Vol.2019-HPC-170 No.32
2019/7/26

Measurement of Main Memory Bandwidth and Memory Access Latency in Intel Processors

Christian Helm[1,a)] Kenjiro Taura[1,b)]

Abstract: The performance of many HPC workloads is limited by the main memory bandwidth. Measurement of the DRAM bandwidth is crucial for diagnosing such problems. Modern processors come with a sophisticated performance monitoring unit (PMU) that can measure the DRAM bandwidth. In addition, it is also possible to measure the memory access latency with the PMU and infer on memory bandwidth problems. There are several ways to measure the memory bandwidth and the memory access latency. The question arises how those methods differ. The differences between those methods are not easily understood from the documentation. In this paper, we discuss the advantages and disadvantages of the various different methods. We use micro-benchmarks to check the correctness of the measurements on current Intel processors. We discovered a case where the documentation is misleading and will lead to incorrect bandwidth measurement if used naively. Overall, we give recommendations on how the main memory bandwidth should be measured for specific use cases and inform about common pitfalls.

1. Introduction

When an HPC application shows bad performance or bad scalability, the limited DRAM bandwidth on today's systems is a potential bottleneck. If a developer knows their application well, then it might be possible to directly confirm this hypothesis based on the source code. However, in many cases, it is easier to use performance analysis tools to verify whether the DRAM bandwidth is limiting an application's performance. Hardware-assisted measurement methods are used in many tools due to their low overhead. Current Intel processors have a powerful performance monitoring unit (PMU). It has support for instruction sampling, trace recording and performance counters. With performance counters hundreds of different events can be monitored. Among them are different ways to measure the main memory bandwidth. In our previous work [5] we introduced an indirect way to diagnose main memory bandwidth limitations. It is based on measuring the memory access latency and offers some advantages compared to the direct bandwidth measurement. Such as a better attribution to source code and objects. We relied on the instruction sampling latency, but there are different ways to measure the latency using the PMU. Naturally, the question arises how those methods differ from each other and if one of them is superior compared to the others. This question is not easy to answer because there is little documentation. Experimental evaluation is required to check if a certain counter actually reports the intended event.

There are tools and libraries to access the PMU from the user space and to program it for performance analysis. The user of such tools has to choose what to monitor and how to interpret the results. For example, such tools include Linux Perf [1], PAPI [2], Intel Performance Counter Monitor [3] and Likwid [4]. They do not solve the above-mentioned problems but are only a low-level interface to the PMU. In this study, we use only Linux Perf and the event names used in this paper follow the convention of Perf.

To address these issues we make the following contributions. We examine different ways of measuring bandwidth and give recommendations about their usage. Our results are based on experiments and a summary of documented characteristics of the different methods. We also show the best way to measure latency for our previously published approach to identify bandwidth saturation. Our work is limited to Intel Xeon processors with Haswell or Broadwell architecture.

2. Hardware PMU Features

Modern PMUs offer different ways to do performance measurements. Out of those, the following two offer ways to measure latency and bandwidth.

2.1 Performance Counters

Performance counters count the occurrence of specific events. In the PMU there is a small number of physical counters. There are hundreds of events that can be programmed to be counted on a specific physical counter. Performance counters cannot be accurately attributed to specific code

[1] The University of Tokyo
[a)] christian@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp

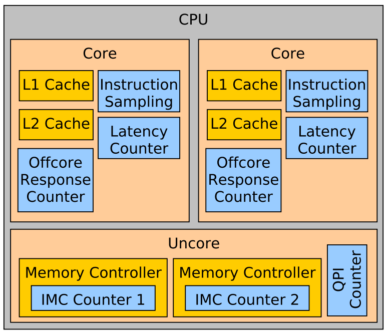

Fig. 1 The location of different bandwidth and latency counters within the CPU.

locations. Performance counters can be read at certain intervals, or at certain points in the program. The change in counter value can then be attributed to that interval. But there can be skid between the actual event occurrence and the counter increase. For short intervals it is not known if the counter increase was actually caused in a specific interval. The counters are located in different parts of the processor. For example, a counter can be inside each core or in the uncore part of the processor as shown in Figure 1. Depending on the location, a counter can either be attributed to a specific core or only to the whole processor.

2.2 Instruction sampling

Instruction sampling works by marking an instruction and observing its execution as it goes through the pipeline of the processor. For load instructions, detailed information can be obtained. For example, the load latency, the actual place where the data was found (L1, L2, L3, remote or local DRAM) and the coherency protocol state at the time of access. AMD calls this method Instruction Based Sampling (IBS). Intel calls it Precise Event Based Sampling (PEBS). The overhead of the sampling method is low since there is dedicated hardware for observing the instructions. The advantage of instruction sampling is that it is precisely attributable to instructions and data. Each sample that is taken contains the instruction pointer of the executed instruction. With debug information in the binary it can be resolved back to the exact source code line in the program. In addition, the accessed data address is also given in the sample. With techniques introduced in [5] it is possible to refer back to the object accesses by this instruction.

3. Related Work

There are some tools that provide more value than just low-level access to the PMU. They provide help by choosing the right events and interpreting PMU data.

DR-BW [6] is a tool which can detect remote memory bandwidth contention in NUMA Systems. It is based on machine learning using features extracted from the performance monitoring unit. One of the features that are used is memory access latency. No further details about the metrics are explained in this paper. This approach is limited to the remote DRAM bandwidth contention on NUMA systems.

Intel VTune Amplifier XE is a general purpose profiling tool, but it also has some specialized memory performance features [7]. It has predefined event types for various measurement objectives. The user does not have to select specific events but is also not informed about the limitations of the event types that are used. Main memory bandwidth can be measured and attributed to the source code. This tool cannot make a decision about whether there is bandwidth contention or not. Which level of bandwidth usage is regarded as too high has to be set by the user.

A tool by Weyers et. al. [8] provides a visualization, which is based on the physical hardware. It can show the communication between different nodes in a NUMA system. It uses Likwid as the backend for reading the hardware performance counters. It uses counters from the memory controller and the QPI interface. An attribution to source code locations is not possible.

The original Intel documentation [9] lists all available events. However, the descriptions are very short and to fully understand what a specific event is actually counting it is often required to have some knowledge about the individual event counter and how it is implemented. Some previous studies take a more detailed look at the individual counters.

Eranian [10] introduces the possibilities of PMUs for performance analysis in 2008. Before that, PMUs were mainly used for verification purposes. He introduces ways to measure bus utilization, cache hit rates, NUMA access ratios, and latency measurement. It is based on older processors and is more of an introduction than a detailed discussion of different measurement methods.

Yasin [11] introduces the top-down approach for structured performance analysis. During the explanation of the approach, one can obtain some information about the inner workings of the counters and which counters can be useful for diagnosing specific problems. It does not go into the details of how to identify memory bandwidth limited applications.

A study by Molka et. al. [12] includes experimental evaluation of different counters. Their approach is to use microbenchmarks to stress certain parts of the memory hierarchy and find correlations with performance counters. They report more detailed information than what is available through official documentation. The points discussed in this paper include the transfers between cache levels, the differentiation of memory and latency bound applications. To the best of our knowledge, this is the most comprehensive prior work on this topic. It does not consider instruction sampling.

4. Experiment Setup

All of the experiments were executed on machines running Ubuntu 18.04 and compiled with gcc 7.4. All measurements were done with Linux Perf version 4.17.8. All event names that appear in this paper refer to the mentioned Perf version and hardware. In other Perf versions or hardware environments, the event names can differ. For some analysis and profiling steps, PerfMemPlus [*1] based on the same Perf version was used. All reported numbers are averages from 6 repeated executions.

4.1 Hardware

The details of the machines, which were used for this evaluation are listed in Table 1. We used numactl to limit the execution to one of the available nodes and to its own memory. Migration of OpenMP threads has been disabled for all experiments. The maximum number of threads used for the experiments is equal to the number of physical cores of a processor.

Table 1 Hardware used for the evaluation.

Name	CPU	DRAM
spica	4x E7-8890v4@2.2 Ghz	DDR4@1600Mhz
comet	2x E5-2699v3@2.3 Ghz	DDR4@1847Mhz
arcturus	2x E5-2699v4@2.2 Ghz	DDR4@2400Mhz

4.2 Hardware Prefetcher Control

We want to run our experiments with and without using the hardware prefetchers. The prefetcher of Intel processors can be turned off and on. It can be done on a running system by writing values into machine specific registers [13]. We use the msr-tools to write to the relevant registers using the following commands:

- Disable prefetching: sudo wrmsr –all 0x1a4 15
- Enable prefetching: sudo wrmsr –all 0x1a4 0
- Read the current status: sudo rdmsr 0x1a4

These commands need root accesses on Linux systems

4.3 Benchmarks

In our experiments, we use two different micro-benchmarks, which are described in the following sections.

4.3.1 Memory Read Benchmark

We use the memory read benchmark to issue read request with a varying bandwidth and then measure the resulting latency. The memory read benchmark sequentially reads a large array from memory. This is repeated several times. After issuing four memory read operations a configurable number of NOPs is inserted. Those NOPs are used for regulating the bandwidth and load on the memory system. The array called A consists of records, each with a size of 64 Bytes. Only the first element in a record is accessed. This means that each cache line is only accessed once. The array A is defined as volatile to force a read memory access. The code is shown in Figure 2. When the hardware prefetchers are turned off, this benchmark allows fine regulation of the memory bandwidth as shown in Figure 3. Because there is a steep change in bandwidth between 50 and 55 delay cycles additional data points were added there. When the hardware prefetchers are turned on, no such fine control is possible. Changing the number of threads still allows for a course regulation of memory bandwidth as shown in Figure 4.

```
#pragma omp parallel
for (long s = 0; s < 200; s++) {
  for (long t = 0; t < 10000000; t++) {
    for (int c = 0; c < 4; c++) {
      A[c][t].next;
    }
    for(unsigned long i = 0; i < delayCount;
         i++) {
          asm volatile("nop");
    }
  }
}
```

Fig. 2 Code of the main loop of the memory read benchmark.

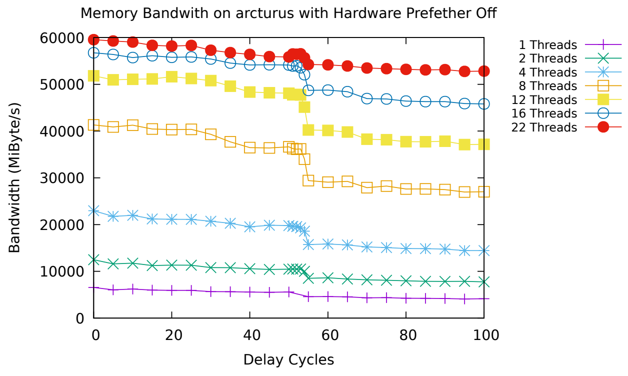

Fig. 3 Main memory bandwidth regulation of the memory read benchmark using threads and delay cycles with hardware prefetchers off.

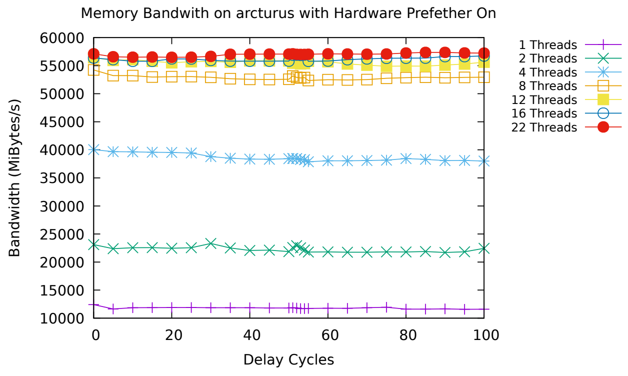

Fig. 4 Main memory bandwidth regulation of the memory read benchmark using threads and delay cycles with hardware prefetchers on.

4.3.2 Stream Triad

We use Stream triad [14] because we can calculate the amount of transferred data. Then we compare it with the measured amount of data transfer to verify the accuracy. Stream triad reads two arrays. Multiplies the value from the first array with a constant scalar and adds another value from the other array. Then it writes the result of this calculation to a third array as shown in Figure 5.

[*1] Available at `github.com/helchr/perfMemPlus`

```
#pragma omp parallel
  for(long i = 0; i < n; i++) {
    c[i] = b[i] + s * a[i]
  }
```

Fig. 5 Code of the main loop of Stream Triad.

All arrays have the same size. The total transferred data volume can be calculated. The array a and b are read. The array c is written. The current Intel architectures use the write allocate scheme, which means that the array c is also read into the caches before it can be written to main memory. We do not use non-temporal stores. Thus, the total read amount of data is three times the array size. It needs to be multiplied by 11 because the benchmark is executed 10 times and for initialization, all arrays are written once. Additionally, the size of the binary is added. We use an array size of of 610.4 MB which results in a total transferred size of 21.974 GB.

5. Direct Bandwidth Measurement

Direct measurement of the memory bandwidth using the PMU is possible. Both methods that we introduce in this section are based on performance counters and have the attribution problem mentioned in Section 2.

5.1 Memory Controller Counters

There are performance counters inside of the memory controller. They are called uncore_imc_*_/cas_count_read and uncore_imc_*/cas_count_write respectively. They count the number of cache lines transferred by the memory controller. There are multiple memory controllers on one chip. In the event name, instead of the asterisk, the number of the memory controller can be used. When using the asterisk the values of all memory controllers are added. To get the number of bytes transferred this value has to be multiplied by 64. Recent versions of Perf already do this for the user and print the data in MiByte. The limitation of these counters is that they can only count in global mode. It is activated in Perf with the –all-cpus flag. That means that they count everything, including memory traffic caused by other applications and the operating system. This introduces additional sources of noise to the measurement. Because those counters measure the whole system, which could allow gathering information about other running applications, extended privileges are required. Either the perf_event_paranoid flag must be set to -3 or root access is required. This may hinder usage on shared systems. Because all cores of a processor share the same memory controllers they also count for one whole socket. This makes attribution to code and data even more difficult because the traffic cannot be attributed to a specific core.

To measure the traffic to remote memories the counters of the QPI interface can be used. This counter is called qpi_data_bandwidth_tx. Similar to the memory controller counters they count all the data that goes in and out of the QPI interface.

The –per-socket flag can be used to gather statistics for every socket individually. For example, the memory bandwidth usage of each socket can be used to diagnose unbalanced usage of the memory in NUMA systems.

5.2 Offcore Response Counters

Another method for bandwidth detection are the offcore counters. Those counters are located at the edge between the core and uncore part of the processor as shown in Figure 1. This means that they can be attributed to specific cores, but still have the attribution problem that is common for all performance counters as explained in Section 2. Because the offcore counters are core local and can be restricted to the profiled application it is possible to use them even on systems with restricted access. Only the bandwidth of the profiled application will be counted. It leads to lower noise in the measurements.

There are different sub-events for the offcore_response events. Their format is always like offocre_response.<access type>.<llc hit or miss>.<response type>. The access type can be further split up in two parts. The first part select either demand of prefetched accesses. The second part select reads, writes, code reads or all accesses. The LLC (Last Level Cache) hit or miss section specifies whether LLC hits or dram hits should be included. The response type can be local DRAM, remote DRAM or various cache coherence dependent options as well as an option to include all responses. Overall, those offcore events allow a very fine selection of specific events. The event we used for the experiments is offcore_response.all_reads.llc_miss.local_dram. According to the documentation, it counts all read accesses (including code reads and reads required for later writes) to the local dram no matter if they are demand or prefetch.

5.3 Bandwidth Experiment Results

To verify whether the bandwidth measurement is accurate we use the Stream benchmark. Figure 6 shows the results. First, we can see that the results are similar in all three systems.

The IMC counters accurately measure the amount of transferred data. The calculated data volume is a little lower because only the part of the main loop in Stream Triad and the initialization add to the total amount of transferred data. The IMC counter counts every memory transfer that happens while the benchmark is running. The IMC counter has higher noise because it not only counts the applications data transfers but all data transfers that occur while the program is running. This includes data transfers due to the operating system and other applications running at the same time.

The offcore counter is limited to the program under test but still counts other memory accesses that occur in the program and thus show a transferred data volume that is slightly higher than the calculated one. The documentation says that prefetched accesses are included [9, Vol 3B p. 18-41] when using this counter. However, our experiments show that the data transferred because of prefetching is not included. When prefetching is turned off the data is the same as with the other counters or the calculated data amount. In contrast, when prefetching is turned on the offcore counters do not report the correct data volume.

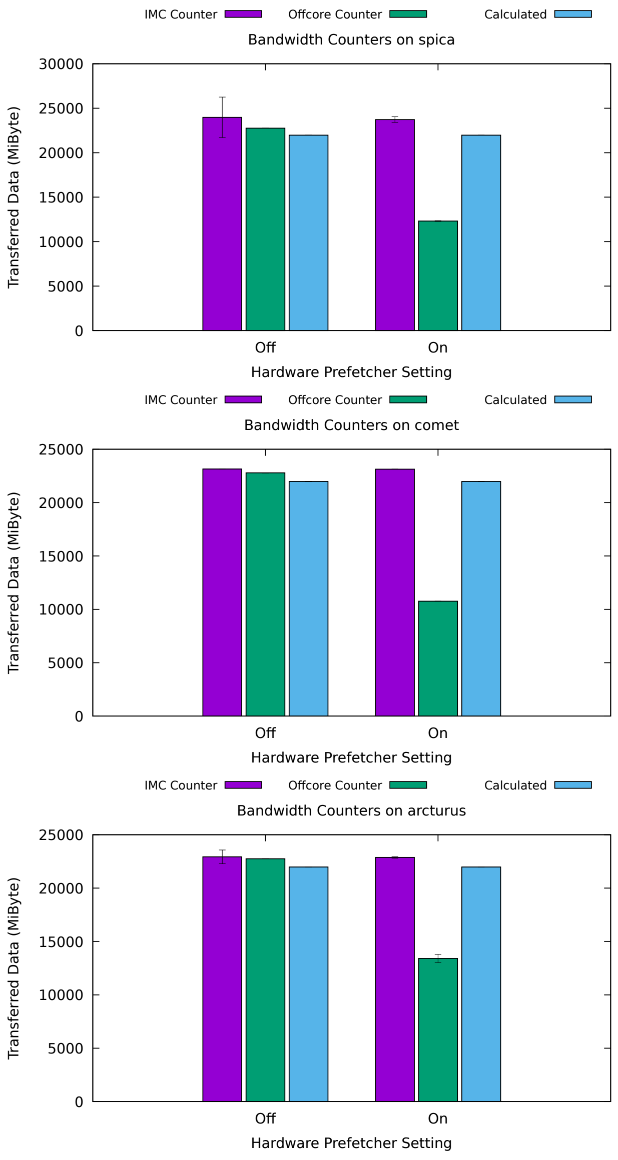

Fig. 6 Transferred data volume in the Stream Triad benchmark measured with offcore and IMC counters compared to the theoretical transferred data. The green bar shows that the offcore counter does not include all of the transferred data when the prefetchers are turned on.

6. Latency Measurement

In our previous publication [5] we introduce a method for detection of main memory bandwidth limitation through latency. We use the latency of load instructions that hit the local DRAM as an indicator for memory bandwidth saturation. The basic idea is that loading data from a memory can be done with a fixed latency. If other issues, like bandwidth saturation, occur the load request is delayed and the total latency to complete the load instruction increases. We observed that the latency stays low with only a small increase until the bandwidth gets close to the hardware limit. At this point, when the system reaches its throughput limit, there is a sharp increase in latency. This relationship is well known in queuing theory. When the arrival rate (bandwidth requirement of the application) is higher than what the system can process in a certain time (maximum hardware memory bandwidth) the time required for queuing and processing (latency) of the requests will increase.

There are different methods to measure the latency. Previously we relied on the instruction sampling latency. In this paper, we also consider a performance counter based measurement.

6.1 Instruction Sampling

The memory access latency is one of the attributes of a sample. This latency is the time from the start of the execution of an instruction until it reaches the globally observable state [9, Vol. 3B p. 18-22]. For the calculation of the latency, only those samples which hit in the DRAM and that hit in the TLB are included. Out of all the those filtered samples, the average latency is used. The instruction sampling latency has the most precise attribution to code and data. Only data from the profiled application is collected and no special privileges are required to use this profiling method. We use PerfMemPlus to do the profiling and extract the data. It is pre-configured to use instruction sampling.

6.2 Performance Counters

Another way to measure the latency is to use performance counters. The counter cpu/l1d_pend_miss.pending counts the time spent for loading data into the L1 cache. It sums up the time for parallel accesses. In other words, this counter is increased by the number of currently outstanding L1 data cache misses in every cycle.

In addition the counters cpu/mem_load_uops_retired.l1_miss and cpu/mem_load_uops_retired.hit_lfb are required. The first one counts the number of load instructions that miss the L1 cache. The second one counts the number of load instructions that hit the line fill buffer (LFB).

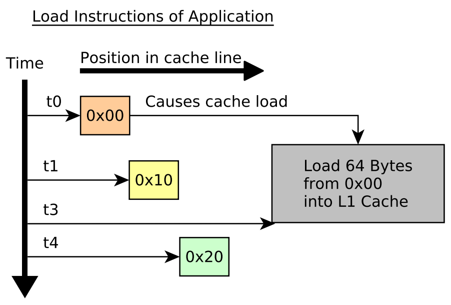
Load Instructions of Application
Time
Position in cache line
t0
0x00
Causes cache load
t1
0x10
Load 64 Bytes from 0x00 into L1 Cache
t3
t4
0x20

Fig. 7 A sequence of memory access requests within the same cache line and their cache hit status. Red = Miss, Yellow = TLB Hit, Green = Hit.

The Intel PMU differentiates between an L1 miss and an LFB Hit. An LFB Hit is an L1 miss, where the requested data is already in the progress of being loaded into the L1 cache. This can either be because the data shares the same cache line or due to prefetching. An example is illustrated in Figure 7. The first load instruction, drawn as a red box, to the address 0x00 accesses the first element of a cache line. Because it is accessed for the first time, it causes an L1 miss. The L1 cache will then start to load a complete cache line of 64 bytes into the L1 cache (gray box). While this load is in progress another load (yellow box) to the address 0x10 is issued by the application. Because the load for the required data is already in progress, this access will be categorized as LFB hit. After the load of a complete cache line into the L1 cache is completed at t_3 another load (green box) to a different element in the same cache line is issued. It will be an L1 cache hit because the complete cache line is already in the L1 cache.

All required counters are part of the core and L1 cache. They can be attributed to specific cores and data recording can be limited to the profiled application. For abbreviation of the long event names the following Greek letters will be used in the metric definition:

$$\alpha = \text{cpu/l1d\_pend\_miss.pending}$$
$$\beta = \text{cpu/mem\_load\_uops\_retired.l1\_miss}$$
$$\gamma = \text{cpu/mem\_load\_uops\_retired.hit\_lfb}$$

Based on those counters two latency metrics are defined as follows:

$$\text{L1 miss latency} = \frac{\alpha}{\beta}$$

$$\text{load miss real latency} = \frac{\alpha}{\beta + \gamma}$$

The l1 miss latency expresses the average time it takes to fulfill a load request that missed the l1 cache. It does not include LFB hits. The load miss real latency is the average time it takes to fulfill a load request that missed the l1 cache or hit in LFB.

6.3 Latency Experiment Results

The results of the experiments with disabled hardware prefetcher are shown in Figure 8. It shows that in some situations the sampling latency can rise high even though the bandwidth is far from the maximum of the system. Just relying on this latency value as an indicator for bandwidth saturation will lead to false positive errors. In contrast, the L1 miss latency and load miss real latency only increase when there is an actual bandwidth limitation. The l1 miss latency and load miss real latency are almost the same in this configuration. This is expected because there will be very few LFB Hits because there is no prefetching and only one element per cache line is accessed. On comet, the system with Haswell architecture, the absolute value of the l1 miss latency and load miss real latency is much higher than on arcturus and spica, which are Broadwell systems.

When the hardware prefetcher is turned on the load miss real latency depends on the number of delay cycles which is visible in Figure 9. Even though the bandwidth is almost the same when the number of delay cycles is low, the latency is higher. This is an advantageous property of the

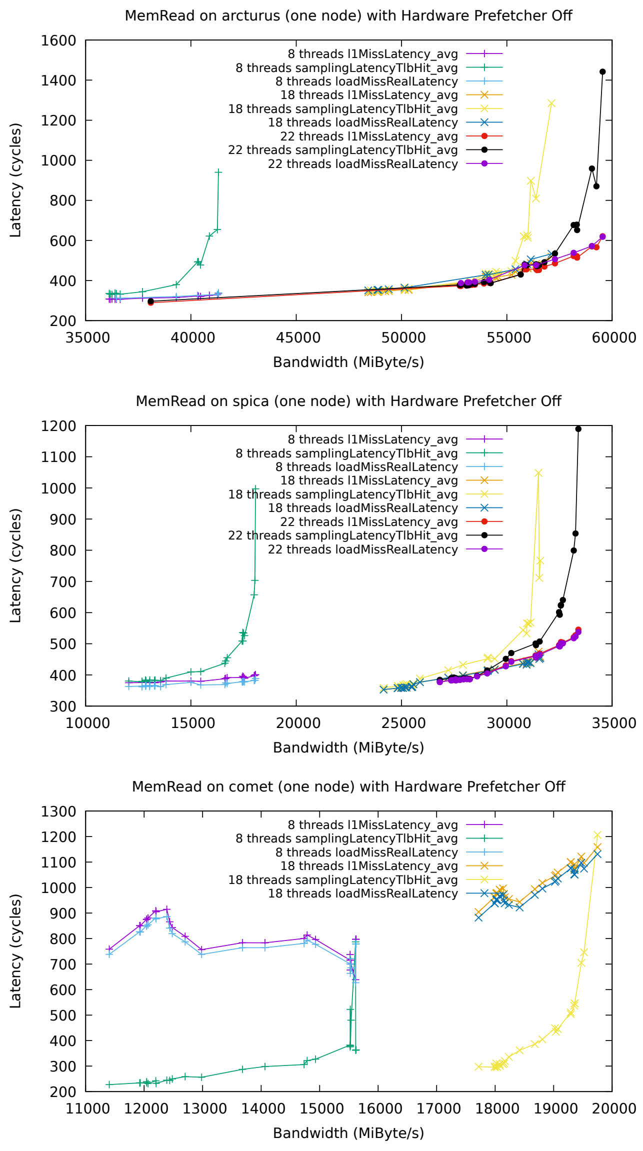

Fig. 8 Latency development of different measurement methods with increasing DRAM bandwidth when hardware prefetchers are disabled. A selection of threads was picked for this diagram to indicate low and high bandwidth situations.

latency measurement, compared to the bandwidth measurement. Because of the prefetcher, the main memory bandwidth is the same no matter how many delay cycles are inserted. However, in cases with a low number of delay cycles, the data is needed quickly. In these situations, the limited main memory bandwidth is worse for performance. Compared to situations with a high number of delay cycles, where the prefetcher has enough time to load the required data.

When the prefetcher is turned on there will be many hits in the LFB and only very few real L1 cache misses. As shown in Figure 10, the l1 miss latency is much higher than the load miss real latency. A high number of delay cycles results in a high l1 miss latency. This is because with a high number of delay cycles the amount of real L1 cache misses will decrease, and thus increase the l1 miss latency.

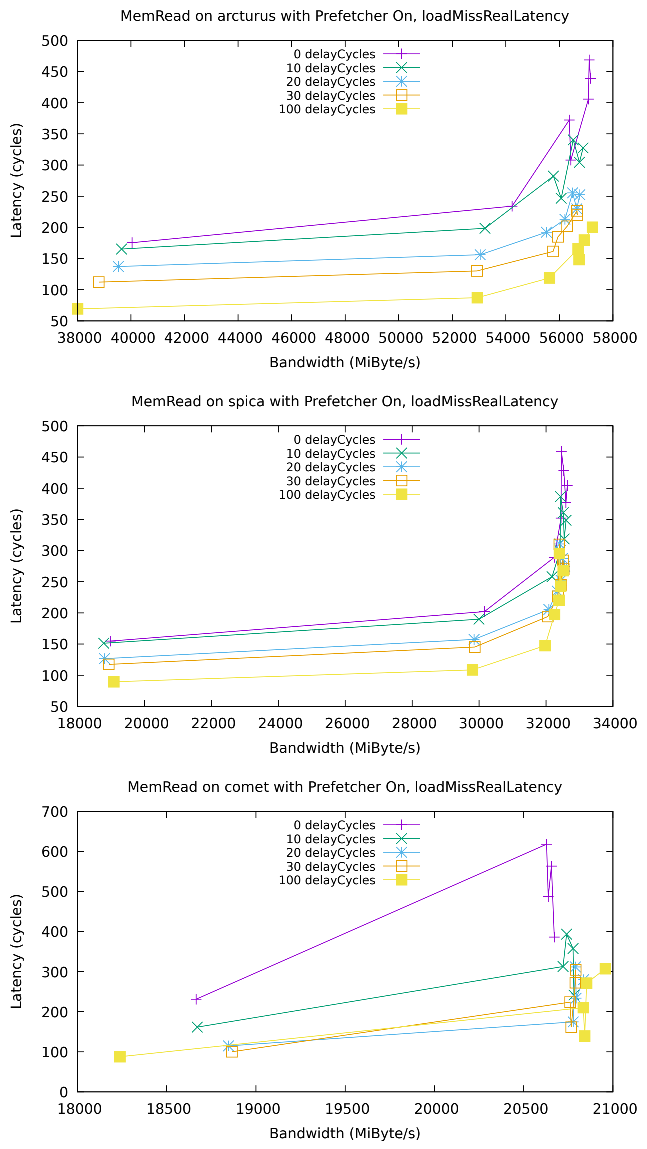

Fig. 9 Load miss real latency development with increasing DRAM bandwidth when hardware prefetchers are disabled. The number of delay cycles has an influence on the measured latency even when the bandwidth is the same.

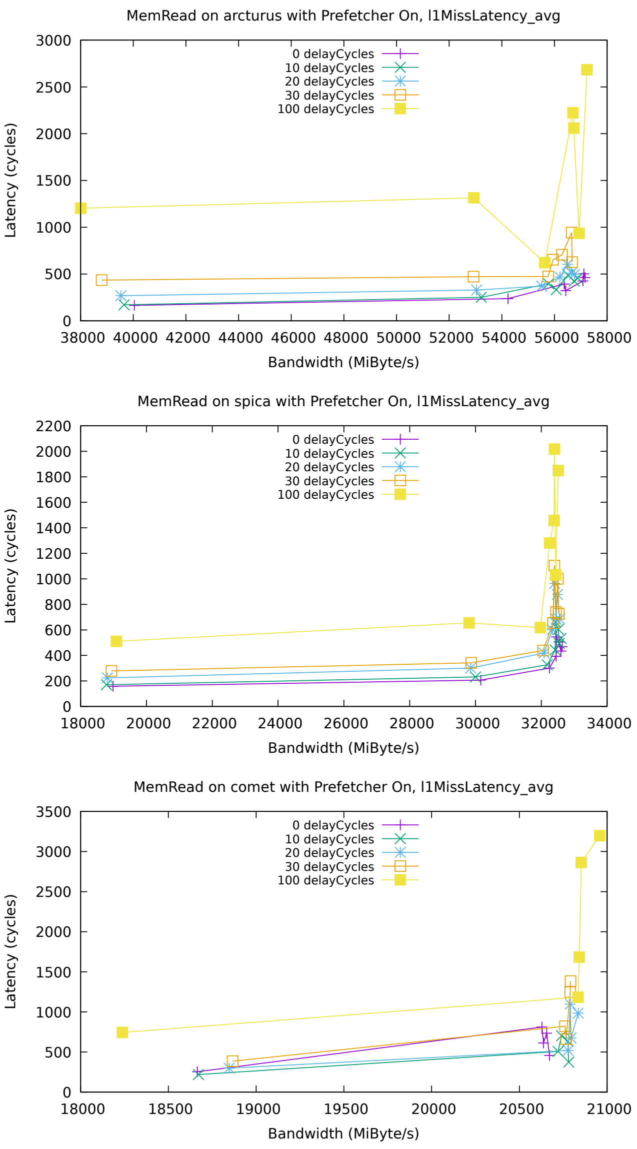

Fig. 10 L1 miss latency development with increasing DRAM bandwidth when hardware prefetchers are disabled. The number of delay cycles has an influence on the measured latency even when the bandwidth is the same.

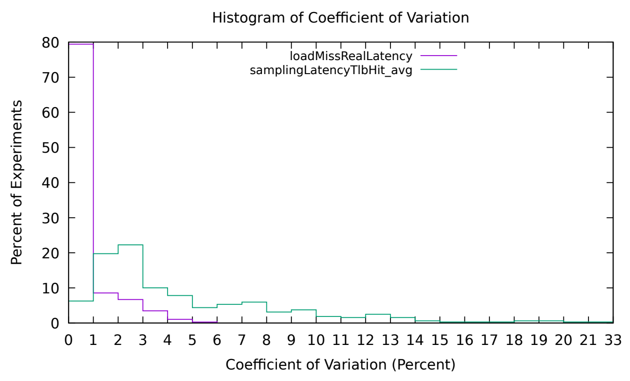

Fig. 11 Histogram showing the coefficient of variance of all experiments.

The error of the load miss real latency measurement is low. In all of the 374 different configurations the maximum coefficient of variation is at most 6.62% and on average 0.68%. The sampling latency measurement shows a higher coefficient of variation. The average is 4.88% and the maximum is 33.17% The distribution is shown in Figure 11. There is no correlation of the measurement accuracy to any of the benchmark parameters. Such as, number of threads, delay cycles, prefetcher settings or hardware system.

7. Conclusion and Future Work

In this paper, we show that depending on conditions, such as access privileges to the system and purpose of the measurement different measurement methods should be used for main memory bandwidth analysis.

If the exact value of bandwidth is of interest the direct bandwidth measurement method should be applied. If privileged access is available and other parts of the system are also under control for the time of the profiling we recommend to use the IMC counters. Otherwise, the offcore counters can be used. But one must be aware of the excluded prefetcher accesses.

For diagnosing bandwidth problems we recommend the latency based methods. Because they allow better attribution of high bandwidth situation to code and data. Additionally, they can distinguish performance degrading high bandwidth situations from harmless high bandwidth situations, that the prefetcher can easily produce.

To improve the latency based detection, that we have proposed in our previous work, we propose the following hybrid instruction sampling and performance counter approach. The load miss real latency is calculated for short intervals of the execution. If the load miss real latency is above a threshold in that interval it is flagged as bandwidth bound. Within this interval, all samples with high latency are selected. Those samples identify the code and data that suffer from the bandwidth limitation. This approach combines the reliability of the load miss real latency with the accuracy of the instruction sampling. We want to implement this approach in an automated detection tool.

References

[1] Linux: Perf, `https://perf.wiki.kernel.org/index.php/Main_Page`.

[2] Terpstra, D., Jagode, H., You, H. and Dongarra, J.: Collecting Performance Data with PAPI-C, *Tools for High Performance Computing* (2010).

[3] Intel Corporation: Performance Counter Monitor, `http://www.intel.com/software/pcm` (2016).

[4] Treibig, J., Hager, G. and Wellein, G.: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments, *Proceedings of the International Conference on Parallel Processing Workshops*, pp. 207–216 (online), DOI: 10.1109/ICPPW.2010.38 (2010).

[5] Helm, C. and Taura, K.: PerfMemPlus : A Tool for Automatic Discovery of Memory Performance Problems, *34th International Conference, ISC High Performance* (2019).

[6] Xu, H., Wen, S., Gimenez, A., Gamblin, T. and Liu, X.: DR-BW: Identifying Bandwidth Contention in NUMA Architectures with Supervised Learning, *IEEE International Parallel and Distributed Processing Symposium, IPDPS* (2017).

[7] Intel Corporation: Finding your memory access performance bottlenecks, `https://software.intel.com/en-us/articles/finding-your-memory-access-performance-bottlenecks` (2016).

[8] Weyers, B., Terboven, C., Schmidl, D., Herber, J., Kuhlen, T. W., Müller, M. S., Hentshel, B., Muller, M. S., Hentschel, B., Müller, M. S. and Hentshel, B.: Visualization of Memory Access Behavior on Hierarchical NUMA Architectures, *Proceedings of VPA 2014: 1st Workshop on Visual Performance Analysis - held in conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 42–49 (online), DOI: 10.1109/VPA.2014.12 (2015).

[9] Intel Croporation: Intel 64 and IA-32 Architectures Software Developer ' s Manual, Vol. 3 (online), DOI: 10.1109/MAHC.2010.22.

[10] Eranian, S.: What can performance counters do for memory subsystem analysis?, *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, No. March, p. 26 (online), DOI: 10.1145/1353522.1353531 (2008).

[11] Yasin, A.: A Top-Down Method for Performance Analysis and Counters Architecture, *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 35–44 (2014).

[12] Molka, D., Schöne, R., Hackenberg, D. and Nagel, W. E.: Detecting Memory-Boundedness with Hardware Performance Counters, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE '17*, pp. 27–38 (online), DOI: 10.1145/3030207.3030223 (2017).

[13] Intel Corporation: Disclosure of H/W prefetcher control on some Intel processors, `https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors` (2014).

[14] McCalpin, John D: STREAM benchmark, `http://www.cs.virginia.edu/stream/` (1995).