Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis

Soramichi Akiyama
Artificial Intelligence Research Center
National Institute of Advanced Industrial Science and Technology (AIST), Japan
s.akiyama@aist.go.jp

Takahiro Hirofuchi
Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology (AIST), Japan
t.hirofuchi@aist.go.jp

ABSTRACT

Analyzing system-noise incurred to high-throughput systems (e.g., Spark, RDBMS) from the underlying machines must be in the granularity of the message- or request-level to find the root causes of performance anomalies, because messages are passed through many components in very short periods. To this end, we consider using Precise Event Based Sampling (PEBS) equipped in Intel CPUs at higher sampling rates than used normally is promising. It saves context information (e.g., the general purpose registers) at occurrences of various hardware events such as cache misses. The information can be used to associate performance anomalies caused by system noise with specific messages. One challenge is that quantitative analysis of PEBS overhead with high sampling rates has not yet been studied. This is critical because high sampling rates can cause severe overhead but performance problems are often reproducible only in real environments. In this paper, we evaluate the overhead of PEBS and show: (1) every time PEBS saves context information, the target workload slows down by 200 − 300 ns due to the CPU overhead of PEBS, (2) the CPU overhead can be used to predict actual overhead incurred with complex workloads including multi-threaded ones with high accuracy, and (3) PEBS incurs cache pollution and extra memory IO since PEBS writes data into the CPU cache, and the severity of cache pollution is affected both by the sampling rate and the buffer size allocated for PEBS. To the best of our knowledge, we are the first to quantitatively analyze the overhead of PEBS.

CCS CONCEPTS

• **Software and its engineering → Software performance**;

KEYWORDS

performance counter, performance analysis, system-noise

ACM Reference format:
Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis. In *Proceedings of ROSS '17, Washington, DC, USA, June 27, 2017,* 8 pages.
DOI: 10.1145/3095770.3095773

1 INTRODUCTION

Analyzing performance anomalies of high-throughput systems in data centers, such as data processing frameworks or RDBMS, must focus on each message or request (instead of each function or code block) to find the causes because of two reasons: (1) messages go through many components running in parallel on different CPUs and hosts (2) each message lives for very short period. For example, Zhao *et al.* [17] constructs control flows of individual requests from logs to find the root causes of performance anomalies of distributed systems (e.g., HDFS, Yarn, HBase). Similarly, Ousterhout *et al.* [9] uses logs to analyze the time during which each Spark SQL query is blocked due to network IO or disk IO, and found that these factors are not necessarily the bottlenecks in Spark.

System-noise is one factor of performance anomalies in high-throughput systems. For example, a large message may unexpectedly cause a larger number of cache misses than others do, resulting in a sudden latency spike that cannot be found by a coarse-grained profiling. We consider using Precise Event Based Sampling (PEBS, provided in Intel CPUs) to analyze system-noise in the granularity of message- or request-level is a promising way to find the root causes of performance anomalies of high-throughput systems. PEBS saves context information of the target workload, such as the general-purpose registers and the target address of a load instruction, at the time of specified hardware event occurrences such as cache misses. PEBS provides much more accurate and fine-grained profiling than normal performance counters, which we believe useful for mapping the sampled hardware events to each message or request that stays inside the system for only a fraction of time.

To use PEBS to analyze system-noise in the message-level, the overhead of PEBS must be studied, although it has not yet been done, because of two reasons. First, mapping collected context information to each message requires much higher sampling rates than used normally because a message is very transient. Second, collecting context information should be online because real world performance problems are often very difficult to reproduce in an offline environment. Therefore, a need arises to quantitatively analyze the overhead of PEBS and to predict how much overhead PEBS incurs to a given target workload.

In this paper, we examine how two important configuration options of PEBS, namely the sampling rate and the buffer size allocated for PEBS, impacts the performance of the target workload. We first show that the time by which the target workload slows down is accurately predictable from the sampling rate for a simple workload. After that, we show this claim is also applicable to complex CPU-intensive workloads including mutli-threaded ones. At last, we study how the CPU overhead and cache pollution caused by PEBS have complex effects to cache-sensitive workloads.

The structure of this paper is as follows. Section 2 shows an overview of PEBS including its difference, when compared to the normal performance counters. Section 3 describes the types of overhead that each of the configuration options of PEBS incurs. Section 4 evaluates the overhead of PEBS using micro and macro benchmarks. Section 5 provides a guide to use PEBS at high sampling rates, considering the knowledge presented in this paper. Section 6 reviews related work and Section 7 summarizes this paper.

2 PRECISE EVENT BASED SAMPLING (PEBS)

2.1 Performance Counters

Performance counters are a hardware-based performance monitoring mechanism, supported not only in Intel CPUs but also by many other vendors. We explain how it works for the Intel case without losing generality. Hardware events are pre-defined such as `ICACHE.MISSES` (the number of instruction cache misses) or `MEM_UOPS_RETIRED.ALL_LOADS` (the number of load micro operations retired). The user can choose a small number of them (up to 2, 4, or 8 depending on the CPU) to let the CPU count the events. The CPU then counts the number of specified events using a designated model-specific register for each event, which we refer to as **counter registers** hereafter. A counter register overflows after the event occurs a specified number of times, R. This is called **reset value**, because it is implemented by setting $-R$ to counter registers at reset. Reset value is also called **sample after value** in some literature. Each core of a CPU has their own counter registers. This enables performance analysis in per core basis.

Profiling using performance counters is interruption-based. The CPU invokes an ACPI interruption when one of the counter registers overflows. The operating system receives the interruption and collects information about the profiled application nearly at the time of the event occurrence that has triggered the interruption. For example, counting `ICACHE.MISSES` and saving the program counter every time an interruption is invoked can tell which program or function causes a lot of instruction cache misses.

2.2 PEBS Overview

Precise Event Based Sampling (PEBS) is an extension to the normal performance counter by Intel [3]. The largest difference is that PEBS saves a set of context information using the hardware when a counter register overflows. This greatly reduces the overhead incurred by interruption handling required every time a counter register overflows with the normal performance counters.

Figure 1 shows how PEBS saves context information.

(1) The user chooses a set of events and specifies if each counter should be PEBS-enabled or not. This means that PEBS counters and normal counters can be used together at the same time. The CPU counts the number of specified events using a designated counter register for each event.
(2) When the counter register for a PEBS-enabled event overflows, the CPU triggers a **PEBS assist** (instead of invoking an interruption) that executes a pre-defined micro-code.
(3) The micro-code saves context information, called a **PEBS record**, into a memory region called the **PEBS buffer** that starts from an address referred to as the **PEBS base**. It can contain multiple PEBS records and the current tail inside the PEBS buffer is called the **PEBS index**.

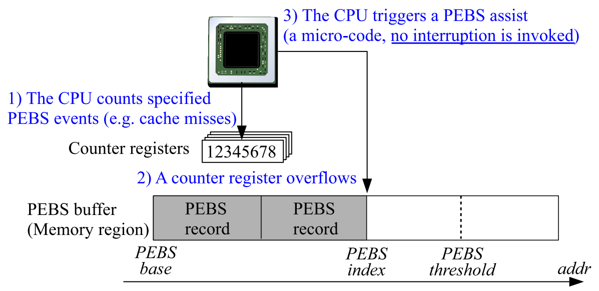
3) The CPU triggers a PEBS assist
(a micro-code, no interruption is invoked)
1) The CPU counts specified
PEBS events (e.g. cache misses)
Counter registers
12345678
2) A counter register overflows
PEBS buffer
(Memory region)
PEBS record
PEBS record
PEBS base
PEBS index
PEBS threshold
addr

Figure 1: Overview of Intel PEBS. A PEBS record is written to the PEBS buffer by the CPU when a PEBS event has occurred specific times and thus the corresponding counter register overflows (no interruption invoked). The PEBS index points the current tail of the PEBS records inside the PEBS buffer.

Once the PEBS index reaches the **PEBS threshold**, a hardware interruption is triggered to tell the operating system that the PEBS buffer is almost full and has to be drained for further processing. By saving context information using hardware-based micro-code, PEBS has two advantages over the normal performance counters:

(1) PEBS reduces the number of interruptions and thus it is expected to reduce performance overhead. For example, if PEBS buffer can contain 100 PEBS records before an interruption is invoked, the number of interruptions is reduced to 1/100 compared to the normal performance counters.
(2) The gap between the time a counter register overflows and the time the context information is saved is much smaller than the interruption-based method. This makes PEBS more precise, as the name suggests.

The content of a PEBS record differs depending on the generation of the CPU cores. In recent CPUs after Skylake micro architecture, a PEBS record has the values of the general purpose registers (`ip`, `rax`, ...), the target address and the latency information if the current instruction is a load, a hardware transaction (tx) abort reason flag, if the event is tx-related, and the hardware time stamp. Note also that not all performance events are usable with PEBS, and each micro architecture has different set of PEBS-able events.

3 PEBS OVERHEAD

PEBS is believed to incur a negligible overhead to target workloads, because saving PEBS records is done by hardware. Many profiling systems claim that their overhead is small enough just because they use PEBS [5, 10]. However, when PEBS is used for analyzing system-noise for each message of request, the sampling rate must be much higher than typical function-level profiling. The overhead PEBS incurs when used at high sampling rates has never been examined, nor documented by Intel. In this section, we explain how each configuration option of PEBS can incur overhead and we evaluate them in Section 4.

3.1 Reset Value

Reset value controls the sampling rate, thus it is the most important parameter for PEBS. Decreasing reset value makes the profiling more fine-grained, but it invokes a larger number of PEBS assists. To our surprise, we found that the CPU overhead of **one** PEBS assist is more than 200 nano seconds. This is because PEBS assist is implemented by a micro-code, rather than a designated ciruit. Therefore, when a PEBS assist is executed on a CPU core, the workload running on the core is suspended and the execution pipeline and the TLB are all flushed.

Given the fact that a typical random access latency to the main memory is around 50 – 100 ns, this overhead is completely non-negligible when the reset value is configured to be very small (equivalent to a very high sampling rate). Therefore, the user must choose a moderate reset value that is enough fine-grained for the purpose of the analysis but does not incur too much overhead.

3.2 PEBS buffer size

Increasing the size of the PEBS buffer decreases the number of interruptions invoked during profiling because larger buffer can have larger number of PEBS records before it becomes full. This obviously decreases the number of context-switches to and from the OS, which causes both the TLB and the execution pipeline to be flushed.

However, we found that merely increasing the size of the PEBS buffer does not always reduce the overhead to the profiled system. This is because PEBS writes PEBS records via the CPU cache, but not directly to the main memory. As a result, increasing the size of the PEBS buffer incurs an overhead for two reasons:

(1) Increased number of cache-misses due to the cache usage by PEBS assists.
(2) Increased amount of memory IO, especially when PEBS assists and the profiled program alternatingly evict the cache lines used by the other.

Therefore, the size of the PEBS buffer should be chosen to minimize the sum of overhead incurred by interruption handling, cache misses, and extra memory IO.

The PEBS buffer size must be equal to the size of a PEBS record when more information than saved by PEBS is needed. For example, because timestamps are not saved by PEBS in CPUs older than Skylake micro-architecture, an interruption is required every time a PEBS assist occurs so that the OS can save timestamps if the user wants them. However, we do not deal with this case in this work and we assume the PEBS buffer size is freely configurable.

4 QUANTITATIVE ANALYSIS

4.1 Set Up

In order to measure the PEBS overhead flexibly and accurately, we created our own kernel module that configures the PEBS functionality with specified parameters. Existing tools such as Linux `perf` are intended to be used for analyzing the performance of the profiled applications but not the performance of PEBS itself, thus they do not allow us to configure every single low-level parameter. For example, `perf` does not support changing the size of the PEBS buffer to an arbitrary value. Another reason to make our own kernel module is the existing tools have non-negligible software overhead [12] to support the easy-to-use interfaces and the rich variety of analysis.

The kernel module recieves the PEBS buffer size, hardware counters to be counted, and reset values for each counter as the parameters of the `init` ioctl call. After it is initialized, the module receives ACPI interruptions every time the PEBS threshold is reached. The module counts the number of PEBS assists that have occurred after the last interruption by $\frac{\text{PEBS\_index}-\text{PEBS\_base}}{\text{sizeof(PEBS\_record)}}$ when an interruption is received. It then resets the PEBS_index to the PEBS_base. This means that the PEBS records written are just ignored. After the target workload has finished, the module prints the number of PEBS assists during the whole run across all cores and exists.

Table 1: Evaluation Environment

	Machine 1	Machine 2	Machine 3
OS	Debian GNU/Linux 8 (Linux kernel 4.9)		
CPU	Xeon E5-2630 v4	Xeon E5-2699 v3	Core i5 6400
Arch.	Broadwell	Haswell	Skylake
# Cores	10	18	4
LLC	25 MB	45 MB	6 MB
CPU freq.	2.2 GHz	2.3 GHz	2.7 GHz
Mem lat.	78 ns	88 ns	56 ns

Table 1 shows the operating system, CPU model name, CPU core micro architecture, the number of cores, the size of the last level cache (LLC), the CPU base frequency (not the “turbo boost” frequency), and the latency of a random memory access of the machines we use in the evaluation. The kernel is updated to a recent one (4.9) from the default one of Debian 8 (3.16). The memory latency was measured by Intel Memory Latency Checker [11]. We mainly use machine 1, which has the latest server-class CPU we have, and use machines 2 and 3 as well in some experiments for supporting the findings.

4.2 CPU Overhead

In this section, we estimate the CPU overhead of PEBS. The CPU overhead consists of CPU cycles spent for PEBS assist itself, the effect of execution pipeline flushes due to the PEBS assist micro-code, and the overhead for interruption handling when the PEBS buffer becomes full. In this section, we set the PEBS buffer size to the largest allocable to make the overhead incurred by interruptions negligible so that we can measure the truly unavoidable overhead of PEBS. As shown in the later sections, PEBS also incurs cache pollution and extra memory IO due to cache evictions because PEBS records are written to the CPU cache. To avoid them and measure only the CPU overhead, we applied PEBS to a busy loop, which has (almost) no data read/write. The PEBS event specified is `UOPS_RETIRED.ALL`. This event “counts the number of micro-ops retired” [3], which means that this is one of the most aggressive usage of PEBS. The PEBS buffer size is set to the largest allocable size by `kmalloc` in our environment (4 MB). The reset values are set to 128000, 64000, 32000, 16000, 8000, 4000, and 2000. The workload executes $10 \times 1024 \times 1024 \times 1024$ (10 G) times of busy loops in

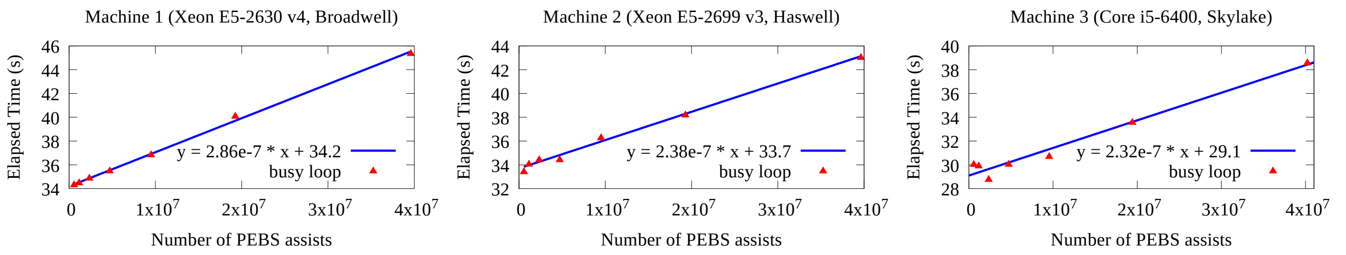

Figure 2: Number of PEBS assists vs. busy loop elapsed time and the best-fit lines.

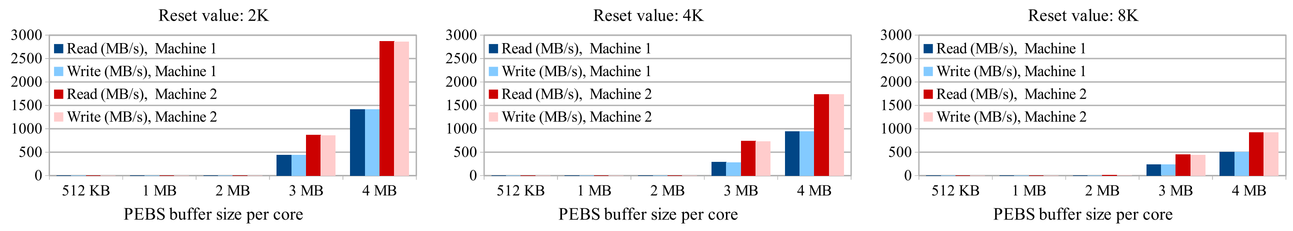

Figure 3: PEBS buffer size per core vs. memory IO caused by cache spill in machine 1 (Broadwell, 10 cores) and machine 2 (Haswell, 18 cores). Both CPUs have 2.5 MB LLC/core. Reset values: 2K (left), 4K (middle), 8K (right).

a single core. We measured the elapsed time of the workload with and without PEBS with various reset values shown above.

Figure 2 shows the elapsed time and the number of PEBS assists occurred in machines 1, 2, and 3 (from left to right). Each point corresponds to each reset value (larger reset value yields smaller amount of overhead). The results are averaged over three runs. The x axes show the number of PEBS assists, and the y axes show the elapsed time. The graphs show that the elapsed time have strong linear correlation with the number of PEBS assists. The blue line in each graph shows the best-fit line that approximates the measured results. In the machine 1 case, the best-fit line is $y = 2.86 \times 10^{-7} x + 34.2$. This means that the CPU overhead per PEBS assist is around 286 nano (286×10^{-9}) seconds in machine 1. Similarly, the CPU overhead is 238 nano seconds / PEBS assist in machine 2, and 232 nano seconds / PEBS assist in machine 3.

The CPU overhead of a PEBS assist is several times larger than a random memory access latency, which is typically 50 ns – 100 ns as shown in Table 1. This is small enough for traditional PEBS usage such as saving PEBS record at every 100K cache misses, but for analyzing system-noise in the message-level graunurality, this value must be carefully considered. We will investigate how to reduce the CPU overhead of PEBS in future work. For example, using huge pages might be effective because the PEBS base is represented as a virtual address thus the page table (or the TLB) is referred before saving context information.

4.3 Data IO

In this section, we estimate the overhead of data IO due to PEBS assists. To our surprise, PEBS assists write PEBS records via the CPU cache, but not directly to the main memory. Therefore, data IO of PEBS assists always incurs cache pollution, and it incurs extra memory IO when the sum of PEBS buffer and the application working set does not fit in the last level cache.

To confirm the fact that PEBS assists write data via the CPU cache, we measured the amount of memory IO of the machine when PEBS is applied to multi-threaded busy loops. The workload creates the same number of threads as the number of cores and each thread executes $10 \times 1024 \times 1024 \times 1024$ (10 G) times of busy loops. Because busy loops incur almost zero memory IO, the observed memory IO should all come from the data write of PEBS assists. The memory IO is measured for machine 1 and machine 2 using a tool provided by Intel [14]. It uses a designated performance monitoring unit implemented in the memory controller, thus measuring memory IO can be orthogonal and in parallel with using PEBS. Please note that this mechanism exists only in server-class CPUs (Xeon E5/E7), thus this experiment cannot be conducted in machine 3 (Core i5).

Figure 3 shows the relationship between the PEBS buffer size per core and observed memory IO. The x axes show the PEBS buffer size **per core**, and the y axes show the observed memory IO in MB/s of read (R) and write (W) traffics. The reset values are set to 2000 in the figure on the left, 4000 in the middle, and 8000 on the right. The size of the last level cache per core is 2.5 MB in both machines 1 and 2, thus allocating more than 2.5 MB of PEBS buffer per core exceeds the capacity of the last level cache.

Figure 3 shows three important things. (1) The observed memory IO is nearly 0 up to the case where the PEBS buffer size per core is 2 MB, but it becomes non-negligible thereafter in both machines and in all reset values. This is because all PEBS records are written to the CPU cache when the PEBS buffer is smaller than the cache, but the CPU cache starts spilling once the PEBS buffer becomes larger than it. (2) When the CPU cache spills, the observed memory IO increases as the size of PEBS buffer increases (c.f. 3 MB cases vs 4 MB cases), although the amount of data PEBS assists write per second is constant regardless of the size of the PEBS buffer. Please be aware that the size of PEBS buffer does not affect the number of PEBS assists per second. (3) Both reads and writes use equal bandwidth, even though PEBS assists only write PEBS records and our kernel module just ignores them in this experiment. This is because a write cache miss requires reading an entire cache line to which the updated word belongs.

In summary, we conclude that PEBS incurs cache pollution because it writes PEBS records to the CPU cache, and larger the PEBS buffer is more severe the cache pollution becomes. This requires special consideration when the profiled application is cache sensitive, and the application and PEBS compete for the shared cache region. We investigate this case in Section 4.6.

4.4 Validation of CPU overhead

In this section, we show that the CPU overhead of PEBS estimated in Section 4.2 can be used to predict the elapsed time of complex CPU intensive workloads with PEBS enabled. We measure elapsed time and the number of PEBS assists for SPEC CPU 2006 integer benchmarks and compare two values:

(1) Measured elapsed time: the actual observed elapsed time when PEBS is enabled. This includes both the time taken for a workload itself and the time used for PEBS assists.
(2) Expected elapsed time: the sum of the elapsed time when a workload is executed without PEBS and the *expected* CPU overhead for PEBS assists.

The *expected* CPU overhead for PEBS assists is calculated by $\alpha \times N$, where α is the CPU overhead per PEBS assist and N is the number of PEBS assists during a workload run. If the values (1) and (2) are close, we can quantitatively predict how much the target system is slowed down without trials and errors. This is because the number of specified events during a workload can be measured by the normal performance counters with negligible overhead. After the number of specified events is measured using the normal performance counters, the user can decide the reset value for PEBS based on the maximum allowed overhead, the required number of samples, and the CPU overhead per PEBS assist. Without a precise prediction of the expected elapsed time including the PEBS overhead, the user must try many reset values to seek a good point that achieves moderate overhead and the required sampling rate.

We use machine 1 in this section; α is 286 nano seconds per PEBS assist (Section 4.2). The PEBS buffer size is 4 MB. Please note that SPEC CPU workloads are single-threaded, thus 4 MB PEBS buffer means 4 MB is allocated for the core on which the workload runs. The reset values are set to 128000, 64000, 32000, 16000, 8000, 4000, and 2000. The PEBS event specified is `UOPS_RETIRED.ALL`.

Figure 4 shows the measured and expected elapsed time for all SPEC CPU 2006 integer benchmarks. The x axes show the number of PEBS assists occurred during the execution, and the y axes show the measured and expected elapsed time. Each point corresponds to reset value of 128K, 64K, 32K, 16K, 8K, 4K, 2K respectively from left to right. The values were averaged over three runs. Figure 5 shows relative error E in % of measured elapsed time against the expected elapsed time for each benchmark and reset value. The values are calculated by:

$$E = \frac{\text{Measured Elapsed Time} - \text{Expected Elapsed Time}}{\text{Expected Elapsed Time}}. \tag{1}$$

A large positive value means that the actual overhead is much larger than expected.

For all benchmarks other than omnetpp, the two lines for each benchmark in Figure 4 almost overlap. The relative errors shown in Figure 5 are also small for these benchmarks; the relative errors are less than 4% in all cases, and less than 2% in many cases. However for omnetpp, the relative errors reach tp to 10+% and they show strong negative correlation with the reset value (smaller the reset value is, larger the relative error becomes). We analyze the reason for this later in Section 4.6.

In summary, we conclude that for many CPU intensive workloads the CPU overhead of PEBS estimated in Section 4.2 is well suited to the prediction of the elapsed time when PEBS is enabled.

4.5 CPU Overhead on Multi-threaded Application

In this section, we show that the CPU overhead per PEBS assist estimated in Section 4.2 can be applied to multi-threaded applications as well. Due to the space limit, we use one multi-threaded application from NAS Parallel Benchmarks, EP. It "generates pairs of Gaussian random deviates" [2] and it is embarrassingly parallel. We use EP because it is highly robust to the CPU cache size thus the effect of data IO shown in Section 4.3 is avoided. The robustness was confirmed by allocating a small portion of the CPU cache by Intel CAT (Cache Allocation Technology) [7] and ensuring that the elapsed time did not change. The size of the PEBS buffer is 1 MB **per core** (instead of 4 MB) because allocating 4 MB per core makes the PEBS buffer alone larger than the CPU cache. Machine 1 is used for this experiment as the representative case.

Figure 6 shows the measured and expected elapsed time of the EP benchmark. The x axis shows the number of PEBS assists during the execution, and the y axis shows the measured and expected elapsed time. Each point corresponds to reset value of 128K, 64K, 32K, 16K, 8K, 4K, and 2K respectively from left to right. The values were averaged over three runs. Because EP invokes the same number of threads as the number of cores (10 threads on 10 cores in this case), PEBS assist occurs in all 10 cores in parallel. Therefore, the expected elapsed time is calculated per core by dividing the number of PEBS assists by 10 (the number of cores). The figure shows the measured and expected elapsed time match very well.

In summary, we conclude that the CPU overhead per PEBS assist estimated in this paper can be applied to multi-threaded applications as well.

4.6 PEBS Buffer size vs. Cache Pollution

In this section, we analyze the reason why the elapsed time of omnetpp is not well predicted compared to the other benchmarks in Section 4.2, and withdraw useful knowledge for applying PEBS to complex workloads.

We believe that one reason is cache pollution caused by the data write from PEBS assist because of two pieces of evidence:

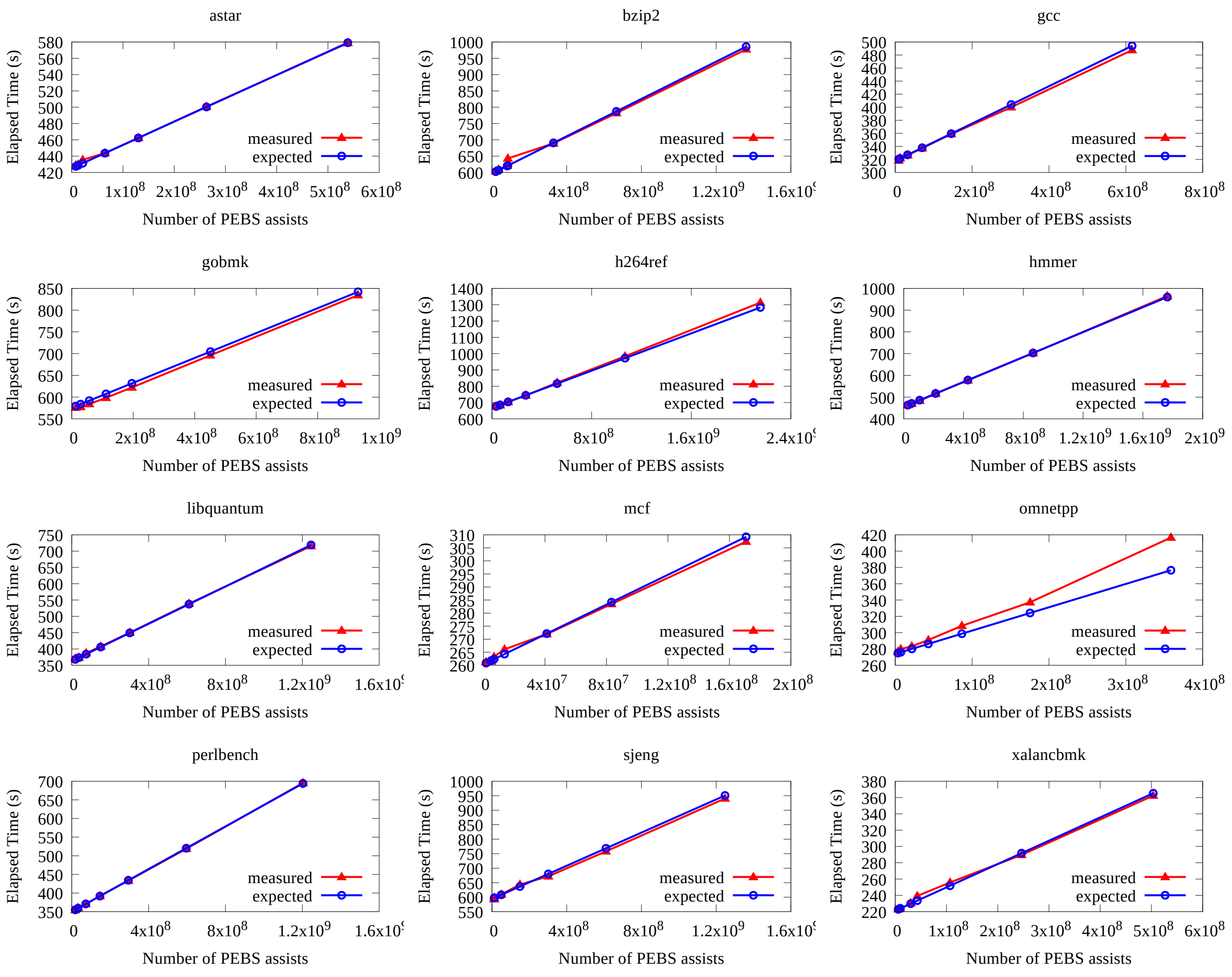

Figure 4: Number of PEBS assists vs. measured and expected elapsed time of SPEC CPU 2006 benchmarks with different reset values (2K, 4K, 8K, 16K, 32K, 64K, 128K). PEBS buffer size is 4MB. Note that the scales of the axes are different in each figure. Note also that the y-axes do not start from 0.

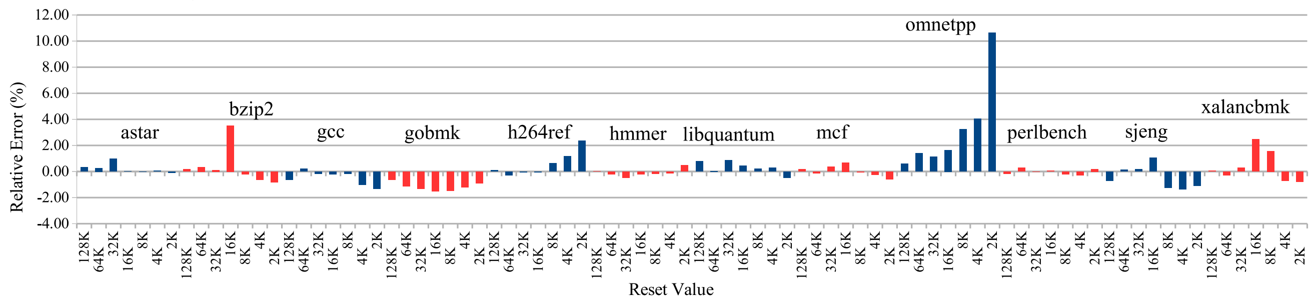

Figure 5: Reset value vs. relative error (%) of measured elapsed time against expected elapsed time of SPEC CPU 2006 benchmarks. The errors are small (within ± 2% for almost all cases) except for omnetpp.

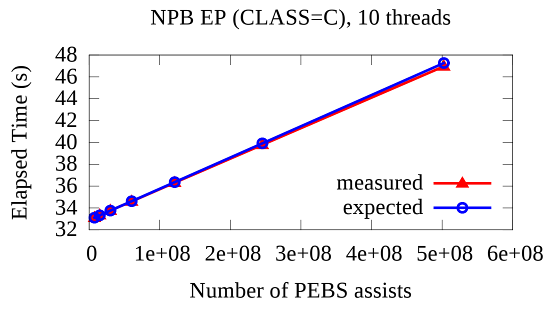

Figure 6: Number of PEBS assist vs. elapsed time for multi threaded application (NPB EP).

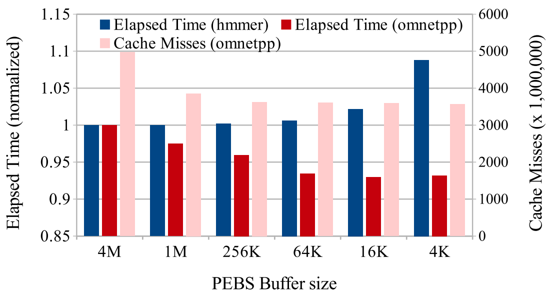

Figure 7: PEBS buffer size vs. elapsed time and cache misses. Omnetpp (cache-sensitive) and hmmer (cache oblivious) show the opposite trends.

(1) Figure 5 shows that the relative error rate increases as the reset value decreases and therefore the data write of PEBS assists gets faster.
(2) We confirmed that omnetpp is one of the most cache sensitive workloads among SPEC CPU (the others are xalancbmk and mcf) by measuring the allocated CPU cache size vs. performance degradation with Intel CAT (detailed values omitted due to the space limit).

Figure 7 shows the elapsed time of omnetpp and hmmer, and the number of cache misses of omnetpp with various PEBS buffer sizes. We use hmmer as a comparison with omnetpp because hmmer is one of the **least** cache sensitive workloads in SPEC CPU (again confirmed with Intel CAT). The reset value is set to 2,000 and the PEBS event specified is `UOPS_RETIRED.ALL`. Note that the figure does not show the number of cache misses for hmmer, because the numbers are 3 orders of magnitude smaller than the ones of omnetpp and the performance of hmmer is not affected by the number of cache-misses. The number of cache misses is counted using a normal performance counter with a large reset value (100K) thus counting cache misses does to affect the results. The elapsed time is normalized by the values with 4 MB PEBS buffer.

The result shows two findings: (1) The elapsed time of omnetpp gets **shorter** when the PEBS buffer becomes **smaller**, although 1000x smaller PEBS buffer incurs a 1000x larger number of interruptions (note that the number of PEBS assists is **the same**). This is because smaller PEBS buffer incurs less severe cache pollution as shown in Section 4.3. (2) The elapsed time of hmmer gets **longer** when the PEBS buffer becomes smaller, which is the opposite trend of the omnetpp results. This is because hmmer is not affected by the cache pollution incurred by PEBS and the effect of interruption handling gets more severe when the PEBS buffer gets smaller.

In summary, the cache pollution incurred by PEBS does affect the performance of the profiled workload, and it can be mitigated by reducing the PEBS buffer size. However, smaller PEBS buffer incurs larger number of interruptions thus the PEBS buffer size must be carefully set depending on the cache sensitivity of the workload. Numerically modeling the relationship between the PEBS buffer size and the severity of cache pollution is a future work, which should unveil why other cache sensitive workloads (xalancbmk, mcf) are not as much affected by large PEBS buffer as omnetpp.

5 USING PEBS FOR ONLINE SYSTEM-NOISE ANALYSIS

Here a guide for using PEBS for online system-noise analysis based on our experimental results is provided. The CPU overhead per PEBS assist must be estimated for the target machine. This can be done with a by running a busy loop with several reset values (e.g., 2K, 4K, ..., 128K) as shown in Section 4.2. The slope of the best-fit line on a graph showing the number of PEBS assists occurred vs. elapsed time of the busy loop represents the CPU overhead per PEBS assist. We showed that the value estimated in this way can be used for more complex workloads.

Then, the number of PEBS events occurs per second during a native run of the profiled workload must be measured. A normal performance counter with a large reset value can be used so that no visible overhead to the profiled workload incurs, because what is needed here is the number of events, but not the context information. This means that the measurement can be applied even to a system serving real users.

Finally, the reset value and the size of the PEBS buffer must be decided with consideration given to (1) the required sampling rate to achieve the purpose of the analysis, (2) the acceptable amount of overhead to the target workload, and (3) the sensitivity of the target workload to cache pollution. Estimating cache sensitivity of a given workload is not a simple task, but several methods are proposed such as using cache partitioning mechanisms [6] or co-locating a hand-crafted cache pressuring program [4]. If the target workload is not cache sensitive, the PEBS buffer can be as large as the CPU cache so that the number of interruptions is minimized. In this case, the reset value can be easily decided by dividing the acceptable overhead per second by the CPU overhead per PEBS assist. Note that the size must be calculated per core, because PEBS assists are executed in each core in parallel (for example, if the profiled workload uses 4 cores, the maximum allowed size of PEBS buffer per core is 1/4 of the last level cache). If the profiled workload is cache sensitive, the size of PEBS buffer is recommended to be small since larger PEBS buffer incurs more severe cache pollution as shown in Section 4.6. However, too small PEBS buffer incurs too many interruptions and degrades the performance of the profiled workload as a result. Therefore the user should find a sweet spot by measuring the performance degradation and the number of cache misses. Systematically finding the best PEBS buffer size for cache sensitive applications is one aspect of our future work.

6 RELATED WORK

Estimating overhead of performance counters have been a big concern. Larysch [5] investigates microscopic behaviors of PEBS and shows that a large portion of PEBS samples are lost when the reset value is very small (e.g., less than 100). This is because of the gap between the time when a counter register overflows and the time when the corresponding PEBS assist is invoked. Any events happening in between this gap are discarded because the counter register is forcefully set to the reset value. The paper also claims that using PEBS introduces little overhead for their profiling system, but it does not quantitatively discuss the overhead of PEBS itself independently. Weaver [12] investigates overheard of normal performance counters when starting and stopping a measurement and reading measured values by existing software. The paper shows that reading a performance counter value using the Linux `perf` interfaces takes 2K – 3K cycles, which comes from page faults and is avoidable by populating memory pages before reading the counter. Although the paper contributes a thorough analysis across many software versions, our work and this paper are orthogonal; this paper focuses on the overhead to observe the measured values, while we focus on the overhead of measurement itself.

The accuracy of performance counters is also important because it directly affects the trustworthiness of the profiling results. Weaver *et al.* [13] discuss non-determinism and overcount of the performance counters. They evaluate the accuracy of performance counters using carefully handcrafted assembly programs (whose number of load/store instructions and branches are known a-priori). Nowak *et al.* [8] target basic-block level profiling and discusses the accuracy of various types of reset values (e.g., even, prime, randomized) on various CPUs. Zaparanuks *et al.* [16] also discuss the accuracy of performance counters with various hardware and software settings. They show that even for the same **binary** on the same CPU, the number of cycles measured differs greatly across the compiler optimization levels used to link it to surrounding codes, because the placement of the binary inside a program differs depending on the optimization level, and this difference affects the performance of caches or branch predictors [15].

7 CONCLUSION AND FUTURE WORK

In this paper, we evaluated the overhead of PEBS at high sampling rates to use it for message-level system-noise analysis of high-throughput systems. We found, despite a wide-spread belief that PEBS incurs negligible overhead, that PEBS incurs 200 – 300 nano seconds of CPU overhead per PEBS assist and also cache pollution due to fast data writes. We also showed a way to predict the actual overhead incurred to complex workloads in many cases, and a guide to configure PEBS for online system-noise analysis.

Future work includes two directions: (1) To further analyze the overhead of PEBS. Modeling the cache pollution is needed to apply online system-noise analysis to cache sensitive applications. Identifying the bottleneck of PEBS overhead inside a CPU is also desirable to find a way to reduce the overhead. (2) To use PEBS for real system-noise analysis based on the knowledge given in this paper. Our DPDK-based high-throughput network latency injector [1] suffers sudden latency spikes. As the first step to tackle this issue, we have found that storing the packet IDs into a rarely-used general purpose register and sampling the register value with PEBS (sampling `UOPS_RETIRED.ALL`) can identify intentionally-delayed packets going through a DPDK-based simple packet forwarder.

ACKNOWLEDGMENTS

This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO). Part of this work is supported by JSPS Grant KAKENHI 16K00115. The authors thank Dr. Steven Lynden for the fruitful comments.

REFERENCES

[1] Shuhei Aketa, Takahiro Hirofuchi, and Ryousei Takano. 2017. DEMU: A DPDK-based Network Latency Emulator. In *Proceedings of IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 1–6.
[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. 1994. *THE NAS PARALLEL BENCHMARKS.* Technical Report RNR-94-007. NASA.
[3] Intel Corpporation. 2016. Intel 64 and IA-32 Architectures Software Developer Manuals. (2016). https://software.intel.com/articles/intel-sdm
[4] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*. 1–14.
[5] Florian Larysch. 2016. *Fine-Grained Estimation of Memory Bandwidth Utilization.* Master Thesis. Karlsruhe Institute of Technology (KIT), Germany.
[6] Jacob Machina and Angela Sodan. 2009. Predicting cache needs and cache sensitivity for applications in cloud computing on CMP servers with configurable caches. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. 1–8.
[7] Khang T Nguyen. 2016. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. (2016). https://software.intel.com/articles/introduction-to-cache-allocation-technology
[8] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. 2015. Establishing a Base of Trust with Performance Counters for Enterprise Workloads. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*. 541–548.
[9] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 293–307.
[10] Harald Servat, Germán Llort, Juan González, Judit Giménez, and Jesús Labarta. 2015. Low-Overhead Detection of Memory Access Patterns and Their Time Evolution. In *Proceedings of International Conference on Parallel and Distributed Computing (Euro-Par)*. 57–69.
[11] Vish Viswanathan. 2016. Intel Memory Latency Checker v3.1a. (2016). https://software.intel.com/articles/intelr-memory-latency-checker
[12] Vincent M. Weaver. 2015. Self-monitoring overhead of the Linux perf_event performance counter interface. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 102–111.
[13] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. 2013. Non-determinism and overcount on modern hardware performance counter implementations. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 215–224.
[14] Thomas Willhalm, Roman Dementiev, and Patrick Fay. 2017. Intel Performance Counter Monitor - A better way to measure CPU utilization. (2017). https://software.intel.com/articles/intel-performance-counter-monitor
[15] Masahiro Yasugi, Yuki Matsuda, and Tomoharu Ugawa. 2013. A Proper Performance Evaluation System That Summarizes Code Placement Effects. In *Proceedings of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 41–48.
[16] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. 2009. Accuracy of performance counter measurements. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 23–32.
[17] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 629–644.