On the Correct Measurement of Application Memory Bandwidth and Memory Access Latency

Christian Helm
The University of Tokyo
Tokyo, Japan
christian@eidos.ic.i.u-tokyo.ac.jp

Kenjiro Taura
The University of Tokyo
Tokyo, Japan
tau@eidos.ic.i.u-tokyo.ac.jp

ABSTRACT

Diagnosing if an application suffers from DRAM contention can be a challenging task. One method is to compare the hardware memory bandwidth limit with the measured memory bandwidth of an application. Another method is based on memory access latency. The latency of a DRAM access in an uncontended state is a hardware characteristic. If an application shows higher DRAM access latency, the increase comes from queuing delays and the application is limited by DRAM bandwidth. Hardware-based measurement of the application's latency and bandwidth can be done with low-overhead and is agnostic of the application's implementation.

But the practical implementation of such a diagnosis system on CPUs is difficult. In modern CPUs, there is an abundance of performance counters and only superficial documentation. Different types of counters for bandwidth or latency, that seemingly measure the same thing, produce different results. There is no in-depth understanding of those performance counters and naive usage may lead to incorrect measurements. Because there is no hardware feature to measure DRAM access latency directly, the implementation of the above-mentioned latency based method may seem impossible.

In this paper, we compare various hardware latency and bandwidth measurement methods on CPUs by using micro-benchmarks. We show results of Intel Haswell, Broadwell and Skylake systems. With our experiments, we show how and why performance counters for bandwidth and latency differ. Only the counters inside of the memory controller correctly measure bandwidth. Latency measured by instruction sampling is suitable to find DRAM contention, even though it is not a pure DRAM access latency.

KEYWORDS

Performance Analysis, Performance Monitoring Unit, Memory Bandwidth, DRAM, Memory Latency

ACM Reference Format:
Christian Helm and Kenjiro Taura. 2020. On the Correct Measurement of Application Memory Bandwidth and Memory Access Latency. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020), January 15–17, 2020, Fukuoka, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3368474.3368476

1 INTRODUCTION

When an HPC application shows bad performance or bad scalability, the limited DRAM bandwidth on today's systems is a potential bottleneck. Often HPC users want to confirm whether their application suffers from the limited DRAM bandwidth of the machine to decide on the optimization direction. Using the performance monitoring unit (PMU) of a processor is an approach that can be applied to any application, regardless of how it is implemented. Hardware-based methods also have low overhead [2].

One approach to find out if an application's bandwidth requirements exceed the hardware limit is based on DRAM bandwidth measurement. The maximum DRAM bandwidth is a machine-specific value that can be determined easily with micro-benchmarks such as Stream [12]. The application's bandwidth can be measured using the PMU. A comparison of the application's bandwidth and the machine limit can reveal a potential bandwidth boundness of the application.

A second approach is based on the memory access latency. The constant hardware characteristic is the latency of a DRAM access in the uncontended state. Loading data from a memory can be done with a fixed latency. If bandwidth saturation occurs, the load request is delayed and the total latency to complete the load instruction increases. This relationship is 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.

The reason why we look at two different methods for diagnosis of memory bandwidth limitations in applications is that hardware measurement options differ between the two. Latency measurement and direct bandwidth measurement have different trade-offs that depend on their hardware implementation. We also discuss those trade-offs in this paper.

Other performance models, like the roofline model [18], the cache line aware roofline model [7] or the ECM model [15] can, in theory, be applied by analyzing the source code and using pen and paper. But real applications are too complex and practical tools rely on performance counters [1, 5] to characterize the applications.

Reliable measurements are the foundation for more advanced performance analysis. As researchers, who work on such performance models and their practical implementation on CPUs, we want to understand exactly what is measured by certain hardware counters. Otherwise, our models will be based on unreliable data.

Memory accesses are issued directly from the code as demand requests or they can be issued by hardware prefetchers [3]. Consideration and differentiation of both are important to get the correct understanding of the memory bandwidth.

Current Intel processors have a powerful performance monitoring unit (PMU) [10, Chapter 18]. It has support for instruction sampling and performance counters. With performance counters, hundreds of different events can be monitored. Among them are different ways to measure the main memory bandwidth and different ways to measure the memory access latency. 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. What those counters really measure and how they are implemented is unknown to the public. Experimental evaluation is required to check if a certain counter reports the intended event accurately.

The latency based concept described above assumes that the latency is the pure DRAM access latency. That means the measured latency is based only on accesses to the DRAM, not access to any other cache levels. And that the latency is measured from the beginning of the request to the memory until the completion of the request. Other parts of the instruction execution, for example, waiting time until an execution port becomes available would also influence the results. Thus, to apply the latency based detection method an exact understanding of what is measured by the hardware latency counters is required.

We use experiments with micro-benchmarks to find out what is actually measured by different PMU based methods. We explain and examine the various trade-offs that come with different PMU measurement methods. Our conclusion shows that there is only one way to correctly measure an application's bandwidth and that other methods do not include accesses caused by the hardware prefetchers. For latency measurement, we show that the instruction sampling latency is not the pure DRAM access latency, because it includes delays that come from the in-core processing of the instruction. Nevertheless, our experiments show that it is suitable for the detection of bandwidth boundness and that it has better attribution to program locations compared to bandwidth measurement and can consider the effects of prefetching. In this paper, we show concrete results for Intel Skylake, Intel Broadwell and Intel Haswell architectures but our method to verify the performance counters can be applied to any type of processor.

2 PERFORMANCE MONITORING UNIT

This section explains basics necessary to understand details of PMU-based measurement. Performance counters and instruction sampling are two fundamentally different ways to measure latency or bandwidth using the PMU. The hardware PMU needs to be programmed. It can be done with existing tools. L1 cache misses are recorded as real L1 cache misses or as line fill buffer hits.

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. Among them are counters for the transferred amount of data and a set of counters that can be used to calculate the memory access latency.

Performance counters can be read in intervals, or at certain points in the program. The change in counter value can then be attributed to that interval. Every time a counter is read, additional overhead is introduced. There can be skid between the actual event occurrence and the counter increase. Thus, attribution to specific points in the application is imprecise. Events can be attributed to functions but not to individual source code lines.

2.2 Instruction Sampling

The instruction sampling hardware marks an instruction and observes its execution as it goes through the pipeline of the processor. 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 [6] it is possible to refer back to the object accesses by this instruction. 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. Direct bandwidth measurement is not possible with instruction sampling. AMD calls this method Instruction Based Sampling (IBS). Intel calls it Precise Event Based Sampling (PEBS).

2.3 PMU Programming

There are tools and libraries to access the PMU from the user space and to program it for performance analysis. They also take care of reading the results from the hardware and storing or printing them for later analysis. The user of such tools has to choose what to monitor and how to interpret the results. They do not provide help with choosing the right measurement methods. For example, such tools include Linux Perf [11], PAPI [14] and Likwid [16]. In this study, we use only Linux Perf and the performance event names used in this paper follow the convention of Perf. We use Perf because it is easy to install and run on Linux systems, no source code modifications are necessary for profiling, and the data export feature allows us to implement our own data analysis methods.

2.4 Cache Miss Classification

Processors are typically equipped with a Line Fill Buffer (LFB). It is placed after the L1 cache and before the L2 cache. It holds the outstanding load requests that missed the L1 cache and are being processed by the memory sub-system.

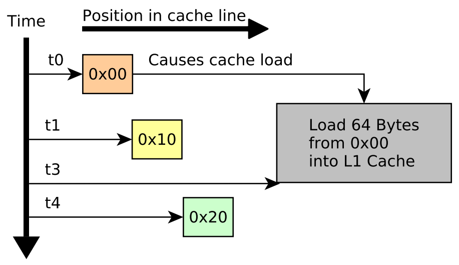

Figure 1: A sequence of memory access requests within the same cache line and their cache hit status. Red = L1 Miss, Yellow = LFB Hit, Green = L1 Hit.

Intel PMUs differentiate 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 1. 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.

3 RELATED WORK

Some tools can do more than just low-level access to the PMU. They provide help by choosing the right events and interpreting PMU data.

DR-BW [19] 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 those features is the 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 [9]. 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 decide 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. [17] 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 [10, Chapter 18] 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 [4] 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 [20] 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. [13] includes experimental evaluation of different counters. Their approach is to use micro-benchmarks 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.

Because there is no detailed information available about the bandwidth and latency measurement methods that we are interested in, we conducted new experiments and present the results in this paper.

4 EXPERIMENT SETUP

Because the existing documentation of performance counters does not provide enough details to clearly understand them we have conducted experiments. We used two different micro-benchmarks with known behavior to check the results of different hardware counters. We compared the results on four different server systems. The setup is explained in this section.

4.1 Software

All of the experiments were executed on machines running Ubuntu 18.04 and the micro-benchmarks were compiled with gcc 7.4. All measurements were done with Linux Perf version 4.17.8. Perf supports multiplexing of events if there are not enough physical counters. We do not use this feature. In our experiments events are always pinned to a physical counter for highest accuracy. For experiments that involve instruction sampling, PerfMemPlus [1] based on the same Perf version was used. All reported numbers are averages from 6 repeated executions. Migration of OpenMP threads has been disabled for all experiments.

4.2 Hardware

We conducted the experiments on the last three Intel server-grade architectures and machines with different DRAM speed. We used one Haswell, two Broadwell, and one Skylake machine. The two Broadwell systems have a big difference in DRAM speed. The details of the machines are listed in Table 1. We used numactl to limit the execution to one of the available nodes and its local memory.

Experiments were conducted with activated and deactivated hardware prefetchers. Prefetcher control was done as described by Intel in [8].

4.3 Micro-Benchmarks

In our experiments, we use two different micro-benchmarks.

[1] Available at https://github.com/helchr/perfMemPlus

Table 1: Hardware used for the evaluation.

Name	Architecture	CPU	DRAM Speed
comet	Haswell	E5-2699v3@2.3Ghz	1847Mhz
spica	Broadwell	E7-8890v4@2.2Ghz	1600Mhz
arcturus	Broadwell	E5-2699v4@2.2Ghz	2400Mhz
rigel	Skylake	Xeon 8176@2.1Ghz	2666Mhz

4.3.1 Memory Read Benchmark. We use the memory read benchmark to issue read request with varying bandwidth and then measure the resulting latency. The memory read benchmark sequentially reads a large array from memory. The hardware prefetchers can predict the access pattern and load data. 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. We only show the graphs for one system because other systems show similar results.

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

Figure 2: Code of the main loop of the memory read benchmark. Each read access is followed by a configurable number of NOPs.

4.3.2 Stream Triad. We use Stream triad [12] because we can calculate the amount of transferred data. Then we compare it with the measured amount of data transfer to verify the accuracy of bandwidth counters. All arrays of the code in Figure 5 have the same size. 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

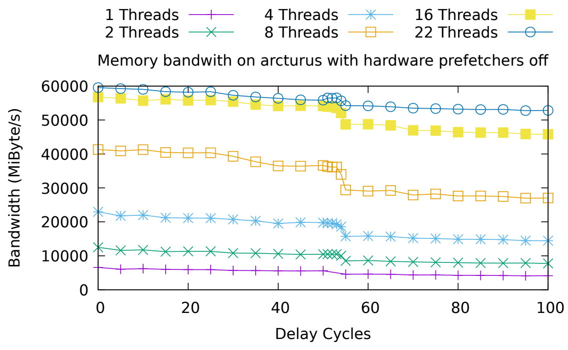

Figure 3: Main memory bandwidth regulation of the memory read benchmark with hardware prefetchers off. Fine control of bandwidth is possible by adjusting the number of delay cycles.

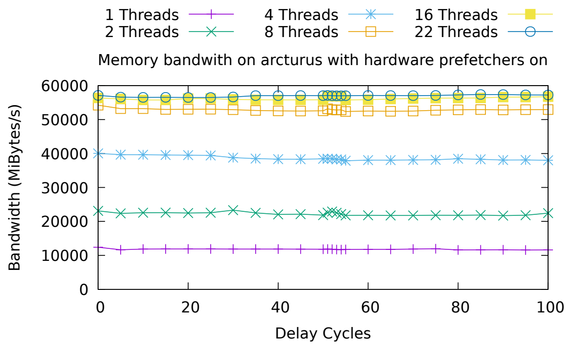

Figure 4: Main memory bandwidth regulation of the memory read benchmark with hardware prefetchers on. The number of delay cycles has no influence on the bandwidth.

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

Figure 5: Code of the main loop of Stream Triad. The arrays a and b are read. The array c is written.

and for initialization, all arrays are written once. Additionally, the size of the binary is added. We use an array size 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. All the methods that we introduce in this section are based on performance counters. Thus attribution to source code and objects is not as good as with the instruction sampling based approaches. Based on the documentation we found the following performance counters that involve DRAM accesses or DRAM bandwidth:

(1) mem_load_uops_l3_miss_retired.local_dram
mem_load_l3_miss_retired.local_dram
(2) unc_h_requests.reads
(3) uncore_imc_*/cas_count_read/
(4) offcore_response.*

The first counter in the list counts the number of uops that load data from the local DRAM. It is not suitable for measuring the DRAM bandwidth. First, a load uop can have different data widths. There is no information about the data size of the load given. Second, loads that are classified as LFB hit, but the data actually comes from the DRAM are not included in this counter.

Counter number 2 and 3 are very similar. Both are in the uncore part of the CPU. They produce similar results and share the same advantages and disadvantages. The difference between them is that the unc_h_requests.reads counter is inside of the home agent and the uncore_imc_*/cas_count_read/ counter is in the memory controller. The offcore_response counters also look like a viable candidate. We focus our detailed evaluation on the memory controller counters and offcore response counters.

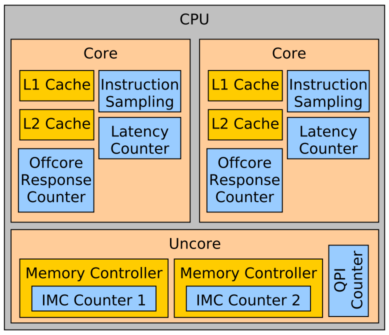

Figure 6: The location of different bandwidth and latency counters within the CPU. Uncore counters have more limitations compared to core local counters.

5.1 Memory Controller Counters

The counter inside of the integrated memory controller (IMC) is called uncore_imc_*/cas_count_read. It counts the number of cache lines transferred by the memory controller from the DRAM to the processor. There are similar counters for written cachelines called uncore_imc_*/cas_count_write.

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 Perf, an asterisk instead of the number will activate counting on all memory controllers and 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.

Because the counters are located in the uncore part of the CPU (Refer to Figure 6) 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 -1 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.

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 option to count the amount of transferred data are the offcore response counters. Those counters are located at the edge between the core and the uncore part of the processor as shown in Figure 6. 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 response 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. It is possible to select either demand accesses, prefetched accesses or all accesses. Memory reads, writes, code reads or all accesses can also be configured. The response type can be configured. It 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 on Haswell and Broadwell 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.

On Skylake the event we used is offcore_response.all_data_rd.l3_miss.any_snoop. The more specific event, which according to the documentation "Counts all demand and prefetch data reads that miss the L3 and the data is returned from local dram" called offcore_response.all_data_rd.l3_miss_local_dram.snoop_miss_or_no_fwd never counts any events in our experiments.

5.3 Bandwidth Experiment Results

The experimental results show that the offcore_response counter does not include accesses due to prefetching, even though the documentation says that demand and prefetched accesses are included. On the Skylake system, even disabled hardware prefetchers do not result in correct measurements.

To verify whether the bandwidth measurement is accurate we use the Stream Triad benchmark as introduced in Section 4.3.2. Figure 7 shows the results. First, we can see that the results are similar on Haswell and Broadwell systems.

5.3.1 Memory Controller Counters. 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.

5.3.2 Offcore Response Counters. 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 [10, Chapter 18, Page 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 and the calculated data amount. In contrast, when prefetching is turned on the offcore response counters do not report the correct data volume.

5.3.3 Exceptions on Skylake. On the Skylake machine (Bottom of Figure 7) even if prefetching is turned off, the offcore counter does not report the correct transferred data. But an effect of the prefetcher setting can be seen when looking at other variants of the offcore counter. There are counters for prefetched data (offcore_response.pf_data_rd.l3_miss.any_snoop) and demand data (offcore_response.demand_data_rd.l3_miss.any_snoop). When hardware prefetchers are turned off, the offcore response counter for the total amount reports the same value as the demand counter. The counter value for prefetched data shows values close to zero. In contrast, when hardware prefetchers are turned on. About two-thirds of the data volume is counted by the prefetcher counter and about one third by the demand counter. The sum of those two counters for prefetched data and demand data is always the sum of the counter that includes all accesses.

6 LATENCY MEASUREMENT

There are two options available for latency measurement. The first one is a metric based on performance counters, and the second one is the latency reported by instruction sampling.

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 [10, Chapter 18 p. 22]. Because the memory level, from which the data was loaded is an attribute of every sample, latency can be calculated for each memory level individually. In these experiments, only those samples which hit in the DRAM and that hit in the TLB are included. Latency of other cache level hits is not included. We calculate the average latency from the filtered samples. 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 view the data because it is designed for instruction sampling and provides easy access to the captured data.

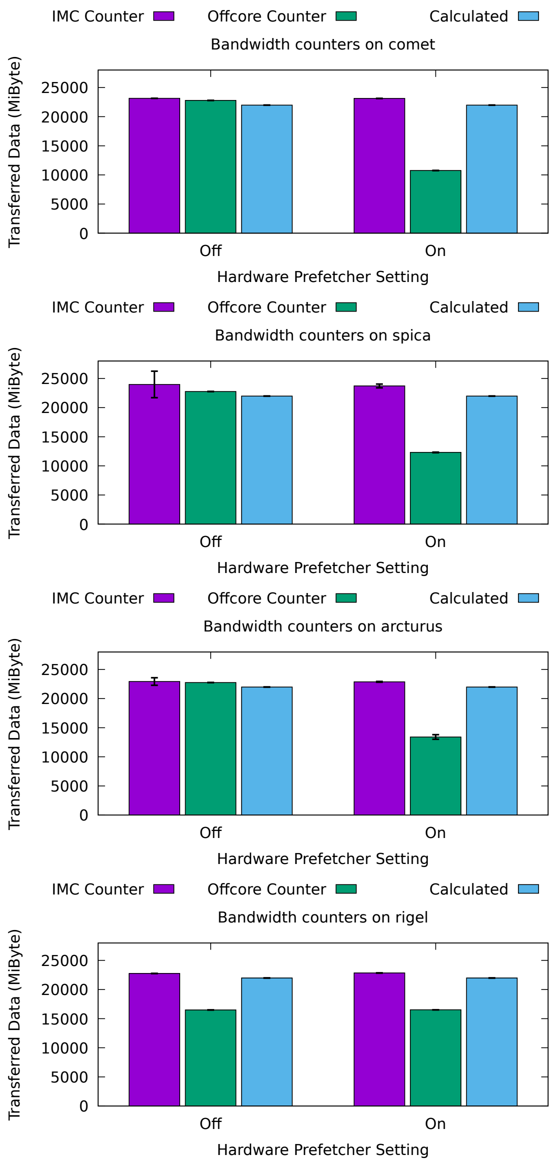

Figure 7: Transferred data volume in the Stream Triad benchmark measured with offcore repsonse counters and IMC counters compared to the calculated transferred data.

6.2 Performance Counters

Another way to measure the latency is to use a metric that is based on performance counters. All required counters are part of the core and L1 cache. Same as the offcore response counters, they can be attributed to specific cores and data recording can be limited to the profiled application.

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).

Based on those counters we define two metrics. 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. The load miss real latency is a metric that is already available in recent Perf versions.

For abbreviation of the long event names the following Greek letters will be used in the metric definition:

$$\alpha = \text{cpu/l1d\_pend\_miss.pending}$$

$$\beta_{Haswell,Broadwell} = \text{cpu/mem\_load\_uops\_retired.l1\_miss}$$

$$\beta_{Skylake} = \text{cpu/mem\_load\_retired.l1\_miss}$$

$$\gamma_{Haswell,Broadwell} = \text{cpu/mem\_load\_uops\_retired.hit\_lfb}$$

$$\gamma_{Skylake} = \text{cpu/mem\_load\_retired.fb\_hit}$$

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 latency is an average for the whole memory hierarchy. It includes the latency of accesses to L2 and L3 caches. In general, it can not be used to measure only the DRAM access latency. Because in our experiments the micro-benchmark only accesses DRAM and no other cache levels we can still use it to measure the DRAM access latency.

6.3 Latency Experiment Results

The main insights from the experiments are: First, memory access latency is indeed a good indicator of bandwidth contention. Second, the effectiveness of prefetching can be seen in the latency. Third, the counter-based metrics only work when including LFB hits. Fourth, the instruction sampling latency consists of in-core instruction processing delays and memory access latency. Despite that, the instruction sampling latency is suitable for diagnosing bandwidth contention.

6.3.1 Hardware Prefetchers Off. Figure 8 shows the measured values of the three different latency measurement methods with rising bandwidth. Common for all four systems is the rise in latency with higher bandwidth. The absolute values of the load miss real latency and L1 miss latency are higher on the Haswell system (Figure 3 top graph) compared to the other systems.

It is also visible that the instruction sampling latency reaches high values even for low bandwidth situations. We will discuss the reason for that in Section 6.3.3. In contrast, the L1 miss latency and load miss real latency only increase when there is an actual bandwidth limitation. The load miss real latency and L1 miss latency have almost the same values. Because there is no prefetching and only one element per cache line is accessed, there are no LFB hits in these experiments. Looking at the metric definitions in 6.2 we can see that in this case, both metrics produce the same value.

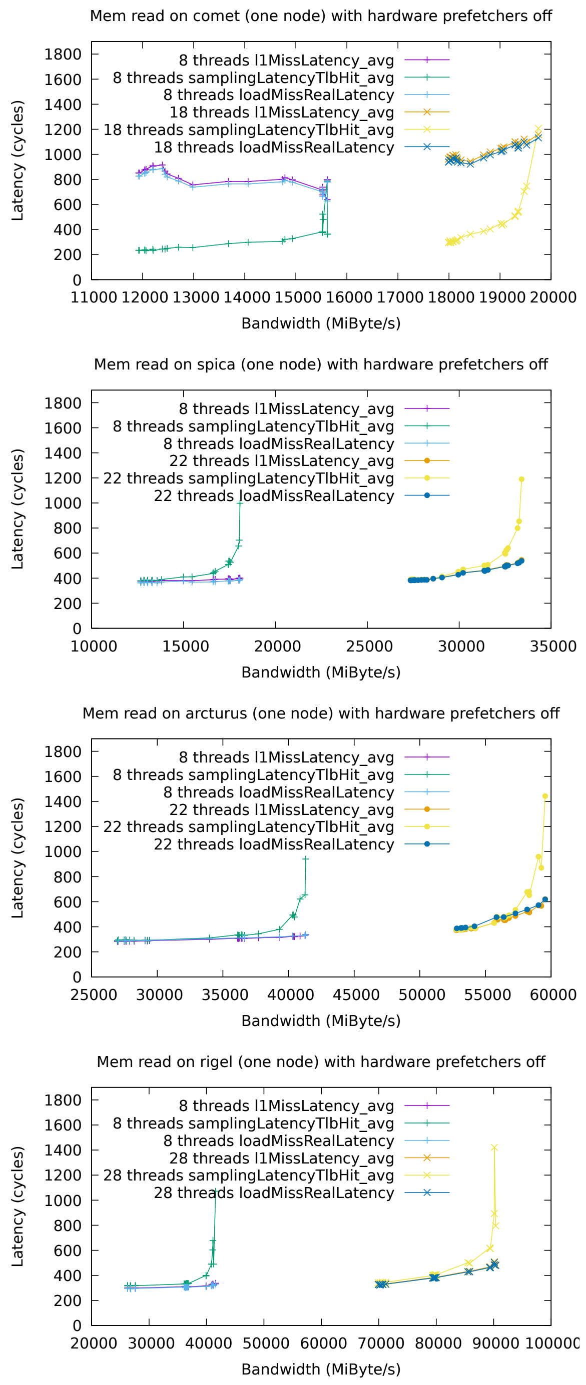

Figure 8: Latency development of different latency measurement methods with increasing DRAM bandwidth when hardware prefetchers are disabled. A selection of threads was picked for this diagram to indicate low and high bandwidth situations. The middle bandwidth section was left out to not obstruct the view. Instruction sampling latency can rise even if the DRAM bandwidth limit is not close.

6.3.2 Hardware Prefetchers On. When the hardware prefetchers are turned on, the number of delay cycles plays an important role. Thus, we show separate diagrams for each type of latency. For each type of latency, it is visible that the latency rises with higher bandwidth. Also, all types of latency show a dependency of latency value and the number of delay cycles. This is a phenomenon that does not appear when the hardware prefetchers are turned off. For the sampling latency in Figure 9 we can see that latency increase at low bandwidth situations does not appear.

The sampling latency (Figure 9) and the load miss real latency (Figure 10) show the same trend regarding delay cycles and latency. Even if the bandwidth is almost the same, when the number of delay cycles is low, the latency is higher. Because of the prefetcher, the main memory bandwidth is the same no matter how many delay cycles are inserted. In cases with a low number of delay cycles, the requests for data are issued quickly. The prefetcher does not have enough time to load the required data. Thus, the requests have to wait a long time until the data finally arrives. When there are many delay cycles, the prefetcher has enough time to load the required data. When the load request is finally issued it will have to wait only a short time.

This is an advantage of the latency measurement compared to the bandwidth measurement. In both situations, many or few delay cycles, the bandwidth is the same. However, the application with few delay cycles has more severe performance problems because there is actual waiting time for the data. In contrast, in the situation with many delay cycles, the prefetcher is simply doing its job and there is no severe performance problem.

When the prefetcher is turned on there will be many hits in the LFB and only very few real L1 cache misses. This ratio will change depending on the number of delay cycles. Many delay cycles will result in many LFB hits and few L1 hits. This effect is visible in Figure 11. A high number of delay cycles results in a high L1 miss latency because the denominator in the metric definition will decrease. We conclude that the load miss real latency is better than the L1 miss latency to measure memory access latency.

6.3.3 Instruction Sampling Latency Analysis. We see in Figure 8 that the instruction sampling latency can be high even though the DRAM bandwidth is far from the machine limit. This is a potential disadvantage for the use of the instruction sampling latency for DRAM contention detection. The instruction sampling latency is the time needed for the whole processing of the instruction. It is not only the time it takes to load data. Thus, the instruction sampling latency will increase when there is an in-core limitation. In contrast, the load miss real latency does not increase because it only starts counting after an L1 miss already occurred.

To verify that we looked at counters that express in-core saturation of different resources. Among them is the counter l1d_pend_miss.fb_full. It counts the number of cycles where a demand request was blocked because of unavailable fill buffer slots. Figure 12 shows that the number of cycles where a request was blocked due to unavailable fill buffer slots is independent of the memory bandwidth. But it depends on the number of delay cycles. When the hardware prefetchers are turned on (No Figure included), there is the same correlation but the value of l1d_pend_miss.fb_full is never higher than 20.000×10^6.

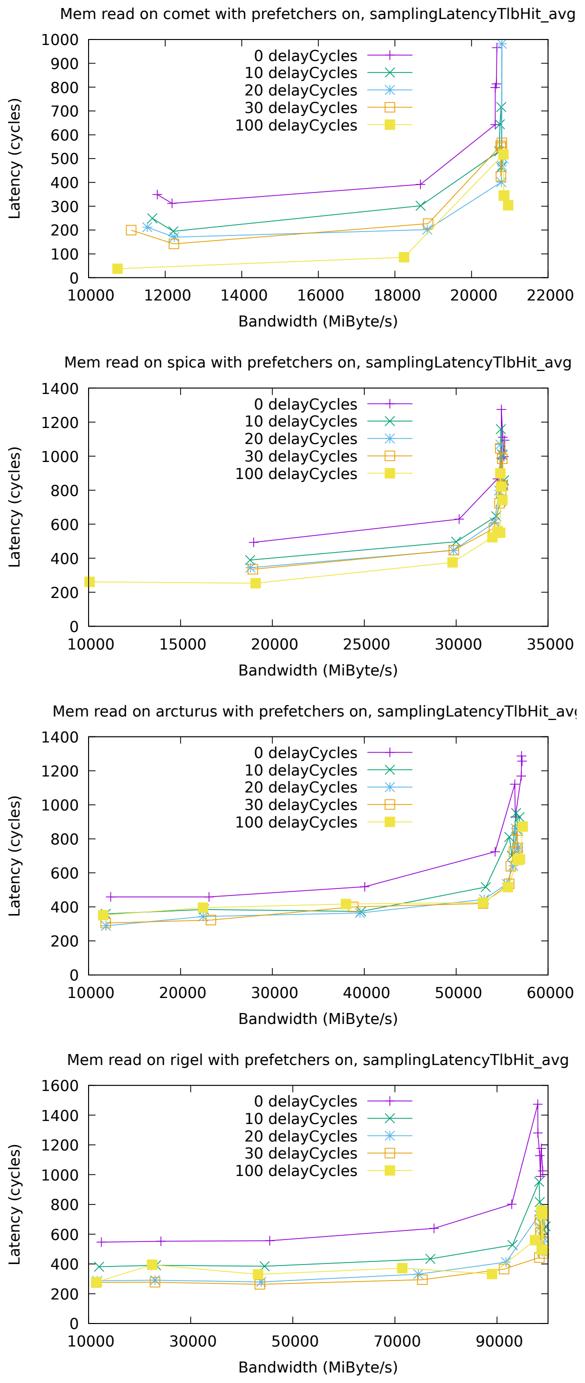

Figure 9: When hardware prefetchers are enabled, higher DRAM bandwidth and fewer delay cycles lead to higher instruction sampling latency. The number of threads controls the bandwidth.

6.3.4 Differentiation of In-Core and DRAM Limitations. To differentiate in-core limitations and DRAM limitations we propose two solutions. First, use the l1d_pend_miss.fb_full counter like we did

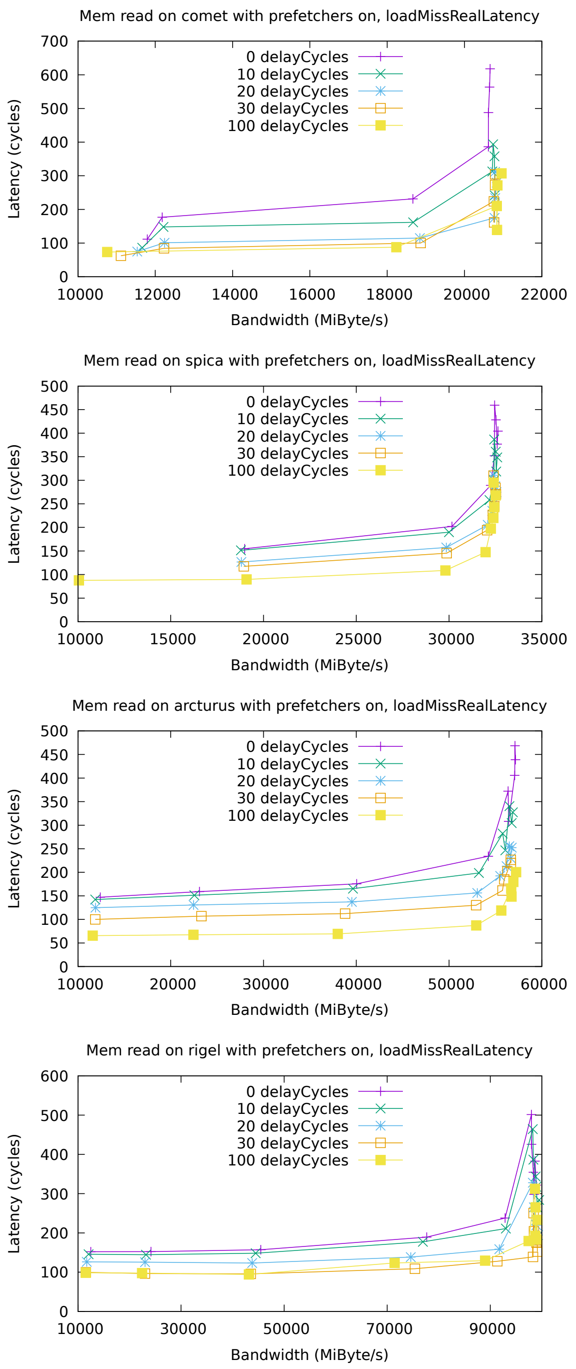

Figure 10: Load miss real latency rise with increasing DRAM bandwidth when hardware prefetchers are enabled. Fewer delay cycles result in higher latency. The number of threads controls the bandwidth.

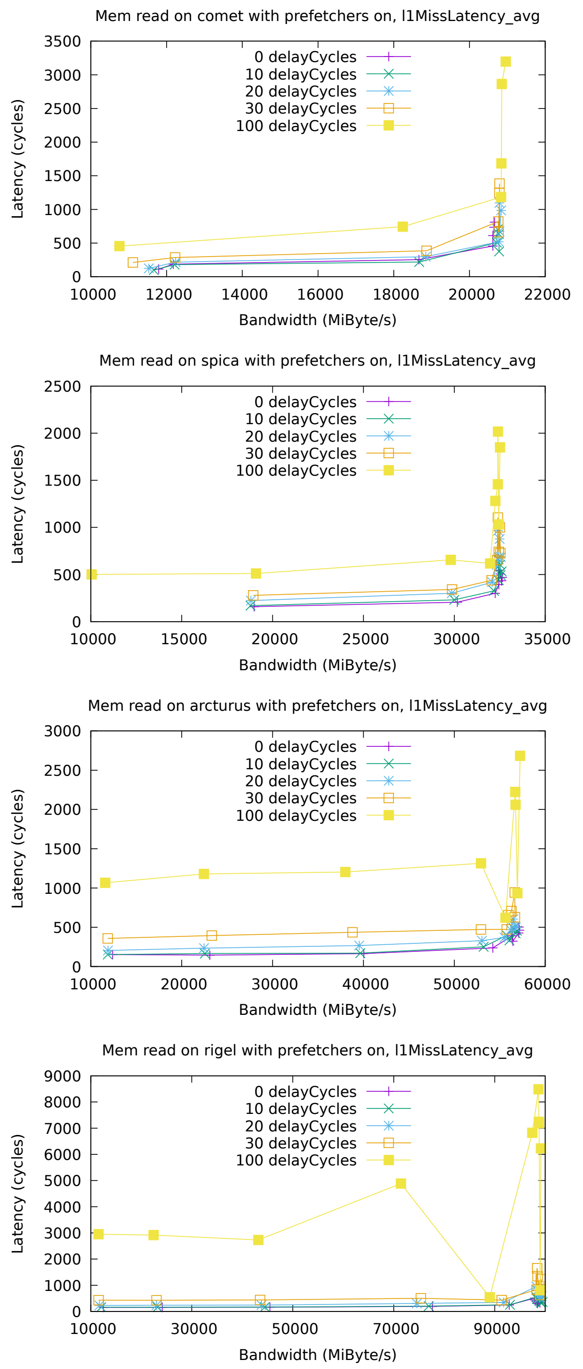

Figure 11: L1 miss latency development with increasing DRAM bandwidth when hardware prefetchers are enabled. Fewer delay cycles lead to lower latency. The number of threads controls the bandwidth.

in our experiments. When high latency is detected it can be used to find out if the core itself is limited. Second, count the number of cores that concurrently access the memory. If high latency is observed but only a few cores access the memory then the limitation is most likely to be in-core because usually a large number of cores is necessary to saturate the DRAM bandwidth. Because the information from which core an access was triggered is also an attribute of a sample, this method is easy to implement based on instruction sampling data. The exact number of cores that is required to saturate the DRAM bandwidth can be determined by a microbenchmark.

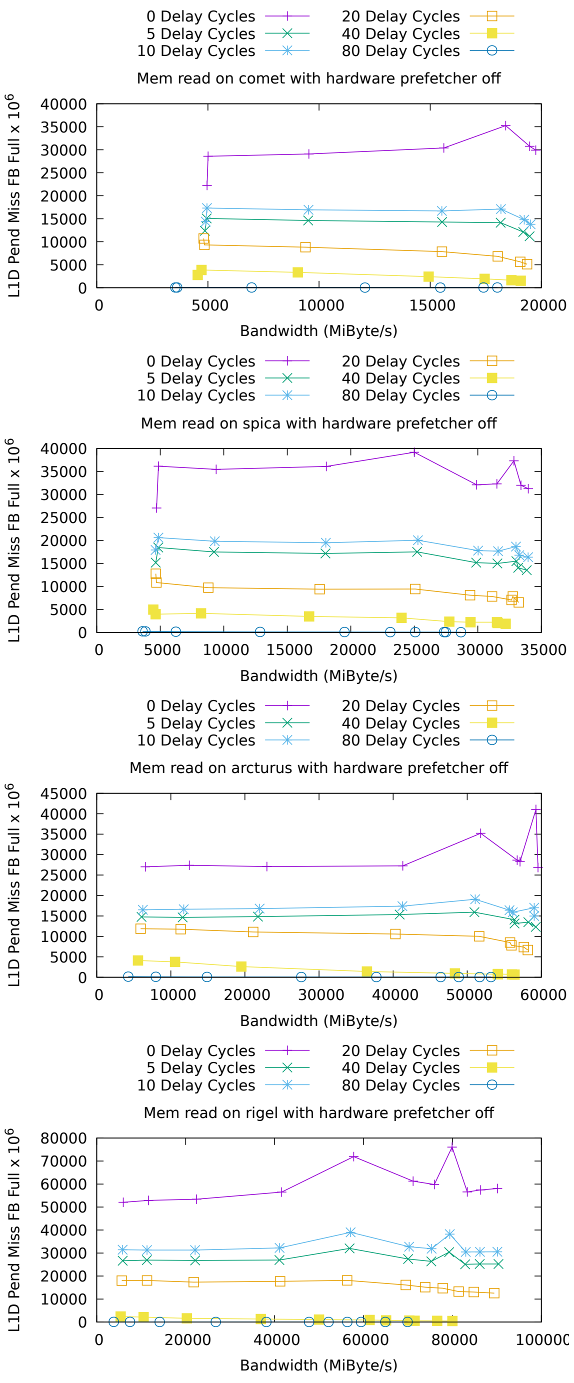

Figure 12: Cycles where fill buffer slots are unavailable do not depend on DRAM bandwidth but depend on the number of delay cycles.

7 CONCLUSION AND FUTURE WORK

Because it allows better attribution of high bandwidth situations to code and data we recommend the instruction sampling latency for the detection of bandwidth contention. It can distinguish performance degrading high bandwidth situations from harmless high bandwidth situations, that the hardware prefetchers can easily produce. Our experiments show the instruction sampling latency consists of in-core delays and DRAM access delays. In another publication [6], we applied this latency based bandwidth contention detection method to real programs, such as PARSEC benchmarks and a machine learning application. We were able to reliably discover DRAM contention and point out the locations of origin in the source code.

However, in some cases, we would like to know the bandwidth consumption of a program and not just the information if there is bandwidth contention or not. In this case, we must rely on performance counters for memory bandwidth. There is a variety of counters that seemingly measure the main memory bandwidth. But we showed that only one of them, the IMC counter measures the correct memory bandwidth. Other counters do not include prefetched accesses. Thus, we suggest using the IMC counters for bandwidth measurements.

In the future, we want to extend our evaluation to AMD processors. They also support instruction sampling with latency measurement. Our approach for verification and the micro-benchmarks can be applied in the same way.

REFERENCES

[1] Antão, D., Taniça, L., Ilic, A., Pratas, F., Tomás, P., and Sousa, L. Monitoring performance and power for application characterization with the cache-aware roofline model. *Lecture Notes in Computer Science 8384 LNCS* (2014), 747–760.
[2] Bitzes, G., and Nowak, A. The overhead of profiling using PMU hardware counters. Tech. Rep. July, CERN, 2014.
[3] Doweck, J. Inside Intel Core Microarchitecture and Smart Memory Access. Tech. rep., Intel Corporation, 2006.
[4] Eranian, S. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN workshop on memory systems performance and correctness* (2008).
[5] Hammer, J., Eitzinger, J., Hager, G., and Wellein, G. Kerncraft : A Tool for Analytic Performance Modeling of Loop Kernels. *Tools for High Performance Computing* (2016).
[6] Helm, C., and Taura, K. PerfMemPlus : A Tool for Automatic Discovery of Memory Performance Problems. In *34th International Conference, ISC High Performance* (2019).
[7] Ilic, A., Pratas, F., and Sousa, L. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters 13*, 1 (2014), 21–24.
[8] 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.
[9] Intel Corporation. Finding your memory access performance bottlenecks. https://software.intel.com/en-us/articles/finding-your-memory-access-performance-bottlenecks.
[10] Intel Croporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3.
[11] Linux. Perf. https://perf.wiki.kernel.org/index.php/Main_Page.
[12] McCalpin, John D. STREAM benchmark. http://www.cs.virginia.edu/stream/.
[13] 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* (2017), 27–38.
[14] Terpstra, D., Jagode, H., You, H., and Dongarra, J. Collecting Performance Data with PAPI-C. *Tools for High Performance Computing* (2010).
[15] Treibig, J., and Hager, G. Introducing a performance model for bandwidth-limited loop kernels. *Lecture Notes in Computer Science 6067 LNCS* (2010), 615–624.
[16] 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* (2010), 207–216.

[17] Weyers, B., Terboven, C., Schmidl, D., Herber, J., Kuhlen, T. W., Müller, M. S., and Hentschel, B. Visualization of Memory Access Behavior on Hierarchical NUMA Architectures. In *Proceedings of VPA 2014: 1st Workshop on Visual Performance Analysis* (2015), pp. 42–49.

[18] Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM 52*, 4 (2009), 65–76.

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

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