Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison

Muhammad Aditya Sasongko, Milind Chabbi, Paul H J Kelly, and Didem Unat, *Member, IEEE*

***Abstract*—Precise event sampling is a profiling feature in commodity processors that can sample hardware events and accurately locate the instructions that trigger the events. This feature has been used in a large number of tools to detect application performance issues. Although precise event sampling is readily supported in modern multicore architectures, vendor supports exhibit great differences that affect their accuracy, stability, overhead, and functionality. This work presents the most comprehensive study to date on benchmarking the event sampling features of Intel PEBS and AMD IBS and performs in-depth analysis on key differences through series of microbenchmarks. Our qualitative and quantitative analysis shows that PEBS allows finer-grained and more accurate sampling of hardware events, while IBS offers richer set of information at each sample though it suffers from lower accuracy and stability. Moreover, OS signal delivery, which is a common method used by the profiling software, introduces significant time overhead to the original overhead incurred by the hardware mechanisms in both PEBS and IBS. We also found that both PEBS and IBS have bias in sampling events across multiple different locations in a code. Lastly, we demonstrate how our findings on microbenchmarks under different thread counts hold for a full-fledged profiling tool that runs on the state-of-the-art Intel and AMD machines. Overall our detailed comparisons serve as a great reference and provide invaluable information for hardware designers and profiling tool developers.**

***Index Terms*—Precise event sampling, PMUs, profiling.**

I. Introduction

PRECISE event sampling (PES) is a powerful profiling feature supported by Performance Monitoring Units (PMUs) in modern CPUs. It has been incorporated in a number of profiling tools that identify performance bottlenecks in shared-memory parallel applications. Some bottlenecks in multithreaded code that these tools can detect are inter-thread coherence traffic [1], [2], [3], false sharing [3], [4], [5], [6], [7], [8], [9], [10], long latency remote memory accesses in NUMA multicore systems [11], [12], [13], data locality problems [14], [15], performance degrading bandwidth consumption[10], and conflict cache misses [16]. In addition to identifying the bottlenecks, PES can help pinpoint the source code lines and data objects causing the bottlenecks through its ability to sample instruction pointers and effective addresses of the operations. Compared to alternatives such as cycle-accurate hardware simulators [17], [18] and binary instrumentation [19], [20], techniques that leverage PES incur much lower time and memory overheads as they employ existing hardware features to capture real hardware events without introducing additional software layer.

Intel supports precise event sampling through Processor Event Based Sampling (PEBS) [21] that is available since Intel Nehalem. Many researchers have developed tools for PEBS [1], [2], [3], [8], [9], [11], [15], [16], [22], [23]. Similarly, AMD processors come with Instruction-Based Sampling (IBS) [24] that is supported in AMD Opteron (microarchitecture family 10h) and its successors. A number of tools have also been developed using IBS [7], [11], [12], [13], [22], [25], [26], [27], [28]. The event sampling in IBM PowerPC architecture is provided through Marked Event Sampling (MRK) [29] that is available since IBM POWER5. This capability is also recently supported in ARM through Statistical Profiling Extension (SPE) introduced in Armv8.2 [30].

Despite the fact that event sampling is commonly used for developing profiling tools, there exists no rigorous study that benchmarks this capability in the microarchitecture. In this paper, we analyze and compare the precise event sampling facilities of two major vendors namely, Intel and AMD, in depth through extensive benchmarks. Intel PEBS and AMD IBS adopt drastically different designs that affect the accuracy, stability, overhead and functionality of the sampling facility. While the outcomes of this work can be used by the profiling tool developers and performance analysts to better understand the behaviours of the tools that they develop or use, hardware designers can leverage the findings to design better PMUs not only for x86-based systems but also for ARM [30], [31] and emerging RISC-V processors [32], [33].

We firstly present qualitative differences between Intel and AMD in terms of usable counters, type of precise events, sampled data, and execution mode. Based on the observations on the qualitative characteristics, we developed a number of microbenchmarks that can assess the effects of the observed qualitative characteristics. The benchmarks were carefully

Manuscript received 24 June 2022; revised 17 February 2023; accepted 9 March 2023. Date of publication 14 March 2023; date of current version 24 March 2023. This work was supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under Grant 949587 and in part by the Scientific and Technological Research Council of Turkey (TUBITAK), under Grant 120E492. The work of Didem Unat was supported in part by the Royal Society-Newton Advanced Fellowship. Recommended for acceptance by T. Estrada. *(Corresponding author: Muhammad Aditya Sasongko.)*

Muhammad Aditya Sasongko and Didem Unat are with the Department of Computer Engineering, Koç University, 34450 Istanbul, Turkey (e-mail: msasongko17@ku.edu.tr; dunat@ku.edu.tr).

Milind Chabbi is with the Scalable Machines Research, San Jose, CA 95134 USA (e-mail: milind@ScalableMachines.org).

Paul H J Kelly is with the Department of Computing, Imperial College London, SW7 2BX London, U.K. (e-mail: p.kelly@imperial.ac.uk).

Digital Object Identifier 10.1109/TPDS.2023.3257105

designed to eliminate any artifacts and have ground truths for the evaluated behaviors. Through these benchmarks, we then quantitatively compare the two facilities in terms of accuracy, time and memory overheads in sampling individual and multiple events, across different thread counts and different sampling intervals. We also evaluate the stability and sampling bias of both facilities, and analyze the ability in attributing samples to the instructions that trigger the samples and the execution modes of those instructions, i.e. kernel mode or user mode. Lastly, to demonstrate how these microarchitectural characteristics impact tools that profile multithreaded applications, we study a full-fledged open-source tool, namely ComDetective [3], [34], that employs precise event sampling to detect inter-thread communication.

A. Findings, Insights and Contributions

Our findings based on the quantitative and qualitative study are as follows. (1) Intel PEBS offers a large set of specific hardware events such as branches, memory loads, etc to select from, while AMD IBS has only two *flavors* of sampling: instruction fetch and micro-operation execution. One impact of this difference is that PEBS is more accurate than IBS in capturing proportional number of samples from specific events, such as loads or branches. This accuracy problem in IBS is especially worse at low thread counts. (2) Though IBS supports fewer sampling choices than PEBS, it offers richer information in each sample, which shows, for example, the origin of accessed data in memory hierarchy. As a consequence, PEBS would have to monitor multiple events simultaneously in order to generate the similar level of information as in one IBS run. (3) Intel PEBS shares the same counters with other non-PEBS PMU events, while AMD IBS has its own internal counters. As a result, the number of different events that PEBS can monitor without multiplexing is limited to the number of available PMU counters per logical core. Multiplexing in PEBS suffers from sample loss, leading to reduced accuracy. (4) Across multiple runs, PEBS displays lower variation in capturing sample counts thus PEBS is more stable than IBS in this regard. (5) AMD Zen3 and Intel Skylake introduce similar time overheads but AMD Zen 2 and Intel Cascade Lake introduces higher overhead than the other two. Thus, the time overhead depends on the specific microarchitecture in use. (6) In both PEBS and IBS, OS signal delivery to user space introduces significant time overhead, which affects the overhead felt by the end-user. Moreover, the time overhead of IBS is sensitive to thread counts as its signal delivery overhead is drastically higher at high thread counts. (7) Both PEBS and IBS exhibit similar and negligible memory overheads on a single thread. However, when running on multiple threads, memory overheads increase as thread count increases except for Cascade Lake. (8) IBS is very sensitive to sampling interval and its accuracy significantly drops at high sampling frequency. PEBS has high accuracy regardless of sampling rate. (9) Both PEBS and IBS are equally biased in sampling an event from multiple different instructions. (10) Lastly, PEBS can be programmed to sample events that execute only in user mode, only in kernel mode, or in any of the two modes, while IBS samples any fetch or micro-operation without discriminating its execution mode.

In summary, our contributions are as follows

- Presenting the most comprehensive study to date on precise event sampling supported by two major vendors. See comparison in Table IV.
- Detailed qualitative and quantitative comparisons of microarchitectural characteristics between Intel PEBS and AMD IBS and demonstrating their accuracy, stability, bias, functionality and overheads
- A suite of synthetic benchmarks that can be used for extending this study to other vendors and multicore architectures
- Providing invaluable information both to the hardware designers and tool developers through our findings that would help understand and improve their designs.

All our codes and data are available to public at: https://github.com/ParCoreLab/pes-artifact.

II. Background

PMU is a special on-chip hardware that can be used to monitor hardware events, such as loads, stores, retired instructions, or software events like page faults, context switches. This section describes the mechanism of PES, a subset of PMUs that is the subject of this work, in Intel and AMD.

A. Precise Event Sampling Support in Intel

On Intel, PES can utilize any programmable counters available in PMUs [35], which consist of a number of registers and counters. For example, global control registers are used to globally enable or disable event counters or PES ability of each counter. Status registers contain info about the capabilities supported by the PMUs or the overflow status of each event counter. Event select registers are used to choose the hardware or software event, such as retired instruction, load instruction, or page fault, to be monitored. A PMC (Performance Monitoring Counter) counts the occurrences of a monitored event.

To enable PES, a programmable counter is enabled along with its PEBS capability by global control registers [35]. This counter is then configured to monitor a targeted hardware event by programming its event select register with the mask and number of the targeted event. The counter is also configured to have a counter overflow in every elapsing of a specified number of events, which is the *sampling interval*. When a counter overflows, PEBS is armed to trap the next occurrence of the monitored event. When the next monitored event occurs, a mechanism called *PEBS assist* will copy the machine state to a *PEBS buffer* in a process we refer to as *sampling*. The machine state and other data such as effective address and load latency collected by the PEBS assist are grouped into a data structure called a *PEBS record* in the PEBS buffer. When the number of PEBS records in the PEBS buffer has reached a predefined threshold, a hardware interrupt is triggered and handled by an interrupt handler that is a part of the OS. The interrupt handler reads the PEBS records in the PEBS buffer, clears the buffer, and sends an OS signal to a user process or thread that will collect the

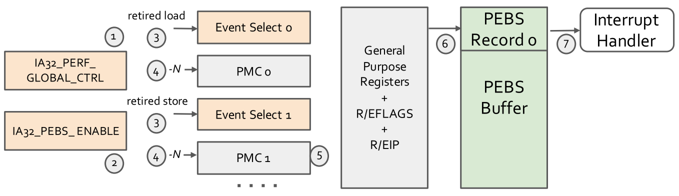

Fig. 1. One possible execution scenario of Intel PEBS: (1) Global control register `IA32_PERF_GLOBAL_CTRL` enables PMC0 and PMC1 by setting its bits corresponding to both counters to one. (2) Global control register `IA32_PEBS_ENABLE` enables PEBS in PMC0 and PMC1 by setting the bits corresponding to both counters to one. (3) The event select registers `IA32_PERFEVTSEL0` and `IA32_PERFEVTSEL1` are programmed to make PMC0 and PMC1 count retired loads and retired stores, respectively. (4) The configured PMCs are preloaded with the sampling interval, -N, so that they overflow on elapsing N events. (5) The profiled program executes for a while; PMC1's counter overflows after N stores occur. PEBS is *armed* to trap the next store. PMC 1 is preloaded with -N again. (6) Another store occurs after the counter overflow. The armed PEBS hardware traps the access and a microcode records the machine state in a PEBS record in PEBS buffer. (7) If the number of records has reached a specified threshold (1 in this case), an interrupt is triggered, and an interrupt handler transfers the PEBS records to user space.

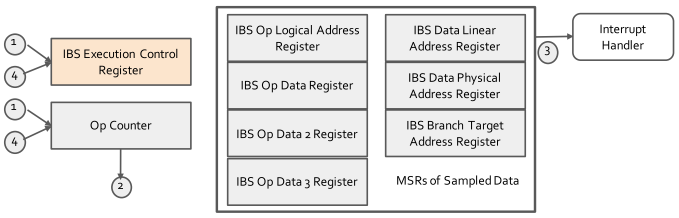
IBS Execution Control Register
Op Counter
IBS Op Logical Address Register
IBS Op Data Register
IBS Op Data 2 Register
IBS Op Data 3 Register
IBS Data Linear Address Register
IBS Data Physical Address Register
IBS Branch Target Address Register
MSRs of Sampled Data
Interrupt Handler

Fig. 2. One execution scenario of AMD IBS: (1) IBS execution control register in CPU is programmed to make the IBS hardware count and sample executed micro-operations. The sampling interval is also written as a field in the control register. The op counter is set to a pseudorandom 7-bit value in the range of 1 to 127. (2) After the profiled thread executes for a while, the value in the op counter equals the sampling interval. As it happens, the next micro-operation will be tagged for sampling. (3) The tagged micro-operation retires. The execution info of the micro-operation is recorded in a number of MSRs. After that, an interrupt is triggered, and the interrupt handler copies the recorded data in the MSRs to a memory buffer in kernel space. (4) Upon copying sampled data, the interrupt handler configures the control register again to re-enable IBS, and the op counter is preloaded with another pseudorandom 7-bit value.

sampled data. On receiving the signal, the user process/thread retrieves the sampled data and uses it for profiling. Fig. 1 shows an example execution of PEBS when profiling retired load and store micro-operations.

PEBS has been used to sample a variety of events to capture performance bottlenecks in multithreaded programs. In [11], PEBS and PEBS with load latency (PEBS-LL) are used to sample retired instructions and long-latency memory loads to approximate NUMA latency per instruction by dividing the total latency of the sampled loads with the number of instruction samples. A work in [8] samples memory loads and stores in each CPU core and has each core monitor effective addresses sampled by other cores using debug registers to detect false sharing. Another work in [16] samples loads that miss in L1 data caches to capture cache access patterns and identify cache conflict misses.

B. Precise Event Sampling Support in AMD

Different from Intel, PES in AMD, i.e. IBS, employs a hardware-based facility that is separate from the PMUs that are commonly used to count or imprecisely sample specific hardware or software events. This mechanism is based on the instruction sampling technique proposed in [36]. The IBS facility in each CPU core consists of a couple of components: two control registers, two internal counters, and a number of MSRs (Model Specific Registers) for sampled data [37].

This facility allows only two flavors of sampling: instruction fetch sampling (IBS fetch) and micro-operation sampling (IBS op) [24], [37]. Either one of the two control registers is programmed to control the IBS hardware depending on the sampling flavor of interest. If instruction fetch sampling is selected, the fetch control register is programmed. Otherwise, the execution control register is programmed. After the control register is programmed, the internal counter that belongs to the selected sampling flavor (the fetch counter or the op counter) will count the monitored event in the CPU core. When an event sampling occurs, information related to the event is recorded in the MSRs of sampled data that belong to the sampling flavor of interest. Another work in

The counter for micro-operation sampling, i.e. op counter, can be programmed to count either clock cycles or dispatched micro-operations. The latter is the default configuration of the IBS driver in [38]. If it is programmed to count clock cycles, it increments for each clock cycle, and when the counter reaches the specified sampling interval, a micro-operation is selected for sampling from the next dispatch line. If the counter is programmed to count dispatched micro-operations, it increments for every dispatched micro-operation, and when the counter reaches the sampling interval, a micro-operation is selected to be sampled in the next cycle. In this work we only consider the op counter to be programmed for counting dispatched micro-operations because it generates predictable number of samples given a known number of micro-operations in an application, which suits our accuracy verification method.

Fig. 2 shows an execution scenario of IBS when counting and sampling micro-operations.

There have been a number of profiling techniques that leverage AMD IBS to identify performance bottlenecks in multi-threaded applications. A work in [15] runs IBS op and filters in only samples from high-latency memory accesses to identify data structures that incur high latency to be accessed. Another work in [24] demonstrates that IBS fetch can be used to detect regions of code that have a large number of killed instruction fetches due to speculative prefetch.

C. Precise Event Sampling Support in ARM

The PES facility in ARM, i.e. Statistical Profiling Extension (SPE), has the same sampling approach as IBS. Its counter counts dispatched micro-operations, and therefore, facilitates sampling from this event [31].

Our work can be extended to cover ARM SPE however our preliminary study on a high-end state-of-the art ARM processor showed that its precise event sampling support is immature

TABLE I
QUALITATIVE COMPARISON OF INTEL PEBS AND AMD IBS

Aspect	*Intel PEBS	AMD IBS
Usable Counters per thread	4 general-purpose performance counters	2 internal counters (1 per sampling flavor)
Event Type	62 subevents of 12 events	2 sampling flavors
Sampled Data	general purpose registers, RFLAGS register, RIP register, applicable counter, data linear address, data src encoding/store status, latency value, timestamp counter eventingIP, TX abort info	16 attributes in each sampled data for inst. fetch sampling 44 attributes in each sampled data for micro-operation sampling
Execution Mode	User or kernel or both modes	Both user & kernel modes

*This info is valid for cascade lake and skylake microarchitectures [35].

and cannot be directly comparable against PEBS or IBS. For example, as quoted in known issues in ARM forge user guide, SPE might observe unexpectedly high sample counts for branch target instructions and unexpectedly low sample counts for some instructions closely following a branch target [39]. As a result, we opt not to include ARM in this study. We plan, in future work, to extend this work to ARM once hardware becomes available that corrects these documented errata.

III. QUALITATIVE COMPARISON

This section presents qualitative differences between the two precise event sampling schemes, Intel PEBS and AMD IBS. Table I highlights these key differences and the section explains them in detail.

A. Usable Counters

Observation 1. PEBS can use up to 4 counters and monitor up to 4 events without multiplexing in microarchitectures. Starting Ice Lake, PEBS can monitor up to 7, while IBS has two dedicated counters, one for each sampling flavor.

Observation 2. PEBS counters are disabled before interrupt delivery, while IBS counters are disabled by software in the interrupt handler.

Observation 3. If the op counter of IBS is programmed to count dispatched micro-operations, it is always preloaded with a pseudorandom 7-bit value after its sampling interrupt is handled.

PEBS shares the PMCs that are also used by other non-precise PMU events. Microarchitectures before Ice Lake (launched in 2019) allow PEBS to use any of the four general-purpose performance counters in each logical core [35]. In Ice Lake, PEBS can use three additional fixed-function performance counters. When the number of events monitored is higher than the number of counters, OS kernel context-switches the events on the counters. When this multiplexing happens, the approximated counter values are inaccurate, and might cause the events to lose some counter overflows.

Different from PEBS, IBS counts instruction fetches and micro-operations using its own internal counters that are separate from other PMCs in each AMD CPU core [37]. It has two internal counters, one for each sampling flavor. Since these counters are not multiplexed, IBS does not miss an overflow.

Another difference is that Intel PMUs have the capability to disable counters when counter overflow occurs[35]. These counters are disabled before an interrupt is delivered. On the other hand, IBS counters do not have this capability, and thus, they have to be disabled by the OS/driver in the interrupt handler. Furthermore, the IBS counters are randomized after each sampling interrupt is handled. The purpose of this randomization is to minimize any correlation between the timing of the interrupts and the locations of instructions in the code. For the fetch counter, randomization is optional, and it is enabled when certain bit in the control register is set to 1. However, the op counter is always randomized with a pseudorandom 7-bit value when programmed to count dispatched micro-operations. This randomization and the fact that IBS counters are not disabled before an interrupt is delivered might cause the number of events detected by IBS to vary across different runs.

B. Type of Precise Events

Observation 4. PEBS has many choices of precise events to monitor, while IBS has two sampling flavors to choose from. As a result, a PEBS counter can be programmed to count only specific hardware event and trigger sampling only of that event, while an IBS counter can only count instruction fetches or micro-operations indiscriminately and trigger sampling of any event that may or may not be of interest.

PEBS has the ability to monitor hardware events at a finer grain than IBS. For example, in Cascade Lake, in total, there are 62 possible hardware sub-events, each of which is identified by a combination of an event number and a unit mask, that can be monitored using PEBS. When monitoring certain sub-event such as memory load, the used hardware counter will increment only for each occurrence of memory load, and it will trigger sampling of only memory load.

In contrast to PEBS, IBS has only two possible sampling choices: instruction fetch sampling and micro-operation execution sampling. In instruction fetch sampling, IBS counts instruction fetches and samples from them, while in micro-operation execution sampling, IBS counts dispatched micro-operations and samples from retired ones. Consequently, an IBS counter can overflow on any instruction fetch or micro-operation, and the sampled fetch or micro-operation does not have to be the event that is targeted by a profiling code. For instance, a profiling code that aims to profile memory accesses might encounter non-memory access samples during profiling.

C. Sampled Data

Observation 5. IBS generates a rich set of attributes for each sampled fetch or micro-operation that record the hardware events during the execution in CPU pipeline. These events might correspond to multiple different precise events in PEBS. Thus, PEBS might have to monitor multiple events simultaneously in order to get the same level of information as IBS.

Though IBS has only two sampling flavors, each sample has a number of attributes. It generates 16 and 44 attributes in each sampled data for instruction fetch and executed micro-operation, respectively. These attributes can help identify the events that are triggered by each sampled instruction fetch or micro-operation (e.g., whether a sampled micro-operation triggers a load, an L1 cache miss, or an L1 DTLB miss).

In each sample, PEBS generates a PEBS record that contains information related to the sampled event, such as the instruction pointer, the architectural state of the logical core after the event is retired, and the effective address in case the sampled event is a memory access. In addition to this general information, PEBS also offers more detailed information on the origin of accessed data in memory hierarchy if load latency sampling facility is enabled.

There is some common information in the data sampled by both PEBS and IBS, such as memory access latency and data misses in L1 or L2 cache. However, there are some attributes that are included in IBS' sampled data but not in PEBS, and vice versa. One example is the width of accessed memory region, which is an attribute in IBS but not in PEBS. To make up for this, a profiler that uses PEBS will have to use a supplementary library such as Intel XED [40]. Moreover, PEBS has to monitor multiple events simultaneously in order to get the same amount of information offered by one sample of IBS. For instance, to profile loads, stores, and branch instructions, PEBS has to monitor these three events separately, while IBS can profile them in a single micro-operation sampling.

D. Execution Mode

Observation 6. While Intel PMUs can be programmed to count events that execute only in user mode, only in kernel mode, or in any mode indiscriminately, IBS counters can only count instruction fetches and micro-operations without regarding their execution mode.

In each event select register (`IA32_PERFEVTSELx`) of Intel, there are bits that determine whether the controlled counter should count events that execute in user mode or kernel mode. Hence, it is possible to program PEBS to sample events that execute only in user, only in kernel, or any of the two modes. In contrast, IBS does not have this ability supported at hardware level. Consequently, IBS always triggers an interrupt regardless of the execution mode of the event that it samples, and the identification of the execution mode has to be done by the interrupt handler by checking the `user_mode(regs)` macro provided by the Linux kernel. This approach might be less accurate than the method of PEBS, which can dedicate a counter to count events that execute only in certain mode. The inaccuracy can affect events that occur very close to mode switches between user and kernel modes. For example, a tagged event that executes in user mode can be counted as an event from kernel space if it occurs just before the mode switches from user to kernel.

IV. Quantitative Comparison

This section quantifies the accuracy of PEBS and IBS under different scenarios: (i) accuracy in monitoring a single event, (ii) accuracy in monitoring multiple events, (iii) accuracy under different sampling intervals, and (iv) the stability of the accuracy across multiple runs. We also study the sampling bias and the functionality of both PEBS and IBS to attribute samples to their instructions and evaluate the time/memory overheads. Lastly, we study the capability of PEBS and IBS in detecting samples from kernel/user mode execution.

Table II shows the specifications of the machines used in the experiments. The default sampling interval in all experiments is set to 100 K. The results are averaged over 5 runs. We disabled Turbo Boost in the Intel machines, Turbo Core in the AMD machines, and hyperthreading in all of the four machines.

We developed the microbenchmarks with great care to eliminate any unintended noise and took a number of measures for a fair study. First, the microbenchmarks were written using `asm` statement [41] as assembly instructions in C in order to fully control the number of hardware events that occur during program execution. Second, unless otherwise stated, they were compiled using `gcc-10.3.0` with `-O0` flag in order to prevent compiler from modifying any part of the code. Third, to minimize overheads at user space, we configure the interrupt handler that handles sampling interrupts not to send any OS signal to user processes or threads by default.

We programmed PEBS using `perf_event_open` system call, and programmed IBS using the IBS driver available in [38]. We chose `perf_event_open` to interface with PEBS as it is the most widely used method that is utilized by the Linux perf tool [42], [43] and a number of other profiling tools [7], [8], [22], [44], [45]. We used the IBS driver in [38] to program IBS because it is the only possible way to interface with IBS in our AMD machine as `perf_event_open` requires certain versions of BIOS that are not commonly available as default in commodity AMD machines [46], [47].

A. Accuracy

This experiment aims to evaluate the accuracy of Intel PEBS and AMD IBS in capturing samples from a benchmark with known number of monitored events. We define accuracy as the closeness of the number of samples captured by PEBS or IBS to the number of expected samples given a sampling interval and a known number of events in the benchmark.

1) Hypothesis: Based on Observations 2, 3, and 4, we expect Intel PEBS to have better accuracy than both sampling flavors of AMD IBS in capturing samples from any hardware event.

2) Methodology: To evaluate the accuracy of both sampling facilities, firstly, we compared the number of retired instruction samples captured by PEBS, the number of executed micro-operations sampled by IBS op, and the number of instruction fetches sampled by IBS fetch against their ground truths. The ground truth for PEBS' retired instruction sampling and IBS op is the number of instructions in the microbenchmark divided by the sampling interval, while the ground truth for IBS fetch is the number of instruction fetches that hit in L1 ITLB as counted by `perf` divided by the sampling interval. The reason for choosing hit count in L1 ITLB is because there is no PMU event in AMD that specifically counts instruction fetches and the number of unique instructions in the microbenchmarks are small enough to fit in L1 ITLB. PEBS' retired instruction sampling and IBS op share the same ground truth because each instruction in the microbenchmark translates to exactly one micro-operation.

TABLE II
SPECIFICATIONS OF AMD AND INTEL MACHINES USED IN EVALUATION

Specification	AMD Zen 2	AMD Zen 3	Intel Skylake	Intel Cascade Lake
CPU Model	7352 CPU	7313 CPU	Core i7-6700 HQ	Xeon 6258R CPU
Microarch Family	Zen 2 (17h)	Zen 3 (19h)	Skylake	Cascade Lake
#Cores x Socket	24 x 2	16 x 2	4 x 1	28 x 2
Cache Sizes	L1i: 32 KB, L1d: 32 KB, L2: 512 KB, L3: 16 MB	L1i: 1MB, L1d: 1MB, L2: 16MB, L3: 256MB	L1i: 128KB, L1d: 128KB, L2: 1MB, l3: 6 MB	L1i: 32 KB, L1d: 32 KB, L2: 1MB, L3: 39MB
Linux Kernel Version	Linux 5.11.0	Linux 5.11.0	Linux 5.11.0	Linux 5.11.0

To evaluate the accuracy of PEBS and IBS, we ran both sampling facilities on a microbenchmark with known number of data cache load misses and known number of instructions, which will serve as the ground truths. We programmed PEBS to sample the precise event version of retired instruction, i.e. `INST_RETIRED:PREC_DIST`, and retired data cache load miss, i.e. `MEM_LOAD_RETIRED.L1_MISS`, in separate runs. We configured IBS to run micro-operation execution sampling, i.e. `IBS op`, and instruction fetch sampling, i.e. `IBS fetch`, also in separate runs.

After evaluating the accuracy in monitoring the most general events, i.e. instructions, instruction fetches, and micro-operations, we evaluated their accuracy in sampling a subset event, which is *data cache load misses* in our case. This is an event of interest for performance analysis tools that aim to detect data locality problems. For this evaluation, the numbers of load miss samples detected by PEBS' data cache load miss monitoring and IBS op are compared against the expected load miss count of the microbenchmark. IBS op is used for this comparison because it can capture data cache misses among its sampled micro-operations. In addition to evaluating the accuracy of PEBS and IBS under a single thread, we also evaluate the sensitivity of their accuracy under different thread counts.

For this experiment we devised a microbenchmark that can be configured to have different ratios of load instructions that miss in L1 data cache with respect to all instructions. The code for the *Accuracy-Bench* benchmark is shown in Listing 1. In each iteration of `loop0` there is exactly one load that misses in L1 data cache, i.e. `movl (%rbx), %ebx`, out of all instructions in the loop. Before `%loop0` is entered, the `%rbx` register is always assigned with the offset address of an array pointed by the `%rcx` register. To ensure that a miss in L1 data cache always occurs whenever the address in the `%rbx` register is accessed within `%loop0`, the array pointed by `%rcx` has the following description.

- Size of the array is 128K bytes, which is larger than the size of the L1 data cache (32K bytes).
- There is a 128 byte gap between any adjacent array elements to ensure that none of the array elements share a cache line.
- All array elements are initialized with their own indices.
- The array is shuffled randomly before `%loop1` is entered to generate a pointer-chasing memory access pattern when the array is accessed in `%loop0`.

By creating pointer-chasing effect whenever `(%rbx)` is accessed and by adding non-memory access instructions, i.e. `addq $1, %%r8`, inside `loop0` as needed, the ratio of L1 load miss to any instruction in the configured benchmark is controllable. By knowing the number of load misses and instructions in the benchmark, given the sampling interval we can easily calculate the number of expected load misses and instruction samples. For example, in the *1/20 Load Miss* case, there is one L1 load miss out of 20 instructions in each iteration, and the inner loop iterates 1000 times while the outer loop iterates 1M times. Thus, in total, there are 1 billion load misses and 20 billions and 4 millions instructions in a single run of the benchmark. As we set up PEBS to monitor load misses in data cache and retired instruction with sampling interval 100K, we can expect 200M and 400 instruction samples to be generated, and among these, the number of load miss samples should be 10K.

```
movq $10000000000, %edx
loop1:
movl $1000, %eax
movq %rcx, %rbx
loop0:
movl (%rbx), %ebx
shlq $7, %rbx
addq %rcx, %rbx\n\t"
addq $1, %%r8
```

```
//loop0 continues
............
addq $1, %%r8
decl %eax
jnz loop0
decl %edx
jnz loop1
```

Listing 1: Code for *Accuracy-Bench* benchmark.

3) Results: Fig. 3 compares the accuracy of Intel PEBS when monitoring retired instruction against both sampling flavors of AMD IBS. In the figure, PEBS, IBS op, and IBS fetch display high accuracy, though IBS op and IBS fetch still show slightly lower accuracy than PEBS. One major reason for the lower accuracy of IBS is that there is a time gap between when the interrupt occurs and when the interrupt handler disables the counter. On the other hand, the counters of PEBS freeze before interrupt delivery, and therefore, they do not suffer from this time gap problem. IBS op displays even lower accuracy than IBS fetch due to its randomization of counter after each sampling interrupt. We also observe an improvement in accuracy for both IBS op and IBS fetch from Zen 2 to Zen 3 microarchitectures. Fig. 4 shows the accuracy of PEBS and IBS when capturing load miss samples from the microbenchmark. The plotted results are produced by PEBS that monitors retired data cache load miss and IBS op. From the figure, it can be seen that PEBS achieves higher accuracy as its sample counts are very close to the expected counts, while IBS deviates more from the ground truth. Unlike the results in Fig. 3, IBS displays lower accuracy in Fig. 4. It shows that, though IBS can capture micro-operation and instruction fetch samples accurately, its detection of subset event, e.g. how many of the detected micro-operation samples are load miss samples, is less accurate.

Next, Fig. 5 presents the accuracy of PEBS and IBS in sampling load misses under different thread counts. The figure shows the results from the Skylake machine only up to 4 threads

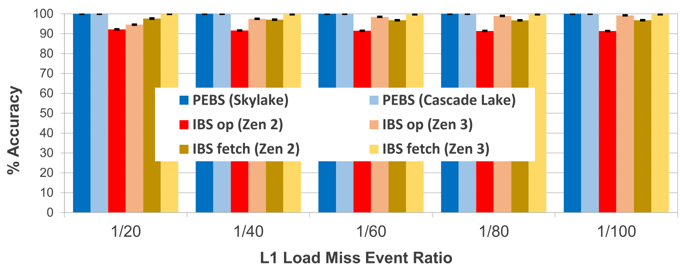

Fig. 3. Comparison of accuracy of PEBS monitoring retired instruction, IBS monitoring micro-operation execution (IBS op), and IBS monitoring instruction fetch (IBS fetch) on the *Accuracy-Bench* benchmark

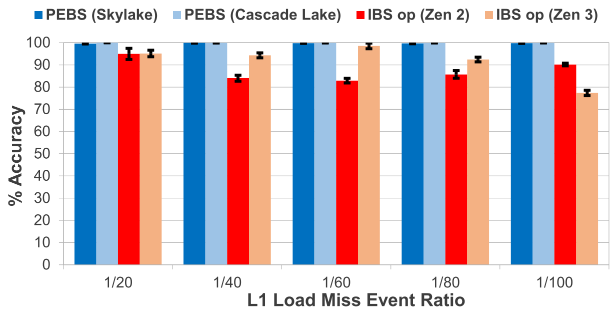

Fig. 4. Accuracy of Intel PEBS and AMD IBS in capturing retired L1 data cache load miss samples from the *Accuracy-Bench* benchmark

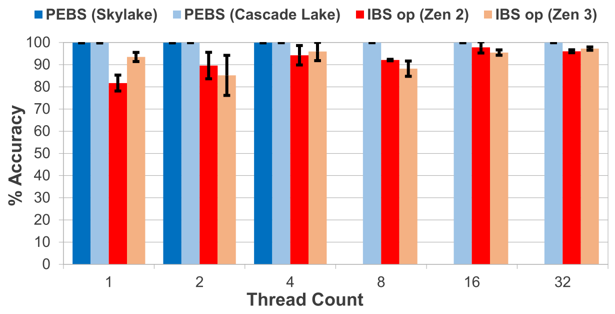

Fig. 5. Accuracy of Intel PEBS and AMD IBS in capturing retired L1 data cache load miss samples from the *Accuracy-Bench* benchmark under different thread counts

as it has only 4 physical cores. As displayed in Fig. 5, the accuracy of PEBS is consistently high across different thread counts, while the accuracy of IBS is lower at low thread counts and becomes better at high thread counts, i.e. 16 and 32 threads. This apparent increase in accuracy results from the summation of the sample counts captured by the multiple threads. The undercounted sample counts in some threads are compensated by the overcounted sample counts in other threads, which results in a total count that is close to the expected.

4) Findings: Intel PEBS always shows high accuracy in sampling any event, while AMD IBS displays lower accuracy. One possible solution to improve the accuracy of IBS is by freezing its counter before a sampling interrupt is triggered similar to how it is implemented in PEBS.

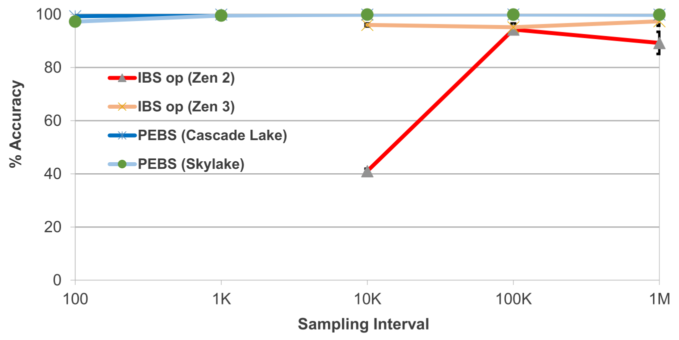

Fig. 6. Accuracy of PEBS and IBS in sampling L1 data cache load misses from the *Accuracy-Bench* benchmark under different sampling intervals.

B. Sensitivity to Sampling Rate and Stability

Next, we evaluate the accuracy under different sampling intervals and the stability of both schemes. Stability here refers to variation of accuracy across multiple runs.

1) Hypothesis: Due to Observations 2, 3, and 4, we expect PEBS to have higher accuracy than IBS under any sampling interval. Based on those observations, we also expect PEBS to be more stable, i.e. having more consistent accuracy, when profiling the same benchmark across different runs.

2) Methodology: To evaluate the accuracy in different sampling intervals, we again use the *Accuracy-Bench* benchmark. We also evaluate the stability of PEBS and IBS by measuring the standard deviation of sample counts detected across multiple runs in these benchmarks.

3) Results: Fig. 6 shows the accuracy of PEBS and IBS when sampling load misses from the *Accuracy-Bench* under different sampling intervals. While PEBS maintains high accuracy across intervals, the accuracy of IBS on Zen 3 is slightly lower than PEBS at high sampling intervals. On Zen 2, the accuracy degrades dramatically at 10K. As the IBS profiler stalls or does not properly work when the sampling interval is $<= 1K$ on the *Accuracy-Bench* , no results from these cases are shown for Zens.

Figs. 3, 4, 5, and 6 present the measured standard deviations as error bars. PEBS displays high stability, i.e. low variation of accuracy, regardless of the event and the number of events that it monitors. In contrast, Figs. 4, 5, and 6 show higher variation of accuracy for IBS in detecting sub-events of executed micro-operations, e.g. the number of load miss samples among detected samples. The maximum standard deviation of IBS op is 2.51% in Fig. 4, 5.96% in Fig. 5, and 4.15% in Fig. 6 at 1M sampling interval, while the maximum standard deviation of PEBS is nearly zero.

However, IBS is quite stable in capturing the cumulative counts of micro-operation when their sub-event types are not considered as shown in Fig. 3.

4) Findings: In capturing samples of a subset event, such as load miss, PEBS has high accuracy under different sampling intervals, while IBS exhibits high accuracy only at large sampling intervals, and it either suffers from lower accuracy or encounters errors when the sampling interval becomes short. PEBS displays high stability across multiple program executions, and IBS is relatively stable in capturing cumulative count of micro-operation samples. However, its stability is much lower in capturing counts of subset event samples.

C. Bias and Instruction Attribution

In this experiment, we evaluate the bias of PEBS and IBS in sampling the same event from multiple different locations in a code, and the accuracy of their ability in attributing samples to the instructions that trigger them.

1) Hypothesis: We expect PEBS and IBS to have no bias in sampling from multiple different instructions that perform the same monitored event. We also expect PEBS and IBS to accurately attribute the sampled events to the actual instructions that trigger those events.

2) Methodology: We evaluate the sampling bias and the instruction attribution of PEBS and IBS by programming them to sample retired loads from a synthetic benchmark, *bias-bench*, shown in Listing 2. If there is no sampling bias, the portion of samples attributed to each load instruction should be 25%. Furthermore, if all samples can be associated with their triggering instructions accurately, there should not be any sample associated with non-load instructions, i.e. *subq*, *cmpq*, and *jne*. We attribute the sampled instruction pointers to the source code lines by comparing the sampled instruction pointers with the instruction addresses of the labels in the benchmark code.

```
movq $10000000000, %rcx
movl $1, %ebx
loop0:
movl (%rax), %ebx // load 1
label1:
movl (%rax), %ebx // load 2
label2:
movl (%rax), %ebx // load 3
label3:
movl (%rax), %ebx // load 4
label4:
subq $1, %rcx // subq
cmpq $0, %rcx // cmpq
jne loop0 // jne
```

Listing 2: Code for the sampling bias and instruction attribution.

3) Results: Fig. 7 presents the percentage of samples attributed to each instruction in *bias-bench*. As PEBS and IBS do not associate any load samples with the non-load instructions, we can infer that both PEBS and IBS accurately attribute the load samples to the instructions that actually trigger them.

Concerning the sampling bias, PEBS detects load samples more unequally across the load instructions than IBS. Most samples detected by PEBS are associated with the *load 2* and *load 3* instructions. For IBS op, there is still inequality in sample distribution across the load instructions. However, the variation is not as large as in PEBS.

4) Findings: Both PEBS and IBS are equally accurate in attributing samples to their triggering instructions. However, PEBS is more biased than IBS in sampling an event from multiple different instructions. From the benchmark, PEBS captures at least 30% of the samples from only 1 out of 4 instructions that execute in a loop.

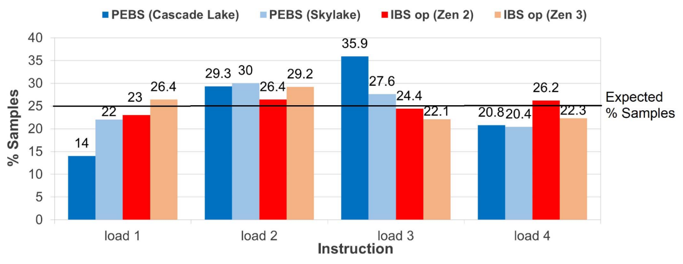

Fig. 7. Expected percentage (25%) vs percentage of samples attributed to each instruction in the *bias-bench* benchmark for PEBS and IBS. No load sample is attributed to non-load instructions.

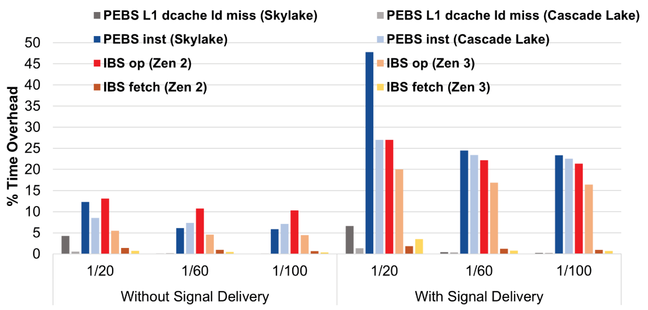

Fig. 8. Comparison of time overheads on *Accuracy-Bench* benchmark with and without signal delivery.

D. Time Overhead

We evaluate the time overheads of PEBS and IBS by running them on simple and more complex benchmarks.

1) Hypothesis: Based on Observation 4, we expect PEBS that monitors a specific hardware event, e.g., L1 data cache load miss, to incur lower overhead than IBS as it will most likely encounter fewer sampling interrupts than IBS. However, if PEBS monitors retired instructions, we expect PEBS to incur similar overhead to IBS because it will encounter approximately the same number of sampling interrupts as IBS.

2) Methodology: We programmed PEBS and IBS to sample from the *Accuracy-Bench* benchmark and the Rodinia benchmark suite [48] as representative real workloads. We configured PEBS to monitor retired L1 data cache load miss and retired instruction in separate runs. In the AMD machines, we programmed IBS to run IBS op and IBS fetch also in separate runs. We ran each Rodinia benchmark on a single thread to avoid microarchitectural factors such as cache line contention that might become a confounding variable that affects time overhead. To evaluate the sensitivity of time overhead to thread counts, we also ran the *Accuracy-Bench* benchmark under different thread counts while being monitored by PEBS and IBS. Furthermore, we ran each benchmark with OS signal delivery to user thread enabled and disabled in separate runs in order to evaluate the extra overhead introduced by the OS signal delivery mechanism in kernel space. Enabling the signal delivery gives us the actual overhead felt by an end-user.

3) Results: Fig. 8 presents the time overheads of PEBS and IBS for all chosen events and sampling flavors when monitoring the *Accuracy-Bench* benchmark, while Fig. 9 shows the time

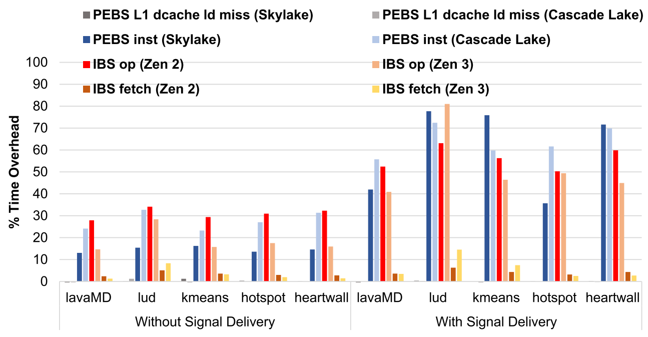

Fig. 9. Comparison of time overheads on Rodinia benchmarks with and without signal delivery.

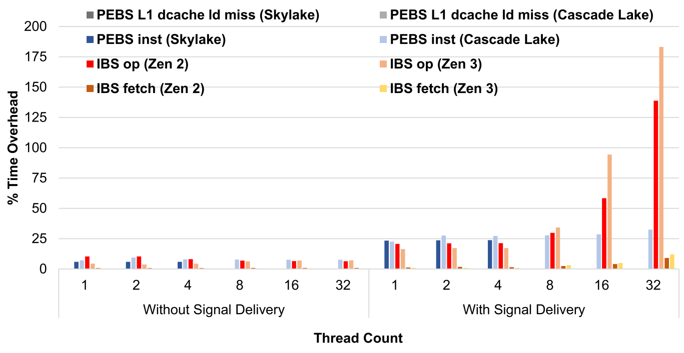

Fig. 10. Comparison of time overheads on *Accuracy-Bench* under different thread counts with and without signal delivery.

overheads on more complex workloads when monitoring selected Rodinia benchmarks. We present the time overheads with and without OS signal delivery enabled. One common pattern that we can observe in Figs. 8 and 9 is that the time overhead is more than doubled when OS signal delivery is enabled. Another pattern is that PEBS' monitoring of L1 data cache load miss and IBS fetch always incur the lowest overheads because they only sample load misses in L1 data cache or instruction fetches and the number of theses events are always lower than the numbers of retired instructions or micro-operations. IBS fetch has lower time overhead than both retired instruction monitoring by PEBS and IBS op. The reason for this is that an instruction fetch actually fetches multiple instructions that lie in the same fetch block from an L1 instruction cache. Therefore, the number of instruction fetches is always fewer than both the numbers of executed instructions and micro-operations. These observed patterns match our hypothesis.

Without signal delivery, the time overheads of PEBS retired instruction and IBS op in the Cascade Lake and Zen 2 machines are higher than the time overheads of PEBS retired instruction and IBS op in the Skylake and Zen 3 machines in nearly all cases except for the *Accuracy-Bench* with ratio 1/20. This observation implies that the hardware and microcode mechanisms of PEBS retired instruction and IBS op in the Cascade Lake and Zen 2 machines are less efficient than the mechanisms in the Skylake and Zen 3 machines. Furthermore, the difference in time overheads between PEBS retired instruction and IBS op in all cases with and without signal delivery contradict the expectation in our hypothesis, which we expected to be similar.

Fig. 10 displays the time overhead of PEBS and IBS on *Accuracy-Bench* with L1 load miss event ratio of 1/100 under different thread counts. Without signal delivery, the time overheads of all of the monitored PEBS events and IBS *flavors* are below 11% without any significant increase across different thread counts. However, when signal delivery is involved, the overheads of PEBS retired instruction, IBS op, and IBS fetch increase along with the increase in thread counts. The overhead of IBS op is more drastic and much higher than the rest when the thread count is 16 and 32.

4) Findings: Both schemes have a significantly higher time overhead when OS signal delivery is enabled, which in turn affects the observed time overhead of a profiling tool that leverages precise event sampling. PEBS and IBS in the Cascade Lake and Zen 2 machines incur higher time overheads than PEBS and IBS in the Skylake and Zen 3 machines for the microbenchmark and the complex benchmarks in nearly all cases when OS signal delivery is not enabled. The lower time overhead of PEBS and IBS in the Skyake and Zen 3 machines can be attributed to its hardware mechanism and the microcode that record sampled data that work more efficiently than the sampling mechanism of PEBS and IBS in the Cascade Lake and Zen 2 machines.

E. Memory Overhead

In this experiment, we evaluate the memory overheads of PEBS and IBS. Measured memory overhead is the maximum resident set size of a process in main memory during the process' lifetime while being monitored by PEBS or IBS.

1) Hypothesis: As we do not process sampled data in any signal handler, we expect low overheads in terms of maximum resident set size in memory from both PEBS and IBS thus their memory overheads should be approximately the same.

2) Methodology: We evaluate the memory overheads by having PEBS and IBS monitor the *Accuracy-Bench* and Rodinia benchmarks in all of the four machines. Using PEBS, we monitored retired L1 data cache load miss and retired instruction in separate runs, and we also ran IBS op and IBS fetch separately. We also evaluate the memory overheads of IBS and PEBS under different thread counts by having them monitor the *Accuracy-Bench* benchmark running on different numbers of threads.

3) Results: The measured memory overheads of PEBS and IBS for the single-threaded cases are less than 4% for the *Accuracy-Bench* benchmark and less than 0.9% for the Rodinia benchmarks (figure is omitted for brevity). There is less overhead in the Rodinia benchmarks as these benchmarks are larger and more complex than *Accuracy-Bench* , and therefore, the memory footprint of sample handling of PEBS and IBS is much smaller in ratio. There is also no huge gap in terms of memory overhead between PEBS and IBS on all of the single-threaded cases for *Accuracy-Bench* and Rodinia benchmarks. The results show that memory overhead of PEBS and IBS in terms of maximum resident set size in main memory is not a function of sample count or signal delivery.

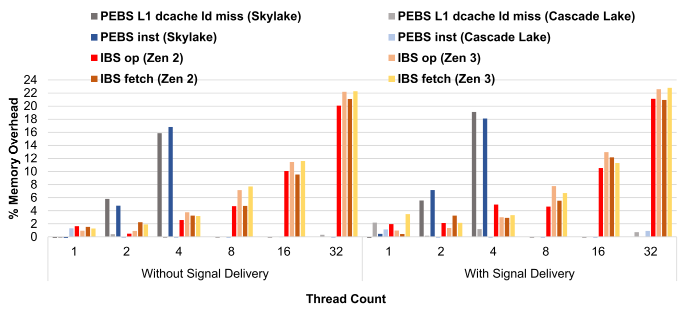

Fig. 11. Comparison of memory overheads on *Accuracy-Bench* under different thread counts with and without signal delivery.

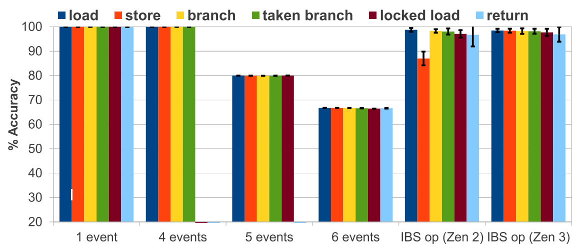

Fig. 12. Comparison of PEBS accuracy in monitoring different numbers of events against IBS op. PEBS monitors multiple events simultaneously except for 1 event case, where it monitors each event in a separate run. IBS can capture all events at once in its microoperation sampling.

Fig. 11 presents the memory overhead of PEBS and IBS on *Accuracy-Bench* with L1 load miss event ratio of 1/100 under different thread counts. The results show that, except for the Cascade Lake machine, memory overhead is a function of thread count – it becomes higher as thread count increases. At 2 and 4 threads, the memory overhead of PEBS in Skylake is much higher than the memory overhead of PEBS in the Cascade Lake. This pattern shows that there is significant improvement in terms of memory overhead from Skylake to Cascade Lake.

4) Findings: On single-threaded cases, PEBS and IBS incur nearly the same amount of memory overhead. However, when multiple threads are used, the memory overhead becomes higher as thread count increases for the Skylake, Zen 2, and Zen 3 machines. PEBS in the Skylake machine incurs much higher memory overhead than IBS in the Zen 2 and Zen 3 machines, while PEBS in the Cascade Lake machine incurs the lowest memory overhead. This result contradicts the expectation in our hypothesis.

F. Multiple Event Monitoring

Next, we compare the accuracy and overhead of PEBS that monitors multiple events against IBS. The accuracy here refers to how close the detected sample counts to the expected.

1) Hypothesis: Based on Observations 1, 2, 3, and 4, we expect PEBS to have better accuracy than IBS as long as the number of monitored events is less than or equal to the number of general-purpose counters. In case the number of monitored events is higher than the number of counters, PEBS will lose samples, thus its accuracy would drop. As a consequence of Observation 5, PEBS might have to monitor multiple events simultaneously to capture the same amount of information that can be captured by IBS in one run.

2) Methodology: To compare the accuracy of PEBS monitoring multiple events against IBS, we developed a microbenchmark, shown in Listing 3, that has known numbers of *load, store, branch, taken branch, return,* and *locked load instructions*. We chose these events as they can be sampled and identified by both PEBS and IBS. Furthermore, these events are also independent of machine configurations such as cache sizes and cache replacement policies, and therefore, their ground truths can be derived only from their instruction counts. We programmed PEBS to monitor one, four, five and six of these events in separate runs to observe the effect of monitoring more events than the number of available general-purpose counters on accuracy. In addition to evaluating accuracy, we also used PEBS to monitor different individual events and different numbers of events on a larger benchmark to see how multiple event monitoring affects the profiling time and memory overheads.

```
void foo1() { return; }
int main () {
        int val = 0;
        __asm__ __volatile__ (
              "movq $100000000, %%rcx\n\t"
              "loop0:\n\t"
              "call foo1\n\t"
              "lock\n\t"
              "addl $1, %0\n\t"
              "lock\n\t"
              "addl $1, %0\n\t"
              "movl %0, %%ebx\n\t"
              "subq $1, %%rcx\n\t"
              "cmpq $0, %%rcx\n\t"
              "je loop1\n\t"
              "cmpq $0, %%rcx\n\t"
              "jne loop0\n\t"
              "loop1:\n\t"
              : "=m" (val)
              :
              : "memory", "%eax", "%ebx", "%ecx"
           );
        return 0;
}
```

Listing 3: Code for multiple event monitoring benchmark.

3) Results: Fig. 12 shows the accuracy of PEBS in the Cascade Lake machine when monitoring multiple events simultaneously on our microbenchmark. We compare the accuracy of PEBS when monitoring 1 event (each event is monitored alone), 4 events (only load, store, branch, and taken branch are monitored together), 5 events (all of them except return), and 6 events (all of them) against the accuracy of the micro-operation sampling of IBS in the Zen 2 and Zen 3 machines. For brevity, we only display the results from these three machines as the results from the Skylake and Cascade Lake machines show similar patterns. Because there are only 4 general-purpose counters that can be used by PEBS in each logical core, it is shown that PEBS loses higher percentage of samples and undercounts when the number of events that it monitors is higher than the number of available counters, i.e. when there are 5 or 6 events monitored. These results confirm our hypothesis.

We also evaluate the effect of multiple event monitoring on time overheads. Table III compares overheads of PEBS across

TABLE III
TIME OVERHEADS OF PEBS MONITORING MULTIPLE EVENTS AND IBS OP ON THE *HEARTWALL* BENCHMARK FROM RODINIA SUITE. SKY: SKYLAKE, CAS: CASCADE LAKE

Monitored Event/ Event Set	Event Count	Without Signal Delivery	With Signal Delivery
IBS op	-	1.323x (Zen 2) 1.16x (Zen 3)	1.599x (Zen 2) 1.449x (Zen 3)
PEBS retired instruction	1	1.146x (Sky) 1.313x (Cas)	1.716x (Sky) 1.699x (Cas)
PEBS retired L1 load miss	1	1.0x (Sky) 1.0x (Cas)	1.002x (Sky) 0.997x (Cas)
PEBS retired load	1	1.124x (Sky) 1.147x (Cas)	1.336x (Sky) 1.327x (Cas)
PEBS set 1 (retired load + retired store)	2	1.142x (Sky) 1.13x (Cas)	1.622x (Sky) 1.574x (Cas)
PEBS set 2 (set 1 + retired L1 load hit + retired conditional branch)	4	1.221x (Sky) 1.145x (Cas)	2.514x (Sky) 2.379x (Cas)
PEBS set 3 (set 2 + retired taken near branch + retired mispredicted conditional branch + retired L1 load miss + retired L2 load hit)	8	1.221x (Sky) 1.142x (Cas)	2.528x (Sky) 2.387x (Cas)
PEBS set 4 (set 3 + retired mispredicted taken near branch + retired L2 load miss + retired L3 load hit + retired far branch + retired L3 load miss + 3 L3 load hit via cross-snooping (none, miss, hit)	16	1.223x (Sky) 1.142x (Cas)	2.558x (Sky) 2.404x (Cas)

different individual events and different event counts when monitoring the *heartwall* benchmark from Rodinia in the Skylake and Cascade Lake machines. To evaluate the time overheads when monitoring different event counts, we programmed PEBS to monitor 4 different sets of events in addition to monitoring individual events, i.e. retired instruction, retired L1 load miss, and retired load, across different runs. As a comparison, we also report the overhead of IBS op when profiling *heartwall* on the Zen 2 and Zen 3 machines.

When no OS signal delivery is involved, IBS op and PEBS retired instruction in the Zen 2 and Cascade Lake machines display the highest time overhead (first and second rows in the table), which is consistent with the result in Fig. 9. When signal delivery is involved, PEBS exhibits significant additional overhead, except when monitoring L1 load miss. The reason for the low overhead when sampling L1 load miss is that this event occurs very rarely in *heartwall* that the total latency of signal deliveries caused by the interrupts does not significantly impact the execution time. Beyond 4 events, the time overhead of PEBS stagnates as there is no increase in the number of sampling interrupts due to event multiplexing in using the limited number of counters.

4) Findings: If the number of events that are simultaneously monitored is higher than the general purpose counters in PEBS, accuracy of PEBS drops. However, its time overhead with OS signal delivery is affected more by the number of sampling interrupts triggered than the number of events monitored because of event multiplexing.

G. Kernel Mode Versus User Mode Identification

In this experiment, we evaluate the methods utilized by PEBS and IBS to identify the execution mode of the sampled events.

1) Hypothesis: Based on Observation 6, we expect the execution mode of the sample detected by Intel PEBS to be more accurate than by AMD IBS.

2) Methodology: To evaluate the accuracy of execution mode detection methods in PEBS and IBS, we developed a microbenchmark and a simple Linux kernel module that run together to cause repetitive execution mode switching during the execution of the microbenchmark. The code of the microbenchmark and the relevant piece of code in the kernel module are shown in Listings 4 and 5. To repeatedly switch from user mode to kernel mode, the microbenchmark calls the *ioctl(fd, TEST_CMD)* function call in every loop iteration. In the assembly code in Listing 4, the system call number of *ioctl*, which is 0x10, is passed as a parameter for the *syscall* instruction in the *%eax* register, the value of *fd* is already stored in the *%edi* register earlier before the shown code, and `TEST_CMD`, which is a macro for 0x67, is placed in the *%esi* register as the third parameter for *syscall*. Upon handling the *ioctl* function call, the piece of code in Listing 5 executes in the kernel mode. Using this pair of microbenchmark and the kernel module, we can expect that 1 billion locked load operations occur in kernel space and no such operation in user space.

```
movq $1000000000, %r8
loop0:
movl $0x10, %eax
movl $0x67, %esi
syscall
subq $1, %r8
cmpq $0, %r8
jne loop0
```

Listing 4: Code in user space.

```
case TEST_CMD:
        lock
        addl $1, (%rax)
break;
```

Listing 5: Code in kernel space.

We ran this experiment by installing the kernel module and running the microbenchmark while being monitored by PEBS and IBS. If the user mode detection method is accurate, we expect no locked load sample to be detected in user space.

3) Results and Findings: Based on the 5 runs of the benchmark, the output from PEBS always shows no detection of locked load sample in user mode, while the output from IBS shows 41 samples in Zen 2 and 84 samples in Zen 3 on average out of 10K expected locked load samples in the user space. These results show that PEBS can detect execution mode precisely, while misattribution of execution mode is possible to occur to IBS samples though with low probability.

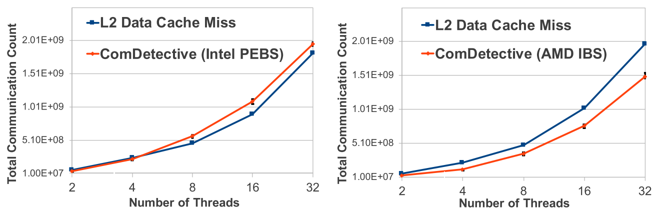

Fig. 13. Total communication counts with sharing faction of 70% for different number of threads in the Intel (Left) and AMD (Right) machines.

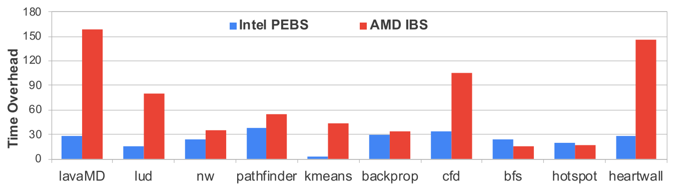

Fig. 14. Time overheads of *ComDetective* when running with PEBS and IBS on 10 Rodinia benchmarks.

V. Full-Fledged Profiling Tool

To compare the precise event sampling capabilities of Intel and AMD for a full-fledged profiling tool, we use an open-source tool, *ComDetective* [3], which captures the inter-thread communication within an application. The main idea of *ComDetective* is to use PMU samples and debug register traps to detect cache line transfers between threads. We performed experiments to compare the accuracy, overheads, and stability of *ComDetective* under PEBS and IBS running on the Intel Cascade Lake and the AMD Zen 2 machines. The sampling interval that we use in each experiment is 500K, which is the default sampling interval in the experiments reported in [3].

Accuracy. To compare the accuracy, we ran *ComDetective* on the *Write-Volume* benchmark developed by the original authors of the tool. In this benchmark, all threads perform an atomic write operation to either a shared or a private variable in a loop that iterates 100 M times. The number of accesses to the shared variable by each thread is controlled by *sharing fraction*. For example, if the sharing fraction is 0.7, each thread writes to the shared variable approximately 70M times. The ground truth for total communication count is the number of L2 data cache misses counted by *perf* since each thread is mapped to have its own L2 cache, and the number of cache line transfers because of atomic writes should be very close to the total number of L2 data cache misses. Fig. 13 shows the total communication counts detected by *ComDetective* and the ground truths under different thread counts. Consistent with our results in Section IV-A, *ComDetective* exhibits higher accuracy with PEBS than AMD as the gaps between the total communication counts and the ground truths are closer in Intel than in AMD.

Time and Memory Overheads. To evaluate the overheads of *ComDetective* when working with PEBS and IBS, we ran it on 10 Rodinia benchmarks. Each benchmark ran with 32 threads, and all threads were bound to cores with compact mapping. Fig. 14 compares the time overheads of *ComDetective* incurred in both machines. Consistent with the results of our experiment on the *Accuracy-Bench* benchmark presented in Table III and Fig. 5, *ComDetective* incurs lower time overheads on nearly all benchmarks when running with PEBS on 32 threads that monitors only retired load and retired store. The only exceptions are *bfs* and *hotspot*, which generate more memory access samples in Intel than AMD. Since both benchmarks produce more memory access samples in Intel, *ComDetective* performs more operations that are intended to detect inter-thread communications such as reading/updating a global data structure and arming debug registers, which incur extra overheads. While running on the Rodinia benchmarks, *ComDetective* incurs similar memory overheads in the Intel and AMD machines, which is 7.8% on average in both machines. The results confirm our results from the memory overhead study that report low memory overhead of PEBS and IBS when profiling large benchmarks.

Stability. Though not very visible in the figure, Fig. 13 also shows the standard deviation error bars indicating the stability when running with PEBS and IBS. Lower stability, i.e. high standard deviation, is observed when *ComDetective* runs on 2 threads. As thread count increases, the stability also gets better. While IBS's stability improves from 20% to 3%, Intel's changes from 13% to 1%. Aligning with our findings in the microbenchmarks, in general IBS exhibits lower stability than PEBS when running *ComDetective*.

VI. Related Work

Our work differs from the previous work first by benchmarking several behavioral aspects of PES through carefully designed microbenchmarks. Second, we evaluate the accuracy and overheads under more complex situations such as multiple event monitoring and monitoring in different modes. Third, we quantify the stability of sample counts generated by PES. Last but not least, to the best of our knowledge, there is no in-depth study on IBS and most studies focus solely on PEBS. Table IV summarizes the comparison between our work with related work.

Larysch [49] evaluated the accuracy and overhead of PEBS in measuring memory bandwidth and used low sampling intervals, which are between 10 and 1000. Through the experiments, the author discovered that PEBS suffers from higher sample losses as sampling interval is decreased.

Nonell et al. [50] evaluated PEBS' accuracy and overhead on applications running on a large numbers of CPUs, ranging from 2048 to 128K cores. By using the PEBS driver that they developed in a lightweight kernel, they could reach low overhead in capturing memory access patterns. Their driver could maintain high accuracy in capturing memory access patterns even under low sampling interval, which is up to 64.

Yi et al. [51] analyzed the accuracy of PEBS, and discovered that PEBS is prone to bias in event sampling due to shadowing. To eliminate the bias, they propose insertion of nop instructions after each monitored event. Gottschall et al. [52] proposed an *Oracle profiler* as a golden reference for time-proportional

TABLE IV
COMPARISON OF OUR WORK WITH RELATED WORK ON PRECISE-EVENT SAMPLING (PES)

Publication	Evaluated Aspect(s)	PES Facilities	Contributions
Larysch [49]	Accuracy, sensitivity to sampling interval	PEBS	They discovered that PEBS suffers from higher sample losses at low sampling intervals.
Nonell et al. [50]	Accuracy, time overhead	PEBS	Their lightweight kernel could maintain high accuracy and low overhead in capturing memory access pattern under low sampling interval.
Yi et al. [51]	Sampling bias	PEBS	They discovered that PEBS suffers from sampling bias due to shadowing and suggested insertion of nop instructions after monitored events.
Gottschall et al. [52]	Sampling bias	PEBS, IBS, ARM SPE	They proposed a golden reference for time-proportional attribution of event sampling as PEBS, IBS, and SPE are not time-proportional in sampling events/instructions.
Akiyama and Hirofuchi [53]	Time overhead, sensitivity to sampling interval	PEBS	They discovered a) CPU overhead of PEBS can be used to predict the profiling overhead of applications b) sampling rate and buffer size can affect cache pollution and the performance of profiled applications.
Xu et al. [54]	Accuracy	PEBS	They proposed a mathematical model to rectify inaccuracy of sampling in PEBS.
Our work	Accuracy, time/memory overheads single/multiple events, stability, sampling bias, instruction attribution, sensitivity to sampling interval and thread count, execution modes	PEBS and IBS	We propose several benchmarks that evaluate various aspects of event sampling using PES and analyze both PEBS and IBS qualitatively and quantitatively. See Section 1.1 for our findings.

attribution of event sampling. They found that existing PES facilities such as PEBS, IBS, and SPE are not time proportional in sampling instructions i.e. the number of samples taken from an instruction is not proportional to the number of CPU cycles incurred by that instruction.

Weaver and McKee [55] evaluated the variation (or stability) of event counting by PMUs in nine x86 architectures and discovered that inter-machine variations could happen because of the double counting of instructions on certain microarchitectures, virtual memory layout of profiled programs or OS activities such as page faults. Weaver et al. [56] also evaluated PMUs in 11 different implementations of x86_64 architecture and discovered sources of variation in their counted events. They explored possible ways to work around these limitations in the machines to produce more deterministic counts.

Akiyama and Hirofuchi [53] analyzed the overhead of PEBS, and demonstrated how the quantified overhead can be used to predict the actual overhead of applications. They also evaluated the effect of sampling rate and PEBS buffer size on cache pollution and the performance of profiled applications. Xu et al. [54] identified the inaccuracies of sampling in PEBS and developed a mathematical model to rectify inaccuracies.

VII. CONCLUSION

In this work, we extensively analyze two precise event sampling facilities from Intel and AMD architectures. In our qualitative analysis, we present their differences in terms of usable counters, types of events that can be sampled, types of data available in each sample, and their abilities to identify the execution mode of each sample. Then, we quantitatively analyze the accuracy, stability, sampling bias, overheads, and functionalities of each sampling facility. We also relate how the qualitative differences that we identified affect some of those aspects that we quantitatively study.

We envision our findings can greatly help tool developers and performance analysts understand the behaviour of their profiling tools and guide hardware designers to better design precise event sampling facility of future CPUs. Several specific usage scenarios are as follows: i) Tool developers could run our benchmarks to verify the accuracy, stability, bias, and overhead of the precise event sampling facility that they are going to use before developing their profiling code. ii) Hardware designers could use our benchmarks to validate the accuracy and to check if there is any sampling bias in the precise event sampling facility that they develop. iii) From the insights that we obtained in our research, hardware designers could also learn the strengths and weaknesses of the dispatch-tagging heuristic used in IBS [24] vs the Next-Committing Instruction (NCI) heuristic used in PEBS[21] and choose which sampling heuristic that they should adopt in their developed micro-architectures.

For future work, this research could be extended in two directions. One direction is to perform exhaustive accuracy comparison of all events supported by both PEBS and IBS such as L2 load miss, L3 load miss, and branch misprediction, and not only L1 load miss. Another direction is to carry out this qualitative and quantitative study on the precise event sampling facilities of other architectures, such as MRK of PowerPC [29] and SPE of ARM [30].

ACKNOWLEDGMENT

Authors would like to thank Barbara Pulmannova for the feedback on ARM SPE and Mohammad Kefah Issa for providing the Skylake machine.

REFERENCES

[1] Intel, *Intel Performance Tuning Utility 3.2 Update*. Santa Clara, CA, USA: Intel Corporation, 2008.

[2] Intel, *Avoiding and identifying false sharing among threads, Intel Corporation*, 2011. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html

[3] M. A. Sasongko, M. Chabbi, P. Akhtar, and D. Unat, “ComDetective: A lightweight communication detection tool for threads,” in *Proc. ACM Int. Conf. High Perform. Comput., Netw., Storage Anal.*, ser. SC ’19. New York, NY, USA, 2019, pp. 1–21. [Online]. Available: https://doi.org/10.1145/3295500.3356214

[4] J. Mario, “C2C - False sharing detection in Linux perf,” 2016. [Online]. Available: https://joemario.github.io/blog/2016/09/01/c2c-blog/

[5] L. Luo et al., “LASER: Light, accurate sharing detection and repair,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 261–273.

[6] X. Liu and B. Wu, "ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12.

[7] T. Liu and X. Liu, "Cheetah: Detecting false sharing efficiently and effectively," in *Proc. ACM Int. Symp. Code Gener. Optim.*, ser. CGO '16. New York, NY, USA, 2016, pp. 1–11. [Online]. Available: https://doi.org/10.1145/2854038.2854039

[8] M. Chabbi, S. Wen, and X. Liu, "Featherlight on-the-fly false-sharing detection," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, ser. PPoPP '18. New York, NY, USA, 2018, pp. 152–167. [Online]. Available: https://doi.org/10.1145/3178487.3178499

[9] A. Eizenberg, S. Hu, G. Pokam, and J. Devietti, "Remix: Online detection and repair of cache contention for the JVM," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, ser. PLDI '16. New York, NY, USA, 2016, pp. 251–265. [Online]. Available: https://doi.org/10.1145/2908080.2908090

[10] C. Helm and K. Taura, "PerfMemPlus: A tool for automatic discovery of memory performance problems," in *High Performance Computing*, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan Eds., Berlin, Germany: Springer, 2019, pp. 209–226.

[11] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on NUMA architectures," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, ser. PPoPP '14. New York, NY, USA, 2014, pp. 259–272. [Online]. Available: https://doi.org/10.1145/2555243.2555271

[12] C. McCurdy and J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 87–96.

[13] R. Lachaize, B. Lepers, and V. Quéma, "MemProf: A memory profiler for NUMA multicore systems," in *Proc. USENIX Conf. Annu. Tech. Conf.*, ser. USENIX ATC'12. USA, 2012, Art. no. 5.

[14] M. A. Sasongko, M. Chabbi, M. B. Marzijarani, and D. Unat, "ReuseTracker: Fast yet accurate multicore reuse distance analyzer," *ACM Trans. Archit. Code Optim.*, vol. 19, pp. 1–25, 2021. [Online]. Available: https://doi.org/10.1145/3484199

[15] P. Roy and X. Liu, "StructSlim: A lightweight profiler to guide structure splitting," in *Proc. ACM Int. Symp. Code Gener. Optim.*, ser. CGO '16. New York, NY, USA, 2016, pp. 36–46. [Online]. Available: https://doi.org/10.1145/2854038.2854053

[16] P. Roy, S. L. Song, S. Krishnamoorthy, and X. Liu, "Lightweight detection of cache conflicts," in *Proc. ACM Int. Symp. Code Gener. Optim.*, ser. CGO 2018. New York, NY, USA, 2018, pp. 200–213. [Online]. Available: https://doi.org/10.1145/3168819

[17] P. Magnusson et al., "Simics: A full system simulation platform," *Comput.*, vol. 35, no. 2, pp. 50–58, 2002.

[18] N. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[19] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proc. IEEE Int. Symp. Code Gener. Optim. Feedback-Directed Runtime Optim.*, ser. CGO '03. USA, 2003, pp. 265–275.

[20] C.-K. Luk et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, ser. PLDI '05. New York, NY, USA, 2005, pp. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[21] Intel, "Intel microarchitecture codename Nehalem performance monitoring unit programming guide," 2010. [Online]. Available: https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf

[22] X. Liu and B. Wu, "ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proc. ACM Int. Conf. High Perform. Comput. Netw. Storage Anal.*, ser. SC '15. New York, NY, USA, 2015, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2807591.2807648

[23] S. Wen, X. Liu, J. Byrne, and M. Chabbi, "Watching for software inefficiencies with witch," *SIGPLAN Not.*, vol. 53, no. 2, pp. 332–347, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296957.3177159

[24] P. J. Drongowski, "Instruction-based sampling: A new performance analysis technique for AMD family 10h processors," Nov. 2007. [Online]. Available: https://composter.com.ua/documents/AMD_IBS_paper_EN.pdf

[25] J. M. Anderson et al., "Continuous profiling: Where have all the cycles gone?," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 357–390, Nov. 1997. [Online]. Available: https://doi.org/10.1145/265924.265925

[26] P. J. Drongowski, "An introduction to analysis and optimization with AMD codeanalyst$^{\text{TM}}$performance analyzer," Advanced Micro Devices, Inc., Tech. Rep., 2008. Accessed: Jun. 1, 2022. [Online]. Available: https://www.iczhiku.com/resourceDetail/HlAI5pTAPEmD9ghdo3s9UA==

[27] AMD, "AMD uProf," Advanced micro devices, inc., Accessed: May 14, 2021. [Online]. Available: https://developer.amd.com/amd-uprof/

[28] X. Liu and J. Mellor-Crummey, "Pinpointing data locality problems using data-centric analysis," in *Proc. IEEE/ACM 9th Annu. Int. Symp. Code Gener. Optim.*, ser. CGO '11. USA, 2011, pp. 171–180.

[29] M. Srinivas et al., "IBM POWER7 performance modeling, verification, and evaluation," *IBM JRD*, vol. 55, no. 3, pp. 4:1–4:19, May/Jun. 2011.

[30] M. Williams, "Statistical profiling extension for ARMv8-A," Jan. 2017. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/statistical-profiling-extension-for-armv8-a

[31] ARM, *Arm Neoverse TM N1 Core. Version R3P1*, ARM, Feb. 2019. Accessed: Jan. 13, 2022. [Online]. Available: https://developer.arm.com/documentation/100616/0301

[32] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, "A RISC-V simulator and benchmark suite for designing and evaluating vector architectures," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, pp. 1–30, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3422667

[33] J. M. Domingos, P. Tomas, and L. Sousa, "Supporting RISC-V performance counters through performance analysis tools for Linux (perf)," in *Proc. 5th Workshop Comput. Archit. Res. RISC-V*, 2021. [Online]. Available: https://www.inesc-id.pt/publications/16626/pdf/

[34] ComDetective: A tool for inter-thread/inter-core communication analysis based on HPC Toolkits, 2019, [Online]. Available: https://github.com/comdetective-tools/hpctoolkit

[35] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B: System Programming Guide, Part 2. Order Number 253669*. Santa Clara, CA, USA: Intel Corporation, May 2020.

[36] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos, "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors," in *Proc. IEEE 30th Annu. Int. Symp. Microarchit.*, 1997, pp. 292–302.

[37] AMD, *AMD64 Technology. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Publication No. 24593 Revision 3.36*. Santa Clara, CA, USA: Advanced Micro Devices, Inc., Oct. 2020.

[38] J. L. Greathouse, "AMD research instruction based sampling toolkit," Jul. 2017. [online]. Available: https://github.com/jlgreathouse/AMD_IBS_Toolkit

[39] ARM, Arm forge user guide version 21.0, 2021. [online]. Available: https://developer.arm.com/documentation/101136/2100/MAP/Arm-Statistical-Profiling-Extension--SPE-/Known-issues

[40] M. J. Charney, "Intel X86 encoder decoder software library," Jul. 2015. [online]. Available: https://software.intel.com/content/www/us/en/develop/articles/xed-x86-encoder-decoder-software-library.html

[41] Extended ASM - Assembler instructions with C expression operands. Accessed: Jun. 5, 2022. [online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

[42] B. Gregg, "Perf examples," Jul. 2020. [online]. Available: http://www.brendangregg.com/perf.html

[43] Perf: Linux profiling with performance counters, Jun. 2020. [online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[44] L. Adhianto et al., "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," *Concurrency Comput. Pract. Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[45] Q. Wang, X. Liu, and M. Chabbi, "Featherlight reuse-distance measurement," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Los Alamitos, CA, USA, Feb. 2019, pp. 440–453. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA.2019.00056

[46] J. L. Greathouse, "Re: Error : IBS profiling is disabled in your BIOS," AMD Community, Accessed: Aug. 13, 2021. [Online]. Available: https://community.amd.com/t5/general-discussions/error-ibs-profiling-is-disabled-in-your-bios/td-p/55043

[47] J. L. Greathouse, "Re: IBS not available on EPYC 7451 ?" AMD Community, Accessed: Aug. 13, 2021. [online]. Available: https://community.amd.com/t5/server-gurus-discussions/ibs-not-available-on-epyc-7451/m-p/258228

[48] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, ser. IISWC '09. USA, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[49] F. Larysch, "Fine-grained estimation of memory bandwidth utilization," Master's thesis, Fac. Inform., Karlsruhe Inst. Technol. (KIT), Germany, 2016.

[50] A. R. Nonell, B. Gerofi, L. Bautista-Gomez, D. Martinet, V. B. Querol, and Y. Ishikawa, "On the applicability of PEBS based online memory access tracking for heterogeneous memory management at scale," in *Proc. ACM Workshop Memory Centric High Perform. Comput.*, ser. MCHPC'18. New York, NY, USA, 2018, pp. 50–57. [Online]. Available: https://doi.org/10.1145/3286475.3286477

[51] J. Yi, B. Dong, M. Dong, and H. Chen, "On the precision of precise event based sampling," in *Proc. 11th ACM SIGOPS Asia-Pacific Workshop Syst.*, ser. APSys '20. New York, NY, USA, 2020, pp. 98–105. [Online]. Available: https://doi.org/10.1145/3409963.3410490

[52] B. Gottschall, L. Eeckhout, and M. Jahre, "Tip: Time-proportional instruction profiling," in *Proc. IEEE/ACM 54th Annu. Int. Symp. Microarchit.*, ser. MICRO '21. New York, NY, USA, 2021, pp. 15–27. [Online]. Available: https://doi.org/10.1145/3466752.3480058

[53] S. Akiyama and T. Hirofuchi, "Quantitative evaluation of Intel PEBS overhead for online system-noise analysis," in *Proc. ACM 7th Int. Workshop Runtime Operating Syst. Supercomput.*, ser. ROSS'17. New York, NY, USA, 2017, pp. 1–8. [Online]. Available: https://doi.org/10.1145/3095770.3095773

[54] H. Xu, Q. Wang, S. Song, L. K. John, and X. Liu, "Can we trust profiling results? Understanding and fixing the inaccuracy in modern profilers," in *Proc. ACM Int. Conf. Supercomput.*, ser. ICS '19. New York, NY, USA, 2019, pp. 284–295. [Online]. Available: https://doi.org/10.1145/3330345.3330371

[55] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?," in *Proc. IEEE Int. Symp. Workload Characterization*, 2008, pp. 141–150.

[56] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2013, pp. 215–224.

Muhammad Aditya Sasongko received the PhD degree from the Department of Computer Engineering at Koç University in 2022 under the supervision of Dr. Didem Unat. He is a postdoctoral researcher with Koç University. His research focuses on performance analysis, PMUs, profiling tools for multicore architectures.

Milind Chabbi received the doctoral degree in computer science from Rice University. He conducts research in the areas of high-performance parallel computing, shared-memory synchronization algorithms, performance analysis tools, and compiler optimizations. He is currently employed as a senior researcher with Uber Technologies in Palo Alto, USA and is also the president of his independent research company Scalable Machine Research.

Paul H J Kelly received graduation with BSc degree in computer science from University College London in 1983, and the PhD degree from Westfield College, University of London, in 1987. He leads the Software Performance Optimisation research group in the Department of Computing with Imperial College London. He has worked in architecture, operating systems, static analysis and algorithms; his main research focus is compiler technology, specifically automated delivery of domain-specific performance optimisations, in particular in computational science and robot vision. He has been on the faculty with Imperial since 1989.

Didem Unat (Member, IEEE) is a faculty member with Koç University and director of Parallel and Multicore Computing Laboratory. She is known for her work on programming models, performance tools, and system software for emerging parallel architectures. She received the Marie Sklodowska-Curie Individual Fellowship from the European Commission in 2015, the BAGEP Award from the Turkish Academy of Sciences in 2019, and the British Royal-Newton Advanced Fellowship in 2020. She is named the Emerging Woman Leader in Technical Computing by ACM SigHPC in 2021. She is also ERC Starting Grant holder.