Data Movement and Workload characterization: Intel Sandy Bridge Core and Uncore PMU features

Tirtha Pratim Bhattacharjee  
Department of Computer Science  
Virginia Tech  
tirtha23@vt.edu

ABSTRACT

This paper provides a detailed on-chip and off-chip data traffic analysis on Intel Sandy Bridge microarchitecture and does workload characterization on a wide spectrum of benchmarks. The paper specifically utilizes the uncore hardware performance counters to generate memory access profile of the evaluated benchmarks. The paper presents a performance comparison across the benchmarks for multi-threaded runs exploiting multiple logical cores and executing instructions in the order of $10^{12}$. The paper also tries to collect application intrinsic profile data like function insight for further research. The paper outlines the scope for further research leading to exhaustive analysis and exploitation of several aspects of the available hardware performance counters with modifications in different performance tools.

Keywords

Intel Sandy Bridge, Uncore Performance analysis

1. INTRODUCTION

The workloads running on our systems are always evolving and subsequently the interdependence of microarchitecture and nature of the application is on the rise. Workload characterization and data traffic analysis is an integral part of performance analysis of computer systems. Performance analysis can be done by identifying the association of processor cycle usage with microoperations issued and their memory access patterns in the context of number and types of instructions as well as data sharing characteristics of the computation. With the advent of parallel computing and the demand of multi-chip processors, these properties influence the design of components at all system levels from the microarchitecture and memory system to compilers and operating environments. The analysis and trending of resource utilizations in the system can influence the performance optimization at software and system level. Such analysis can be beneficial in guiding the usage of this microarchitecture in a high computation platform responsible for a specific nature of workload.

Performance analysis on a microarchitecture is the experimental investigation of the microarchitecture’s response to a set of instructions and corresponding data stream. It’s important that we analyze a broad and diverse portfolio of workloads that represent the usage of the modeled architecture and given data movement. In this paper, the attention is focused on computationally-intensive but variably data-parallel and data-sharing workloads. The benchmarks have been carefully selected to represent the most important and computationally demanding programs. The workload characterizations are not noble individually but it is believed that the results of a study of the performance properties of this set of benchmarks on the evaluated microarchitecture i.e. Intel Sandy Bridge using special hardware performance counters make this valuable. This study is also remarkable in its approach to quantify the data-movement profile of this micro-architecture and unveils some of the significantly inherent cache and memory characteristics.

There have been several research on performance analysis of microarchitectures [1, 2] and workload characterization separately. There have been active areas of research in memory and cache analysis and characterizing memory access locality in applications. For parallel workloads, the amount and types of data sharing and exchange among the threads have significant implications on the cache coherency traffic and ultimately on the performance of SMP systems. Unlike other related works [3], we tried to focus on exploiting vendor-provided uncore hardware performance counters to validate a new microarchitecture, Intel Sandy Bridge, with a wide portfolio of benchmarks and generate a microarchitecture specific workload characterization. We also intend to join dots in the related on-core and uncore memory traffic with valid performance information in the micro-operations level rather than approximation.

2. INTEL SANDY BRIDGE MICRO-ARCHITECTURE

Sandy Bridge [4] is the microarchitecture introduced by Intel to substitute the older Nehalem microarchitecture. Intel core i7 processors and Intel Xeon 5500 processors constitute the top-line of the processors implementing this microarchitecture. Several upgrades have been included in this architecture, remarkable are those like shared last level cache, two load/store operations per CPU cycle for each memory channel, 32 KB each data and instruction L1 cache with a 256KB L2 cache. Each socket has one to eight cores, which share the L3 cache, a local integrated memory controller and an Intel QuickPath Interconnects. 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. The integrated memory controller enables an enormous increase in memory access bandwidth by separating the cache coherency traffic from the memory access...
traffic. Even the memory control logic can run at processor frequencies and thereby reduce the latency.

2.1. Out of Order Execution Pipeline

A block diagram of the core out of order pipeline is shown in Figure 1. Instructions are decoded into the executable microoperations (uops) and assigned to free resources in the form of Reservation Station (RS), Reorder Buffer (ROB) or load/store buffers. Retirement and writeback of state to visible registers is only done for instructions and uops that are on the correct execution path.

![Figure 1: Block diagram for core out of order pipeline](image1)

2.2. Core and Uncore Memory Subsystems

In the core memory subsystem, data is transferred around the components like registers and caches in cachelines of 64 bytes. The level 1 caches are supported by TLBs with 64 and 128 entries respectively for the 32KB each, data and instruction caches. There is a shared 512 entry second level TLB for the 256 kB L2 cache. Cacheline coherency is maintained in this multi-core multi-socket system through the four state (MESI) cacheline protocol. 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 one 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 L2 through a 16 element superqueue and allocates a line fill buffer. If the line is not found in the L2 CACHE, then it must be retrieved from the uncore.

![Figure 2: Block Diagram for one socket of uncore memory subsystem](image2)

The uncore memory subsystem [5] (as shown in Figure 2) comprises of a shared 8/4 MB last level cache (L3 CACHE), a memory access chipset with 2-3 channels, and a socket interconnection interface (Intel QuickPath Interconnect, QPI) 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 through the processor uncore’s Global Queue (GQ). Each socket has a Caching agent (GQ plus the L3 cache) and a Home agent (the Integrated Memory Controller, IMC). In the case of a L3 hit, GQ snoops other cores for a modified copy before updating. An L3 cache miss results in simultaneous queries for the line from all the Caching Agents and the Home agent to the remote IMC’s through the QPI.

2.3. Core and Uncore Performance Monitoring Unit (PMU)

Each core has its own PMU. There are 3 fixed counters and 4 general counters for each hyper-thread and events are counted on a per thread basis. Sandy Bridge has a significant step-up with a separate PMU unit for the uncore subsystem tightly coupled to core PMU for interrupts. It consists of 8 general counters for the events and one fixed counter for the uncore frequency.

![Figure 3: Block diagram for Intel QPI protocol subsystem](image3)

Uncore PMU is the prime focus of this project as the exploration of these events can draw a clear picture of the off-core traffic. Uncore PMU’s are mainly per socket level and count of events can be extracted from a set of local counter registers. The event bits and the unit mask (umask) bits together identify the raw data counter for a particular event. The performance monitoring in the uncore subsystem can be done in several stages and components (shown in Figure 3) like per-socket uncore, central U-box, caching agent (LLC Coherence engine, Cbo), home agent (coherent/non-coherent memory reads/writes, HA box), IMC (interfacing to DRAM), Power Controller Unit (PCU), QPI box (aggregate of QPI link layer and R3QPI to ring) and R2PCIe box. The registers within Cbo, PCU and U-box can be monitored by accessing MSR and the rest by PCIe device configuration space. Irrespective of address-space difference, the uniformity in programming event code, umask, threshold, edge detect, enable/disable is maintained for the uncore PMU’s.

Cbo events are beneficial in counting the LLC related events like access rates, hit/miss rates and MESI state transitions. Though the events record the aggregate LLC transactions from all the cores in
the socket, there are separate counters for the Cbo instances also. IMC performance counters provide vital information on transaction to DRAM like DRAM clocks in the IMC, CAS Read/write and several memory commands and page related events.

3. EXPERIMENTAL SETUP

We evaluate the Sandy Bridge microarchitecture with a 4-core at one socket, Intel core i7 processor (model 2600) which comprises of an 8MB LLC, 2 channel IMC and an integrated PCIe. The system was running 3.11 Linux kernel from the Debian package. This version of Linux comes with in-built Perf. Intel uncore related perf support is patched in this Linux kernel to get the particular uncore events. The kernel ensures that uncore PMU’s are available under /sys/bus/event_source/devices/ and will have the separate uncore coherence boxes for all physical cores.

3.1. Evaluated Benchmarks

We have evaluated PARSEC [6] benchmark suite and chosen eight appropriate benchmarks with relatively high read/write microoperations. PARSEC provides a wide choice of benchmarks covering different application domains, pipeline/data-parallel models and runtime behaviors. They also provide ample provision for different working sets on parallel models (thread based), suitable for microarchitectural performance analysis and studies. We have also evaluated CoMD [7], Classical molecular dynamics proxy application which provides strong MPI scaling. We have also included STREAM [8], Sustainable Memory Bandwidth in High Performance Computers from HPC benchmark suite in our evaluation for memory bandwidth approximation. The tabular classification of the evaluated benchmarks has been presented in Table 1 on the basis of domain, parallel model and data-sharing/exchange behavior.

The experimental evaluation for memory access properties of these workloads. In highly parallel applications, caching behavior along with the memory coherence traffic that it causes, is as important to examine as the performance of the microarchitecture. Given that the performance of these applications is often governed by cache miss latency and the ability to stream data into the processor, it is logical to deduce that properly placed pre-fetch instructions can have a tremendous effect on application performance. For a particular micro-architecture it’s very essential to examine the sensitivity of the applications to the size of the inlying cache and the cachelines which can be analyses with the proper decomposition of miss-rates at different levels of cache. All the results in the following graphs are normalized to numbers corresponding to 1 trillion (10^{12}) instructions) per application run and the benchmarks are run with 4 threads on the 4 cores (if not stated otherwise).

### Table 1: Benchmark classification

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Domain</th>
<th>Parallelization Model</th>
<th>Parallelism</th>
<th>Working Set</th>
<th>DataSharing</th>
<th>DateExchange</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face3D</td>
<td>Animation</td>
<td>Data-parallel</td>
<td>Coarse</td>
<td>Large</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Fenris</td>
<td>Data-mapping</td>
<td>Data-parallel</td>
<td>Medium</td>
<td>Unboundedly high</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>X264</td>
<td>Media Processing</td>
<td>Pipeline</td>
<td>Coarse</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>Data-mapping</td>
<td>Data-parallel</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Fuddsmate</td>
<td>Astronomical</td>
<td>Data-parallel</td>
<td>Fine</td>
<td>Large</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Swaptions</td>
<td>Financial</td>
<td>Data-parallel</td>
<td>Coarse</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>bodytrack</td>
<td>Computer Vision</td>
<td>Data-parallel</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>CoMD</td>
<td>Molecular</td>
<td>Parallel</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>STREAM</td>
<td>Memory-bandwidth</td>
<td>Parallel</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

3.2. Profiling Strategy

We have mainly used Linux-tool perf [9] with libpfm4.3 which has support for the uncore events. Perf can read the raw registers for extracting the counters. Another tool, Likwid [10] has also been evaluated and it poses a huge potential for future extensions in pinning the benchmark threads to particular cores in a multi-socket environment and read the performance counters. We have also evaluated Intel VTune Analyzer [11] in a limited scope and observed an appreciable feature of counters segregation per function level or threads level.

The profiling strategy as adopted in the subsequent sections can be listed as follows:

- Collect the information from the available counters for different sets of inputs for different benchmarks.
- Vary the input size or scale the benchmarks to execute 1 trillion (10^{12}) instructions per run.
- Run the benchmarks on 1 core or 4 cores or 8 logical cores (with Hyper-Thread enabled) and characterize the on-chip and off-chip data movement.
- Use Perf record/report profiling to identify code regions with CPU/memory intensive functions.

4. EXPERIMENTAL EVALUATION

The performance of computational applications can be highly attributed to the memory access properties of these workloads. In highly parallel applications, caching behavior along with the memory coherence traffic that it causes, is as important to examine as the performance of the microarchitecture. Given that the performance of these applications is often governed by cache miss latency and the ability to stream data into the processor, it is logical to deduce that properly placed pre-fetch instructions can have a tremendous effect on application performance. For a particular micro-architecture it’s very essential to examine the sensitivity of the applications to the size of the inlying cache and the cachelines which can be analyses with the proper decomposition of miss-rates at different levels of cache. All the results in the following graphs are normalized to numbers corresponding to 1 trillion (10^{12}) instructions) per application run and the benchmarks are run with 4 threads on the 4 cores (if not stated otherwise).

4.1. Stall Decomposition

Though there are provision for two instructions to be executed per CPU cycle, the analysis of the performance counters regarding CPU cycles used and the uops issued/retired reveal interesting aspects of the workloads. The stalled-cycles for the front-end and the back-end are also measured. The front-end stalls can be attributed to the branch mispredictions, code generation, invocation of microcode and exceptions. The back-end stalls are crucial to be eradicated as the system waits for the resources and provide application developers, important microarchitecture insight for better programming flow. The decomposition of the stall events to get resource stalls is the primary step to characterize back-end stalls. In figure 4, we have shown the total number of resource stalls, RESOURCESTALLS_ANY and the subsequent breakage into important components like load/store based stalls or Reservation buffer (RS) or Reorder Buffer (ROB) full stalls. We observe that streamcluster exhibits high-order of stalls (close to 180% of CPU cycles) and subsequently we notice that a huge portion is consumed by Load based resource stalls (refer figure 5) and can be beneficial in further memory traffic analysis.

Figure 4: Core statistics like cpu-cycles, resource stalls branch instructions etc., comparison for the benchmarks (normalized for 10^{12} Instructions)
We can also get better idea of the branch instructions and LLC accesses and overall idea of cache references and cache misses (see in figure 5). The cache misses are approximately of the same order of the cache references and attribute to high load. We also observe that the LLC loads are in higher order for almost all the applications which makes the Sandy Bridge architecture better suited for it’s inclusive nature of LLC for the cores in a socket and QPI for the remote packages.

By the virtue of on-chip DRAM control, DRAM paging policy statistics can also be collected from the uncore registers. The data is collected for all the benchmarks for the important events like DRAM channel opens on the three available channels (refer figure 7) and access control commands like read cache agents (refer figure 6) and auto page close. We observe a significant spike in read caching agent DRAM access commands in streamcluster which can be attributed to high DRAM open access channels and are followed by moderate commands in fluidanimate and x264. Whereas auto page close commands prevail in bodytrack and least IMC busy states appears to be in swaptions.

In figure 8, we gain some valuable insight into the LLC miss rates through uncore performance counters. LLC transactions have been significantly high for freqmine and contribute to LLC write misses. This calls for an in-depth analysis of the impact of such off-chip data movement on resource stalls for such applications. We also observe that the highly data-parallel and data-sharing nature attribute to high order of LLC hits from the peer probes (LLC shares of other cores in the package) which again indicates the suitability of inclusive LLC nature of this microarchitecture.

In figure 9, we see two prominent LLC events which are LLC miss write events and LLC Hits from peer probe events. Most of the benchmarks with the exception of streamcluster and x264 have shown ~85 to 98% of LLC transactions to credited to write misses. Whereas streamcluster behaves differently with close to 90% of LLC transactions being peer probe hits which means that the inclusive LLC for the socket helped for a benchmark with highly data-sharing threads for reading inputs.

By the virtue of on-chip DRAM control, DRAM paging policy statistics can also be collected from the uncore registers. The data is collected for all the benchmarks for the important events like DRAM channel opens on the three available channels (refer figure 7) and access control commands like read cache agents (refer figure 6) and auto page close. We observe a significant spike in read caching agent DRAM access commands in streamcluster which can be attributed to high DRAM open access channels and are followed by moderate commands in fluidanimate and x264. Whereas auto page close commands prevail in bodytrack and least IMC busy states appears to be in swaptions.

In figure 8, we gain some valuable insight into the LLC miss rates through uncore performance counters. LLC transactions have been significantly high for freqmine and contribute to LLC write misses. This calls for an in-depth analysis of the impact of such off-chip data movement on resource stalls for such applications. We also observe that the highly data-parallel and data-sharing nature attribute to high order of LLC hits from the peer probes (LLC shares of other cores in the package) which again indicates the suitability of inclusive LLC nature of this microarchitecture.

In the figure 9, we see two prominent LLC events which are LLC miss write events and LLC Hits from peer probe events. Most of the benchmarks with the exception of streamcluster and x264 have shown ~85 to 98% of LLC transactions to credited to write misses. Whereas streamcluster behaves differently with close to 90% of LLC transactions being peer probe hits which means that the inclusive LLC for the socket helped for a benchmark with highly data-sharing threads for reading inputs.
benchmarks exhibit good number of L2 hits (about 50% to 70% of the instruction count), whereas *streamcluster* exhibit whooping 150% L2 hits which can be correlated to the prior results of huge resource-stalls and can be attributed to the nature that the benchmark processes huge chunk of read-points. These precise memory load retired events should be extremely effective at identifying cachelines that result in access contention in threaded applications in a single or multi-socket environment.

**Figure 10:** Statistics for uncore memory load retired instructions at different cache levels, comparison for the benchmarks (normalized for $10^{12}$ Instructions)

### 4.3. Evaluation of Uncore Performance with Multiple Threads

We have evaluated the benchmarks scaling them from 1 thread on one core to 4 threads on 4 physical cores and 8 threads on 8 logical cores (hyper-thread enabled). We have noticed interesting patterns in memory access traffic and the core characteristics. We have not included the detailed analysis of the benchmarks in the interest of space but we will discuss some interesting observations in this section. Benchmark *freqmine* exhibited around 100% increase in the hits from the peer shares of LLC when ran with 8 threads enabled rather than 1 thread. Though the stalled cycles remain the same but it significantly doubled the portion of resource stalls including stalls due to full RS or ROB, especially 150% rise in store related resource stalls. It also exhibited pressure on offcore traffic doubling the DRAM open channels for read operations.

Similarly, benchmarks *streamcluster* and *CoMD* have exhibited interesting change in performance for the off-core traffic with multi-threading. Strong scaling in *CoMD* with 8 threads significantly increased the off-core traffic (specifically memory access) and can be attributed to the nature of the benchmark. On the other hand *streamcluster* have exhibited significant increase in number of hits in LLC transactions when scaled to run with 8 multiple threads.

### 4.4. Analysis with Function Insight

Uncore performance counters were also evaluated to correlate with the insight into the functions called and the loop section calls. The evaluation was mainly done with *perf record* and *perf report*. The observations seem to be interesting and are potential candidates for more advanced research to be intelligently integrated into the workload profile. Such function insight for benchmarks like *streamcluster* have indicated considerable concentration of CPU cycles in one function like *pgain* (~94%). Further analysis of the function micro-operations and instruction types like *movx, add* can give valuable heads-up towards the causal modeling of profile generation. We have evaluated Intel VTune Analyzer [11] in a limited scope and observed an appreciable amount of profile data in this sub-category. The features seem to be promising which were hard to record in Linux perf tool but perf scores high in its excellent and lucid interface to program the counters according to the needs.

## 5. FUTURE WORK

There have been some obvious challenges which will form the basic podium to further our research. We are continuing the work as an endeavor to extend the profiling as well the benchmark portfolio for Sandy Bridge microarchitecture to understand the on-chip and off-chip data movement. Uncore performance events provides an opportunity to peep deep into the hardware intricacies and progress from real counter information rather than approximation of the data-movement modeling. We have faced some issues where the register counters are still returning no count. We intend to adopt a dynamic approach towards the initial programming of the counters and on the fly polling. We also intend to arrange some program injections for the benchmarks where we poll these event counters repetitively or read the msr for the cores at interesting code-points. We are trying to count the events actively depending on msr address space, aim is to script it in perf tool for appropriate scheduling.

We have based this analysis on first-level data collection and we will be extending our work to draw correlation among the events and derive formulae to calculate higher level parameters like off-core traffic, read/write bandwidth, latency caused by cache coherence etc. to characterize the workloads. Though thread level event counters, reading counters inside the program and getting a function level memory traffic analysis are not possible at this time. We also want to extend the characterization to uncover some important behaviors like true execution stalls per thread. A complete breakdown of cacheline traffic due to loads and some overview of total offcore traffic can be achieved from the memory access events with event multiplexing enabled and requires multiple runs. We intend to target loop dominated benchmarks with frequent call sites and analyze the relevant predefined ratios like lost stall cycles, block execution and function call counts, and specific loop related saturation effects.

We will be including more microbenchmarks and kernels in the portfolio of our workload. We have plans to further extend our research on a comparative analysis of data spatial vs. temporal locality of the workloads on this microarchitecture. Such analysis will be helpful in characterizing the behavior of different types of benchmarks in the data locality space on this microarchitecture and conclude if advances in Sandy Bridge architecture support such applications.

## 6. CONCLUSION

In this study we have presented a detailed characterization of computationally intensive workloads with significant data-sharing parallel models based on the performance on the Sandy Bridge microarchitecture. Several aspects were examined thoroughly with focus on the uncore performance events which were newly added in this microarchitecture. Though the techniques in this paper are not noble but exploration of these performance counters on a wide spectrum of workloads and draw conclusion can prove to be valuable in guiding architecture design suited for a set of applications in the HPC environment.

We found that the uncore performance events are capable to generate important profiles on memory access in terms of cacheline traffic due to loads/stores and total offcore traffic. They can guide
us in identifying a micro-operation delivery problem and subsequent resource stalls. On accurate sampling of the address space we can get a valuable profile of the workloads including the address profiles. With exhaustive suite of perf event selection techniques we can draw better correlations of the instruction stalls with on-chip or off-chip data traffic. We intend to expand this characterization work to encompass a larger, more diverse group of applications with microkernels and microbenchmarks for Sandy bridge architecture and justify the behavior in high-end desktops to HPC environment.

7. REFERENCES


