRED.STORES	Instructions retired which contains a store	02	
MEM_LOAD_RETIRED.L1D_HIT	Retired loads that hit the L1 data cache	01	CB
MEM_LOAD_RETIRED.L2_HIT	Retired loads that hit the L2 cache	02	
MEM_LOAD_RETIRED. LLC_UNSHARED_HIT	Retired loads that hit the LL3 cache	04	
MEM_LOAD_RETIRED. OTHER_CORE_L2_HIT_HITM	Retired loads that hit sibling core's L2	08	
MEM_LOAD_RETIRED.LLC _MISS	Retired loads that miss the LL3 cache	10	
MEM_LOAD_RETIRED. DROPPED_EVENTS	Retired load info dropped due to data breakpoint	20	
MEM_LOAD_RETIRED.HIT_LFB	Retired loads that miss the L1 data cache and hit a line fill buffer	40	
MEM_LOAD_RETIRED.DTLB_MISS	Retired loads that miss the DTLB	80	
MEM_UNCORE_RETIRED. OTHER_CORE_L2_HITM	Memory instructions retired LL3 Cache hit and HITM in sibling core	02	0F
MEM_UNCORE_RETIRED. REMOTE_CACHE_LOCAL_HOME_HIT	Memory instructions retired remote cache HIT	08	
MEM_UNCORE_RETIRED. REMOTE_DRAM	Memory instructions retired remote DRAM	10	
MEM_UNCORE_RETIRED.LOCAL_DRAM	Memory instructions retired local DRAM	20	
MEM_UNCORE_RETIRED.UNCACHEABLE	Memory instructions retired IO	80	
MEM_STORE_RETIRED.DTLB_MISS	Retired stores that miss the DTLB	01	0C
MEM_STORE_RETIRED. DROPPED_EVENTS	Retired stores dropped due to data breakpoint	02	
ITLB_MISS_RETIRED	Retired instructions that missed the ITLB	20	C8

Strictly speaking the ITLB miss event is really an execution event but is listed here as it is associated with cacheline access.
The precise events listed above allow load driven cache misses to be identified by data source. This does not identify the “home” location of the cachelines with respect to the NUMA configuration. The exceptions to this statement are the events MEM_UNCORE_RETIRED.LOCAL_DRAM and MEM_UNCORE_RETIRED.NON_LOCAL_DRAM. These can be used in conjunction with instrumented malloc invocations to identify the NUMA “home” for the critical contiguous buffers used in an application.
The sum of all the MEM_LOAD_RETIRED events will equal the MEM_INST_RETIRED.LOADS count.
A count of L1D misses can be achieved with the use of all the MEM_LOAD_RETIRED events, except MEM_LOAD_RETIRED.L1D_HIT. It is better to use all of the individual MEM_LOAD_RETIRED events to do this, rather than the difference of MEM_INST_RETIRED.LOADS-MEM_LOAD_RETIRED.L1D_HIT because while the total counts of precise events will be correct, and they will correctly identify instructions that caused the event in question, the distribution of the events may not be correct due to PEBS SHADOWING, discussed later in this section.

L1D_MISSES = MEM_LOAD_RETIRED.HIT_LFB + MEM_LOAD_RETIRED.L2_HIT + MEM_LOAD_RETIRED.LLC_UNSHARED_HIT + MEM_LOAD_RETIRED.OTHER_CORE_HIT_HITM + MEM_LOAD_RETIRED.LLC_MISS

MEM_LOAD_RETIRED.LLC_UNSHARED_HIT is not well named. The inclusive L3 CACHE has a bit pattern to identify which core has a copy of the line. If the only bit set is for the requesting core (unshared hit) then the line can be returned from the L3 CACHE with no snooping of the other cores. If multiple bits are set, then the line is in a shared state and the copy in the L3 CACHE is current and can also be returned without snooping the other cores. If the line is read for ownership (RFO) by another core, this will put the copy in the L3 CACHE into an exclusive state. If the line is then modified by that core and later evicted, the written back copy in the L3 CACHE will be in a modified state and snooping will not be required.
MEM_LOAD_RETIRED.LLC_UNSHARED_HIT counts all of these. The event should really have been called MEM_LOAD_RETIRED.LLC_HIT_NO_SNOOP.
Similarly, MEM_LOAD_RETIRED.LLC_HIT_OTHER_CORE_HIT_HITM would have been better named as MEM_LOAD_RETIRED.LLC_HIT_SNOOP. The author apologizes for this, having been the one responsible for the poor naming.

When a modified line is retrieved from another socket it is also written back to memory. This causes remote HITM access to appear as coming from the home dram. The MEM_UNCORE_RETIRED.LOCAL_DRAM and MEM_UNCORE_RETIRED.REMOTE_DRAM thus also count the L3 CACHE misses satisfied by modified lines in the caches of the remote socket.

There is a difference in the behavior of MEM_LOAD_RETIRED.DTLB_MISSES with respect to that on Intel® Core™2 processors. Previously the event only counted the first miss to the page, as do the imprecise events. The event now counts all loads that result in a miss, thus it includes the secondary misses as well.

Latency Event

Saving the best for last, the Intel® Core™ i7 processor has a "latency event" which is very similar to the Itanium® Processor Family Data EAR event. This event samples loads, recording the number of cycles between the execution of the instruction and actual deliver of the data. If the measured latency is larger than the minimum latency programmed into MSR 0x3f6, bits 15:0, then the counter is incremented. Counter overflow arms the PEBS mechanism and on the next event satisfying the latency threshold, the measured latency, the virtual or linear address and the data source are copied into 3 additional registers in the PEBS buffer. Because the virtual address is captured into a known location, the sampling driver could also execute a virtual to physical translation and capture the physical address. The physical address identifies the NUMA home location and in principle allows an analysis of the details of the cache occupancies.

Further, as the address is captured before retirement even the pointer chasing encodings

```
MOV RAX, [RAX+const]
```

have their addresses captured.

Because an MSR is used to program the latency only one minimum latency value can be sampled on a core during a given period. To enable this, the Intel performance tools restrict the programming of this event to counter 4 to simplify the scheduling.

The preprogrammed event files used by the Intel® PTU and Vtune™ Performance Analyzer contain the following latency events, differing in the minimum latencies required to make them count. Both tools do the required programming of MSR 0x3f6.

Table 4

Event Name	Description	umask	Event
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_0	Load instructions retired above 0 cycles	10	0B
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_4	Load instructions retired above 4 cycles	10	
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8	Load instructions retired above 8 cycles	10	
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_10	Load instructions retired above 16 cycles	10	
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_20	Load instructions retired above 32 cycles	10	
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_40	Load instructions retired above 64 cycles	10	
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_80	Load instructions retired above 128 cycles	10	
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_100	Load instructions retired above 256 cycles	10	
MEM_INST_RETIRED.LATENCY_ABOVE	Load instructions retired above	10	

_THRESHOLD_200	512 cycles	
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_400	Load instructions retired above 1024 cycles	10
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_800	Load instructions retired above 2048 cycles	10
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_1000	Load instructions retired above 4096 cycles	10
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_2000	Load instructions retired above 8192 cycles	10
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_4000	Load instructions retired above 16384 cycles	10
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8000	Load instructions retired above 32768 cycles	10

The Data Source register captured in the PEBS buffer with the Latency Event is interpreted as follows:

Table 5

Intel® Core™ i7 Processor Data Source Encoding	**Data Source short description**	**Data Source Longer Description**
0x0	Unknown Miss	Unknown cache miss.
0x1	L1 Hit	Minimal latency core cache hit. This request was satisfied by the data cache.
0x2	Fill Buffer Hit	Pending core cache hit. The data is not yet in the data cache, but is located in a line fill buffer and will soon be committed to cache. The data request was satisfied from the line fill buffer.
0x3	L2 CACHE Hit	Highest latency core cache hit. This data request was satisfied by the L2 CACHE.
0x4	L3 CACHE Hit	L3 CACHE Hit: Hit the last level cache in the uncore with no coherency actions required (snooping).
0x5	L3 CACHE Hit Other Core Hit Snp	L3 CACHE Hit: Hit the last level cache and was serviced by another core with a cross core snoop where no modified copies found. (clean)
0x6	L3 CACHE Hit Other Core HitM	L3 CACHE Hit: Hit the last level cache and was serviced by another core with a cross core snoop where modified copies found. (HITM)
0x7	L3 CACHE_No_Details	Encoding not supported
0x8	Remote_Cache_FWD	L3 CACHE Miss: Local home requests that missed the last level cache and was serviced by forwarded data following a cross package snoop where no modified copies found. **(Remote home requests are not counted)**

0x9	Remote_Cache_HITM	L3 CACHE Miss: Local or Remote home requests that missed the last level cache and was serviced by forwarded data following a cross package snoop where a modified copy was found and coherency actions taken. **(Not supported at this time)**
0xA	Local_Dram_GoTo_S	L3 CACHE Miss: Local home requests that missed the last level cache and was serviced by local dram. (goto Shared state)
0xB	Remote_Dram_GoTo_S	L3 CACHE Miss: Remote home requests that missed the last level cache and was serviced by remote dram. (goto Shared state)
0xC	Local_Dram_GoTo_E	L3 CACHE Miss: Local home requests that missed the last level cache and was serviced by local dram. (goto Execlusive state)
0xD	Remote_Dram_GoTo_E	L3 CACHE Miss: Remote home requests that missed the last level cache and was serviced by remote dram. (goto Exclusive state)
0xE	I/O	None: The request was a result of an input or output operation.
0xF	Uncacheable	The request was to un-cacheable memory.

The latency event is the recommended method to measure the penalties for a cycle accounting decomposition. Each time a PMI is raised by this PEBS event a load to use latency and a data source for the cacheline is recorded in the PEBS buffer. The data source for the cacheline can be deduced from the low order 4 bits of the data source field and the table shown above. Thus an average latency for each of the 16 sources can be evaluated from the collected data. As only one minimum latency at a time can be collected it may be awkward to evaluate the latency for an L2 CACHE hit and a remote socket dram. A minimum latency of 32 cycles should give a reasonable distribution for all the offcore sources however. The Intel® PTU version 3.2 performance tool can display the latency distribution in the data profiling mode and allows sophisticated event filtering capabilities for this event.

Precise Execution Events

There are a wide variety of precise events monitoring other instructions than load and store instructions. Of particular note are the precise branch events that have been added. All branches, near calls and conditional branches can all be counted with precise events, for both retired and mispredicted (and retired) branches of the type selected. For these events, the PEBS buffer will contain the target of the branch. If the Last Branch Record (LBR) is also captured then the location of the branch instruction can also be determined. When the branch is taken the IP value in the PEBS buffer will also appear as the last target in the LBR. If the branch was not taken (conditional branches only) then it won’t

and the branch that was not taken and retired is the instruction before the IP in the PEBS buffer.
In the case of near calls retired, this means that Event Based Sampling (EBS) can be used to collect accurate function call counts. As this is the primary measurement for driving the decision to inline a function, this is an important improvement. In order to measure call counts, you must sample on calls. Any other trigger introduces a bias that cannot be guaranteed to be corrected properly.
The precise branch events are shown in the table below:

Table 6

Event Name	Description	umask	Event
BR_INST_RETIRED.CONDITIONAL	Retired conditional branch instructions	01	C4
BR_INST_RETIRED.NEAR_CALL	Retired near call instructions	02	
BR_INST_RETIRED.ALL_BRANCHES	Retired branch instructions	04	

Shadowing

There is one source of sampling bias associated with precise events. It is due to the time delay between the PMU counter overflow and the arming of the PEBS hardware. During this period events cannot be detected due to the timing shadow. To illustrate the effect consider a function call chain where a long duration function, foo, which calls a chain of 3 very short duration functions, foo1 calling foo2 which calls foo3, followed by a long duration function foo4. If the durations of foo1,foo2 and foo3 are less than the shadow period the distribution of pebs sampled calls will be severely distorted.

1) if the overflow occurs on the call to foo, the pebs mechanism is armed by the time the call to foo1 is executed and samples will be taken showing the call to foo1 from foo.
2) If the overflow occurs due to the call to foo1, foo2 or foo3 however, the PEBS mechanism will not be armed until execution is in the body of foo4. Thus the calls to foo2, foo3 and foo4 cannot appear as pebs sampled calls

Shadowing can effect the distribution of all PEBS events. It will also effect the distribution of basic block execution counts identified by using the combination of a branch retired event (PEBS or not) and the last entry in the LBR. If there were no delay between the PMU counter overflow and the LBR freeze, the last LBR entry could be used to sample taken retired branches and from that the basic block execution counts. All the instructions between the last taken branch and the previous target are executed once. Such a sampling could be used to generate a “software” instruction retired event with uniform sampling, which in turn can be used to identify basic block execution counts. Unfortunately the shadowing causes the branches at the end of short basic blocks to not be the last entry in the LBR, distorting the measurement. Since all the instructions in a basic block are by definition executed the same number of times.

The shadowing effect on call counts and basic block execution counts can be alleviated to a large degree by averaging over the entries in the LBR. This will be discussed in the section on LBRs.

Loop Tripcounts

The available options for optimizing loops are completely constrained by the loop tripcount. For counted loops it is very common for the induction variable to be compared to the tripcount in the termination condition evaluation. This is particularly true if the induction variable is used within the body of the loop, even in the face of heavy optimization. Thus a sequence like the following will appear in the disassembly:

```
addq $0x8, %rcx
cmpq %rax, %rcx
jnge    triad+0x27
```

This was from a heavily optimized triad loop that the compiler had unrolled by 8X. In this case the two registers, rax and rcx are the tripcount and induction variable. If the PEBS buffer is captured for the conditional branches retired event, the average values of the two registers in the compare can be evaluated. The one with the larger average will be the tripcount. Thus the average, RMS, min and max can be evaluated and even a distribution of the recorded values.

Last Branch Record (LBR)

The LBR captures the source and target of each retired taken branch. It keeps 16 source-target sets in a rotating buffer. In Intel® Core™ i7 processor, the types and privilege levels of the branch instructions that are captured can be filtered. This means that the LBR mechanism can be programmed to capture branches occurring at ring0 or ring3 or both (default) privilege levels. Further the types of taken branches that are recorded can also be filtered. The table below lists the filtering options

Table 7

LBR Filter Bit Name	Bit Description	bit
CPL_EQ_0	Exclude ring 0	0
CPL_NEQ_0	Exclude ring3	1
JCC	Exclude taken conditional branches	2
NEAR_REL_CALL	Exclude near relative calls	3
NEAR_INDIRECT_CALL	Exclude near indirect calls	4
NEAR_RET	Exclude near returns	5
NEAR_INDIRECT_JMP	Exclude near unconditional near branches	6
NEAR_REL_JMP	Exclude near unconditional relative branches	7
FAR_BRANCH	Exclude far branches	8

The default is to capture all branches at all privilege levels (all bits zero). Another reasonable programming would set all bits to 1 except bit 1 (capture ring 3) and bit 3 (capture near calls) and bits 6 and 7. This would leave only ring 3 calls and unconditional jumps in the LBR. Such a programming would result in the LBR having the last 16 taken calls and unconditional jumps retired and their targets in the buffer. A PMU sampling driver could then capture this restricted “call chain” with any event, thereby providing a “call tree” context. The inclusion of the unconditional jumps will unfortunately cause problems, particularly when there are if-else structures within loops. In a case where there were particularly frequent function calls at all levels, the inclusion of returns could be added to clarify the context. However this would reduce the call chain depth that could

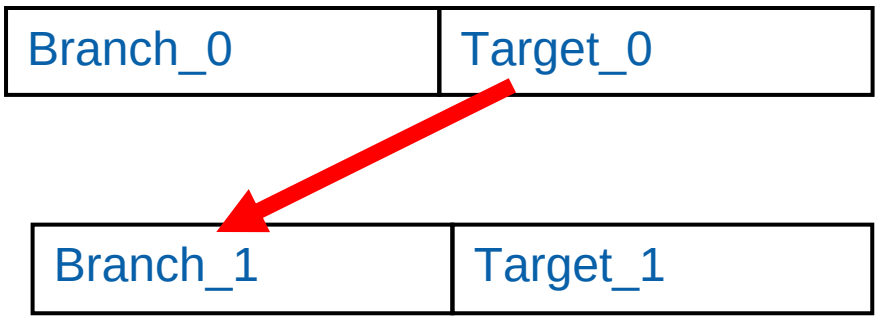
be captured. A fairly obvious usage would be to trigger the sampling on extremely long latency loads, to enrich the sample with accesses to heavily contended locked variables, and then capture the call chain to identify the context of the lock usage.

Call Counts and Function Arguments

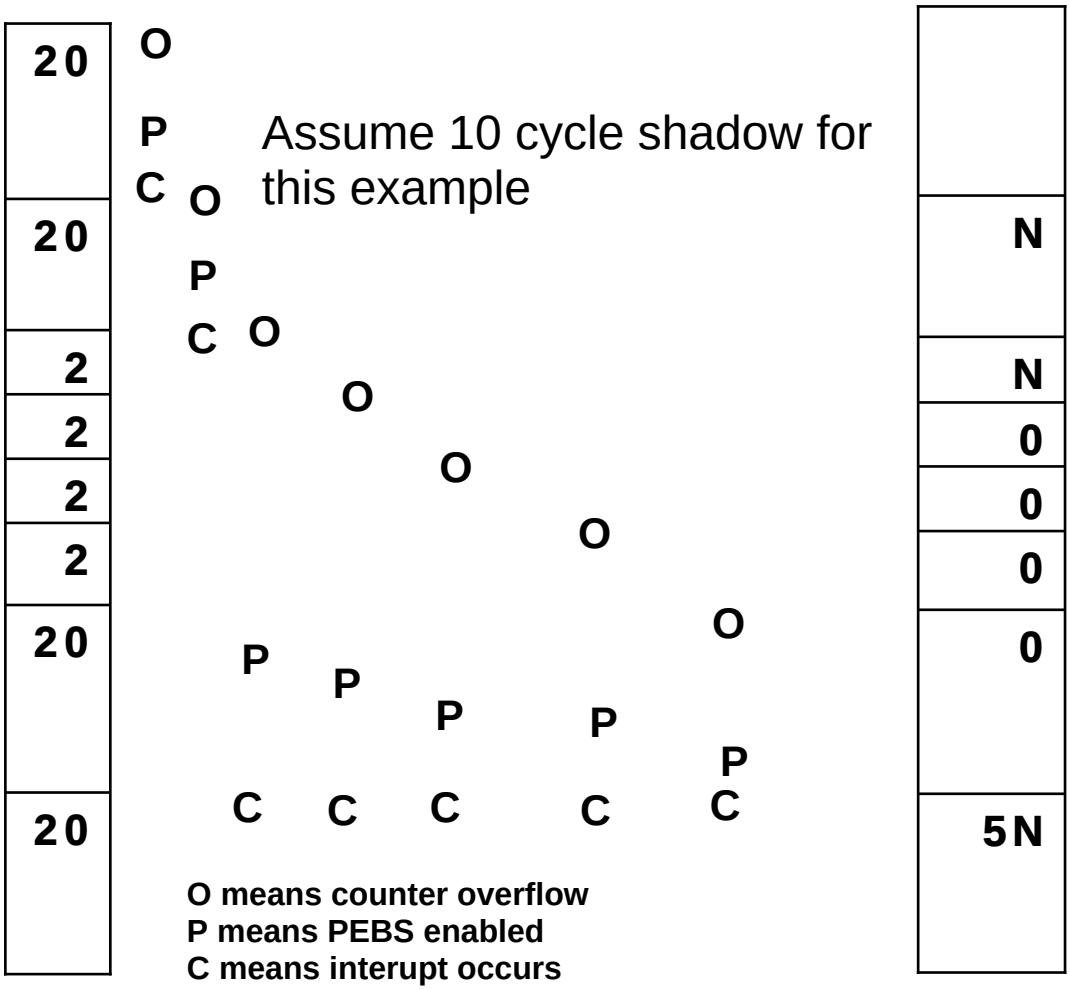
If the LBRs are captured for PMIs triggered by the BR_INST_RETIRED.NEAR_CALL event, then the call count per calling function can be determined by simply using the last entry in LBR.As the PEBS IP will equal the last target IP in the LBR, it is the entry point of the calling function. Similarly, the last source in the LNR buffer was the call site from within the calling function. If the full PEBS record is captured as well, then for functions with limited numbers of arguments on Intel64 OS's, you can sample both the call counts and the function arguments.

LBRs and Basic Block Execution Counts

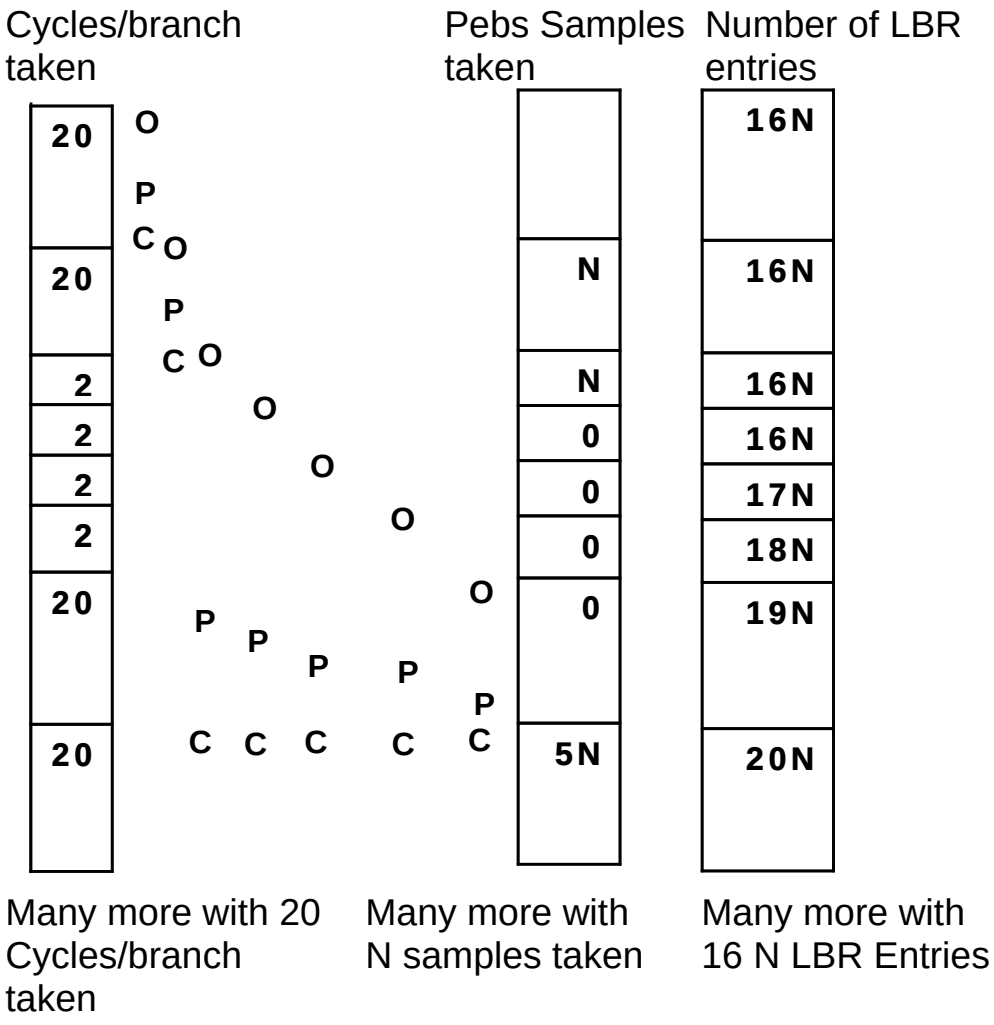
Another interesting usage is to use the BR_INST_RETIRED.ALL_BRANCHES event and the LBRs with no filter to evaluate the execution rate of basic blocks. As the LBRs capture all taken branches, all the basic blocks between a branch IP (source) and the previous target in the LBR buffer were executed one time. Thus a simple way to evaluate the basic block execution counts for a given load module is to make a map of the starting locations of every basic block. Then for each sample triggered by the PEBS collection of BR_INST_RETIRED.ALL_BRANCHES, starting from the PEBS address (a target but perhaps for a not taken branch and thus not necessarily in the LBR buffer) and walking backwards through the LBRs until finding an address not corresponding to the load module of interest, count all the basic blocks that were executed. Calling this value "number_of_basic_blocks", increment the execution counts for all of those blocks by 1/(number_of_basic_blocks). This technique also yields the taken and not taken rates for the active branches. All branch instructions between a source IP and the previous target IP (within the same module) were not taken, while the branches listed in the LBR were taken. This is illustrated in the graphics below



All instructions between Target_0 and Branch_1 are retired 1 time
All Basic Blocks between Target_0 and Branch_1 are executed 1 time
All Branch Instructions between Target_0 and Branch_1 are not taken



Assume every branch is taken, and average over the basic blocks in the LBR trajectory results in:



This illustrates a situation where some basic blocks would appear to never get samples and some have many times too many. Weighting each entry by 1/(num of basic blocks in the LBR trajectory), in this example would result in dividing the numbers in the right most table by 16. Thus we end up with far more accurate execution counts ((1.25-> 1.0) * N) in all of the basic blocks, even those that never directly caused a PEBS sample.

As on Intel® Core™2 processors there is a precise instructions retired event that can be used in a wide variety of ways. In addition there are precise events for uops_retired, various SSE instruction classes, FP assists. It should be noted that the FP assist events only detect x87 FP assists, not those involving SSE FP instructions. Detecting all assists will be discussed in the section on the pipeline Front End.
The instructions retired event has a few special uses. While its distribution is not uniform, the totals are correct. If the values recorded for all the instructions in a basic block are averaged, a measure of the basic block execution count can be extracted. The ratios of basic block executions can be used to estimate loop tripcounts when the counted loop technique discussed above cannot be applied.
The PEBS version (general counter) instructions retired event can further be used to profile OS execution accurately even in the face of STI/CLI semantics, because the pebs

buffer retains the IP value of the OS code section where the overflow (+1) occurred. The interrupt then occurs after the critical section has completed, but the data was frozen correctly. If the cmask value is set to some very high value and the invert condition is applied, the result is always true, and the event will count core cycles (halted + unhalted). Consequently both cycles and instructions retired can be accurately profiled. The UOPS_RETIRED.ANY event, which is also precise can also be used to profile Ring 0 execution and really gives a more accurate display of execution.

The events are shown in the table below:

Table 8

Event Name	Description	Umask	Event	Cmask	Inv	Edge
INST_RETIRED.ANY_P	Instructions retired (general counter)	01	C0	0	0	0
INST_RETIRED.TOTAL_CYCLES	Total cycles	01	C0	10	1	0
INST_RETIRED.TOTAL_CYCLES_R0	Total cycles in ring 0	01	C0	10	1	0
INST_RETIRED.TOTAL_CYCLES_R3	Total cycles in ring 3	01	C0	10	1	0
INST_RETIRED.X87	Retired floating-point operations	02	C0	0	0	0
INST_RETIRED.MMX	Retired MMX instructions	04	C0	0	0	0
UOPS_RETIRED.ACTIVE_CYCLES	Cycles Uops are being retired	01	C2	1	0	0
UOPS_RETIRED.ANY	Uops retired	01	C2	0	0	0
UOPS_RETIRED.STALL_CYCLES	Cycles Uops are not retiring	01	C2	1	1	0
UOPS_RETIRED.RETIRE_SLOTS	Retirement slots used	02	C2	0	0	0
UOPS_RETIRED.MACRO_FUSED	Macro-fused Uops retired	04	C2	0	0	0
SSEX_UOPS_RETIRED.PACKED_SINGLE	SIMD Packed-Single Uops retired	01	C7	0	0	0
SSEX_UOPS_RETIRED.SCALAR_SINGLE	SIMD Scalar-Single Uops retired	02	C7	0	0	0
SSEX_UOPS_RETIRED.PACKED_DOUBLE	SIMD Packed-Double Uops retired	04	C7	0	0	0
SSEX_UOPS_RETIRED.SCALAR_DOUBLE	SIMD Scalar-Double Uops retired	08	C7	0	0	0
SSEX_UOPS_RETIRED.VECTOR_INTEGER	SIMD Vector Integer Uops retired	10	C7	0	0	0
FP_ASSIST.ANY	X87 Floating point assists	01	F7	0	0	0
FP_ASSIST.OUTPUT	X87 FP assist on input values	02	F7	0	0	0
FP_ASSIST.INPUT	X87 FP assist on output values	04	F7	0	0	0

Non-PEBS Core Memory Access Events

On the Intel® Core™ i7 processor the mid level cache (L2 CACHE) misses and traffic with the uncore and beyond can be measured with a large variety of metrics. Besides the PEBS events discussed earlier which monitor loads there are a wide variety of regular/non-PEBS events as well.
A special event to decompose the L2 CACHE misses by data source exists among the Intel® Core™ i7 processor core events. It is organized as a matrix of request type by response type. Multiple requests and responses can be selected and if any of the requests and any of the responses are satisfied by a particular L2 CACHE miss then the event increments. A list of request and response types are shown below.

Table 9

	Bit position	Description
Request	0	Demand Data Rd = DCU reads (includes partials, DCU Prefetch)
Type	1	Demand RFO = DCU RFOs
	2	Demand Ifetch = IFU Fetches
	3	Writeback = L2 CACHE_EVICT/DCUWB
	4	PF Data Rd = MPL Reads
	5	PF RFO = MPL RFOs
	6	PF Ifetch = MPL Fetches
	7	OTHER
Response	8	L3 CACHE_HIT_UNCORE_HIT
Type	9	L3 CACHE_HIT_OTHER_CORE_HIT_SNP
	10	L3 CACHE_HIT_OTHER_CORE_HITM
	11	L3 CACHE_MISS_REMOTE_HIT_SCRUB
	12	L3 CACHE_MISS_REMOTE_FWD
	13	L3 CACHE_MISS_REMOTE_DRAM
	14	L3 CACHE_MISS_LOCAL_DRAM
	15	IO_CSR_MMIO

The request type “other”, selected by enabling bit 7 includes things like non temporal SSE stores. The three L3 CACHE hit options correspond to an unshared line, a shared clean line and a shared line that is modified in one of the other cores. The L3 CACHE_miss_remote options correspond to lines that must have the cores snooped (ie used by multiple cores) or clean lines that are used by only one core and can be safely forwarded directly from the remote L3 CACHE.
Due to the number of bits required to program the matrix selection, a dedicated MSR (1A6) is used. However, the global nature of the MSR results in only one version of the event being able to be collected during a given collection period. The Intel Performance Tools (the VTune™ Performance Analyzer and the Intel Performance Tuning Utility) constrain this by requiring the event be programmed into counter 2.

In order to make data collection maximally efficient a large set of predefined bit combinations were included in the default event lists to minimize the number of data collection runs needed. The predefined combinations for requests are shown below

Multiple request type and response type bits can be set (there are approximately 65K non-zero programmings of the event possible) to allow data collection with the minimum number of data collection runs. The predefined combinations used by the Intel Performance Tools are shown below. The event names are constructed as OFFCORE_RESPONSE_0.REQUEST.RESPONSE. Where the REQUEST and RESPONSE strings are defined to correspond to unique programmings of the lower 8 bits or the upper 8 bits of the 16 bit field. The *DEMAND* events discussed in this document also include any requests made by the L1D cache hardware prefetchers.

Request Type	MSR Encoding
ANY_DATA	xx11
ANY_IFETCH	xx44
ANY_REQUEST	xxFF
ANY_RFO	xx22
COREWB	xx08
DATA_IFETCH	xx77
DATA_IN	xx33
DEMAND_DATA	xx03
DEMAND_DATA_RD	xx01
DEMAND_IFETCH	xx04
DEMAND_RFO	xx02
OTHER	xx80
PF_DATA	xx30
PF_DATA_RD	xx10
PF_IFETCH	xx40
PF_RFO	xx20
PREFETCH	xx70

Response Type	MSR Encoding
ANY_CACHE_DRAM	7Fxx
ANY_DRAM	60xx
ANY_LLC_MISS	F8xx
ANY_LOCATION	FFxx
IO_CSR_MMIO	80xx
LLC_HIT_NO_OTHER_CORE	01xx
LLC_OTHER_CORE_HIT	02xx
LLC_OTHER_CORE_HITM	04xx
LOCAL_CACHE	07xx
LOCAL_CACHE_DRAM	47xx
LOCAL_DRAM	40xx
REMOTE_CACHE	18xx
REMOTE_CACHE_DRAM	38xx
REMOTE_CACHE_HIT	10xx
REMOTE_CACHE_HITM	08xx
REMOTE_DRAM	20xx

Errata:

Non temporal stores to locally homed cachelines would be thought to increment the offcore_response_0 event when the MSR was set to a value of 0x4080 (other.local_dram). These NT writes in fact increment the event when the MSR is programmed to 0x280 (other.llc_other_core_hit). This can make the analysis of total traffic to dram a bit clumsy.

Cacheline requests to the uncore are staged in the Super Queue
There are two events that monitor when the super queue is full and when it is full and more critically the cycles when it tries to allocate a slot but cannot.

Table 10

Event Name	Definition	Umask	Event
OFFCORE_REQUESTS_BUFFER_FULL	Offcore request queue (SQ) full	01	B2
SQ_STALL	Cycles SQ allocation stalled, SQ full	01	F6

Bandwidth per core

Measuring the bandwidth for an individual core is complicated on Intel® Core™ i7 processors. The problem is the writebacks/evictions from the L2 CACHE and to some degree the non temporal SSE writes. The eviction of modified lines from the L2 CACHE causes a write of the line back to the L3 CACHE. The line is only written to memory when it is evicted from the L3 CACHE some time later (if at all). The writebacks to memory due to eviction of modified lines from the L3 CACHE cannot be associated with an individual core. The net result of this is that the total write bandwidth for all the cores can be measured with events in the uncore. The read bandwidth and the SSE non-temporal write bandwidth can be measured on a per core basis. Further due to the intrinsically NUMA nature of memory bandwidth, the measurement for those 2 components has to be divided into bandwidths for the core on a per target socket basis. It can be measured for the loads with the events

OFFCORE_RESPONSE_0.DATA_IFETCH.LLC_MISS_LOCAL_DRAM
OFFCORE_RESPONSE_0.DATA_IFETCH.LLC_MISS_REMOTE_DRAM

And for both target sockets with

OFFCORE_RESPONSE_0.DATA_IFETCH.ANY_DRAM

While the SSE stores would be measured with

OFFCORE_RESPONSE_0.OTHER.LLC_MISS_LOCAL_CACHE_DRAM
OFFCORE_RESPONSE_0.OTHER.LLC_MISS_REMOTE_DRAM

And for both sockets with

OFFCORE_RESPONSE_0.OTHER.ANY_CACHE_DRAM

The use of the “CACHE_DRAM” encodings is to work around the errata mentioned earlier in this section. Non temporal stores of course write to dram.

But of course, none of the above includes the bandwidth associated with writebacks of modified cacheable lines.

If you are only concerned with data then the events
OFFCORE_RESPONSE_0.DATA_IN.LLC_MISS_LOCAL_DRAM
OFFCORE_RESPONSE_0.DATA_IN.LLC_MISS_REMOTE_DRAM
May prove the most useful, along with
OFFCORE_RESPONSE_0.DATA_IN.LLC_MISS
Which would include the lines forwarded from the remote cache along with the two dram sources just above.

L1D, L2 Cache Access and More Offcore events

There is a certain redundancy in some of the offcore/L2 CACHE miss events. The events discussed below are of a secondary nature and are discussed here mostly for completeness. The most obvious use may be to help decompose the impact of the hardware prefetchers, how often their requests hit in the L2 CACHE, how frequently they evict modified lines and so on.

In addition to the OFFCORE_RESPONSE_0 event and the precise events that will be discussed later, there are several other events that can be used as well. These can be used to supplement the offcore_response_0 events, which can only be programmed in a one/run manner, to minimize data collection time in some cases. L2 CACHE misses can also be counted with the architecturally defined event LONGEST_LAT_CACHE_ACCESS, however as this event also includes requests due to the L1D and L2 CACHE hardware prefetchers its utility may be limited. Some of the L2 CACHE access events can be used for both breaking down L2 CACHE accesses and L2 CACHE misses by type, in addition to the OFFCORE_REQUESTS events discussed earlier. The L2_RQSTS and L2_DATA_RQSTS events are listed below with the assorted umask values that can be used to discern the assorted access types. In all of the L2 CACHE access events the designation PREFETCH only refers to the L2 CACHE hardware prefetch. The designation DEMAND includes loads, stores, SW prefetches and requests due to the L1D hardware prefetchers.

Table 11

Event Name	Definition	Umask	Event
L2_RQSTS.LD_HIT	Load requests that hit the L2	1	24
L2_RQSTS.LD_MISS	Load requests that miss the L2	2	
L2_RQSTS.LOADS	All L2 load requests	3	
L2_RQSTS.RFO_HIT	Store RFO requests that hit the L2	4	
L2_RQSTS.RFO_MISS	Store RFO requests that miss the L2	8	
L2_RQSTS.IFETCH_HIT	Code requests that hit the L2	10	
L2_RQSTS.IFETCH_MISS	Code requests that miss the L2	20	
L2_RQSTS.IFETCHES	All L2 code requests	30	
L2_RQSTS.PREFETCH_HIT	Prefetch requests that hit the L2	40	
L2_RQSTS.PREFETCH_MISS	Prefetch requests that miss the L2	80	
L2_RQSTS.RFOS	All L2 store RFO requests	0C	
L2_RQSTS.MISS	All L2 misses	AA	
L2_RQSTS.PREFETCHES	All L2 prefetches	C0	

L2_RQSTS.REFERENCES	All L2 requests	FF	
L2_DATA_RQSTS.DEMAND.I_STATE	L2 data demand in I state (misses)	1	26
L2_DATA_RQSTS.DEMAND.S_STATE	L2 data demand in S state	2	
L2_DATA_RQSTS.DEMAND.E_STATE	L2 data demand in E state	4	
L2_DATA_RQSTS.DEMAND.M_STATE	L2 data demand in M state	8	
L2_DATA_RQSTS.PREFETCH.I_STATE	L2 data prefetches in the I state (misses)	10	
L2_DATA_RQSTS.PREFETCH.S_STATE	L2 data prefetches in the S state	20	
L2_DATA_RQSTS.PREFETCH.E_STATE	L2 data prefetches in E state	40	
L2_DATA_RQSTS.PREFETCH.M_STATE	L2 data prefetches in M state	80	
L2_DATA_RQSTS.DEMAND.MESI	L2 data demand requests	0F	
L2_DATA_RQSTS.PREFETCH.MESI	All L2 data prefetches	F0	
L2_DATA_RQSTS.ANY	All L2 data references	FF	

The L2_LINES_IN and L2_LINES_OUT events have been decomposed slightly differently than on Intel® Core™2 processors as can be seen in the table below. The L2_LINES_OUT event can now be used to decompose the evicted lines by clean and dirty (ie a Writeback) and whether they were evicted by an L1D request or an L2 CACHE HW prefetch.

Table 12

Event Name	Definition	Umask	Event
L2_LINES_IN.S_STATE	L2 lines allocated in the S state	2	F1
L2_LINES_IN.E_STATE	L2 lines allocated in the E state	4	
L2_LINES_IN.ANY	Lines allocated in the L2 cache	7	
L2_LINES_OUT.ANY	All L2 lines evicted	0F	F2
L2_LINES_OUT.DEMAND_CLEAN	L2 lines evicted by a demand request	1	
L2_LINES_OUT.DEMAND_DIRTY	L2 modified lines evicted by a demand request	2	
L2_LINES_OUT.PREFETCH_CLEAN	L2 lines evicted by a prefetch request	4	
L2_LINES_OUT.PREFETCH_DIRTY	L2 modified lines evicted by a prefetch request	8	

The event L2_TRANSACTIONS counts all interactions with the L2 CACHE and is divided up as follows

Table 13

Event Name	Definition	Umask	Event
L2_TRANSACTIONS.LOAD	Load, SW prefetch and L1D prefetcher requests	1	F0
L2_TRANSACTIONS.RFO	RFO requests from L1D	2	
L2_TRANSACTIONS.IFETCH	Cachelines requested from L1I	4	
L2_TRANSACTIONS.PREFETCH	L2 Hw prefetches, includes L2 hits and Misses	8	
L2_TRANSACTIONS.L1D_WB	Writebacks from L1D	10	
L2_TRANSACTIONS.FILL	Cachelines brought in from L3 CACHE	20	
L2_TRANSACTIONS.WB	Writebacks to the L3 CACHE	40	
L2_TRANSACTIONS.ANY	All actions taken by the L2	80	

Writes and locked writes are counted with a combined event.

Table 14

Event Name	Definition	Umask	Event
L2_WRITE.RFO.I_STATE	L2 store RFOs in I state (misses)	1	27
L2_WRITE.RFO.S_STATE	L2 store RFOs in S state	2	
L2_WRITE.RFO.E_STATE	L2 store RFOs in E state	4	
L2_WRITE.RFO.M_STATE	L2 store RFOs in M state	8	
L2_WRITE.LOCK.I_STATE	L2 lock RFOs in I state (misses)	10	
L2_WRITE.LOCK.S_STATE	L2 lock RFOs in S state	20	
L2_WRITE.LOCK.E_STATE	L2 lock RFOs in E state	40	
L2_WRITE.LOCK.M_STATE	L2 lock RFOs in M state	80	
L2_WRITE.RFO.HIT	All L2 store RFOs that hit the cache	0E	
L2_WRITE.RFO.MESI	All L2 store RFOs	0F	
L2_WRITE.LOCK.HIT	All L2 lock RFOs that hit the cache	E0	
L2_WRITE.LOCK.MESI	All L2 lock RFOs	F0	

The next largest set of penalties is associated with the TLBs and accessing more physical pages than can be mapped with their finite number of entries. A miss in the first level TLBs results in a very small penalty that can usually be hidden by the OOO execution and compiler's scheduling. A miss in the shared TLB results in the Page Walker being invoked and this penalty can be noticeable in the execution.
The (non pebs) TLB miss events break down into three sets: DTLB misses, Load DTLB misses and ITLB misses. Store DTLB misses can be evaluated from the difference of the DTLB misses and the Load DTLB misses. Each then has a set of sub events programmed with the umask value. A summary of the non PEBS TLB miss events is in the table below.

Table 15

Event Name	Definition	Umask	Event
DTLB_LOAD_MISSES.ANY	DTLB load miss	1	8
DTLB_LOAD_MISSES. WALK_COMPLETED	DTLB load miss page walks	2	
DTLB_LOAD_MISSES. PMH_BUSY_CYCLES	Page walk blocked due to PMH busy	8	
DTLB_LOAD_MISSES.STLB_HIT	DTLB first level load miss but second level hit	10	
DTLB_LOAD_MISSES.PDE_MISS	DTLB load miss caused by low part of address	20	
DTLB_LOAD_MISSES.PDP_MISS	DTLB load miss caused by high part of address	40	
DTLB_LOAD_MISSES. LARGE_WALK_COMPLETED	DTLB load miss large page walks	80	
DTLB_MISSES.ANY	DTLB miss	1	49
DTLB_MISSES. WALK_COMPLETED	DTLB miss page walks	2	
DTLB_MISSES. PMH_BUSY_CYCLES	Page walk blocked due to PMH busy	8	
DTLB_MISSES.STLB_HIT	DTLB first level miss but second level hit	10	
DTLB_MISSES.PDE_MISS	DTLB miss caused	20	

	by low part of address		
DTLB_MISSES.PDP_MISS	DTLB miss caused by high part of address	40	
DTLB_MISSES. LARGE_WALK_COMPLETED	DTLB miss large page walks	80	
ITLB_MISSES.ANY	ITLB miss	1	85
ITLB_MISSES. WALK_COMPLETED	ITLB miss page walks	2	
ITLB_MISSES. PMH_BUSY_CYCLES	Page walk blocked due to PMH busy	8	
ITLB_MISSES.STLB_HIT	ITLB first level miss but second level hit	10	
ITLB_MISSES.PDE_MISS	ITLB miss caused by low part of address	20	
ITLB_MISSES.PDP_MISS	ITLB miss caused by high part of address	40	
ITLB_MISSES. LARGE_WALK_COMPLETED	ITLB miss large page walks	80	

The L1 data cache, L1D, is the final component to be discussed. These events can only be counted with the first 2 of the 4 general counters. Most of the events are self explanatory. The total number of references to the L1D can be counted with L1D_ALL_REF, either just cacheable references or all. The cacheable references can be divided into loads and stores with L1D_CACHE_LOAD and L1D_CACHE.STORE. These events are further subdivided by MESI states through their umask values, with the I state references indicating the cache misses.
The evictions of modified lines in the L1D result in writebacks to the L2 CACHE. These are counted with the L1D_WB_L2 events. The umask values break these down by the MESI state of the version of the line in the L2 CACHE.
The locked references can be counted also with the L1D_CACHE_LOCK events. Again these are broken down by MES states for the lines in L1D.
The total number of lines brought into L1D, the number that arrived in an M state and the number of modified lines that get evicted due to receiving a snoop are counted with the L1D event and its umask variations.
NOTE: many of these events are known to overcount (l1d_cache_ld, l1d_cache_lock) so they can only be used for qualitative analysis.
These events and a few others are summarized below.

Table 16

Event Name	Definition	Umask	Event
L1D_WB_L2.I_STATE	L1 writebacks to L2 in I state (misses)	01	28
L1D_WB_L2.S_STATE	L1 writebacks to L2 in S state	02	28
L1D_WB_L2.E_STATE	L1 writebacks to L2 in E state	04	28
L1D_WB_L2.M_STATE	L1 writebacks to L2 in M state	08	28
L1D_WB_L2.MESI	All L1 writebacks to L2	0F	28
L1D_CACHE_LD.I_STATE	L1 data cache read in I state (misses)	01	40
L1D_CACHE_LD.S_STATE	L1 data cache read in S state	02	40
L1D_CACHE_LD.E_STATE	L1 data cache read in E state	04	40

L1D_CACHE_LD.M_STATE	L1 data cache read in M state	08	40
L1D_CACHE_LD.MESI	L1 data cache reads	0F	40
L1D_CACHE_ST.I_STATE	L1 data cache stores in I state	01	41
L1D_CACHE_ST.S_STATE	L1 data cache stores in S state	02	41
L1D_CACHE_ST.E_STATE	L1 data cache stores in E state	04	41
L1D_CACHE_ST.M_STATE	L1 data cache stores in M state	08	41
L1D_CACHE_ST.MESI	All L1 data cache stores	0F	41
L1D_CACHE_LOCK.HIT	L1 data cache load lock hits	01	42
L1D_CACHE_LOCK.S_STATE	L1 data cache load locks in S state	02	42
L1D_CACHE_LOCK.E_STATE	L1 data cache load locks in E state	04	42
L1D_CACHE_LOCK.M_STATE	L1 data cache load locks in M state	08	42
L1D_ALL_REF.ANY	All references to the L1 data cache	01	43
L1D_ALL_REF.CACHEABLE	L1 data cacheable reads and writes	02	43
L1D_PEND_MISS.PENDING	Outstanding L1 data cache misses at any cycle	01	48
L1D_PEND_MISS.LOAD_BUFFERS_FULL	L1 data cache load fill buffer full	02	48
L1D.REPL	L1 data cache lines allocated	01	51
L1D.M_REPL	L1D cache lines allocated in the M state	02	51
L1D.M_SNOOP_EVICT	L1D snoop eviction of cache lines in M state	08	51
L1D_CACHE_PREFETCH_LOCK_FB_HIT	L1D prefetch load lock accepted in fill buffer	01	52
L1D_CACHE_LOCK_FB_HIT	L1D load lock accepted in fill buffer	01	53
L1I.HITS	L1I instruction fetch hits	01	80
L1I.MISSES	L1I instruction fetch misses	02	80
L1I.READS	L1I Instruction fetches	03	80
L1I.CYCLES_STALLED	L1I instruction fetch stall cycles	04	80
L1I_OPPORTUNISTIC_HITS	Opportunistic hits in the streaming buffer	01	83
L1D_PREFETCH.REQUESTS	L1D hardware prefetch requests	01	4E
L1D_PREFETCH.MISS	L1D hardware prefetch misses	02	4E
L1D_PREFETCH.TRIGGERS	L1D hardware prefetch requests triggered by FSM	04	4E

Store Forwarding

There are few cases of loads not being able to forward from active store buffers in Intel® Core™ i7 processors. The predominant remaining case has to do with larger loads overlapping smaller stores. There is not event that detects when this occurs. There is also a “false store forwarding” case where the addresses only match in the lower 12 address bits. This is sometimes referred to as 4K aliasing. This can be detected with the following event

Table 17

Event Name	Description	Event Code	Umask
PARTIAL_ADDRESS_ALIAS	False dependencies due to partial address aliasing	7	1

Front End Events

Branch misprediction effects can sometimes be reduced through code changes and enhanced inlining. Most other front end performance limitations have to be dealt with by the code generation. The analysis of such issues is mostly of use by compiler developers.

Branch Mispredictions

As discussed earlier there is good coverage of branch mispredictions with precise events. These are enhanced by use of the LBR to identify the branch location to go along with the target location captured in the PEBS buffer. It is not clear to the author that there is more information to be garnered from the front end events that can be used by code developers, but they are certainly of use to chip designers and architects. These events are listed below.

Table 18

Event Name	Description	Event Code	Umask
BACLEAR.CLEAR	BAclears asserted, regardless of cause	E6	1
BACLEAR.BAD_TARGET	BACLEAR asserted with bad target address	E6	2
BPU_CLEARS.EARLY	Early Branch Prediciton Unit clears	E8	1
BPU_CLEARS.LATE	Late Branch Prediction Unit clears	E8	2
BPU_MISSED_CALL_RET	Branch prediction unit missed call or return	E5	1
BR_INST_DECODED	Branch instructions decoded	E0	1
BR_INST_EXEC.COND	Conditional branch instructions executed	88	1
BR_INST_EXEC.DIRECT	Unconditional branches executed	88	2
BR_INST_EXEC.INDIRECT_NON_CALL	Indirect non call branches executed	88	4
BR_INST_EXEC.RETURN_NEAR	Indirect return branches executed	88	8
BR_INST_EXEC.DIRECT_NEAR_CALL	Unconditional call branches executed	88	10
BR_INST_EXEC.INDIRECT_NEAR_CALL	Indirect call branches executed	88	20
BR_INST_EXEC.TAKEN	Taken branches executed	88	40
BR_INST_EXEC.ANY	Branch instructions executed	88	7F
BR_INST_EXEC.NON_CALLS	All non call branches executed	88	3
BR_INST_EXEC.NEAR_CALLS	Call branches executed	88	30
BR_MISP_EXEC.COND	Mispredicted conditional branches executed	89	1
BR_MISP_EXEC.DIRECT	Mispredicted unconditional branches executed	89	2
BR_MISP_EXEC.INDIRECT_NON_CALL	Mispredicted indirect non call branches executed	89	4
BR_MISP_EXEC.RETURN_NEAR	Mispredicted return branches executed	89	8
BR_MISP_EXEC.DIRECT_NEAR_CALL	Mispredicted non call branches executed	89	10
BR_MISP_EXEC.INDIRECT_NEAR_CALL	Mispredicted indirect call branches executed	89	20
BR_MISP_EXEC.TAKEN	Mispredicted taken branches executed	89	40
BR_MISP_EXEC.ANY	Mispredicted branches executed	89	7F
BR_MISP_EXEC.NON_CALLS	Mispredicted non call branches executed	89	3
BR_MISP_EXEC.NEAR_CALLS	Mispredicted call branches executed	89	30

Branch mispredictions are not in and of themselves an indication of a performance bottleneck. They have to be associated with dispatch stalls and the instruction starvation condition, UOPS_ISSUED:C1:I1 – RESOURCE_STALLS.ANY. Such stalls are likely to be associated with icache misses and ITLB misses. The precise ITLB miss event can be useful for such issues. The icache and ITLB miss events are listed below

Table 19

Event Name	Description	Event Code	Umask
L1I.HITS	L1I instruction fetch hits	80	1
L1I.MISSES	L1I instruction fetch misses	80	2
L1I.CYCLES_STALLED	L1I instruction fetch stall cycles	80	4
L1I.READS	L1I Instruction fetches	80	3
IFU_IVC.FULL	victim cache full	81	1
IFU_IVC.L1I_EVICTION	L1I eviction	81	2
ITLB_FLUSH	ITLB flushes	AE	1
ITLB_MISSES.ANY	ITLB miss	85	1
ITLB_MISSES.WALK_COMPLETED	ITLB miss page walks	85	2
ITLB_MISSES.LARGE_WALK_COMPLETED	ITLB miss large page walks	85	80
LARGE_TLB.HIT	large TLB hit	82	1

FE Code Generation Metrics

The remaining front end events are mostly of use in identifying when details of the code generation interact poorly with the instructions decoding and uop issue to the OOO engine. Examples are length changing prefix issues associated with the use of 16 bit immediates, rob read port stalls, instruction alignment interfering with the loop detection and instruction decoding bandwidth limitations. The activity of the LSD is monitored using CMASK values on a signal monitoring activity. Some of these events are listed below:

Table 20

Event Name	Description	Event Code	Umask
ILD_STALL.LCP	Length Change Prefix stall cycles	87	1
ILD_STALL.MRU	MRU stall cycles	87	2
ILD_STALL.IQ_FULL	Instruction Queue full stall cycles	87	4
ILD_STALL.REGEN	Regen stall cycles	87	8
ILD_STALL.ANY	Any Instruction Length Decoder stall cycles	87	0F
INST_DECODED.DEC0	Instructions that must be decoded by decoder 0	18	1
INST_QUEUE_ WRITE_CYCLES	Cycles instructions are written to the instruction queue	1E	1
INST_QUEUE_WRITES	Instructions written to instruction queue.	17	1
LSD.ACTIVE	Cycles when uops were delivered by LSD	A8	1
LSD.INACTIVE	Cycles no Uops delivered by the LSD	A8	1
LSD_OVERFLOW	Loops that can't stream from the instruction queue	20	1
LSD_REPLAY.ABORT	Loops aborted by the LSD	1F	2
LSD_REPLAY.COUNT	Loops replayed by the LSD	1F	1
MACRO_INSTS.DECODED	Instructions decoded	D0	1
MACRO_INSTS. FUSIONS_DECODED	Macro-fused instructions decoded	A6	1
RAT_STALLS.ANY	All RAT stall cycles	D2	0F
RAT_STALLS.FLAGS	Flag stall cycles	D2	1

RAT_STALLS.REGISTERS	Partial register stall cycles	D2	2
RAT_STALLS. ROB_READ_PORT	ROB read port stalls cycles	D2	4
RAT_STALLS. SCOREBOARD	Scoreboard stall cycles	D2	8

Microcode and Exceptions

The invocation of microcode can significantly effect the retirement of instructions. Most instructions are decoded to a single uop. There are a few that can decode to many uops, for example fsincos, fptan or rep mov. When long strings of uops are required, they are inserted into the pipeline by the microcode sequencer. This can be monitored with the UOPS_DECODED.MS_CYCLES_ACTIVE event. Which uses the base event, with the cmask set to 1, to count the cycles the microcode sequencer is active. Consequently regions of execution where this is significant can be easily identified.

Another more nefarious source is due to FP assists, like the processing of denormalized FP values or QNaNs. In such cases the penalty is essentially the uops required for the assist plus the pipeline clearing required to ensure the correct state. Consequently there is a very clear signature consisting of MACHINE_CLEAR.CYCLES and uops being inserted by the microcode sequencer, UOPS_DECODED.MS_CYCLES_ACTIVE. The execution penalty being approximately the sum of these two.

Table 21

Event Name	Description	Event Code	Umask
UOPS_DECODED.MS	Uops decoded by Microcode Sequencer	D1	2
MACHINE_CLEARS.CYCLES	Cycles Machine Clears Asserted	C3	1

Uncore Performance Events

The performance monitoring of the uncore is mostly accomplished through the uncore PMU and the events it monitors. As stated earlier the PMU has 8 general counters and a fixed counter that monitors the uncore's unhalted clock, which runs at a different frequency than the core. The uncore cannot by itself generate an interrupt. The cores do that. When an uncore counter overflows, a bit pattern is used to determine which cores should be signaled to raise a PMI. The uncore PMU is unaware of the core, PID or TID that caused the event that overflowed a counter. Consequently the most reasonable approach for sampling on uncore events is to raise a PMI on all the logical cores in the package.

There are a wide variety of events that monitor queue occupancies and inserts. There are others that count cacheline transfers, dram paging policy statistics, snoop types and responses, and so on. The uncore is the only place the total bandwidth to memory can be measured. This will be discussed explicitly after all the uncore components and their events are described.

The Global Queue

L2 CACHE misses and writebacks from all the cores of a package result in requests being sent to the uncore's Global Queue (GQ). There are 3 "trackers" in the GQ that are the queues for on package read and writeback requests and requests that arrive from a "peer", meaning anything coming from the Intel® QuickPath Interconnect. As mentioned before these have 32, 16 and 12 entries respectively. The occupancies, inserts, cycles full and cycles not empty for all three trackers can be monitored. Further as load requests go through a series of stages the occupancy and inserts associated with the stages can also be monitored, enabling a "cycle accounting" breakdown of the uncore memory accesses due to loads.

When a counter is first programmed to monitor a queue occupancy, for any of the uncore queues, the queue must first be emptied. This is accomplished by the driver issuing a bus lock. This only needs to be done when the counter is first programmed. From that point on the counter will correctly reflect the state of the queue, so it can be repeatedly sampled for example without another bus lock being issued.

The following events monitor the 3 GQ trackers, with RT signifying the read tracker, WT, the write tracker and PPT the peer probe tracker.

Table 22

Event Name	Definition	Umask	Event
UNC_GQ_TRACKER_OCCUP.RT	GQ read tracker occupancy	01	02
UNC_GQ_TRACKER_OCCUP.RT_LLC_MISS	GQ read tracker L3 CACHE miss occupancy	02	
UNC_GQ_TRACKER_OCCUP.RT_TO_LLC_RESP	GQ read tracker L3 CACHE request occupancy	04	
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACQUIRED	GQ read tracker from no RTID to RTID acquired occupancy	08	
UNC_GQ_TRACKER_OCCUP.WT_TO_RTID_ACQUIRED	GQ write tracker from no RTID to RTID acquired occupancy	10	
UNC_GQ_TRACKER_OCCUP.WT	GQ write tracker alloc to deallocate occupancy	20	
UNC_GQ_TRACKER_OCCUP.PPT	GQ peer probe tracker alloc to deallocate occupancy	40	
UNC_GQ_ALLOC.RT	GQ read tracker requests	01	03
UNC_GQ_ALLOC.RT_LLC_MISS	GQ read tracker L3 CACHE misses	02	
UNC_GQ_ALLOC.RT_TO_LLC_RESP	GQ read tracker L3 CACHE requests	04	
UNC_GQ_ALLOC.RT_TO_RTID_ACQUIRED	GQ read tracker L3 CACHE miss to RTID acquired	08	
UNC_GQ_ALLOC.WT_TO_RTID_ACQUIRED	GQ write tracker L3 CACHE miss to RTID acquired	10	
UNC_GQ_ALLOC.WT	GQ write tracker requests	20	
UNC_GQ_ALLOC.PPT	GQ peer probe tracker requests	40	

A latency can measured by the average duration of the queue occupancy, if the occupancy stops as soon as the data has been delivered. Thus the ratio of UNC_GQ_TRACKER_OCCUP.X/UNC_GQ_ALLOC.X measures an average duration of queue occupancy. The total occupancy period measured by

Total Read Period = UNC_GQ_TRACKER_OCCUP.RT/UNC_GQ_ALLOC.RT
Is longer than the data delivery latency due to it including time for extra General Queue bookkeeping and cleanup.

Similary, the
L3 CACHE response Latency = UNC_GQ_TRACKER_OCCUP.RT_TO_LLC_RESP
/UNC_GQ_ALLOC.RT_TO_LLC_RESP
This term is essentially a constant. It does not include the total time to snoop and retrieve a modified line from another core for example, just the time to scan the L3 CACHE and see if the line is or is not present on the socket.

Miss to fill latency = UNC_GQ_TRACKER_OCCUP.RT_LLC_MISS
/UNC_GQ_ALLOC.RT_LLC_MISS
This ratio will overcount the latency as it will include time for eviction of any modified lines that must be written back to dram on eviction.
An overall latency for an L3 CACHE hit is the weighted average over the latency of a simple hit, where the line has only been used by the core making the request and the latencies for accessing clean and dirty lines that have been accessed by multiple cores. These three components of the L3 CACHE hit for loads can be decomposed using the OFFCORE_RESPONSE events.

Table 15

OFFCORE_RESPONSE_0.DEMAND_DATA.LLC_HIT_NO_OTHER_CORE
OFFCORE_RESPONSE_0.DEMAND_DATA.LLC_HIT_OTHER_CORE_HIT
OFFCORE_RESPONSE_0.DEMAND_DATA.LLC_HIT_OTHER_CORE_HITM
OFFCORE_RESPONSE_0.DEMAND_DATA.LOCAL_CACHE

The *LOCAL_CACHE event should be used as the denominator. The individual latencies could have to be measured with microbenchmarks, but the use of the precise latency event will be far more effective as any bandwidth loading effects will be included.
The L3 CACHE miss component is the weighted average over the latencies of hits in a cache on another socket, with the multiple latencies as in the L3 CACHE hit just discussed. It also includes in the weighted average the latencies to local and remote dram. The local dram access and the remote socket access can be decomposed with more uncore events. This will be discussed a bit later in this paper.
The *RTID* events allow the monitoring of a sub component of the Miss to fill latency associated with the communications between the GQ and the QHL.

The write and peer probe latencies are the L3 CACHE response time + any other time required. This can also include a fraction due to L3 CACHE misses and retrievals from

dram. The breakdowns cannot be done in the way discussed above as the extra required events do not exist.

There are events which monitor if the three trackers are not empty (>= 1 entry) or full.
Table 23

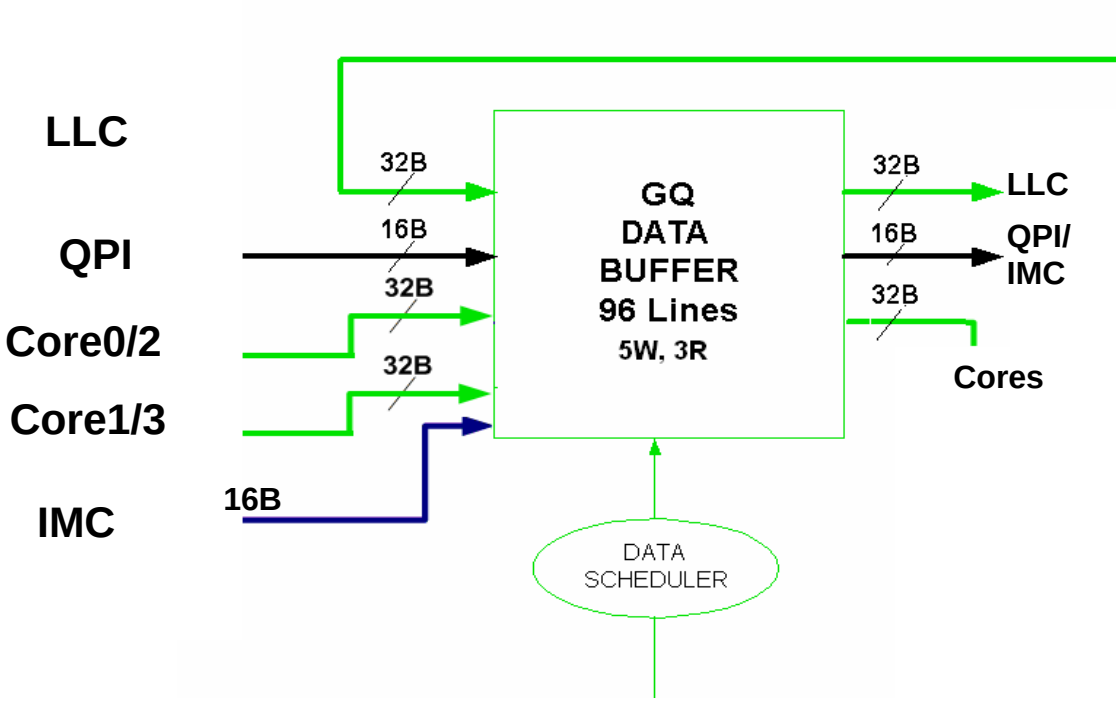
Event Name	Definition	Umask	Event
UNC_GQ_CYCLES_FULL.READ_TRACKER	Cycles GQ read tracker is full.	01	00
UNC_GQ_CYCLES_FULL.WRITE_TRACKER	Cycles GQ write tracker is full.	02	
UNC_GQ_CYCLES_FULL.PEER_PROBE_TRACKER	Cycles GQ peer probe tracker is full.	04	
UNC_GQ_CYCLES_BUSY.READ_TRACKER	Cycles GQ read tracker is busy	01	01
UNC_GQ_CYCLES_BUSY.WRITE_TRACKER	Cycles GQ write tracker is busy	02	
UNC_GQ_CYCLES_BUSY.PEER_PROBE_TRACKER	Cycles GQ peer probe tracker is busy	04	

The technique of dividing the latencies by the average queue occupancy in order to determine a penalty does not work for the uncore. Overlapping entries from different cores do not result in overlapping penalties and thus a reduction in stalled cycles. Each core suffers the full latency independently. To evaluate the correction on a per core basis one needs the number of cycles there is an entry from the core in question. A *NOT_EMPTY_CORE_N type event would needed. There is no such event. Consequently, in the cycle decomposition one must use the full latency for the estimate of the penalty. As has been stated before it is best to use the PEBS latency event as the data sources are also collected with the latency for the individual sample.
The individual components of the read tracker, discussed above, can also be monitored as busy or full by setting the cmask value to 1 or 32 and applying it to the assorted RT occupancy events.

Table 24

Event Name	CMASK	Umask	Event
UNC_GQ_TRACKER_OCCUP.RT_LLC_MISS_FULL	32	02	02
UNC_GQ_TRACKER_OCCUP.RT_TO_LLC_RESP_FULL	32	04	
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACQUIRED_FULL	32	08	
UNC_GQ_TRACKER_OCCUP.RT_LLC_MISS_BUSY	1	02	02
UNC_GQ_TRACKER_OCCUP.RT_TO_LLC_RESP_BUSY	1	04	
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACQUIRED_BUSY	1	08	

The GQ data buffer traffic controls the flow of data through the uncore. Diagramatically it can be shown as follows



The input and output flows can be monitored with the following events. They measure the cycles that the ports they are monitoring are busy. Most of the ports transfer a fixed number of bits per cycle, however the Intel® QuickPath Interconnect protocols can result in either 8 or 16 bytes being transferred on the read Intel QPI and IMC ports. Consequently these events cannot be used to measure total data transfers and bandwidths.

Table 25

Event Name	Definition	Umask	Event
UNC_GQ_DATA.FROM_QPI	Cycles GQ data is imported from Quickpath interconnect	01	04
UNC_GQ_DATA.FROM_IMC	Cycles GQ data is imported from integrated memory interface	02	
UNC_GQ_DATA.FROM_LLC	Cycles GQ data is imported from L3 CACHE	04	
UNC_GQ_DATA.FROM_CORES_02	Cycles GQ data is imported from Cores 0 and 2	08	
UNC_GQ_DATA.FROM_CORES_13	Cycles GQ data is imported from Cores 1 and 3	10	
UNC_GQ_DATA.TO_QPI_IMC	Cycles GQ data sent to the QPI or IMC	01	05
UNC_GQ_DATA.TO_LLC	Cycles GQ data sent to L3 CACHE	02	
UNC_GQ_DATA.TO_CORES	Cycles GQ data sent to cores	04	

The GQ handles the snoop responses for the cacheline requests that come in from the Intel® QuickPath Interconnect. These correspond to the queue entries in the peer probe tracker.
They are divided into requests for locally homed data and remotely homed data. If the line is in a modified state and the the GQ is responding to a read request the line also

must be written back to memory. This would be a wasted effort for a response to a RFO as the line will just be modified again, so no Writeback is done for RFOs.
The local home events:

Table 26

Event Name	Definition	Umask	Event
	Local home snoop response		
UNC_SNP_RESP_TO_LOCAL_HOME.I_STATE	L3 CACHE does not have cache line	01	06
UNC_SNP_RESP_TO_LOCAL_HOME.S_STATE	L3 CACHE has cache line in S state	02	
UNC_SNP_RESP_TO_LOCAL_HOME.FWD_S_STATE	L3 CACHE in E state, changed to S state and forwarded	04	
UNC_SNP_RESP_TO_LOCAL_HOME.FWD_I_STATE	L3 CACHE has forwarded a modified cache line responding to RFO	08	
UNC_SNP_RESP_TO_LOCAL_HOME.CONFLICT	Local home conflict snoop response	10	
UNC_SNP_RESP_TO_LOCAL_HOME.WB	L3 CACHE has cache line in the M state responding to read	20	
UNC_SNP_RESP_TO_LOCAL_HOME.HITM	L3 CACHE HITM (WB, FWD_S_STATE)	24	
UNC_SNP_RESP_TO_LOCAL_HOME.HIT	L3 CACHE HIT (S, FWD_I_STATE, conflict)	1A	

And the snoop responses for the remotely homed lines

Table 27

Event Name	Definition	Umask	Event
	Remote home snoop response		
UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE	L3 CACHE does not have cache line	01	07
UNC_SNP_RESP_TO_REMOTE_HOME.S_STATE	L3 CACHE has cache line in S state	02	
UNC_SNP_RESP_TO_REMOTE_HOME.FWD_S_STATE	L3 CACHE in E state, changed to S state and forwarded	04	
UNC_SNP_RESP_TO_REMOTE_HOME.FWD_I_STATE	L3 CACHE has forwarded a modified cache line responding to rfo	08	
UNC_SNP_RESP_TO_REMOTE_HOME.CONFLICT	Remote home conflict snoop response	10	
UNC_SNP_RESP_TO_REMOTE_HOME.WB	L3 CACHE has cache line in the M state responding to read	20	
UNC_SNP_RESP_TO_REMOTE_HOME.HITM	L3 CACHE HITM (WB, FWD_S_STATE)	24	
UNC_SNP_RESP_TO_REMOTE_HOME.HIT	L3 CACHE HIT (S, FWD_I_STATE, conflict)	1A	

Some related events count the MESI transitions in response to snoops from other caching agents (processors or IOH). Some of these rely on an MSR so they can only be measured one at a time, as there is only one MSR. The Intel performance tools will schedule this correctly by restricting these events to a single general uncore counter.

Table 28

Event Name	Definition	Umask	Event	msr	msr value
UNC_GQ_SNOOP.GOTO_S	change cache line to S state	01	0C	0	0
UNC_GQ_SNOOP.GOTO_I	change cache line to I state	02		0	0
UNC_GQ_SNOOP.GOTO_S_HIT_M	change cache line from M to S state	04		301	1
UNC_GQ_SNOOP.GOTO_S_HIT_E	change cache line from E to S state	04		301	2
UNC_GQ_SNOOP.GOTO_S_HIT_S	change cache line from S to S state	04		301	4
UNC_GQ_SNOOP.GOTO_S_HIT_F	change cache line from F to S state	04		301	8
UNC_GQ_SNOOP.GOTO_I_HIT_M	change cache line from M to I state	08		301	10
UNC_GQ_SNOOP.GOTO_I_HIT_E	change cache line from E to I state	08		301	20
UNC_GQ_SNOOP.GOTO_I_HIT_S	change cache line from S to I state	08		301	40
UNC_GQ_SNOOP.GOTO_I_HIT_F	change cache line from F to I state	08		301	80

L3 CACHE Events

The number of hits and misses can be determined from the GQ tracker allocation events, but it is simpler with the following list:

Table 29

Event Name	Definition	Umask	Event
UNC_LLC_HITS.READ	Number of L3 CACHE read hits	01	08
UNC_LLC_HITS.WRITE	Number of L3 CACHE write hits	02	
UNC_LLC_HITS.ANY	Number of L3 CACHE hits	03	
UNC_LLC_HITS.PROBE	Number of L3 CACHE peer probe hits	04	
UNC_LLC_MISS.READ	Number of L3 CACHE read misses	01	09
UNC_LLC_MISS.WRITE	Number of L3 CACHE write misses	02	
UNC_LLC_MISS.ANY	Number of L3 CACHE misses	03	
UNC_LLC_MISS.PROBE	Number of L3 CACHE peer probe misses	04	

Note that the *.any events only refer to requests from the cores of the local package and do not include the requests that arrive from the Intel® QuickPath Interconnect.

The MESI breakdown of lines allocated and victimized can also be monitored with LINES_IN, LINES_OUT:

Table 30

Event Name	Definition	Umask	Event
UNC_LLC_LINES_IN.M_STATE	L3 CACHE lines allocated in M state	01	0A
UNC_LLC_LINES_IN.E_STATE	L3 CACHE lines allocated in E state	02	
UNC_LLC_LINES_IN.S_STATE	L3 CACHE lines allocated in S state	04	
UNC_LLC_LINES_IN.F_STATE	L3 CACHE lines allocated in F state	08	
UNC_LLC_LINES_IN.ANY	L3 CACHE lines allocated	0F	
UNC_LLC_LINES_OUT.M_STATE	L3 CACHE lines victimized in M state	01	0B
UNC_LLC_LINES_OUT.E_STATE	L3 CACHE lines victimized in E state	02	
UNC_LLC_LINES_OUT.S_STATE	L3 CACHE lines victimized in S state	04	
UNC_LLC_LINES_OUT.I_STATE	L3 CACHE lines victimized in I state	08	
UNC_LLC_LINES_OUT.F_STATE	L3 CACHE lines victimized in F state	10	
UNC_LLC_LINES_OUT.ANY	L3 CACHE lines victimized	1F	

Intel® QuickPath Interconnect Home Logic (QHL)

Data access requests that miss the L3 CACHE are sent to the Intel® QuickPath Interconnect home logic unit to retrieve the data from the local NUMA dram. Such request are speculative by nature, as a hit(m) response to a snoop requests to the other caching agents may return the line more quickly and supersede the request to the QHL. Again this process starts with 3 request queues for requests from local requests, requests from remote processor sockets and from the IOH. The requests are broken down by reads and writes. This allows the measurement of the latency to local dram for all 3 request source classes for reads. Further the total bandwidth to the local dram can also be measured. In addition the contributions from the three sources and the fraction of read and write bandwidth/source can also be evaluated.

Again we have events that indicate when there are entries in the queues (*BUSY*) allowing the penalties to be evaluated (busy/requests). The number of cycles the queues are full can also be measured.

The request queues are set differently for DP and UP systems. On a DP system the local tracker has 24 entries the, the remote queue has 16 and the IOH has 24. In the UP mode there are only the local and IOH queues and both have 32 entries.

Using the L3 CACHE miss latency computed from the GQ occupancies and the QHL events we can decompose the miss latency into a component handled by the QHL and the component handled by the QPI.

The local QHL read latency = UNC_QHL_OCCUPANCY.LOCAL/
UNC_QHL_REQUESTS.LOCAL_READS

The total miss latency (discussed during the GQ section) can be decomposed (thereby defining the terms) into

Miss latency = L3 CACHE response Latency + M_{QPI} * QPI response latency + M_{QHL} * QHL latency

Where M_{QPI} and M_{QHL} are the fractions of L3 CACHE misses that get their responses from either the QPI interface or the local dram. The best way to compute these fractions is to simply use the core event

OFFCORE_RESPONSE_0.DATA_IFETCH.REMOTE_CACHE_DRAM

And

OFFCORE_RESPONSE_0.DATA_IFETCH.LOCAL_DRAM

The events are listed below.

Table 31

Event Name	Definition	Umask	Event
UNC_QHL_OCCUPANCY.IOH	QHL IOH tracker allocate to deallocate read occupancy	01	23
UNC_QHL_OCCUPANCY.REMOTE	QHL remote tracker allocate to deallocate read occupancy	02	23
UNC_QHL_OCCUPANCY.LOCAL	QHL local tracker allocate to deallocate read occupancy	04	23
UNC_QHL_REQUESTS.IOH_READS	QPI Home Logic IOH read requests	01	20
UNC_QHL_REQUESTS.IOH_WRITES	QPI Home Logic IOH write requests	02	20
UNC_QHL_REQUESTS.REMOTE_READS	QPI Home Logic remote read requests	04	20
UNC_QHL_REQUESTS.REMOTE_WRITES	QPI Home Logic remote write requests	08	20
UNC_QHL_REQUESTS.LOCAL_READS	QPI Home Logic local read requests	10	20
UNC_QHL_REQUESTS.LOCAL_WRITES	QPI Home Logic local write requests	20	20
UNC_QHL_CYCLES_FULL.IOH	Cycles QHL IOH Tracker is full	01	21
UNC_QHL_CYCLES_FULL.REMOTE	Cycles QHL Remote Tracker is full	02	21
UNC_QHL_CYCLES_FULL.LOCAL	Cycles QHL Local Tracker is full	04	21
UNC_QHL_CYCLES_NOT_EMPTY.IOH	Cycles QHL IOH tracker is busy	01	22
UNC_QHL_CYCLES_NOT_EMPTY.REMOTE	Cycles QHL remote tracker is busy	02	22
UNC_QHL_CYCLES_NOT_EMPTY.LOCAL	Cycles QHL local tracker is busy	04	22

Integrated Memory Controller (IMC)

Access to dram is controlled directly from the processor package. The performance monitoring capabilities associated with the memory controller are extensive, extending far beyond typical expected needs. The memory controller supports up to three memory channels and each channel is monitored individually.

The first thing is to measure the queue occupancies, queue inserts (allocations) and the cycles where the queues are full for normal application accesses to the dram. This reveals the latencies and bandwidths per memory channel. For this the following events are particularly useful.

Note: When first programming the queue occupancy events, a bus lock must be issued to correctly initialize the occupancy. This is the same requirement encountered with the GQ and QHL occupancy events.

Table 32

Event Name	Definition	Umask	Event
UNC_IMC_NORMAL_OCCUPANCY.CH0	IMC channel 0 normal read request occupancy	01	2A
UNC_IMC_NORMAL_OCCUPANCY.CH1	IMC channel 1 normal read request occupancy	02	
UNC_IMC_NORMAL_OCCUPANCY.CH2	IMC channel 2 normal read request occupancy	04	
UNC_IMC_NORMAL_READS.CH0	IMC channel 0 normal read requests	01	2C

UNC_IMC_NORMAL_READS.CH1	IMC channel 1 normal read requests	02	
UNC_IMC_NORMAL_READS.CH2	IMC channel 2 normal read requests	04	
UNC_IMC_NORMAL_READS.ANY	IMC normal read requests	07	
UNC_IMC_WRITES.FULL.CH0	IMC channel 0 full cache line writes	01	2F
UNC_IMC_WRITES.FULL.CH1	IMC channel 1 full cache line writes	02	
UNC_IMC_WRITES.FULL.CH2	IMC channel 2 full cache line writes	04	
UNC_IMC_WRITES.FULL.ANY	IMC full cache line writes	07	
UNC_IMC_WRITES.PARTIAL.CH0	IMC channel 0 partial cache line writes	08	
UNC_IMC_WRITES.PARTIAL.CH1	IMC channel 1 partial cache line writes	10	
UNC_IMC_WRITES.PARTIAL.CH2	IMC channel 2 partial cache line writes	20	
UNC_IMC_WRITES.PARTIAL.ANY	IMC partial cache line writes	38	

The bandwidths due to normal application dram access can be evaluated as follows:
Read Bandwidth (ch0) = 64*UNC_IMC_NORMAL_READS.CH0* Frequency/Cycles
Where any of the cycle events (core or uncore) can be used as long as the corresponding frequency is used also.
Similarly the write bandwidth can be evaluated (ignoring partial writes) as:
Write Bandwidth (ch0) = 64*UNC_IMC_WRITES.FULL.CH0* Frequency/Cycles

The penalty can be evaluated using the cycles that there are entries in the queues as usual. Similarly there are events for counting the cycles during which the associated queues were full.

Table 33

Event Name	Definition	Umask	Event
UNC_IMC_BUSY.READ.CH0	Cycles IMC channel 0 busy with a read request	01	29
UNC_IMC_BUSY.READ.CH1	Cycles IMC channel 1 busy with a read request	02	29
UNC_IMC_BUSY.READ.CH2	Cycles IMC channel 2 busy with a read request	04	29
UNC_IMC_BUSY.WRITE.CH0	Cycles IMC channel 0 busy with a write request	08	29
UNC_IMC_BUSY.WRITE.CH1	Cycles IMC channel 1 busy with a write request	10	29
UNC_IMC_BUSY.WRITE.CH2	Cycles IMC channel 2 busy with a write request	20	29

As the dram control is on the processor the dram paging policy statistics can also be collected. Some of the events related to this are listed below.

Table 34

Event Name	Definition	Umask	Event
UNC_DRAM_OPEN.CH0	DRAM Channel 0 open commands	01	60
UNC_DRAM_OPEN.CH1	DRAM Channel 1 open commands	02	60
UNC_DRAM_OPEN.CH2	DRAM Channel 2 open commands	04	60
UNC_DRAM_PAGE_CLOSE.CH0	DRAM Channel 0 page close	01	61

UNC_DRAM_PAGE_CLOSE.CH1	DRAM Channel 1 page close	02	61
UNC_DRAM_PAGE_CLOSE.CH2	DRAM Channel 2 page close	04	61
UNC_DRAM_PAGE_MISS.CH0	DRAM Channel 0 page miss	01	62
UNC_DRAM_PAGE_MISS.CH1	DRAM Channel 1 page miss	02	62
UNC_DRAM_PAGE_MISS.CH2	DRAM Channel 2 page miss	04	62

There are also queues to monitor the high priority accesses like those associated with device drivers for video and network adapters.

Table 35

Event Name	Definition	Umask	Event
UNC_IMC_ISOC_OCCUPANCY.CH0	IMC channel 0 ISOC read request occupancy	01	2B
UNC_IMC_ISOC_OCCUPANCY.CH1	IMC channel 1 ISOC read request occupancy	02	2B
UNC_IMC_ISOC_OCCUPANCY.CH2	IMC channel 2 ISOC read request occupancy	04	2B
UNC_IMC_ISOC_OCCUPANCY.ANY	IMC ISOC read request occupancy	07	2B
UNC_IMC_HIGH_PRIORITY_READS.CH0	IMC channel 0 high priority read requests	01	2D
UNC_IMC_HIGH_PRIORITY_READS.CH1	IMC channel 1 high priority read requests	02	2D
UNC_IMC_HIGH_PRIORITY_READS.CH2	IMC channel 2 high priority read requests	04	2D
UNC_IMC_HIGH_PRIORITY_READS.ANY	IMC high priority read requests	07	2D
UNC_IMC_CRITICAL_PRIORITY_READS.CH0	IMC channel 0 critical priority read requests	01	2E
UNC_IMC_CRITICAL_PRIORITY_READS.CH1	IMC channel 1 critical priority read requests	02	2E
UNC_IMC_CRITICAL_PRIORITY_READS.CH2	IMC channel 2 critical priority read requests	04	2E
UNC_IMC_CRITICAL_PRIORITY_READS.ANY	IMC critical priority read requests	07	2E
UNC_IMC_ISOC_FULL.READ.CH0	Cycles DRAM channel 0 full with ISOC read requests	01	28
UNC_IMC_ISOC_FULL.READ.CH1	Cycles DRAM channel 1 full with ISOC read requests	02	28
UNC_IMC_ISOC_FULL.READ.CH2	Cycles DRAM channel 2 full with ISOC read requests	04	28
UNC_IMC_ISOC_FULL.WRITE.CH0	Cycles DRAM channel 0 full with ISOC write requests	08	28
UNC_IMC_ISOC_FULL.WRITE.CH1	Cycles DRAM channel 1 full with ISOC write requests	10	28
UNC_IMC_ISOC_FULL.WRITE.CH2	Cycles DRAM channel 2 full with ISOC write requests	20	28

Dram access control commands can be monitored with:

Table 36

Event Name	Definition	Umask	Event
UNC_DRAM_READ_CAS.CH0	DRAM Channel 0 read CAS commands	01	63
UNC_DRAM_READ_CAS.AUTOPRE_CH0	DRAM Channel 0 read CAS auto page close commands	02	63
UNC_DRAM_READ_CAS.CH1	DRAM Channel 1 read CAS commands	04	63
UNC_DRAM_READ_CAS.AUTOPRE_CH1	DRAM Channel 1 read CAS auto page close commands	08	63
UNC_DRAM_READ_CAS.CH2	DRAM Channel 2 read CAS commands	10	63

UNC_DRAM_READ_CAS.AUTOPRE_CH2	DRAM Channel 2 read CAS auto page close commands	20	63
UNC_DRAM_WRITE_CAS.CH0	DRAM Channel 0 write CAS commands	01	64
UNC_DRAM_WRITE_CAS.AUTOPRE_CH0	DRAM Channel 0 write CAS auto page close commands	02	64
UNC_DRAM_WRITE_CAS.CH1	DRAM Channel 1 write CAS commands	04	64
UNC_DRAM_WRITE_CAS.AUTOPRE_CH1	DRAM Channel 1 write CAS auto page close commands	08	64
UNC_DRAM_WRITE_CAS.CH2	DRAM Channel 2 write CAS commands	10	64
UNC_DRAM_WRITE_CAS.AUTOPRE_CH2	DRAM Channel 2 write CAS auto page close commands	20	64
UNC_DRAM_REFRESH.CH0	DRAM Channel 0 refresh commands	01	65
UNC_DRAM_REFRESH.CH1	DRAM Channel 1 refresh commands	02	65
UNC_DRAM_REFRESH.CH2	DRAM Channel 2 refresh commands	04	65
UNC_DRAM_PRE_ALL.CH0	DRAM Channel 0 precharge all commands	01	66
UNC_DRAM_PRE_ALL.CH1	DRAM Channel 1 precharge all commands	02	66
UNC_DRAM_PRE_ALL.CH2	DRAM Channel 2 precharge all commands	04	66

Intel® QuickPath Interconnect Home Logic Opcode Matching

One rather different form of monitoring the cacheline access and writeback traffic in the uncore is to use the QHL opcode matching capability. QHL requests can be superseded when another source can supply the required line more quickly.

L3 CACHE misses to locally homed lines, due to on package requests, are simultaneously directed to the QHL and QPI. If a remote caching agent supplies the line first then the request to the QHL is sent a signal that the transaction is complete. If the remote caching agent returns a modified line in response to a read request then the data in dram must be updated with a writeback of the new version of the line.

There is a similar flow of control signals when the QPI simultaneously sends a snoop request for a locally homed line to both the GQ and the QHL. If the L3 CACHE has the line the QHL must be signaled that the transaction was completely by the L3 CACHE/GQ. If the line in the L3 CACHE (or the cores) was modified and the snoop request from the remote package was for a load, then a writeback must be completed by the QHL and the QHL forwards the line to the QPI to complete the transaction.

The cases listed above (and others) can be monitored by using the opcode matcher in the QHL to monitor which protocol signals it receives and use these to count the occurrences of the assorted cases. The opcode matcher is programmed with MSR 396h as described in the table below:

Table 37

Bit Position	Bit Name	Access Method	Reset Type	Reset Value	Bit Description
39:03:00	Address	state	reset_reset	0	An Address Match PerfMon event is generated if the incoming address for a request matches these bits.
47:40:00	Opcode	state	reset_reset	0	An Opcode Match PerfMon event is generated if the incoming Opcode for a request matches these bits.
63:61	Match_Select	state	reset_reset	0	This match select field allows for the following sub-event combinations: Bits [63:61] = '10- ' Address match only Bits [63:61] = '01- ' Opcode match only Bits [63:61] = '11- ' (Address match) OR (Opcode match) Bits [63:61] = '001 ' (Address match) AND (Opcode match) Bits [63:61] = '000 ' 0 Upon hardware reset, this field is all zeroes thus powering down the local event generation logic in the CHL section. This encoding scheme helps minimize design impact in the CHL (only 2 additional stages of logic needed).

There is no reasonable way to predefine any address matching of course but several opcodes that identify writebacks and forwards from the caches that are certainly useful as they identify (un)modified lines that were forwarded from the remote socket or to the remote socket. Not all the predefined entries currently make sense.

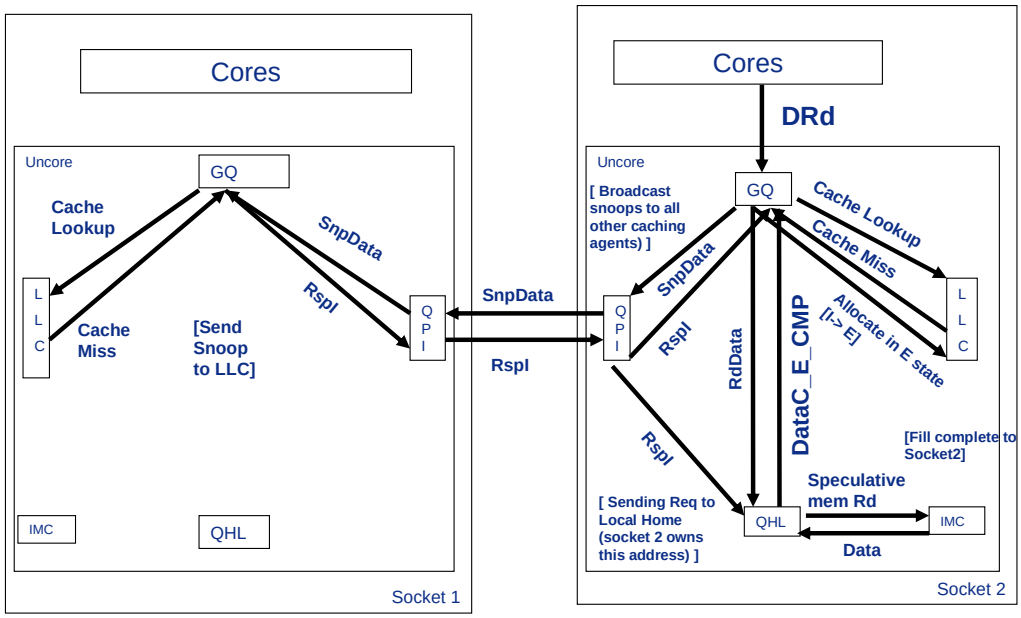
Table 38

Event Name	Description	umask	Event	MSR	MSR Value
UNC_ADDR_OPCODE_MATCH.IOH.NONE	No opcode match	01	35	396	0
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDI	Hitm in IOH Cache, RFO snoop	01	35	396	4000190000000000
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDS	??	01	35	396	40001A0000000000
UNC_ADDR_OPCODE_MATCH.IOH.RSPIWB	??	01	35	396	40001D0000000000
UNC_ADDR_OPCODE_MATCH.LOCAL.NONE	none	04	35	396	0

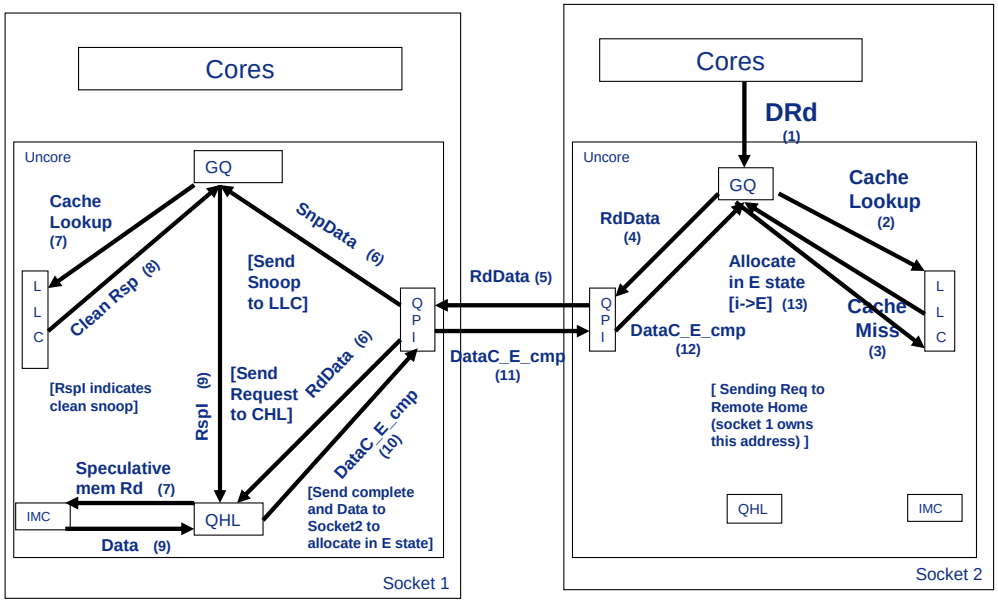
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDI	Hitm in local L3 CACHE, RFO snoop	04	35	396	4000190000000000
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDS	Local L3 CACHE in F or S, load snoop	04	35	396	40001A0000000000
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPIWB	Hitm in local L3 CACHE, load snoop	04	35	396	40001D0000000000
UNC_ADDR_OPCODE_MATCH.REMOTE.NONE	none	02	35	396	0
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDI	Hitm in remote L3 CACHE, RFO	02	35	396	4000190000000000
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDS	Remote L3 CACHE in F or S, load	02	35	396	40001A0000000000
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPIWB	Hitm in remote L3 CACHE, load	02	35	396	40001D0000000000

These opcode uses can be seen from the dual socket QPI communications diagrams below. These predefined opcode match encodings can be used to monitor HITM accesses in particular and serve as the only event that allows profiling the requesting code on the basis of the HITM transfers its requests generate.

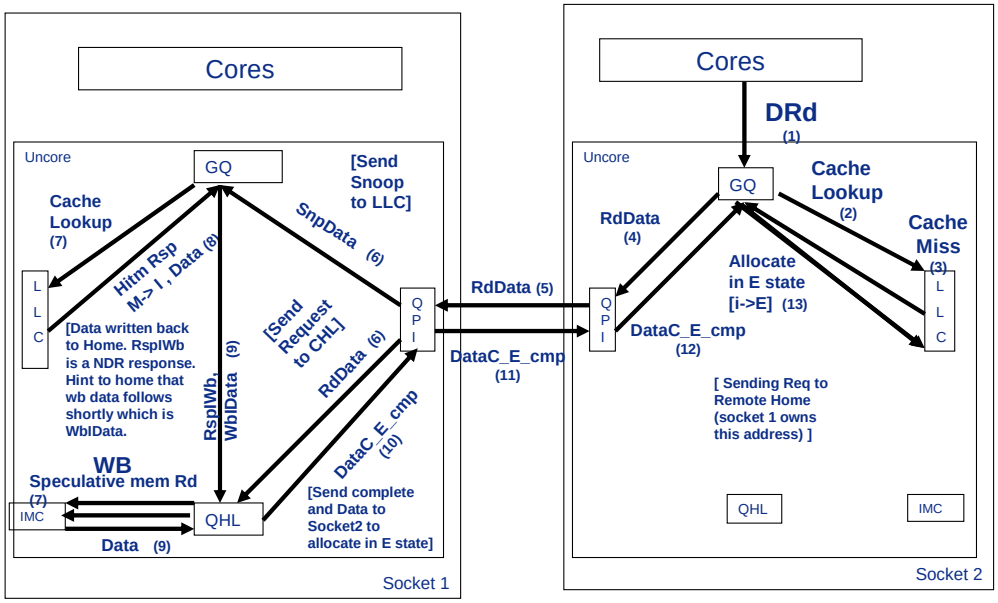
RdData request after LLC Miss to Local Home (Clean Rsp)



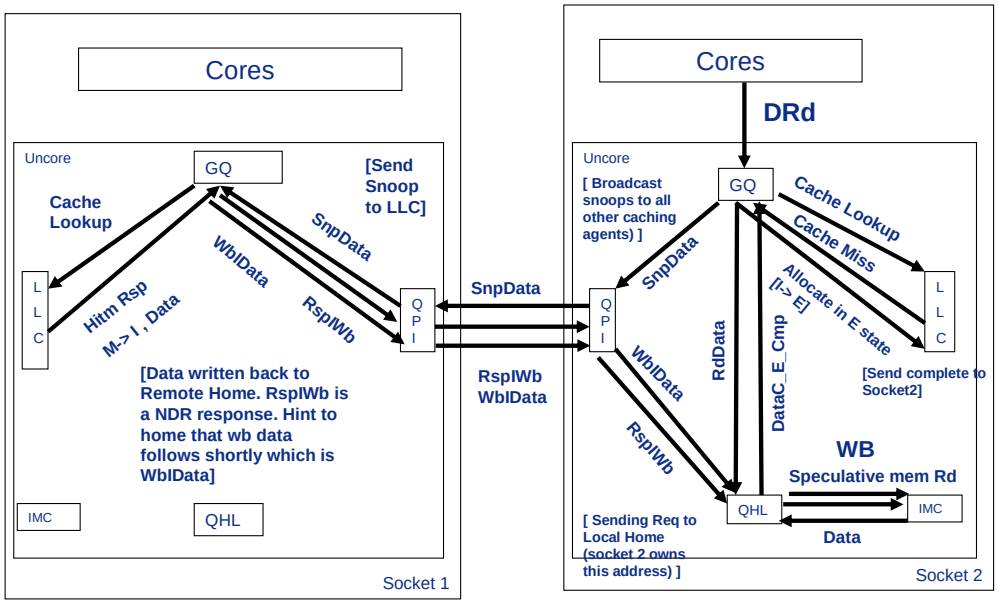
RdData request after LLC Miss to Remote Home (Clean Rsp)



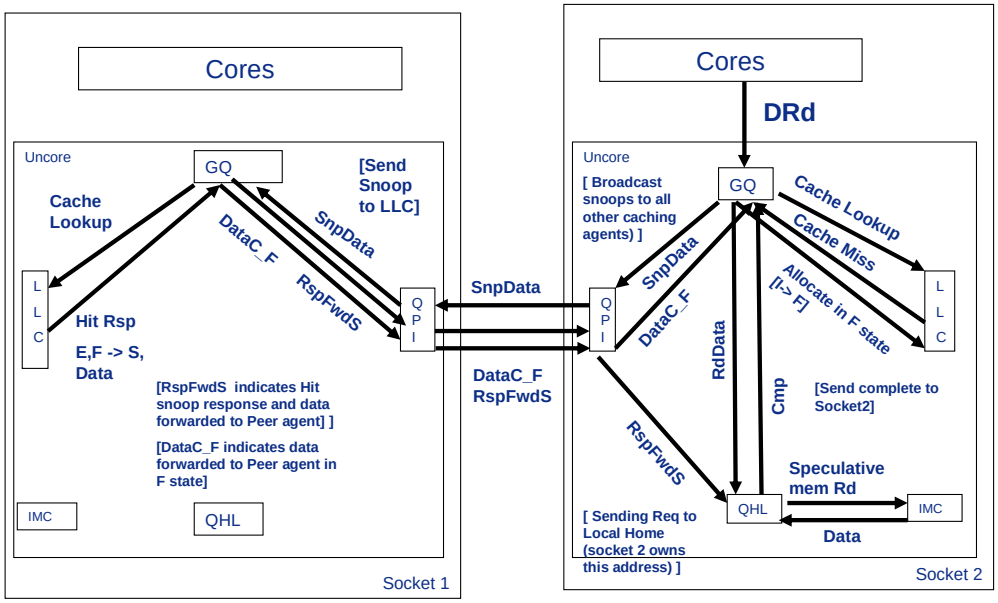
RdData request after LLC Miss to Remote Home (Hitm Res)



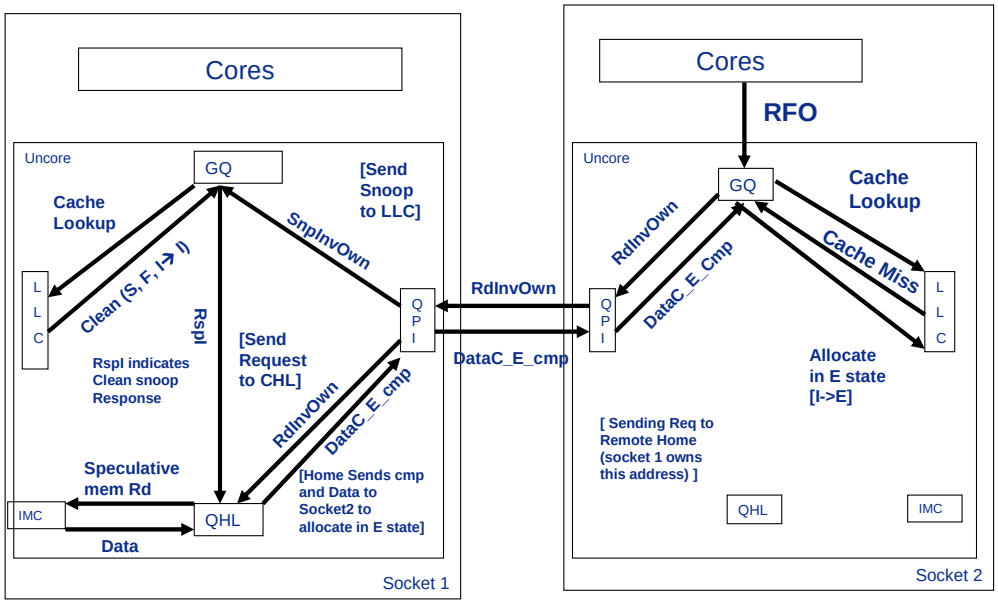
RdData request after LLC Miss to Local Home (Hitm Response)

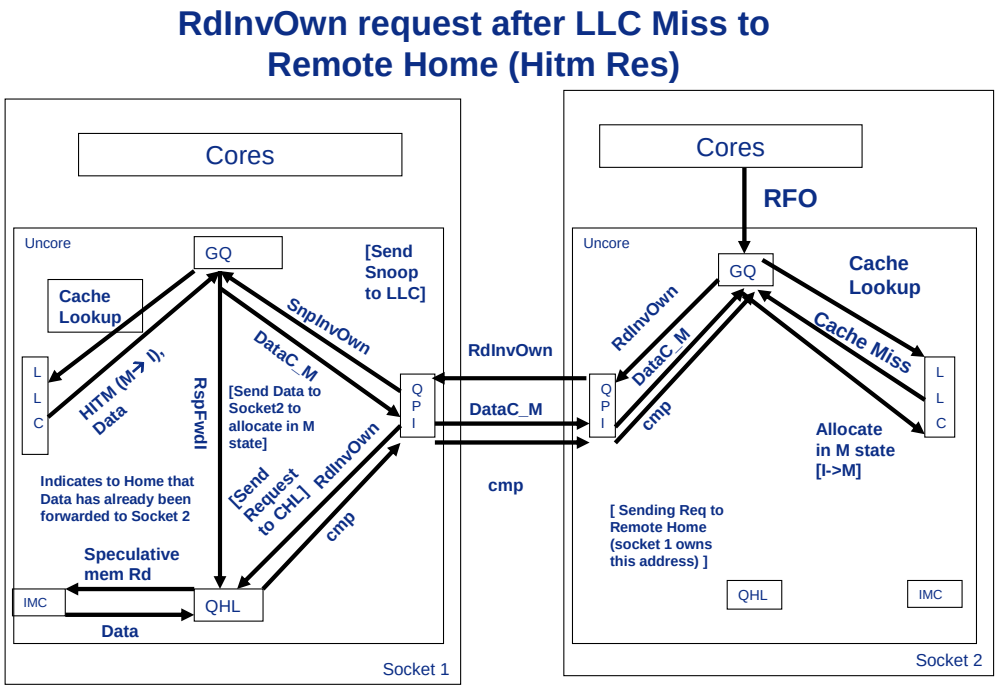


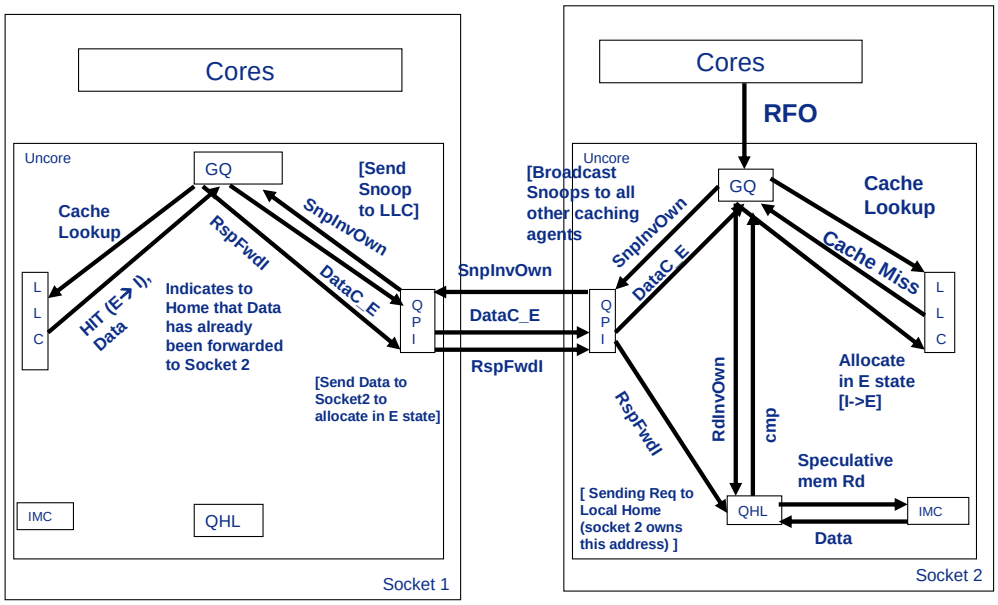
RdData request after LLC Miss to Local Home (Hit Response)



RdInvOwn request after LLC Miss to Remote Home (Clean Res)







The diagrams show a series of QPI protocol exchanges associated with Data Reads and Reads for Ownership (RFO), after an L3 CACHE miss, under a variety of combinations

of the local home of the cacheline, and the MESI state in the remote cache. Of particular note are the cases where the data comes from the remote QHL even when the data was in the remote L3 CACHE. These are the Read Data with the remote L3 CACHE having the line in an M state. Whether the line is locally or remotely “homed” it has to be written back to dram before the originating GQ receives the line, so it always appears to come from a QHL. The RFO does not do this. However, when responding to a remote RFO (SnpInvOwn) and the line is in an S or F state, the cacheline gets invalidated and the line is sent from the QHL.
The point is that the data source might not always be so obvious.

Measuring Bandwidth From the Uncore

Read bandwidth can be measured on a per core basis using events like OFFCORE_RESPONSE_0.DATA_IN.LOCAL_DRAM and OFFCORE_RESPONSE_0.DATA_IN.REMOTE_DRAM. The total bandwidth includes writes and these cannot be monitored from the core as they are mostly caused by evictions of modified lines in the L3 CACHE. Thus a line used and modified by one core can end up being written back to dram when it is evicted due to a read on another core doing some completely unrelated task. Modified cached lines and writebacks of uncached lines (ex: written with non temporal streaming stores) are handled differently in the uncore and their writebacks increment various events in different ways.
All full lines written to dram are counted by the UNC_IMC_WRITES.FULL.* events. This includes the writebacks of modified cached lines and the writes of uncached lines, for example generated by NT SSE stores. The uncached line writebacks from a remote socket will be counted by UNC_QHL_REQUESTS.REMOTE_WRITES. The uncached writebacks from the local cores are not counted by UNC_QHL_REQUESTS.LOCAL_WRITES, as this event only counts writebacks of locally cached lines.
The UNC_IMC_NORMAL_READS.* events only count the reads. The UNC_QHL_REQUESTS.LOCAL_READS and the UNC_QHL_REQUESTS.REMOTE_READS count the reads and the InvtoE transactions, which are issued for the uncacheable writes, eg USWC/UC writes. This allows the evaluation of the uncacheable writes, by computing the difference of UNC_QHL_REQUESTS.LOCAL_READS + UNC_QHL_REQUESTS.REMOTE_READS – UNC_IMC_NORMAL_READS.ANY. These events are summarized in the following table

Table 39

Event	Description
UNC_IMC_WRITES.FULL.ANY	All writes of full cachelines (cached and uncached)
UNC_IMC_WRITES.FULL.CH0	Writes of full lines to channel 0
UNC_IMC_WRITES.FULL.CH1	Writes of full lines to channel 1
UNC_IMC_WRITES.FULL.CH2	Writes of full lines to channel 2
UNC_QHL_REQUESTS. LOCAL_WRITES	Writes of modified cached lines from local cores
UNC_QHL_REQUESTS.	Writes of modified cached lines AND uncached lines from

REMOTE_WRITES	remote cores
UNC_IMC_NORMAL_READS.ANY	Total normal priority reads
UNC_IMC_NORMAL_READS.CH0	Total normal priority reads on Channel 0
UNC_IMC_NORMAL_READS.CH1	Total normal priority reads on Channel 1
UNC_IMC_NORMAL_READS.CH2	Total normal priority reads on Channel 2
UNC_QHL_REQUESTS. LOCAL_READS	Total reads plus I to E for writebacks from local cores
UNC_QHL_REQUESTS. REMOTE_READS	Total reads plus I to E for writebacks from remote cores

Conclusion:

Intel® Core™ i7 Processors and Intel® Xeon™ 5500 Processors open a new class of performance analysis capablitlies

Appendix 1

Profiles

Basic Intel® PTU Profiles

General Exploration

A six event set that can be captured in a single run. The following events are included:
Cpu_clk_unhalted.core
Inst_retired.any
Br_inst_retired.all_branches
Mem_inst_retired.latency_above_threshold_32
Mem_load_retired.llc_miss
Uops_executed.core_stall_cycles
Thus this profile gives cycle usage and stalled cycles for the core (ie works best with HT disabled). Instructions retired can be used for basic block execution counts, particularly in conjunction with the precise branch retired event
Using the latency event with a 32 cycle threshold measures the distribution of offcore accesses though the data source encoding captured with the event. As the latency events capture data sources, latencies and linear addresses, this profile can also yield a breakdown of data sources for loads that miss the core's data caches and the latencies that result. This event is randomly samples loads and the sampling fraction is dependent on the application. This can be measured by normalizing the sum of the L3 CACHE miss data sources with the Mem_uncore_retired.llc_miss event, which counts them all.

Branch Analysis

This profile is designed for detailed branch analysis. The 4 events:
br_inst_retired.all_branches
br_inst_retired.near_call:LBR=user_calls
cpu_clk_unhalted.thread
inst_retired.any
allow basic execution analysis and a variety of deatiled loop and call analyses. The call counts per source can be extracted as the LBRs are captured. If the call counts are low you may need to make a copy of the profile and decrease the SAV value. As the LBRs are captured this will take 2 runs and cannot be multiplexed. As the registers are captured, on Intel(r) Core(tm) i7 systems running in Intel(r) 64 enabled mode, the integer arguments of functions can be extracted from the register values display in the asm display, for functions with limited numbers of arguments. Further the register values can be used to get average tripcount values for counter loops where an induction variable is compared to a tripcount, when using the all_branches event. The SAV value for the call retired event must be tuned to the application as this can vary by orders of magnitude between applications.

Cycles and Uops

This list of 14 events, thus collected in 3 runs, assists in the analysis of cycle usage and uop flow through the core pipeline, and execution flow through the program. The events are:
Br_inst_retired.conditional
Br_inst_retired.near_call
Cpu_clk_unhalted.core
Inst_retired.any
Resource_stalls.any
Uops_decoded.any
Uops_decoded.stall_cycles
Uops_executed.core_stall_cycles
Uops_executed.port015
Uops_executed.port234_core
Uops_issued.any
Uops_issued.stall_cycles
Uops_retired.any
Uops_retired.stall_cycles
This set of events can be used to identify a large number of performance issues. It identifies where cycles are consumed in the code and what fraction of them corresponded to stalls in the execution stages. Uops_executed.core_stall_cycles count cycles that no uops were dispatched to the execution units. Used in conjunction with the precise event, uops_retired.stall_cycles, the nature of the stall can usually be discerned with the disassembly display. The control flow of the program can be monitored with the inst_retired and the two PEBS branch events. Function call counts and basic block execution counts can be extracted with these 3 events. Uops_issued.stall_cycles-resource_stalls.any monitors uop delivery starvation due to FE issues. Uops_executed.port015 + uops_executed.core234.core – uops_retired.any measures the

wasted work due to speculatively dispatched instructions.
Uops_retired.any/inst_retired.any can be used to identify when (FP) exception handlers are being frequently invoked, and causing a high ratio (>>1). The comparison of the stall cycle counts at the various pipeline stages can yield insight into the uop flow efficiency.

Memory Access

A set of 13 events that require 3 runs, or three groups if event multiplexing is enabled. These events were selected to give a reasonably complete breakdown of cacheline traffic due to loads and some overview of total offcore traffic. As cycles and instructions retired have dedicated counters, they are also included. The additional events are
Mem_inst_retired.loads
Mem_inst_retired.stores
Mem_inst_retired.latency_above_threshold_32
Mem_inst_retired.latency_above_threshold_128
Mem_load_retired.llc_miss
Mem_load_retired.llc_unshared_hit
Mem_load_retired.other_core_l2_hit_hitm
Mem_uncore_retired.local_dram
Mem_uncore_retired.remote_dram
Offcore_response_0.data_in.local_dram
Offcore_response_0.data_in.remote_dram
The use of the offcore_response_0.any_request.local_dram/remote dram events was selected because non temporal stores to local dram are miscounted by the "other_core_hit_hitm" data source. This does not happen for the remote dram.
Using two latency thresholds allows simultaneous monitoring of L3 CACHE hits and L3 CACHE misses with reasonable statistical accuracy. As the latency events capture data sources, latencies and linear addresses, it was decided that full data profiling with all the events would not be needed. A copy that includes this option is also provided.

False- True Sharing

These 2 precise events should be extremely effective at identifying cachelines that result in access contention in threaded applications. This profile can be used on either single or dual socket configurations.
Mem_inst_retired.stores
Mem_uncore_retired.other_core_l2_hitm
The SAV values are lowered with respect to nominal values as the objective is to capture as many addresses as possible to enable the data profiling analysis of these contended accesses. If data acquisition runs last much longer than 1 minute, invoke the sampling multipliers in the project properties in order to keep the number of events under 10 million. The tb5 file will not be created if the size exceeds 4GBs.

FE Investigation

A list of 14 events, thus collected in 3 runs, that yields a reasonably complete breakdown of instruction delivery related performance issues.
Br_inst_exec.any
Br_misp_exec.any
Cpu_clk_unhalted.core
Inst_retired.any
Ild_stall.any
Ild_stall.lcp
Itlb_miss.retired
L1I.cycles_stalled
L1I.misses
Rat_stalls.flags
Rat_stalls.registers
Rat_stalls.rob_read_port
Resource_stalls.any
Uops_issued.stall_cycles
The difference of uops_issued.stall_cycles - resource_stalls.any yields the instruction starvation cycles when the machine is booted with HT disabled. This can be used as an overall guide for identifying a uop delivery problem. The main causes for such issues are usually, branch mispredictions causing incorrect instruction “prefetching”, uop decoding and resource allocation bandwidth issues and excessively large active binaries. The selected events should assist in the identification of these issues.

Working Set

A single run data collection gathering PEBS data on all loads and stores retired. In addition, cycles, inst_retired.any, and conditional branches executed are also collected. The Sample After Values for the load and store instructions are lowered from the default values by a factor of 100. This will result in a severe performance distortion and an enormous amount of data being collected. This is needed to accurately sample the address space of a real application. The data profiling is enabled and the Intel® PTU Data Access Analysis package can be used to get an idea of the working set size of the program in the utility histogram pane. Address profiles can also be nicely extracted with this profile. An application that normally runs in one minute on a single core will produce approximately 2GBs of data, so it is wise to use the SAV multipliers if the run time is longer than a couple minutes.

Loop Analysis with call sites

A reasonably complete list of events for analyzing loop dominated codes
Arith.cycles_div_busy
Br_inst_retired.all_branches
Br_inst_retired.any.near_call
Br_misp_exec.any

Cpu_clk_unhalted.thread
DTLB_misses.any
Inst_retired.any
Load_hit_pre
Mem_inst_retired.Latency_above_threshold_32
Mem_load_retired.L2_hit
Mem_load_retired.llc_miss
Mem_load_retired.llc_unshared_hit
Mem_load_retired.other_core_l2_hit_hitm
Mem_uncore_retired.local_dram
Mem_uncore_retired.other_core_l2_hitm
Mem_uncore_retired.remote_dram
Offcore_response_0.data_in.any_dram
Offcore_response_0.data_in.local_dram
Offcore_response_0.data_in.remote_dram
Sq_full_stall_cycles
Rat_stalls.any
Rat_stalls.rob_read_port
Resource_stalls.load
Resource_stalls.ROB_full
Resource_stalls.RS_full
Resource_stalls.store
Uops_executed.core_stall_cycles
Uops_issued.any
Uops_issued.stall_cycles
Uops_retired.any
Uops_retired.stall_cycles
This list of events will allow computation of many of the more relevant predefined ratios for loop execution. These include the cycles lost to assorted load latencies, stalls at execution, FE stalls, stalls at retirement, wasted work, branch misprediction rates, basic block execution counts, function call counts, a few specific loop related FE and saturation effects and input bandwidth. The precise events will be collected with the full PEBS buffer enabling data address profiling.
This profile was added in Intel® PTU with and without call site collection to allow multiplexing. However, currently multiplexing in intel® PTU will not work with this many events. Further, Intel® PTU 3.2 multiplexing may crash some OS’s when HT is enabled.

Client Analysis with/without call sites

A reasonably complete list of events for analyzing typical client applications
Arith.cycles_div_busy
Arith.div
Arith.mul
Br_inst_retired.all_branches
Br_inst_retired.any.near_call

Br_misp_exec.any
Cache_lock_cycles.l1d
Cpu_clk_unhalted.thread
DTLB_misses.any
Fp_mmx_trans.any
Ild_stall.any
ild_stalls.iq_full
Ild_stall.lcp
ild_stalls.mru
ild_stalls.regen
Inst_retired.any
Itlb_miss_retired
L1i.cycles_stalled
L1I.misses
Load_hit_pre
Machine_clears.cycles
Mem_inst_retired.Latency_above_threshold_32
Mem_inst_retired.loads
mem_inst_retired.stores
Mem_load_retired.hit_lfb
mem_load_retired.L1d_hit
Mem_load_retired.L2_hit
Mem_load_retired.llc_miss
Mem_load_retired.llc_unshared_hit
Mem_load_retired.other_core_l2_hit_hitm
Mem_uncore_retired.local_dram
Mem_uncore_retired.other_core_l2_hitm
Misalign_mem_ref.load
Misalign_mem_ref.store
Offcore_request.uncached_mem
Offcore_response_0.data_in.any_dram
Partial_address_alias
Sq_full_stall_cycles
Rat_stalls.any
Rat_stalls.flags
Rat_stalls.registers
Rat_stalls.rob_read_port
Resource_stalls.any
Resource_stalls.load
Resource_stalls.ROB_full
Resource_stalls.RS_full
Resource_stalls.store
Uops_executed.core_stall_cycles
Uops_issued.any
Uops_issued.stall_cycles
Uops_retired.any

Uops_retired.stall_cycles
Uops_issued.core_stall_cycles
This list of events wil allow computation of many of the more relevant predefined ratios of interest in client application execution. These include the cycles lost to assorted load latencies, stalls at execution, FE stalls and most of the causes of FE stalls, stalls at retirement, wasted work, branch misprediction rates, basic block execution counts, function call counts, a few specific low penalty issues seen in client applications, saturation effects and input bandwidth. With HT disabled FE stalls can be evaluated with uops_issued.stall_cycles - resource_stalls.any, with HT enabled use uops_issued.core_stall_cycles-resource_stalls.any. The precise events will be collected with the full PEBS buffer enabling data address profiling.
This profile was added in Intel® PTU with and without call site collection to allow multiplexing. However, currently multiplexing in intel® PTU will not work with this many events. Further, Intel® PTU 3.2 multiplexing may crash some OS's when HT is enabled.

Appendix II PMU Programming

Figure 1: PerfEvtSelX MSR Definition

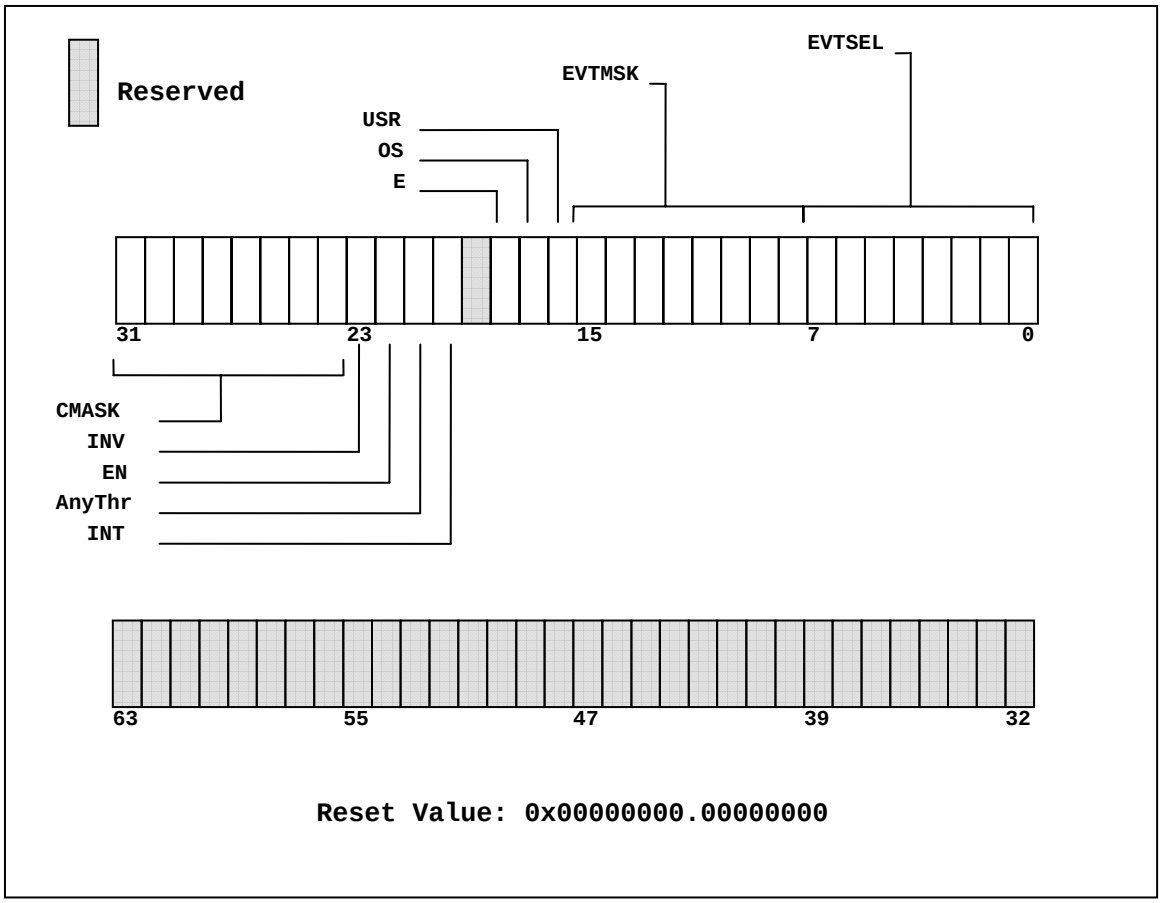


Table 1: PerfEvtSelX Programming

Bit	Bit Position	Access	Description
EVTSEL	7:0	RW	Selects the event logic unit used to detect micro-architectural conditions.
EVTMSK	15:8	RW	Condition qualifiers for the event selection logic specified in the EVTSEL field.
USR	16	RW	When set, indicates that the event specified by bit fields EVTSEL and EVTMSK is counted only when the logical processor is operating and privilege level 1, 2, or 3.
OS	17	RW	When set, indicates that the event specified by bit fields EVTSEL and EVTMSK is counted only when the logical processor is operating and privilege level 0.
E	18	RW	When set, causes the counter to increment when a deasserted to asserted transition occurs for the conditions that can be expressed by any of the fields in this register.
INT	20	RW	When set, the logical processor generates an exception through its local APIC on counter overflow. Counters only count up, and interrupts are generated on a transition from maximum count to zero. There will be some latency from the time the counter triggers the interrupt until the interrupt handler is invoked.
AnyThr	21	RW	When clear, the counter increments only when event conditions are satisfied in its logical processor. When Set, the counter increments when event conditions are satisfied for any logical processor in the core in which this counter resides.
EN	22	RW	When clear, this counter is locally disabled. When set, this counter is locally enabled.
INV	23	RW	When clear, the CMASK field is interpreted as greater than or equal to. When set, the CMASK field is interpreted as less than.
CMASK	31:24	RW	When this field is clear, it has no effect on counting. When set to a value other than zero, the logical processor compares this field to the event counts on each core clock cycle. If INV is clear and the event counts are greater than or equal to this field, the counter is incremented by one. If INV is set and the event counts are less than this field, the counter is incremented by one. Otherwise the counter is not incremented.

Software must read-modify-write or explicitly clear reserved bits.