On the Precision of Precise Event Based Sampling

Jifei Yi, Benchao Dong, Mingkai Dong, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

ABSTRACT

Many performance studies rely on Intel's Precise Event Based Sampling (PEBS) to collect processor events, where precision is a key for the reliability of analysis. In this paper, we make a study on the precision of PEBS and show that, while by its name being precise, PEBS can cause mistakes under shadowing, which may make the analysis unreliable. We then show how to remedy such imprecision by artificially inserting bogus instructions. Evaluation shows that our remedy leads to more precise event samples and thus more reliable performance analysis.

KEYWORDS

PEBS, Sampling, Accuracy

ACM Reference Format:
Jifei Yi, Benchao Dong, Mingkai Dong, Haibo Chen. 2020. On the Precision of Precise Event Based Sampling. In *11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20), August 24–25, 2020, Tsukuba, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409963.3410490

1 INTRODUCTION

The complexity of software systems has made it notoriously hard to diagnose performance problems. To this end, modern processors provide performance monitoring units (PMU) [8] to collect hardware-level performance events, such as CPU cycles, cache misses, and memory accesses, for online or offline performance analysis.

To strike a good balance between intrusiveness and precision, PMU widely used sampling for profiling. With a configured event and threshold, PMU triggers an *overflow* interrupt after the number of specified event occurrences exceeds the threshold. Then, the software interrupt handler can take the chance to record useful information about the context of the interrupt, which is usually called a sample. Ideally, all monitored events should be sampled with exactly equal probability. However, PMU sampling is usually imprecise. Due to hardware limits, there is an inevitable time delay from the PMU overflow interrupt to the signal handler, which is called skid. As a result, the software handler might, by mistake, sample an instruction following the one that should be sampled.

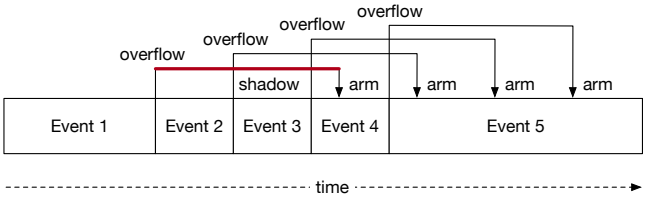


Figure 1: Event1-Event5 are five same events to be monitored. Due to the shadow effect, the PMU counter overflow caused by event 1 will record event 4, while those generated by events 2–4 will take a sample of event 5.

To make sampling more accurate and efficient, modern Intel processors are equipped with Precise Event-Based Sampling (PEBS)[1] technique. When a PMU counter overflow occurs, instead of triggering interrupts, the PEBS hardware is armed. The hardware will then be ready to record the next monitored event. It will record the event in a software-configured memory region [14]. System software can then inspect the samples later.

However, PEBS is not a silver bullet. Our study shows that it suffers from *the shadow effect* — a time delay between the PMU counter overflow and the arming of the PEBS hardware, during which any event[2] that occurs cannot be captured. Figure 1 illustrates this problem. Assuming that we have five consecutive events if a PMU counter overflow is generated when the first event occurs, the shadow hides event2 and event3. So the PEBS hardware will record event4. For the same reason, the PMU counter overflows generated by events 2-4 will finally take the sample of event 5.

No previous work has specifically studied how the shadow effect affect PEBS. In this paper, we systematically studied PEBS to better understand the precision of PEBS. With carefully-designed experiments, we demonstrate that the shadow effect can introduce significant biases in sampling

APSys '20, August 24–25, 2020, Tsukuba, Japan

ACM ISBN 978-1-4503-8069-0/20/08…$15.00
https://doi.org/10.1145/3409963.3410490

[1] Also called Processor Event-Based Sampling in some versions of Intel documents.
[2] An event in this paper indicates an instruction that triggers an increase in the PMU counter of the monitored performance event.

results under some circumstances. After analyzing the phenomenon, we propose the sampling bias rule of PEBS: with a reasonable sampling period, *an event has a higher probability of being sampled when further away from the previous adjacent event.* This probability reaches its maximum after the distance is greater than the shadow's length and no longer increases with the distance. We then further propose a remedy method to the shadow effect problem by simply inserting nop instructions at the appropriate positions for some performance events. Evaluation shows that our remedy can effectively eliminate the shadow effect and provide accurate and precise sampling for performance tuning.

In summary, the contributions of this paper include:

- A study uncovering the problem of PEBS that leads to bias sampling results (§2);
- An analysis that uncovers the reason for the occurrence of PEBS bias and under what conditions it may occur (§3);
- A remedy to alleviate this problem over many performance events (§4) with evaluation to illustrate the effectiveness and overhead (§5).

2 BACKGROUND AND MOTIVATION

2.1 PMU and PEBS

PEBS is capable of recording more information than conventional PMU sampling, including the instruction pointer (IP) of each sample, the value in general registers, and so on. For the memory read and write events, it can also record the destination address of the memory accesses.

Sampling technology is widely used in mainstream performance tools, including perf [1], Oprofile [4], Intel VTune [9], etc. There is also a lot of work based on conventional PMU sampling or PEBS [2, 7, 13, 17]. Sampler [13] utilizes the memory access samples obtained by PEBS to detect memory errors.

Unfortunately, neither conventional PMU sampling nor PEBS is precise [15]. According to Chen et al. [3], Levinthal [10], and Nowak et al. [11], the imprecision comes from three sources: the synchronization of the monitored program with the sampling period, and the aforementioned skid and the shadow effect. Nowak et al. [11] uses PEBS to do the analysis of basic blocks. They do not need the instruction level precision. PEBS is able to get more accurate results than conventional PMU sampling for basic block analysis. Xu et al. [16] solved the inaccurate problem of conventional PMU sampling but did not study the problem of PEBS because the results of PEBS sampling is unpredictable and difficult to reason.

```
while (offset not reach the end) {
    lfence();
    read8bytes(base+offset);        // R1
    read8bytes(base+offset+8);      // R2
    read8bytes(base+offset+16);     // R3
    read8bytes(base+offset+24);     // R4
    offset += 32;
}
```

Snippet 1: Pseudo code of mem_loads main loop

2.2 Perf

Perf is a performance analyzing tool in Linux from kernel v2.6.31 [6]. It provides many commands to profile from a simple program like hello world to the entire system, including the kernel code. It supports performance counters, tracepoints, and dynamic probes [5].

We can use utilities provided by perf to easily count and sample specific performance events to analyze program performance and locate the performance bottlenecks. All "precise events" in perf are sampled with the PEBS hardware.

2.3 Imprecision of PEBS Sampling Results

PEBS supports various events, including retired memory instructions, cache hit or miss (L1, L2, and L3), branch instructions, retired instructions, etc [8].

However, PEBS is not always accurate. We demonstrate the accuracy issue of PEBS by testing the `mem_inst_retired.all_loads` event with the code shown in Snippet 1. In the test, we mmap a 1GB file in tmpfs and read the mapped area in a loop with a granularity of 8 bytes. Within each loop round, we use an `lfence` instruction to ensure that all reads in the previous round have finished, and then we read the next four 8-byte data. To prevent irrelevant memory read instructions from affecting the sampling results, we store the unrelated variables in registers to ensure that only the four memory read instructions inside the loop can trigger `mem_inst_retired.all_loads` events.

We execute the code and record samples of the `mem_inst_retired.all_loads` event using perf with PEBS enabled. We distinguish the four memory reads by inspecting the IP in the samples. To avoid the influence of a particular sampling period on the results, we conduct the test three times, with three prime numbers, 10007, 2347, 991, as the sampling period, respectively. Taking the sampling period 10007 as an example, accessing 1GB memory at 8-byte granularity should yield a total of about $2^{30}/8/10007 = 13412$ samples. With a precise sampling, the four instructions should have a similar probability to be sampled, i.e., about $13412/4 = 3353$ samples per instruction. However, the results in Figure 2 show that all samples turn out to be the first read instruction

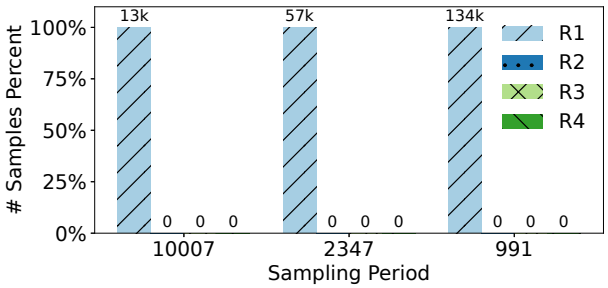


Figure 2: PEBS sampling results for the code in Snippet 1. The height of the bar represents the percentage of the instruction with the sampling period on the x-axis. The number above the bar is the absolute number of samples for this instruction. Different colors represent the results of different instructions. Regardless of the sampling period, all samples are focused on instruction R1, and no samples fall on R2, R3, or R4.

(R1), and the remaining three read instructions are completely uncaught. Since the total number of samples is correct, the extremely biased distribution suggests that samples which should fall on R2–R4 end up on R1 instead. The other two sampling periods present similar results.

3 DIVING INTO THE SHADOW

As the shadow effect, the time delay from the PMU counter overflow to the arming of the PEBS hardware, shades any event in between, it is easy to put the blame on the shadow effect for the abnormality in Figure 2. However, no one has ever studied the issue in depth and tried to explain how the shadow effect affects. In the section, we dive into the shadow effect and look for a general rule that can explain the imprecision of PEBS.

Since the shadow is timing-related, we first measure the number of cycles for `lfence` and the 4 read instructions in Snippet 1 using `rdtsc` [12]. The instruction `lfence` takes approximately 12–14 cycles to complete. The read instructions are so fast that `rdtsc` can barely measure their execution time, except for a small number of R1, which are slower due to cache misses.

We then try to understand how the shadow effect leads to the abnormality and show our explanation in Figure 3. For brevity, we denote the length of the shadow as S during the explanation. Our basic assumption is that S is longer than the time of 3 fast read instructions and shorter than the execution time of `lfence`. As a result, the PMU counter overflow generated by whichever read instruction ends up arming the PEBS hardware during the execution of `lfence` and subsequently sampling the next event, i.e., R1. It is also reasonable if S is longer than the execution time of the `lfence` instruction. That is, the overflow generated by a read instruction

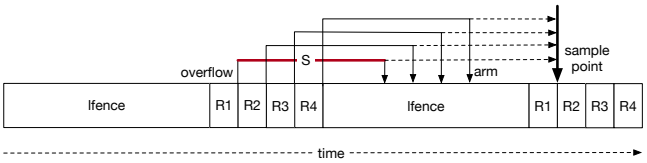


Figure 3: Whenever the PMU counter overflow is generated, it ends up arming the PEBS hardware during the execution of the `lfence` instruction and eventually records R1.

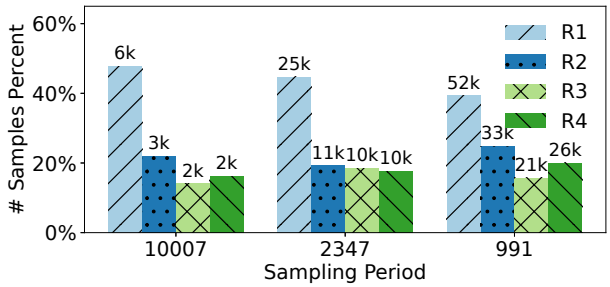


Figure 4: After removing the `lfence` instruction, regardless of the sampling period, the proportion of R1 is always significantly higher than the other three instructions.

may span multiple `lfence` instructions before arming the PEBS hardware. However, our following experiments in §5.2 show that it's not the latter case.

Since the extremely biased distribution is caused by the relatively slow `lfence` instruction, we remove the `lfence` instruction in Snippet 1 and repeat the experiment to see whether the shadow effect vanishes.

To our surprise, the results in Figure 4 show that the distribution is still not fair enough. Regardless of the sampling period, the proportion of R1 samples (45%–54%) is much higher than that of the other three instructions (all less than 22%). As there is no slow instruction like `lfence`, why are the sampling results still biased?

The answer turns out to be cache misses. Since the four instructions are accessing 32-byte-aligned contiguous memory, only the first instruction (R1) may cause a cache miss, which will take more time to read the data from the main memory. The next three read instructions (R2–R4) are served by the cache directly. On cache hits, the probability of the four instructions being sampled is theoretically equal. However, since R1 may be prolonged due to cache misses, the PMU counter overflows generated sometime before R1 will eventually sample R1. Thus, R1 has a higher probability of being sampled than the other three instructions. Figure 5 demonstrates the scenario. The first four black arrows are all able to sample correctly, but all the blue arrows will eventually

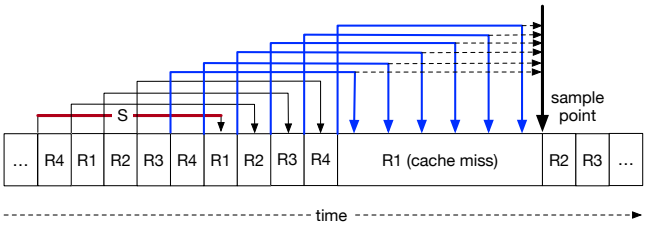


Figure 5: Due to the shadow effect, PMU counter overflows generated some time before the cache miss (the blue arrows) end up recording the cache miss instruction.

```
for (i = 0; i < 1<<27; i++) {          // Br1
    if (i % 2 == 0)                    // Br2
        nop();
    /* do something time-consuming */
    if (i % 3 == 0)                    // Br3
        nop();
}
```

Snippet 2: Code of branch instructions test

sample R1, which makes R1 has a higher sample percentage than others.

According to these observations, we summarize the sampling bias rule of PEBS as below. With a reasonable sampling period, **an event has a higher probability of being sampled when it is further away from the previous adjacent event.** This probability reaches its maximum after the distance is greater than the length of the shadow and no longer increases with the distance.

We then conduct similar experiments on other PEBS performance events to check whether the bias also exists and whether the rule applies.

The performance event `br_inst_retired.all_branches_pebs` tracks all retired branch instructions [8]. We construct a piece of code (in Snippet 2) that loops 128M times. Three branch instructions are executed in each loop round, including a loop condition branch (indicated as Br1) and two more branch instructions (Br2 and Br3). To validate the sampling rule we observed, we add some time-consuming code (50 nops in the experiment) between Br2 and Br3.

Again, all three branch instructions should be sampled similarly with a precise sampling mechanism. But the results in Figure 6 show that almost all samples are Br3, which has the furthest distance to its previous branching event, regardless of the sampling period.

All the previous performance events have the same characteristic of fast execution. The previous analysis shows that the biased distribution in Figure 4 is caused by cache misses.

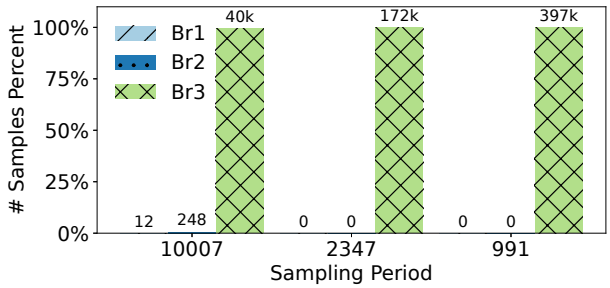


Figure 6: Regardless of the sampling period, all samples fall on Br3. Theoretically, all three instructions should have similar numbers of samples.

So we construct multiple consecutive cache misses to test whether the same problem occurs again.

There are many PEBS performance events related to cache. We select the L3 cache miss event called `mem_load_retired.l3_miss`, and construct four consecutive L3 cache miss instructions as the code in Snippet 3 shows. The four addresses are flushed out of the cache with four `clflush` instructions at the beginning of each loop. Then one `mfence` is used to ensure the data not in the cache before executing the subsequent memory read instructions. We then measure the execution times of these four read instructions with `rdtsc`. Their execution time is all around 200 cycles, which further confirms that all of the instructions (R1–R4) can cause cache misses events.

```
while (offset not reach the end) {
    clflush(base1+offset);
    clflush(base2+offset);
    clflush(base3+offset);
    clflush(base4+offset);
    mfence();
    read8bytes(base1+offset);      // R1
    read8bytes(base2+offset);      // R2
    read8bytes(base3+offset);      // R3
    read8bytes(base4+offset);      // R4
    offset += 8;
}
```

Snippet 3: Code of L3 cache miss test

We then sample the code and find that it is still biased, as shown in Figure 7. While it does not appear extremely biased as in the previous events, the results are still very significantly off and the rule still applies.

4 REMEDY

The main reason for the inaccuracy of PEBS is the shadow effect. Events that appear in the shadow cannot be captured, so the most intuitive solution to this problem is to keep the

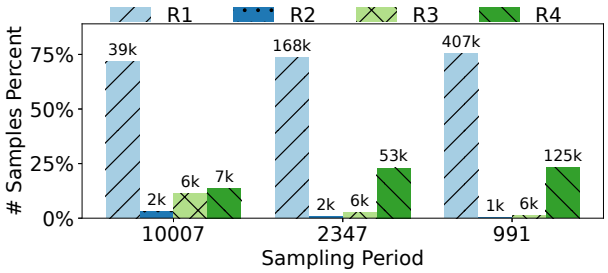


Figure 7: The sample proportion of L3 miss performance event. For all sampling periods, R1 has a significantly higher sampling proportion than that of the other three instructions.

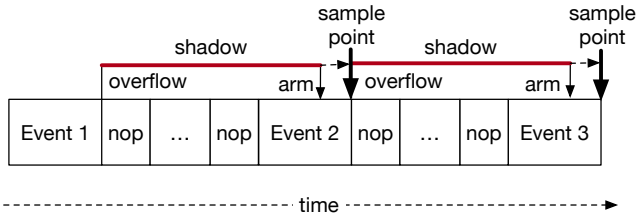


Figure 8: Ensure that the PMU counter overflow caused by the previous event must record the next event by inserting enough nop instructions between two adjacent events.

event out of the shadow, i.e., using other instructions to separate two adjacent events to be tracked. For example, we can insert some unrelated instructions into two identical events that are close together so that the interval time between them is longer than the length of the shadow.

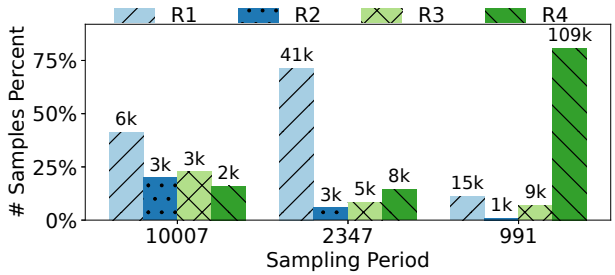
Take the experiment in Snippet 1 without the `lfence` instruction as an example. We keep inserting more nop instructions into the middle of two adjacent instructions until the sampling results just become exactly accurate. We then measure the total time required for instruction and the added nop, which should be exactly just greater than or equal to the length of the shadow.

When we add 43 nops to the end of each read instruction, the sampling results, as shown in Figure 9(a), are still biased. But when we add an extra nop, i.e., 44 nops after each read instruction, the sampling becomes very accurate. As shown in Figure 9(b).

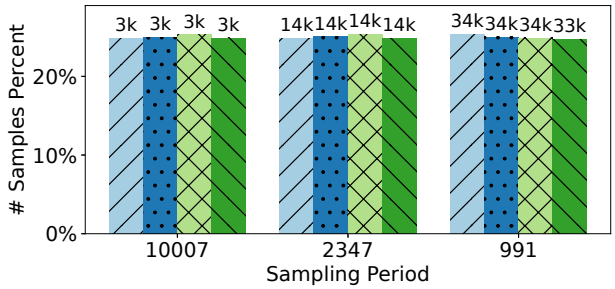
For some of the PEBS performance events, we can add some nop instructions to the end of each instruction that may trigger a PMU counter overflow to achieve accurate sampling at the instruction level.

5 EVALUATION

In this section, we repeat all the previous tests using the method of adding nops The correctness of this method is



(a) The sample proportion of memory reads events with 43 nop instructions added after every read instruction.



(b) The sample proportion of memory reads events with 44 nop instructions added after every read instruction.

Figure 9: The sampling ratio after inserting different numbers of nop instructions after each read instruction in Snippet 1 without the `lfence` instruction.

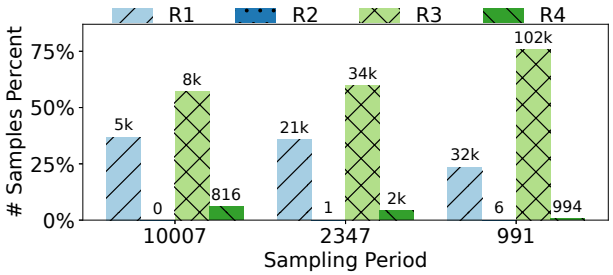
first verified and evaluated to see how many nop instruction needed to be added for each performance event to ensure accurate sampling. The overhead of the inserted nop instructions in the corresponding experiment is then given. All results in this section are averages of five experiments to reduce chance errors.

5.1 Evaluation Setup

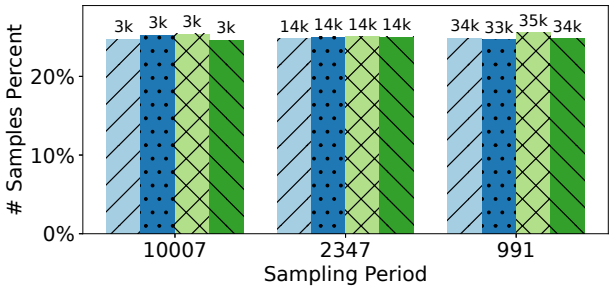
Experiments are conducted on a server with two ten-core Intel® Xeon® Gold 5215M CPUs. The server is equipped with 375GB DDR4 DRAM distributed on two NUMA nodes. The version of Linux and perf is 5.3.11.

5.2 Accuracy

First, we add some nop instructions to the end of each read instruction in Snippet 1 to see if we can get sufficiently accurate sampling results. When we add 42 nops, the sampling results are shown in Figure 10(a), with almost no samples falling on R2, very few on R4, and more than half on R3. After adding 43 nops, the sampling results become accurate, as shown in Figure 10(b), no matter what sampling period we choose. The percentage of samples for each of the four read instructions is around 25%. The total execution time

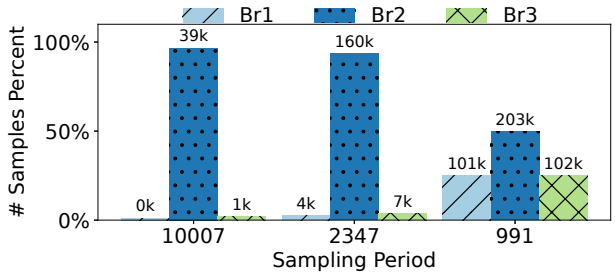


(a) The sample proportion of memory reads events with 42 nop instructions added after every read instruction.

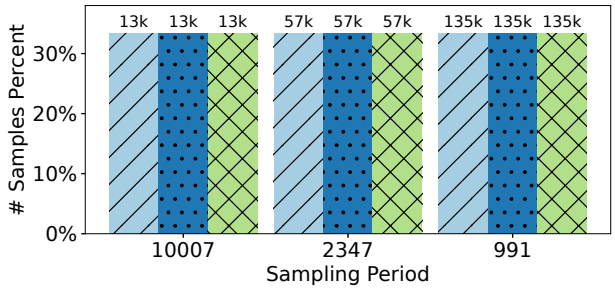


(b) The sample proportion of memory reads events with 43 nop instructions added after every read instruction.

Figure 10: The sampling ratio after inserting different numbers of nop instructions after each read instruction in Snippet 1.



(a) The sample proportion of branching events with 45 nop instructions added after every branch.



(b) The sample proportion of branching events with 46 nop instructions added after every branch.

Figure 11: The sampling results after inserting different nop instructions in Snippet 2.

of one read and 43 nops measured by `rdtsc` is about 8–10 cycles.

The result of the branch instructions is shown in Figure 11. When we insert 45 nop instructions into the middle of two branches, the sampling results are still very biased (Figure 11(a)). But when the number of nop instructions reaches 46, the results become very accurate, as shown in Figure 11(b). The cycles required for a branch instruction plus 46 nop instructions measured by `rdtsc` is about 10, which suggests that the length of the shadow of the branching event is around 10 cycles.

Figure 12 shows the result for the L3 cache miss events in Snippet 3. Surprisingly, for the longer cache miss instructions, the number of nop instructions required to ensure accurate sampling results is also greater. 220 nop instructions cannot guarantee accurate sampling results, and the sampling results obtained from different sampling periods have large deviations (Figure 12(a)). But 221 nop instructions can ensure accurate sampling results at any sampling period (not coincident with pattern of the workload) (Figure 12(b)).

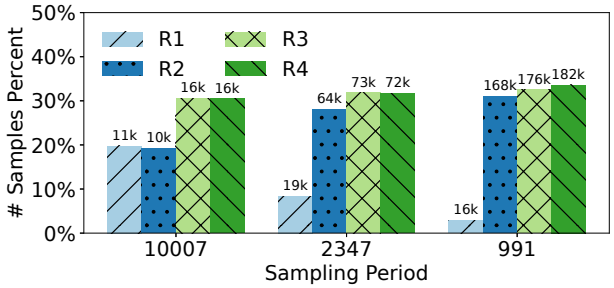
The number cycles required for one cache miss instruction and 221 nop instructions is about 240, which shows that the shadow of the cache miss events is inherently longer than other instructions in PEBS. These results show that the shadow lengths of different PEBS performance events are inconsistent. For many PEBS performance events, the remedy by inserting nop instructions is able to solve the PEBS sampling bias problem.

5.3 Overhead

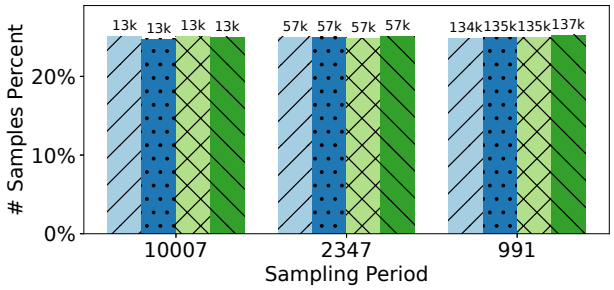
This approach has the obvious drawback of incurring the additional overhead associated with the code belonging to the tracked event related to the percentage of instructions. Below we test how much performance overhead the method incurs by adding nop instructions after each instruction.

In this part, we first measure the execution time of codes from Snippet 1 to Snippet 3 separately. Then we insert enough nop instructions in the same way as previous that ensure accurate sampling of PEBS and measure its running time again. The difference between the two execution times is the additional performance overhead of our approach, as shown in Table 1.

For the previous tests, the performance overhead caused by the nop instructions is around 33%–222%. Since not every instruction in the actual scenario can trigger the event to be tracked, it is not necessary to insert nop instructions after every instruction. So the cost in the actual scenario should be lower than the overhead in our experiments. Simultaneously, the sampling technique is typically used to analyze



(a) The sample proportion of L3 cache miss events with 220 nop instructions added after every read instruction.



(b) The sample proportion of L3 cache miss events with 221 nop instructions added after every read instruction.

Figure 12: The sampling results after inserting different nop instructions in Snippet 3.

Table 1: Execution time before and after inserting enough nop instructions on Intel® Xeon® Gold 5215M Processors.

Code in the main loop	Overhead
lfence → 4 reads (Snippet 1)	134%
4 reads (Snippet 1 without lfence)	222%
3 branches (Snippet 2)	130%
4 clflush → mfence → 4 reads (Snippet 3)	33%

and optimize program performance and is not used in the production environment; this overhead is acceptable.

5.4 Experiment Results on Other Platforms

We repeat all the above experiments on two other platforms. The first has two twenty-core Intel® Xeon® Gold 6138 CPUs and 188GB DDR4 DRAM on two NUMA nodes. The version of Linux kernel and perf is 4.19.32.

The second has a six-core Intel® Core™ i7-8700 processor and 32GB DDR4 DRAM on one NUMA node. The version of kernel and perf is 5.2.5.

The results of PEBS sampling also suffer from the same bias problem, which can be fixed when we insert enough nop instructions after each instruction that may trigger an event, as shown in Table 2. The corresponding overheads introduced by these nop instructions are shown in Table 3.

Table 2: The minimal number of nop instructions to be inserted after each event to ensure proper sampling on Intel® Xeon® Gold 6138 and Intel® Core™ i7-8700 Processors.

PEBS Performance Event	Gold 6138	i7-8700
mem_inst_retired.all_loads (Snippet 1)	43	43
mem_inst_retired.all_loads (Snippet 1 without lfence)	44	44
br_inst_retired.all_branches_pebs (Snippet 2)	46	46
mem_load_retired.l3_miss (Snippet 3)	221	221

Table 3: The overhead of inserting enough nop instructions for different performance events on Intel® Xeon® Gold 6138 and Intel® Core™ i7-8700 Processors.

Code in the main loop	Gold 6138	i7-8700
lfence → 4 reads (Snippet 1)	131%	146%
4 reads (Snippet 1 without lfence)	231%	242%
3 branches (Snippet 2)	139%	123%
4 clflush → mfence → 4 reads (Snippet 3)	58%	127%

To ensure all the performance events aforementioned in the same code to be sampled accurately, the number of nop instructions to be inserted remain the same across platforms. This indicates that the shadow length of some performance events may be the same on different hardware platforms. But before we use PEBS on a new hardware platform, we should still measure how many nop instructions need to be inserted in order to make it really "precise".

6 CONCLUSION

The sampling results of the Precise Event Based Sampling (PEBS) are not precise. This paper makes a systematic study on the precision of PEBS and finds the sampling bias rule of PEBS. We then present how to remedy such imprecision by inserting nop instructions. Evaluation shows that the remedy can make PEBS produce real "precise" sampling results with acceptable overhead.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for the constructive comments and suggestions. This work is supported in part by China National Natural Science Foundation (No. 61925206). Haibo Chen (haibochen@sjtu.edu.cn) is the corresponding author.

REFERENCES

[1] 2019. perf - Performance analysis tools for Linux. https://man7.org/linux/man-pages/man1/perf.1.html.

[2] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative evaluation of intel pebs overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*. 1–8.

[3] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. 2010. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 42–52.

[4] William E Cohen. 2004. Tuning programs with OProfile. *Wide Open Magazine* 1 (2004), 53–62.

[5] Arnaldo Carvalho De Melo. 2010. The new linux 'perf' tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[6] Jake Edge. 2009. Perfcounters added to the mainline. https://lwn.net/Articles/339361/.

[7] Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, and Bernd Hamann. 2014. Dissecting on-node memory access performance: a semantic approach. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 166–176.

[8] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html. *Volume 3B: System Programming Guide, Part 2* (2016).

[9] Intel. 2020. Intel® VTune™ Profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[10] David Levinthal. 2009. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide* 30 (2009), 18.

[11] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. 2015. Establishing a base of trust with performance counters for enterprise workloads. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 541–548.

[12] Gabriele Paoloni. 2010. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation* 123 (2010).

[13] Sam Silvestro, Hongyu Liu, Tong Zhang, Changhee Jung, Dongyoon Lee, and Tongping Liu. 2018. Sampler: Pmu-based sampling to detect memory errors latent in production software. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 231–244.

[14] Vincent M Weaver. 2016. *Advanced hardware profiling and sampling (PEBS, IBS, etc.): creating a new PAPI sampling interface*. Technical Report. Tech. rep., University of Maine.

[15] V. M. Weaver, D. Terpstra, and S. Moore. 2013. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 215–224.

[16] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can We Trust Profiling Results? Understanding and Fixing the Inaccuracy in Modern Profilers. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. Association for Computing Machinery, New York, NY, USA, 284–295. https://doi.org/10.1145/3330345.3330371

[17] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. 2011. Security breaches as PMU deviation: detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. 1–5.