Diagnosing Performance Fluctuations of High-throughput Software for Multi-core CPUs

Soramichi Akiyama, Takahiro Hirofuchi, Ryousei Takano
National Institute of Advanced Industrial Science and Technology (AIST), Japan
{s.akiyama, t.hirofuchi, takano-ryousei}@aist.go.jp

***Abstract*—Performance fluctuations are common in various software such as databases and software networking stacks. A fluctuation refers different performance (latency, throughput) for similar or identical data-items (e.g. requests, queries, packets) due to non-functional states such as cache warmth. While tail latency caused by fluctuations badly affect user experiences, diagnosing them is difficult as reproducing non-functional states in a controlled environment is not feasible. To this end, we estimate elapsed time of each function for each data-item individually to observe a single fluctuation occurrence online so that reproducing non-functional states is no longer needed. The issue is that instrumentation-based tracing methods are too heavy because a function takes a few micro seconds in high-throughput software systems for the multi-core age. We propose a hybrid approach of instrumentation and hardware-based sampling. It enables to diagnose performance fluctuations of high-throughput software systems with acceptable and adjustable overhead. Our evaluations show that it can clearly show a performance fluctuation that occurs by different cache-warmth in a sample application, and that it can be also applied to realistic software.**

I. Introduction

Performance fluctuations are common across a variety of software such as databases and software networking stacks. Performance (e.g. latency, throughput) of such software often fluctuates for even two identical *data-items* (e.g. queries, packets, requests) depending on non-functional states of the software such as cache warmth, resource contention, and other factors. Huang *et al.* ran the TPC-C workload on popular database engines and confirmed that *"the standard deviation was twice the mean"* [1]. Dobrescu *et al.* reported that the performance of a software packet-processing platform drops by 27% in the worst case due to shared resource contentions [2].

Diagnosing performance fluctuations is challenging. Fluctuations often occur only with a complex set of non-functional states that appear during a compound test or in a production environment. Analyzing the phenomena later offline is very difficult because reproducing the non-functional states of the time when a fluctuation has occurred is not feasible due to three reasons: (1) pinpointing a specific part of non-functional states as the root cause before an analysis is impossible thus as many non-functional states as possible should be reproduced, (2) non-functional states cannot be easily quantified, and (3) non-functional states change every time a new data-item is processed. For example, if performance of a database engine fluctuates only when its on-memory cache is fragmented and the fragmentation is fixed after processing few queries, then reproducing the phenomenon is a hard task.

This work is supported in part by the Project for Developing Innovation Systems of the MEXT, Japan.

To diagnose performance fluctuations without reproducing them in a controlled environment, *we estimate the period of time each function takes for each data-item individually.* This enables to observe a single occurrence of a performance fluctuation online and reproducing the fluctuation in a controlled environment is no longer required. In the aforementioned example of a database engine, the function that fetches tables from the cache may take much longer time for a specific data-item, implying that the cache performance is degraded when the data-item has been processed.

High-throughput software systems in the multi-core age process many data-items per time unit, thus a function takes a very short period of time such as several micro seconds or even shorter. This arises a challenge to obtain elapsed time of each function per data-item. A common approach that inserts instrumentation code into the beginning and the end of each function (e.g. `gprof` [3], `Vampir` [4], `cProfile` [5]) cannot be applied as-is. First, instrumenting every function is too heavy for our purpose because a function may take only several micro seconds. Second, selecting which function to instrument to reduce the overhead is not doable because a function that takes only a short period of time in a test run may become the root cause of a fluctuation in a production run, which is the nature of performance fluctuations.

We propose a hybrid approach of coarse-grained instrumentation and hardware-based low-overhead sampling to tackle the challenge. The key ideas are as follows:

1) Instrumentation is applied only to code points where the target program starts and finishes processing a data-item. This reduces the overhead by reducing the number of times the instrumentation code is invoked to 'twice per data-item' from 'twice per function'.
2) Elapsed time of each function per data-item is estimated by a low-overhead hardware-based sampling mechanism. The overhead is adjustable via its sampling rate without specifying which function to estimate.

As the sampling mechanism, we utilize *Precise Event Base Sampling (PEBS)*, a hardware functionality of Intel CPUs that samples the timestamp and the value of the instruction pointer (and other data that are irrelevant to this paper). The overhead of PEBS is approximately 250 *nano* seconds per sample [6], and the overall overhead is adjustable by controlling the sampling rate without specifying which function to sample.

The technical issue is that mapping each PEBS sample to a particular data-item is not straight-forward. A PEBS sample does not include any information about the data-item being processed at the time the sample has been taken. Furthermore, because PEBS is hardware-based, extending it to sample other information than pre-defined is not doable. We solve this issue by leveraging characteristics of a modern software architecture that aims to be scalable on multi-core CPUs. To reduce context switching overhead, modern high-throughput software systems divide the work to process a data-item to smaller tasks, each of which is pinned to designated a core. By recording timestamps when a data-item enters and leaves a CPU core, we can map PEBS samples taken between the two timestamps on the core to that data-item. Our evaluations show that our method can diagnose a performance fluctuation caused by different cache warmth for each data-item in our sample application, and it can also be applied to a real application using DPDK [7].

This paper is structured as follows. Section II explains the context and challenges of this work. Section III describes our hybrid approach and its prototype implementation. Section IV shows the results of our feasibility study. Section V discusses possible further improvement on our method. Section VI reviews the related work and Section VII concludes the paper.

II. Background

A. Performance Fluctuations

A performance fluctuation is a phenomenon where performance (e.g. throughput, latency) of a software system fluctuates even for similar or identical data-items (e.g. queries, packets, requests) due to non-functional states of the software such as cache-warmth and shared resource contentions. For example, when a database engine processes two identical queries within a short interval, the first one can take significantly longer time than the second one because the target table may not be cached on memory when the first one is processed.

Performance fluctuations are of great concern because tail latencies can greatly affect the user experience and approached by researchers [8], [9]. Huang *et al.* [1] measured the latencies of queries of the TPC-C workload on widely used database engines (MySQL, Postgres and VoltDB). They reported that *"the standard deviation was twice the mean"* and *"the 99th percentile was an order of magnitude greater than the mean"*. Dobrescu *et al.* [2] reported that the performance of a software packet-processing platform drops by 27% in the worst case due to shared resource contentions. Another example that we found is the access control list (ACL) functionality of DPDK (Data Place Development Kit) [7]. The latencies of two very similar packets differ by several micro seconds depending on the number of ACL rules applied to each packet.

B. Leveraging Traces to Diagnose Fluctuations

Performance analysis tools provide either (or both) *profiles* or/and *traces* of target programs. A trace includes events that happen during a program run (e.g. function calls, cache misses) with timestamps. A profile summarizes performance metrics averaged over a whole run or for some time period (e.g. 10 sec, 5 min). A profile is useful for understanding overall performance characteristics, while a trace is useful to analyze each event with a deeper focus. Figure 1 illustrates a trace and a profile of a web server that invokes three functions to process a request. Note that it is an imaginary example to illustrate the concept. The profile shows elapsed time of each function (A, B, C) that are accumulated over the whole run. The trace, on the other hand, shows events (entering / leaving functions in this example) with a timestamp for each request.

Trace

Request	Function	Event	Timestamp (us)
#1	A	Enter	00010
#1	A	Leave	00100
#2	A	Enter	00145
#2	A	Leave	00155
...	...	...	...
#50	C	Enter	04918
#50	C	Leave	04923

Profile

Function	Total Time
A	250 us
B	100 us
C	50 us

Fig. 1. A trace (left) and a profile (right). A profile shows "averaged" results only and cannot find performance fluctuations. A trace shows each event with timestamps that can help finding performance fluctuations.

A performance fluctuation occurs in a complex set of non-functional state that appears during a compound test workload or a production run. Reproducing the non-functional states of the time when a fluctuation has occurred is not feasible due to three reasons. (1) Pinpointing a specific part of non-functional states as the root cause before an analysis is impossible thus as many non-functional states as possible should be reproduced. (2) Non-functional states cannot be easily quantified. (3) Non-functional states can change every time a new data-item is processed. For example, reproducing the cache warmth of a machine into another machine is not easily conducted.

We diagnose performance fluctuations of a software system with a trace that records elapsed time of each function for each data-item. A trace can catch a single occurrence of a fluctuation thus reproducing non-functional states into a controlled environment is no longer required. For example in Figure 1, the trace clearly shows a performance fluctuation where function A takes $100 - 10 = 90$ us for request #1 while it takes only $155 - 145 = 10$ us for request #2. Achieving our goal requires a trace to be per-data-item and per-function, which arises challenges that we explain in the next section.

C. Challenges on Obtaining Traces

Existing tracing tools such as `gprof` [3], `Vampir` [4] and `cProfile` [5] rely on instrumentation either into the source code or the binary of a target program. They taint code points under interest by means of marking function calls. For example, gprof inserts calls to a marking function named `mcount` to the beginning of every function. It is called every time the target program calls a function, so that the timestamp (and other program states) is recorded every time a function is invoked. cProfile, the built-in profiler of Python, also inserts marking function calls to trace program states.

Challenges appear when existing tracing tools based on instrumentation are used to diagnose performance fluctuations.

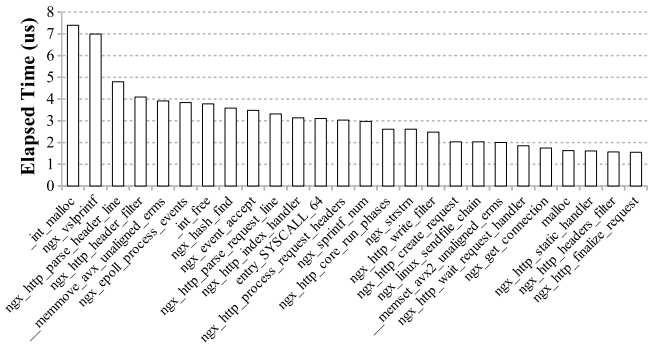

Fig. 2. Per-request elapsed time of each function of NGINX.

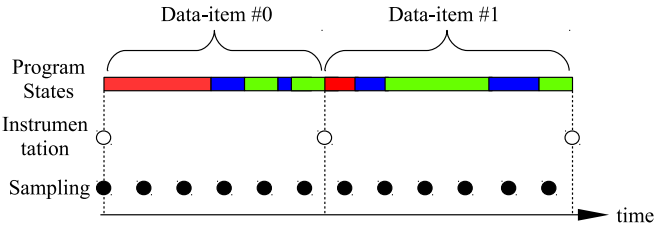

Fig. 3. Overview of the proposed hybrid approach.

First, instrumenting every function is too heavy a function may take very short period of time in high-throughput software systems for the multi-core age. Figure 2 shows per-request elapsed time of functions of a web server. We set up NGINX and executed the Apache benchmark from a different machine connected by a 1 Gbps link. The default index page (612 bytes) is loaded with 1 K simultaneous connections up to 300 K requests in total. For simplicity, we used one worker thread and all requests are processed in one CPU core. The workload took 44.8 seconds, which means the average elapsed time per request was 149 us (the network latency affects only once and can be ignored). We used perf [10] to measure cycles spent on each function and the per-request elapsed time of function f is estimated by $149 \times \frac{c_f}{c_a}$, where c_f is the number of cycles f takes and c_a is the all cycles spent by the server. The results show that many functions take less than 4 us and instrumenting every function to obtain per-request traces is too heavy. Second, selecting which function to instrument to reduce the overhead cannot be done a-priori to solving performance fluctuations. Due to the nature of performance fluctuations, a function that takes only a small period of time in a test run may take much longer time for a specific data-item. We need a novel way to obtain traces with low and adjustable overhead without selecting which function to trace.

III. Proposal: Hybrid Approach

A. Overview

In order to diagnose performance fluctuations of high-throughput software systems, we propose a hybrid approach that combines instrumentation and sampling working complementary to each other. The main idea is to use instrumentation only when it is necessary and to use hardware-based light-weight sampling mechanism in other places. Data retrieved by the two means are integrated together to form a single trace.

Figure 3 shows an overview of our approach. The colored bars on the top show an execution path of a target program, which our method traces. Each bar shows that the program executes a specific function during the length of the bar and different colors indicate different functions. The white circles in the middle show data retrieved by instrumentation and the black circles at the bottom show ones retrieved by a hardware-based sampling mechanism. A marking function is instrumented and invoked every time the target program starts processing a new data-item and finishes processing it. The marking function records the timestamp and the data-item ID so that mappings between a time period and the data-item being processed during the period is obtained. Because the marking function is called only twice per data-item, the overhead by instrumentation is greatly reduced compared to existing tracing mechanisms that invoke a marking function every time a function is called. Sampling is applied to the target program simultaneously with instrumentation. Instruction pointers with timestamps are periodically recorded (or "sampled") so that mappings between timestamps and functions being executed at the time points are obtained. The two mappings are later integrated to estimate elapsed time of each function per data-item to diagnose performance fluctuations.

TABLE I
Characteristics by Each Tracing Mechanism

	Sampling	Instrumentation
Implemented by	hardware	software
Overhead	low	high
Timing	periodic	per each data-item
Adjustable	yes	no
What to trace	pre-defined	software-controlled
Traced data includes	timestamp, instruction pointer	timestamp, data-item ID

Table I shows characteristics of the two mechanisms. Instrumented code has to be invoked every time a data-item is started / finished processing and the timing is not adjustable, while sampling is periodic and the sampling rate is adjustable without specifying which function to sample. On the other hand, instrumentation can trace everything that is visible from software including the data-item ID being processed, while the sampling can trace a pre-defined set of information including a timestamp and a value of the instruction pointer. The overhead of instrumentation is relatively high because it is software-based, while the sampling mechanism utilized in this paper is all hardware-based and light-weight. The details of the mechanisms and how to integration the two types of data are described in the following sections.

B. PEBS: Details, Importance, and Issue

For the sampling part, we utilize *Precise Event Based Sampling (PEBS)*, a hardware-based sampling mechanism of Intel

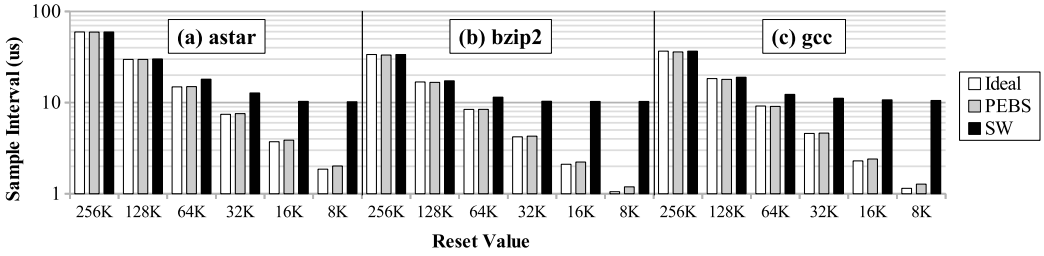

Fig. 4. Sample intervals of PEBS and a software-based sampling mechanism. perf with the traditional performance counters is used as a representative of software-based mechanisms. It asks the OS to sample program states. The throttling mechanism of perf is disabled. The sample interval of perf is as long as 10 us no matter how high the sampling rate is. The sample interval of PEBS can be almost 1 us and close to the ideal case.

CPUs. It is promising because our previous work [6] showed that the overhead is approximately 250 ns per sample (note that the unit is *nano* second). On the other hand, software-based sampling mechanisms incur much overhead due to execution switches between a target program and the sampling software. Since PEBS is a part of the performance counter functionality, we hereafter refer non-PEBS performance counters to as the *traditional performance counters* to avoid confusions.

To configure PEBS, pairs of *hardware events* and their *reset values* (referred to as R) are specified. The number of pairs that can be specified simultaneously depends on the CPU model, but we use only one pair in our approach. A hardware event, such as "retired micro operations" and "cache misses", is chosen from the pre-defined set of events that are listed in the CPU specification [11]. After PEBS is enabled, the CPU counts the number of occurrences of the specified event for each CPU core using a designated counter register for the core. The counter registers are set to $-R$ at first. When a counter register overflows, the CPU samples the values of general-purpose registers (e.g. `eax`), the instruction pointer, and the hardware timestamp of the core and reset the counter register to $-R$. Therefore, a sample is taken every R occurrences of the specified event. The sampled values are stored into a memory region referred to as the *PEBS buffer*. An interruption is raised from the CPU when (and only when) the PEBS buffer has become full. The operating system receives the interruption so that it can process the samples in the PEBS buffer.

Our method utilizes PEBS because software-based sampling mechanisms cannot sample at a short enough sample interval. A *sample interval* is the time difference between two consecutive samples. Sample intervals of software-based mechanisms include overhead to switch execution from target programs to sampling software that save program states. Our approach requires the sample interval to be in the level of one micro second. A function may take only several micro seconds in high-throughput software systems (as observed in Figure 2), and several (at least two) samples must be taken for a function to estimate elapsed time of it using the samples.

Figure 4 shows the relationship between achieved sample intervals and the configured sampling rates for a software-based sampling mechanism and PEBS. The evaluation environment is in Table II. The linux perf tool is used as a representative of software-based sampling mechanisms. It supports both PEBS and the traditional performance counters, but we use the latter in this experiment. Note that even though the traditional performance counters themselves are hardware, they rely on software to sample program states every time a counter register overflows. We disable the throttling mechanism of perf that automatically limits the sampling rate. For PEBS, we used a simple kernel module described in Section III-E. The samples were stored in memory to avoid IO overhead. The x axis shows the reset value (which controls the sampling rate) and the y axis shows the achieved sample interval. The hardware event specified was `UOPS_RETIRED.ALL`, which *"counts the number of micro-ops retired"* [11]. Three workloads (astar, bzip2, gcc) from SPEC CPU 2006 are used. "Ideal" indicates the ideal case where doubling the sampling rate (halving the reset value) makes the sample interval halved. Note that the sample intervals for the same reset value are different across benchmarks because the average "instructions per cycle" are different for each benchmark. The results show that the sample interval of PEBS (hardware-based) can be almost 1 us while the one of perf (software-based) cannot be shorter than 10 us no matter how high the configured sampling rate is, which means that PEBS is promising for our purpose.

The technical issue in using PEBS for our purpose is that PEBS only samples pre-defined set of information and it is not extensible because PEBS is all hardware-based. PEBS samples the values of the general purpose registers such as `eax` and `ebp`, the hardware timestamp, the instruction pointer, and some additional information that are not useful for our purpose such as the abort reason of hardware transactions. Therefore, a PEBS sample cannot be directly mapped with a data-item by itself. This contrasts with software-based sampling mechanisms that can sample any information but are much slower. We solve this challenge by integrating data retrieved by PEBS with ones retrieved by coarse-grained instrumentation.

C. Instrumentation

We instrument a marking function that records timestamps and data-item IDs into code points where the target program starts and finishes processing a data-item. We refer these points to as *data-item switches* hereafter. Finding data-item switches to instrument is not always straight-forward if the target program is built with an arbitrary software architecture.

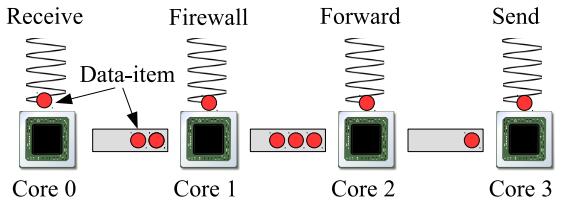

Fig. 5. Modern software architecture for high scalability. Each core has only one thread and each thread executes a part of the task required to process a data-item. The number of data-item processed on a core at a time is one.

We leverage the characteristics of a modern software architecture that aims to be highly scalable on multi-core CPUs to find data-item switches to instrument. Figure 5 illustrates the software architecture. Each core invokes at most one thread and there are the same (or fewer) number of threads as the number of cores in one CPU. A thread executes a part of the whole task that is required to process one data-item (e.g. packet receiving, packet forwarding) and is connected with other threads by software queues. This architecture is better in scalability than a traditional architecture where the same number of threads as the number of data-items being processed are invoked [12]. It is used by many major software systems that aim very high throughput such as DPDK, MariaDB and NGINX. In the development model of DPDK, a thread is pinned to each core and packets are passed through the threads using software queues. MariaDB claims in its manual that *"there should be a single active thread for each CPU on the machine"* [13]. NGINX also uses the same architecture [14].

To concretely discuss how to detect data-item switches, we further categorize the software architecture into two types:

1) *Self-switching*: in this architecture, data-item switches are explicitly written in the source code in small number of places such as the beginning of a worker thread and an event handler.
2) *Timer-switching*: in this architecture, data-item switches are forcefully incurred by timers, in addition to explicit places. The idea is to guarantee a latency threshold when a data-item is taking too much time to be processed.

The difference between the two is explained by a case where consecutive heavy and light data-items are processed. In the self-switching architecture, the light data-item cannot be processed until the system finishes processing the heavy one. It is good for throughput because there is no overhead of extra data-item switching that typically causes context switches. On the other hand, in the timer-switching architecture, the system can finish processing the light data-item before it finishes the heavy one. The self-switching one is adopted by DPDK and MariaDB, and the timer-switching one is used in NGINX.

Self-switching architecture switches data-items only when it explicitly wants to. This corresponds a code point where a thread starts processing a new data-item or an event handler starts processing a new event, typically at the top of a busy loop that includes the actual work of the thread or the event handler. We detect this type of data-item switches by recording timestamps in these code points. We discuss the whole procedure our method in Section III-D.

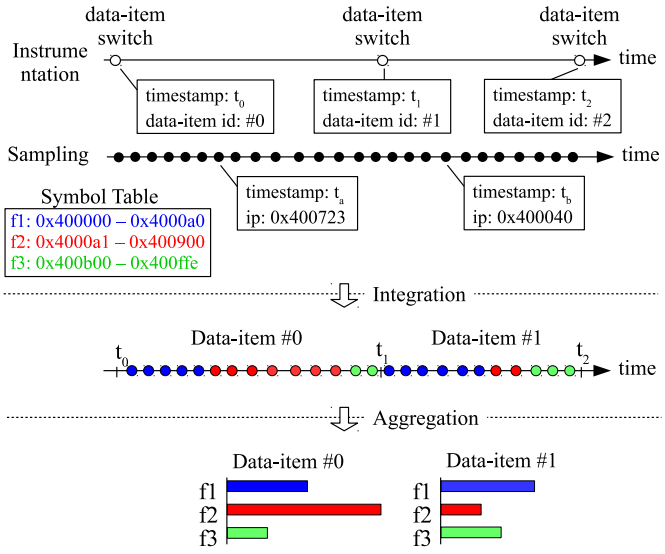

Fig. 6. Proposed Hybrid Approach

Timer-switching architecture switches data-items when the amount of time spent for a particular data-item has reached the threshold, in addition to data-item switches explicitly incurred. These switches are typically implemented by a scheduling mechanism such as a combination of a timer and user-level threading. Therefore, detecting data-item switches requires to record the activities of the scheduler of a target program. We discuss how we can potentially extend our method to this type of data-switches in Section V.

D. The Procedure of Our Proposal

This section describes our hybrid approach to diagnose performance fluctuations for high-throughput software systems built with the self-switching architecture. Figure 6 illustrates the overall procedure. It shows the case where there is only one CPU core for simplicity, but the same procedure is executed on every core of a multi-core CPU. Note that PEBS supports sampling core-related events for every core simultaneously.

1) The target program is executed with instrumentation and sampling enabled. Every time a data-item switch occurs, the timestamp and the data-item id are recorded by the instrumented code. For example, the timestamp t_0 and the data-item id #0 are recorded on the first data-item switch in the figure. At the same time, timestamps and the value of the instruction pointer are sampled and recorded periodically by PEBS.
2) The data obtained in the above step are integrated. First, the timestamp included in each PEBS sample (such as t_a and t_b in the figure) are compared with the ones recorded at data-item switches (t_0, t_1, t_2). The PEBS sample with the timestamp t_a belongs to data-item #1 because $t_0 < t_a < t_1$. Second, the values of the instruction pointer included in each PEBS sample are compared with the

symbol table of the target program. Symbols are the names of functions and the addresses of their beginning and ending points that are obtained from the binary of the target program.

3) The elapsed time of function f_n for data-item #M is calculated by the difference between the timestamps of the first and the last PEBS sample that belong to $\{f_n$, data-item #M$\}$.

E. Implementation

PEBS is configured by a small kernel module and helper programs called simple-pebs [15]. We modified it to support Skylake micro-architecture. The reason to use a module with minimum set of functionalities is that linux perf APIs have non-negligible overhead due to the too rich features [16].

The kernel module allocates PEBS buffer and enables PEBS functionality with the given reset value (R) and hardware event. PEBS samples program states every time the specified event has occurred R times, and the CPU invokes an interruption when the PEBS buffer has become full. The kernel module receives the interruption and asks a helper program to copy the data from PEBS buffer to userspace memory. In our prototype, we output the copied data into an SSD and then notifies the kernel module that the data is safely saved so that PEBS can be re-enabled. There are several possible ways to reduce the storage-related overhead including double buffering (so that the helper program can re-enable PEBS immediately), but the focus of this paper is to prove that our approach is promising and we leave these optimization for future work.

To trace data-item switches, we inserted log-printing statements that output the timestamp and the data-item ID (specific content differs depending on each workload). The prototype outputs them directly to SSD, but again it is possible to temporarily store them to the main memory and periodically dump them to minimize the overhead.

IV. Feasibility Studies

A. Methodology

This section shows that our approach can obtain elapsed time of each function in per-data-item basis to help diagnose performance fluctuations. Our feasibility studies are twofold:

1) First, we use a sample application that we build as a proof-of-concept of the approach. It mimics query processing applications with an in-memory cache mechanism. The performance fluctuates depending on the cache warmth even for queries with the same content. Our method can clearly observe the fluctuation.
2) Second, we target an access control list (ACL) application of DPDK. The packets going through it experience different latencies depending on their TPC headers. This fluctuation occurs in a specific non-functional state that is hard to reproduce in an offline environment. Our method accurately shows elapsed time of the bottleneck function for each packet individually and helps understanding the root cause.

TABLE II
Evaluation Environment

CPU	Core i7 6700K (Skylake Micro arch.)
Motherboard	Supermicro X11SAE-F
OS	Debian GNU/Linux 8.9 (Linux kernel 4.9)
NIC	10 Gbps Intel X520-DA2 × 2
Memory	64 GB (16 GB DDR4 × 4)
SSD	512 GB (Crucial M4 CT512M4SSD2)

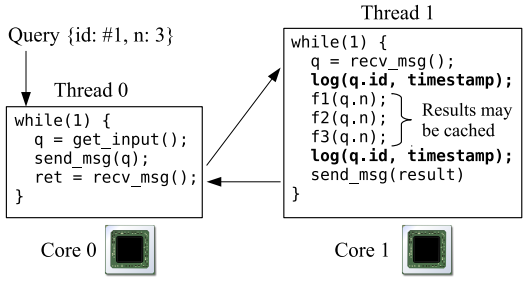

Fig. 7. Overview of the sample app. Each thread is pinned to a designated CPU core. Thread 0 receives queries and pass them one by one to Thread 1. Thread 1 does some work whose amount increases in proportion to the n in the queries. The performance fluctuates due to the caching mechanism.

Table II shows the evaluation environment. We use a CPU with the Skylake micro architecture because sampling timestamps with PEBS is only supported since Skylake. Other hardware and software have no special requirement and everything is a commodity. The machine has two NICs to receive packets from one NIC and send them to another one after processing.

B. Proof of Concept

In this section, we use our sample application as a proof-of-concept of our approach. The sample application mimics query answering applications and it is based on the self-switching architecture (Section III-C). Figure 7 illustrates how it works. It invokes two threads (Thread 0 and Thread 1), each of which is pinned to a designated CPU core. Thread 0 receives queries as inputs, and pass them one by one to Thread 1. A query consists of its id and a number (n) and Thread 1 applies linear transformations to $n \times 1000$ ($= N$) points $\{x_i, y_i\}$ ($i = 1...N$) and returns the results. The application's performance fluctuates because it has an in-memory cache mechanism. If the results of some points to be processed are already cached, the elapsed time is shortened because there is no need to re-calculate the results. To apply our method, Thread 1 is instrumented by two `log(d.id, timestamp)` lines to detect data-item switches. The logging code is inserted only at the beginning and the end of the while loop although the while loop includes three function calls (`f1`, `f2` and `f3`).

Figure 8 shows per-data-item elapsed time of each function of the sample application obtained by our method. The hardware event specified for PEBS is `UOPS_RETIRED.ALL` and the reset value is 8000. The x axis shows queries, each of which includes a unique id and a number n. The y axis shows the elapsed time of each query that are broken down

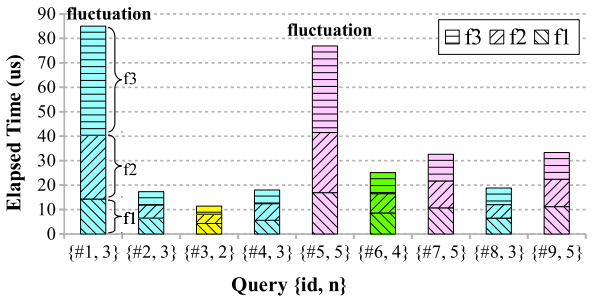

Fig. 8. Per-data-item elapsed time of each function obtained by our approach. It visualizes performance fluctuations where the 1st and 5th queries take much longer time. Queries with the same n have the same color.

into functions. Results for queries that have the same n are filled with the same color.

Figure 8 clearly shows that the performance of the sample application fluctuates even for queries with the same n. The 1^{st} query has the same n ($= 3$) with the 2^{nd}, 4^{th} and 8^{th} queries. However, the 1^{st} one takes much longer time than the others because the results are not yet cached when the 1^{st} query is input. The same is observed for the 5^{th}, 7^{th} and 9^{th} queries, which have the same n ($= 5$). The 5^{th} one takes much longer time than the others because only 3000 points are previously processed and the other 2000 points have to be newly processed. These fluctuations are only visible by traces with per-data-item basis. Even though the fact that the 1^{st} query is slower than the others can be detected by means such as service level logging, our method provides richer information than service level logging can. For example, the results show that `f3` takes much longer time than `f1` when the cache does not hit, and this knowledge is given because our method obtains elapsed time of each function per data-item.

C. Applying to a Realistic Application

1) The Application and its Performance Fluctuations: In this section, we show that our method can be applied to a real application. We use a sample application included in the DPDK [7] framework. DPDK is a framework for building high throughput software networking stacks. The application uses the Access Control List (ACL) functionality of DPDK to implement a simple firewall.

The ACL application has three worker threads (RX, ACL and TX) pinned to designated CPU cores. The RX thread receives packets and pushes them into a software queue connected with the ACL thread. The ACL thread retrieves the packets from the queue, and checks the installed ACL rules to judge if the packets should be forwarded or not. The packets are pushed to another software queue connected with the TX thread when the packets pass the ACL rules. The TX thread retrieves the packets and sends them to another NIC.

Packets going through the application experience different latencies (performance fluctuation) when a complex set of ACL rules that hold a specific condition is installed. Even though that this issue can be observed by just logging timestamps of ingress and egress packets, to further analyze the phenomenon requires traces that show performance metrics for each function per data-item. Although the phenomenon is reproducible by installing the same ACL rules into a controlled environment and sending the same packet, it is a very hard task if not impossible. Because the root cause of the fluctuation is not known a-priori, not only the ACL rules but also as many non-functional states have to be reproduced.

The performance fluctuations stem from implementation designs of the ACL functionality of DPDK:

1) It stores the ACL rules into trie structures to efficiently treat many of ACL rules. This is because a firewall of a large-scale datacenter tends to have thousands or even tens of thousands of ACL rules.
2) It divides the ACL rules into multiple trie structures. This is because storing all ACL rules into a single trie consumes too much memory when there are many rules.
3) A key of the trie structure consists of three parts: the source address (4 bytes), the destination address (4 bytes), and a combination of the source and the destination ports (2 + 2 = 4 bytes) of the TCP header.

From design (3), the computational cost to find rules that match a given packet can differ depending on how many bytes within a key has to be checked. For example, if a trie does not include any rule that matches the source address, the destination address and the source/destination ports need not to be checked anymore. However, if a trie includes a rule that matches the source address, the destination address and the source/destination ports, the full length of the key is checked. In addition, this difference is amplified by the number of tries because the same is applicable to every trie.

2) Tracing the ACL Application: We apply our method to the ACL application. We instrument and sample the ACL thread within the application because the other two threads (RX, TX) does almost nothing. The hardware event specified is `UOPS_RETIRED.ALL`. The source code of the ACL thread is instrumented to detect data-item switches. Because DPDK uses the self-switching software architecture, the instrumentation is trivial. The ACL thread is modified to output the timestamp right after it retrieves a packet from the RX thread and right before it pushes a packet to the TX thread.

Table III shows the installed ACL rules. The source and the destination addresses are the same across the all rules. For the rules with the source port of 1 to 666, the destination port ranges from 1 to 750. For the rules with the source port of 667, the destination port ranges from 1 to 500. The number of rules is $666 \times 750 + 500 = 50,000$ in total. The rules are stored in 247 trie structures. The vanilla DPDK stores ACL rules into at most 8 trie structures no matter how many rules exist, but we modified the source code to enlarge the limit.

Test packets are sent from GNET [17], a hardware network tester. It has three 10 Gbps NICs and two of them are connected to the test machine. The packets are sent from NIC 0 of GNET to NIC 0 of the test machine, and sent back from NIC 1 of the test machine to NIC 1 of GNET after passing the ACL application. Packets are sent one by one with a short

TABLE III
INSTALLED ACL RULES ($666 \times 750 + 500 = 50000$ rules)

Src Addr	Dst Addr	Src Port	Dst Port	Action
192.168.10.0/24	192.168.11.0/24	1	1	Drop
...	...	...	...	...
192.168.10.0/24	192.168.11.0/24	1	750	Drop
192.168.10.0/24	192.168.11.0/24	2	1	Drop
...	...	...	...	...
192.168.10.0/24	192.168.11.0/24	2	750	Drop
...	...	...	...	...
192.168.10.0/24	192.168.11.0/24	666	1	Drop
...	...	...	...	...
192.168.10.0/24	192.168.11.0/24	666	750	Drop
192.168.10.0/24	192.168.11.0/24	667	1	Drop
...	...	...	...	...
192.168.10.0/24	192.168.11.0/24	667	500	Drop

interval (not burstly) so that DPDK does not batch them. How to retrieve the IDs from batched data-items is future work.

TABLE IV
TEST PACKET TYPES

Type	Src Addr	Dst Addr	Src Port	Dst Port
A	192.168.10.4	192.168.11.5	10001	10002
B	192.168.10.4	192.168.22.2	10001	10002
C	192.168.12.4	192.168.22.2	10001	10002

Table IV shows the three types of test packets. For type A, both the source and destination addresses match some rules. To check if they pass through the firewall, the tries are traversed using all the three parts of the keys (src addr, dst addr and src/dst ports). For type B, the source address match some rules but the destination address does not match any rule. To check if they pass through the firewall, the tries are traversed using two components of the keys (src addr and dst addr). For type C, neither part of the key matches any rule. To check if they pass through the ACL, the tries are traversed only using the first part of the key (src addr). Packet latencies differ depending on the number of parts used to traverse the tries. Thus, the type A packets experience the longest latency and the type C ones experience the shortest latency.

3) Results and Findings: We estimate the elapsed time of the `rte_acl_classify` function for each type of the packet using our method. This function includes the main loop for the task to investigate if each packet can pass through the ACL rules or not.

Figure 9 shows the estimated per-packet elapsed time of the `rte_acl_classify` function. The x axis shows reset values and the y axis shows the elapsed time. The values are averaged over 10,000 runs and the error bars show the standard deviations. The "baseline" were obtained by inserting instrumentation code that output the timestamp at the beginning and the end of `rte_acl_classify` function. The "baseline" shows the golden data to which we compare the estimation

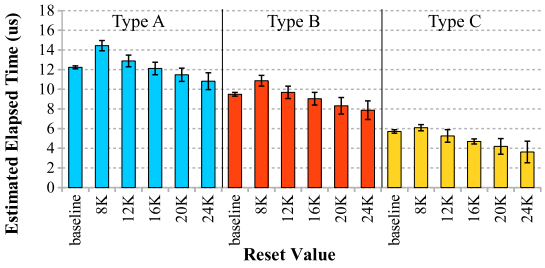

Fig. 9. Estimated per-packet elapsed time of rte_acl_classify function. The x axis shows reset values, and the y axis shows the estimated elapsed time. The error bars show the standard deviations. The label "baseline" for the x axis means that the value was estimated by a log-based method for comparison.

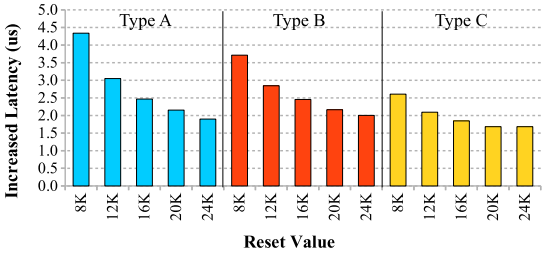

Fig. 10. Overhead of our method for different reset values. The x axis shows reset values and the y axis shows the increased latency for the reset value.

by our method. The results clearly show that the performance fluctuates by more than 100% (approx. 6 us for packet type C and 12 – 14 us for packet type A). Please carefully note two things: (1) We calculated the average to discuss the accuracy of our estimation, but our method estimates elapsed time of each function for every packet individually. (2) The baseline is obtainable by instrumentation method in this case because we a-priori know that this function fluctuates. In normal cases where the bottleneck is not known, instrumentation-only methods cannot be applied as we discuss in Section II-C.

Figure 10 shows the overhead of our method for each reset value. The x axis shows reset values and the y axis shows the latency increase. The latency was measured by the hardware tester (GNET). The overhead for reset value R is calculated by $L_R - L*$, where L_R refers the average latency of the packets for the reset value and L_* refers the average latency of the packets when no profiling is applied.

The total size of the samples generated by PEBS was 270 MB/s, 194 MB/s, 153 MB/s, 125 MB/s, and 106 MB/s, respectively for the reset values of 8K, 12K, 16K, 20K, and 24K. Suppose the CPU has 16 cores and the same analysis is applied to all the cores, the accumulated size per CPU are 4.3 GB/s, 3.1 GB/s, 2.5 GB/s, 2.0 GB/s and 1.7 GB/s, respectively for each reset value. Directly dumping all samples to durable storage (as we do in our prototype implementation) can be expensive, but this cost can be amortized by online processing of the samples. For example, one can estimate the elapsed time of each function online and dump raw samples only when the estimation diverges from the average by a threshold in order to analyze the phenomenon later offline.

Note that 4.3 GB/s is only less than 4% of memory bandwidth per socket of recent CPUs. Intel Xeon Platinum 8153 processor has 16 cores and 6 memory channels, resulting in 127.8 GB/s memory bandwidth with DDR4-2666 modules.

The results show that both accurate estimation and moderate overhead are achieved together with a proper reset value (e.g. 16K in this experiment). We conclude from this that our method is useful to diagnose performance fluctuations of real applications that occur in a production environment with a specific non-functional state.

V. Discussions

A. Applicability to Timer-Switching Architecture

We discuss how our method can be extended for the timer-switching architecture. In the timer-switching architecture, a data-item switch may occur even when processing of a data-item has not yet finished. This mechanism is typically implemented by a user-level lightweight scheduling mechanism such as user-level threading (e.g. [18]).

The key idea is to use a general-purpose register to store the data-item ID that is currently processed. When a data-item enters a core, the data-item id is stored to a general-purpose register such as `r13`. User-level threading mechanisms switch the values of general-purpose registers when a new user-level thread is scheduled. Therefore, once a general-purpose register has a data-item id, it is automatically switched to the id of a different data-item when the underlying user-level threads are switched. In this way, every PEBS sample has the id of a data-item that should be mapped to it. This requires that `r13` is never overwritten by anyone including the C library and the OS. As a first step, we confirmed that linux kernel and glibc can be built and work normally with no `r13` usage (except in the assembly code) by adding a compiler option.

B. Limitations

1) Limitation of Sampling-based Traces: Sampling based traces cannot provide meaningful information on functions that take shorter than the sample interval. The number of samples that belong to such functions is at most one and we cannot estimate the elapsed time. Thus, the sampling rate must be high enough to sample functions that are potential bottlenecks. In contrast, profiles can estimate "averaged" elapsed time of a function that is shorter than the sample interval. The same function is executed many times within a profile because the profile is averaged over a sufficient amount of time. The elapsed time of a function is calculated by $T \times \frac{n}{N}$, where T is the total elapsed time, n is the number of samples mapped to the function and N is the number of all samples.

2) Limitation of Hardware-based Sampling: Hardware-based sampling mechanisms can only sample a pre-defined set of information. One of the drawbacks is that PEBS does not support recording the call graph. As a result, if a sample mapped to function `g` exists between samples mapped to another function `f`, we can only "guess" that `g` is called by `f` but cannot guarantee it. This may lead to wrong understanding when a small utility function is called many times.

C. Choice of Reset Values

Our approach has a trade-off between overhead and accuracy as seen in the experiment for the ACL application. Lowering the overhead requires a smaller sampling rate and lower accuracy, and higher accuracy results in larger overhead. PEBS does not support specifying the sample interval with a time period. Thus, finding a right spot within the trade-off needs two relationships: (1) between reset values and overhead and (2) between reset values and sample intervals. The former is analyzed in our previous work [6]. It revealed that the extra execution time target programs take is accurately predictable from the number of samples taken during an execution, (almost) regardless of the application characteristics.

The relationship between reset values and sample intervals is more complex because PEBS does not support counting bare cycles that can be used as a timer. HW events other than cycles (e.g. micro operations retired) do not occur with a constant interval, thus sample intervals can neither be a complete constant. However, we confirmed for the PEBS samples taken for the ACL application that the sample intervals have a strong linearity with the reset values and the deviations are very small, which means that the sample interval is predictable from a given reset value. For finding the best reset-value for a given overhead requirement, one can refer our previous work [6].

D. Measuring Other Performance Metrics

In this paper focuses on how to estimate elapsed time of each function in a per-data-item basis. It is straight-forward to extend our method to retrieve other performance metrics such as the number of cache misses. One can just specify the hardware event that counts cache misses to be counted by PEBS. In this case, a PEBS sample (with a timestamp) is taken every time cache misses occur specified amount of times. The number of PEBS samples associated with each function (per data-item) by our integration approach means that how many cache misses the function has incurred. For example, if the number of PEBS samples that belong to function f_1 and data-item #1 is 10 and the number for f_1 and data-item #2 is 2, it means that the number of cache misses incurred by $f1$ fluctuates. Besides cache misses, PEBS supports counting various metrics for each core including the number of branch mis-predictions and the number of load instructions [11].

VI. Related Work

A. Instrumentation-only Methods

Per-data-item analysis of high-throughput software systems only with instrumentation are widely tackled. Ousterhout *et al.* [19] propose *blocked time analysis* that measures the amount of time a Spark task is blocked due to IO. They use logs (either existing or added) to record the blocked time for each Spark task individually. Their method requires to add logs to appropriate places (they reported that *"existing logging was often incorrect or incomplete"*), while in our scenario we cannot decide where to insert logs before solving a fluctuation.

VProfiler [20] instruments large scale software to find the root cause of performance fluctuations. To avoid the overhead

of instrumenting every function, it starts from instrumenting functions near the root and refines the result by instrumenting each of the child functions run by run. However in our scenario, performance fluctuations occur in a specific set of non-functional state and it is difficult to reproduce the fluctuations to refine the results for VProfiler.

Zhao *et al.* propose a non-intrusive method for per-data-item tracing [21], [22]. They do not require any modification to the source code of target programs. Their analysis on major large-scale software (e.g. Hadoop) found that they already output enough logs to distinguish each data-item in the default logging level. However, software that are used internally in enterprises or laboratories, or software that are under development may not output enough logs to apply their methods. They also propose Log20 [23], which automatically finds the best placement of log-printing statements. Although they provide a light-weight instrumentation library, their purpose is to find the root cause of system failures but not performance fluctuations. Their evaluation shows that they only insert 0.08 log-printing statements per data-item, which is too few to diagnose performance fluctuations.

B. Software-based Sampling

Software-based sampling is widely used to obtain traces, while we use a hardware-based method in this paper. The difference between software- and hardware-based sampling mechanisms is that the former can sample any information that is visible from software but it incurs large overhead (and the other way around; the latter is light-weight but not flexible).

vTune [24] and perf [10] are widely used performance analysis tools that utilize software-based sampling. They use the traditional performance counters (see Section III-B for the definition) to count hardware events such as cache misses, and sample program states by software every time an event occurs the specified number of times. Pyflame [25] is a sampling-based performance analysis tool for Python, which uses the `ptrace` system call to periodically attach and detach the target Python process. In software-based sampling mechanisms, target programs must briefly be suspended to allow the tools to sample consistent program states. In perf and vTune, the performance counter mechanism raises an interruption to the OS and the execution is forcefully switched from the target program. In Pyflame, target programs are suspended by the OS every time they are attached using the `ptrace` system call. The overhead by these brief suspensions of the execution can be negligible for profiles or coarse-grained traces, but it cannot be afforded in our approach as we have shown in Section II-C.

VII. Conclusions

Performance fluctuations are common across a variety of software. Traces are useful to diagnose performance fluctuations that occur only with a specific non-functional state. However, obtaining a trace for every function and every data-item incurs too much overhead for high-throughput software. We proposed a hybrid approach that combines coarse-grained instrumentation and sampling with adjustable overhead. Case studies showed that it pinpoints performance fluctuations of our sample application a real DPDK application.

References

[1] J. Huang, B. Mozafari, G. Schoenebeck, and T. F. Wenisch, "A top-down approach to achieving performance predictability in database systems," in *ACM International Conference on Management of Data (SIGMOD)*, 2017, pp. 745–758.

[2] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 141–154.

[3] J. Fenlason and R. Stallman, "GNU gprof The GNU Profiler," https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html.

[4] "VAMPIR 9.4," https://www.vampir.eu/.

[5] "The python profilers," https://docs.python.org/3.6/library/profile.html.

[6] S. Akiyama and T. Hirofuchi, "Quantitative evaluation of intel pebs overhead for online system-noise analysis," in *Int'l Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2017, pp. 3:1–3:8.

[7] "DPDK data plane development kit," http://dpdk.org/.

[8] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 161–175.

[9] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Prioritymeister: Tail latency qos for shared networked storage," in *ACM Symposium on Cloud Computing (SoCC)*, 2014, pp. 29:1–29:14.

[10] "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page.

[11] Intel Corporporation, "Intel 64 and ia-32 architectures software developer manuals," https://software.intel.com/articles/intel-sdm.

[12] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *the Symposium on Operating Systems Principles (SOSP)*, 2001, pp. 230–243.

[13] "Thread pool in mariadb," https://mariadb.com/kb/en/library/thread-pool-in-mariadb/.

[14] "Inside nginx: How we designed for performance & scale," https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/.

[15] "simple-pebs," https://github.com/andikleen/pmu-tools/tree/master/simple-pebs.

[16] V. M. Weaver, "Self-monitoring overhead of the linux perf_event performance counter interface," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 102–111.

[17] Y. Kodama, T. Kudoh, R. Takano, H. Sato, O. Tatebe, and S. Sekiguchi, "Gnet-1: gigabit ethernet network testbed," in *International Conference on Cluster Computing (IEEE Cluster)*, 2004, pp. 185–192.

[18] J. Nakashima and K. Taura, "Massivethreads: A thread library for high productivity languages," *Concurrent Objects and Beyond*, vol. 8665, pp. 1–18, 2014.

[19] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 293–307.

[20] J. Huang, B. Mozafari, and T. F. Wenisch, "Statistical analysis of latency through semantic profiling," in *the European Conference on Computer Systems (EuroSys)*, 2017, pp. 64–79.

[21] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A non-intrusive request flow profiler for distributed systems," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 629–644.

[22] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 603–618.

[23] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 565–581.

[24] Intel Corporporation, "Intel vtune amplifier," https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[25] E. Klitzke, "Pyflame: Uber Engineering's Ptracing Profiler for Python," https://eng.uber.com/pyflame/.