Adaptive Traffic Monitoring for Software Dataplanes

Gioacchino Tangari, Marinos Charalambides, Daphne Tuncer, George Pavlou
Department of Electronic and Electrical Engineering, University College London, UK

***Abstract*—Network operators have recently been developing multi-Gbps traffic monitoring tools that execute on commodity hardware and are part of the packet-processing pipelines realizing software dataplanes. These solutions allow sophisticated tasks to be performed on a per-packet basis, without relying on sampling or passive trace analysis, by leveraging the processing power available on servers. Although advances in packet capture have enabled intercepting packets from network cards at high rates, bottlenecks can still arise in the monitoring process as a result of concurrent access to shared processor resources, variations of the traffic skew, and unbalanced packet-rate spikes. In this paper we present an adaptive traffic monitoring approach that copes with emerging bottlenecks by timely detecting changes in the operational conditions and reconfiguring monitoring-related operations for subsets of traffic flows. Our solution performs responsive adaptations at the time scale of milliseconds and does not require a significant amount of resources. To demonstrate the capabilities of our approach we implemented it as part of a generic packet-processing pipeline and show that lossless traffic monitoring can be achieved for a wide range of conditions.**

I. Introduction

Recent advances in network management allow for frequent resource reconfigurations in response to a variety of events concerning network utilization, security, and changing user demand. This capability poses strict requirements on traffic monitoring, which cannot be satified by existing tools. Packet sampling, for example, has limitations in capturing transient events [1], OpenFlow counters can only report flow-aggregate information due to well-known scalability issues [2][3], while SNMP provides restricted visibility given its coarse report granularity and frequency.

To facilitate a detailed view of network-wide events, the research community has been recently investigating solutions where sophisticated measurement tasks are executed on a *per-packet* basis within the packet-processing pipeline, without relying on packet mirroring and passive trace analysis, thus enabling timely reports of fine granularity [4]. The development of such solutions on hardware switches, using sketch-based measurements [5][6] and novel languages like P4 [7], are promising but still at the proof-of-concept stage. At the same time, network operators have embraced the deployment of fine-grained traffic monitoring on commodity hardware, incorporated in the software packet-processing pipelines, also known as software *dataplanes*. This allows for enhanced flexibility and low cost since traffic monitoring can exploit the processing power of servers to execute complex per-packet measurements.

Achieving lossless packet processing when performing software-based traffic monitoring is a challenging task. On one hand it needs to cope with increasing data rates supported by network cards (10+ Gbps), which squeeze the admissible packet processing times to a few tens of nanoseconds. On the other hand it should satisfy the operator's requirement of assigning limited resources to the monitoring process (*e.g.*, 1 processor core per 10 Gbps [4]) while performing advanced measurement tasks on a per-packet basis. Although recent packet capture engines [9][8] and technologies such as Receive-Side Scaling (RSS) [10] cope well with packet capture at wire-speed, short-lived bottlenecks can still arise in the monitoring process. While packet rate spikes affecting one or a subset of cores can impose unsustainable workload burdens, dynamic changes in traffic skew and concurrent access to shared server resources can inflate the per-packet latency.

A possible approach to limit the risk of packet loss is to allocate more cores to traffic monitoring. An alternative is to restrict the available measurement-related operations to a minimal set such as byte and packet count. These solutions however result to inefficient use of the resources. In this paper we address these limitations by proposing a novel *adaptive* traffic monitoring solution for software dataplanes, where measurement operations are reconfigured in a responsive manner in the face of emerging bottlenecks to reduce packet processing times. Compared to state-of-the-art platforms that use multi-core architectures [18] [19], our work takes a new step towards lossless packet processing by investigating how the operations of a monitoring process can be adapted at run time in order to meet dynamic resource availability at individual cores. More specifically, our solution relies on the frequent estimation of the operating conditions coupled with extensive offline analysis of the different per-packet latencies involved in the monitoring process. To avoid the consumption of additional resources, we designed the monitoring adaptations to operate together with the packet-processing pipeline on the same core. This is achieved using procedures that run to completion in short times, so as to avoid starvation in the packet capture buffers, and generate a small overhead in terms of CPU-time.

We investigate the benefits of the proposed adaptive monitoring approach by implementing it based on a generic and widely-used [4][11] traffic monitoring pipeline relying on a hash table. The results of the experiments demonstrate that our solution can significantly reduce the risk of packet loss under various events such as multi-Gbps traffic rate spikes, increasing processor concurrency and changes in traffic skew. Moreover, we show that monitoring adaptations can be performed in short time-scales (10 ms), while incurring a small CPU-time overhead (≈1%).

The remainder of this paper is as follows. We provide background information on software-based traffic monitoring

978-3-901882-98-2 © 2017 IFIP

in Sec.II. We then describe the proposed approach in Sec.III-V and evaluate its performance in Sec.VI. Related work is discussed in Sec.VII and final remarks are presented in Sec.VIII.

II. Background

A number of research approaches have recently embraced the use of commodity hardware to realize a wide range of network functions, as this entails improved flexibility and reduced costs. Traffic monitoring, in particular, is a good candidate for such an implementation, as it can benefit from the processing capability of powerful servers in order to perform complex measurement tasks at the granularity of a single packet, without the need to employ sampling techniques. As such, network operators have started developing monitoring solutions that are part of the software packet processing pipeline, *e.g.*, in a software switch.

Compared to monitoring operations in hardware switches, where memory availability is the main shortage, traffic measurements on commodity hardware is constrained by the CPU-time and the *working set*, *i.e.*, the data most frequently accessed for the measurements [11]. This clearly reflects on the design choices for such monitoring tools. Instead of using more complex measurement techniques like *heap-based* [12] solutions and *sketches* [6][13], which reduce the total memory usage, traffic monitoring for software dataplanes can just rely on simple hash tables for storing the traffic flow statistics, as they guarantee the best performance in terms of CPU-time and working set [11]. Hence, the monitoring process consists in hashing the header of each packet, *e.g.*, on the 5-tuple, and storing a set of statistics in the hash table.

To satisfy the operator's requirements, traffic monitoring on commodity hardware has to combine three main features: handling high traffic rates at a limited cost (*e.g.*, 1 core for 10 Gbps [4]), achieving zero packet loss and supporting diverse, sophisticated forms of analysis. However, to collectively meet these requirements is not a trivial task. In order to sustain high throughputs (10+ Gbps), the monitoring process should ensure total packet processing times in the order of few tens of nanoseconds, *e.g.*, no more than 70 ns for 10 Gbps of 64-byte packets. Current state-of-the-art practices, such as RSS and capture engines (frameworks like DPDK and Netmap) provide essential support by ensuring packet capture at wire-speed. While these techniques can get packets from the network card to the monitoring process at a high rate, they only solve half the challenge since monitoring bottlenecks can emerge after packets have been captured. These are described below.

Traffic rate variations High-speed packet processing servers use multiple cores and RSS to deal with multi-Gbps traffic. However, these setups are still prone to performance degradation. If the amount of resources devoted to monitoring is limited (*e.g.*, in small-scale deployments), a single core can still face unsustainable workloads at high traffic rates. In addition several events such as fast variation of user demand [15], sub-second congestion [16], or DoS attacks [4] can result to traffic rate spikes affecting one or more cores, even for deployments with multi-queue packet capture (such as RSS).

Shared resource contention A monitoring process usually coexists with other tasks on the same machine, often on the same processor, including other monitoring processes running on different cores. Resulting hardware resource contention [14] involves caches, the memory controller and buses. Among these, the L3 cache, shared by multiple cores in modern platforms, accounts for most of the performance degradation in traffic monitoring, since measurement tasks are particularly aggressive in terms of L3 references per second. Assuming that on a n core processor the monitoring process executes on core 1, variation in data access patterns of other processes running on cores 2 to n can affect the monitoring time per packet due to cache entry replacements, resulting to a higher miss ratio.

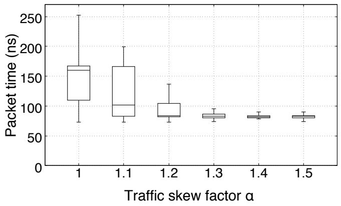


Fig. 1: Packet processing time *vs.* traffic skew

Change of traffic skew The skewness of the traffic distribution plays a key role at run time as it defines the monitoring working set. For traffic with lower skew a higher fraction of packets cannot be served from the processor caches, resulting in higher packet processing latencies. As an example, we measure the packet completion time of a simple monitoring process that updates packet and byte counts using packet traces with different skewness.[1] As shown in Fig.1, reductions of the skew factor α can double the per-packet latency.

III. Adaptive monitoring

Bottlenecks in the monitoring process, resulting from the aforementioned conditions, translate into longer queues in the packet capture stack, which leads to higher chances of packet loss. This is an important problem given also the reduced size of RSS queues (no more than 4K packets) and packet I/O rings [17], enough to absorb only less than one millisecond of traffic at 10 Gbps. To ensure resilience to potential bottlenecks and achieve lossless packet-processing, the operator can only count on resource overprovisioning, or restrict the measurement-related operations to a minimal set, *e.g.*, packet and byte counting only. However, the former approach violates the requirement of allocating a limited amount of resources for monitoring and the latter penalizes the granularity and expressivity of monitoring reports.

To address these issues we propose a novel *adaptive* monitoring approach, with which measurement operations are reconfigured at run time to cope with emerging conditions. Our solution is based on two main tasks: (i) timely detection of changes in operating conditions and estimation of available time for measurements, and (ii) swift response to emerging bottlenecks by reconfiguring the set of monitoring operations to obtain *light-weight* packet processing in cases where adverse conditions reduce the available time. Compared to existing approaches that use multi-core packet scheduling [18][19], our solution tackles the challenge of lossless traffic analysis from a different angle, by leveraging adaptations in the monitoring process itself, *i.e.*, at the level of a single core.

We illustrate the main concept of the approach through a simple example. Consider a monitoring application for TCP traffic supporting two monitoring setups. One (*light*) only

[1] We use a Zipf distribution and 10^5 flows

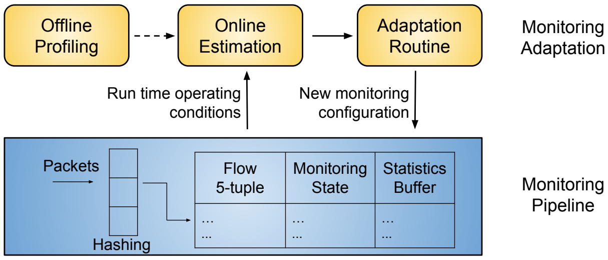


Fig. 2: Overview of the proposed approach

involves byte and packet counting for each traffic flow and consumes a limited time, while the other (*heavy*) analyzes TCP flows in depth, thus requiring many more CPU cycles for each incoming packet. The second setup is initially selected for all flows, which allows detailed measurements, without incurring packet loss given the current conditions. At a certain point in time, a bottleneck is created due to the conditions described in Section II, and the time for monitoring now exceeds the current availability. With existing systems this brings considerable packet loss as soon as the input queue is saturated. Instead, our solution re-estimates the amount of time available for monitoring at run time, and subsequently shifts enough flows from the *heavy* to the *light* setup to resolve the bottleneck.

Approach overview We model the traffic monitoring process as a simple packet-processing *pipeline* based on a single hash table, where incoming packets are hashed on their 5-tuple to match a corresponding *flow-entry*. Flow-entries contain an identifier called the Monitoring State, that indicates which measurement operations should be performed for each packet of a flow. Each operation entails a different cost in terms of processing time. At run time, flow-entries can shift between the available Monitoring States.

Monitoring adaptation operates together with the pipeline as part of the same process to avoid usage of additional resources (*i.e.*, more cores) as well as synchronization overheads. As shown in Fig.2, our solution relies on a three-phase procedure. The first one, *Offline Profiling*, runs at the initialization of the monitoring pipeline, before the start of an incoming packet stream. Its role is to profile the various processing times involved in the monitoring pipeline. The other two, namely *Online Estimation* and *Adaptation Routine*, execute at run time on a small time-window basis.

In each time-window, the Online Estimation procedure extracts the current run time conditions of the monitoring pipeline using limited additional measurements and a few key results from the Offline Profiling phase. At the end of each time-window, based on the extracted knowledge an estimate of the available monitoring time *per-packet* is generated, which is set as the target for the next epoch. The *Adaptation Routine* takes this value as input and, if the actual monitoring configuration exceeds the target time, it produces a new configuration of flow Monitoring States that achieves an adequate time reduction.

Challenges Three main challenges are addressed for the design of this solution. Achieving a reasonable overhead at run time is the first challenge. The Online Estimation and the Adaptation Routine time-share the CPU with packet processing since they operate on the same core. As such, their consumption of time can directly affect the monitoring pipeline by decreasing the sustainable packet-rate and incurring additional loss. For this reason, we design all operations involved in the monitoring adaptation so that (i) they generate limited CPU-time overhead, and (ii) they all run to completion in short times (no more than $10\mu s$) to avoid starvation in the packet capture queue. The second challenge is how to accurately estimate the operational conditions, which we address by using tools that provide high level of precision and are light-weight enough to be deployed at run time. The last challenge is about achieving reactive adaptations, for which we design each component of our solution to work with time-windows as small as $10ms$, still with a reasonable overhead.

IV. Offline and online time estimations

To enable monitoring reconfigurations, the proposed approach is first responsible for detecting on a short time-window basis changes in the operating conditions and estimating the amount of time available to perform monitoring operations for each packet. Based on this estimation it then adjusts the set of monitoring operations for subsets of the flow-table entries to ensure that all packets can be processed on time.

A. Expected Time per Packet

The total expected time associated with each packet in the monitoring pipeline depends on whether the packet belongs to a new flow, for which no entries exist in the flow-table, or to an existing flow. In the first case, this represents the total processing time for a new-flow packet (including the new flow-entry insertion), denoted here as T_i. In the second case, the time can be decomposed in two main components: the retrieval time T_r, *i.e.*, the time for retrieving the measurement data for the packet, including hashing and accessing the matching flow-table entry, and the processing time T_p, *i.e.*, the time needed to perform, after the necessary data has been retrieved, the operations in the current Monitoring State of the matching flow-entry. The total expected packet time T_{pkt} can then be estimated based on the following equation:

$$T_{pkt} = (1 - \lambda_f)(T_r + T_p) + \lambda_f T_i \tag{1}$$

where λ_f represents the ratio of packets belonging to new flows over the total number of packets processed in the current time-window. Based on the findings reported in [14][11], which show that the probability of retrieving data from L3 processor cache is by far the dominant factor affecting the retrieval time, T_r can be further decomposed as:

$$T_r = T_r^H \cdot P + T_r^M \cdot (1 - P) \tag{2}$$

where T_r^H and T_r^M represent the retrieval time in case the data is accessed from the processor cache and from memory, respectively, P is the probability of cache hit (*i.e.*, the matching flow-table entry is retrieved from the cache), and $(1-P)$ is the probability of cache miss (*i.e.*, access to memory). Combining equations (1) and (2), the time per packet T_{pkt} is given by:

$$T_{pkt} = (1-\lambda_f)[T_r^H \cdot P + T_r^M \cdot (1-P) + T_p] + \lambda_f T_i \tag{3}$$

As observed from equation (3), T_{pkt} can be obtained based on the estimation of six variable values. To keep the run time adaptation cost as low as possible, the best approach to determine these values is to perform the estimation offline, for example based on benchmarking. While this works well for T_p, T_r^H, T_r^M and T_i, it cannot apply to P and λ_f given that both variables strongly depend on the run time conditions. In this case, an online procedure is required.

Operation	5th Quantile Time (ns)	95th Quantile Time (ns)	Std dev.
Timing	19	28	2.73
T_p: Pkt & Byte counting	22	31	2.78
T_p: Tcp diagnosis	83	98	4.01
T_r^H	80	128	8.87
T_r^M	149	272	39.20

TABLE I: Statistics for T_r and T_p datasets

B. *Offline Profiling*

The objective of the Offline Profiling phase is to characterize the resource utilization of traffic monitoring by analyzing the execution times T_p, T_r^H, T_r^M and T_i through a set of benchmarks. While resource consumption can be easily derived online in the case of adaptive monitoring solutions dedicated to hardware switches (each monitored flow strictly maps to a single flow-entry in TCAM), it is a much harder task to achieve in software deployments, where the focus is on the CPU time rather than the total memory usage. Not only do the processing times depend on the server hardware (*e.g.*, clock rate), they also vary based on what monitoring operation must be performed on a packet. Offline Profiling overcomes this limitation by building the knowledge with which resource utilization can be tracked at run time with a limited cost.

Estimating the processing and retrieval times To determine the value of the processing and retrieval times, we leverage the observation that in practice the processing time T_p is much more predictable than the retrieval times T_r^H and T_r^M [2]. Intuitively, the processing time for each Monitoring State is proportional to the number of boolean/arithmetic operations executed in that state. In contrast, T_r^H and T_r^M, which are dominated by the flow-entry retrieval time, can be affected by possible hash collisions, the use of different processor caches in the available hierarchy, or unwanted episodes regarding memory allocation, such as TLB (Translation Lookaside Buffer) misses, whose impact also depends on the server hardware and kernel configuration (*e.g.*, memory page size).

To illustrate these effects, Table I shows the statistics of the distribution of the execution times for data retrieval (T_r) as well as various monitoring operations (T_p).[3] A timing operation is also included, which measures the duration of the different operations using a high definition timer. As observed, while T_r^H and T_r^M exhibit high statistical dispersion, the values of T_p for the two considered Monitoring States are characterized by low standard deviation that can partly be attributed to the bias introduced by the timer. Based on these results, we develop two different strategies to estimate the values of T_p and T_r^H/T_r^M, respectively.

Processing time estimation. Given the predictability of T_p, the processing time required for each Monitoring State s_i ($T_p^{s_i}$) can be estimated by collecting samples of $T_p^{s_i}$ over a large packet trace and setting the value of $T_p^{s_i}$ to the sample mean.

Retrieval time estimation. Due to the sensitivity of the retrieval times, we propose a different approach based on statistical model fitting to estimate the value of T_r^H and T_r^M. The proposed approach works in two steps as described below.

	Normal (baseline)	Weibull (BIC to baseline)	Gumbel (BIC to baseline)
T_r^H	0%	+1.86%	-3%
T_r^M	0%	+8.1%	-7%

TABLE II: Model selection for T_r^H and T_r^M

The objective of the first step is to collect two datasets of time samples, one for T_r^H and one for T_r^M. When collecting the relevant datasets, it is essential to ensure that flow entries reside in either the fast processor caches (for T_r^H) or in memory (for T_r^M). This can be achieved by modulating the size of the monitoring working set used by the input packet stream (*i.e.*, number of unique flows in each trace). Given the L3 processor cache size S, with N_H and N_M denoting the number of different flows in each trace (*i.e.*, for T_r^H and T_r^M, repectively), and the size of the flow-table entry F, the size of the monitoring working set should be so that the following conditions are satisfied: $N_H.S < L3$ (for T_r^H to force cache hit) and $N_M.S \gg L3$ (for T_r^M to ensure cache miss).

The second step consists in selecting, using a standard fitting strategy, the most appropriate statistical model to represent the distribution of each variable. Although different distributions can be taken into account, we simplify the process and restrict our choice to three representative cases capturing well various degrees of sample asymmetry. In particular, we select the Normal distribution as the baseline, as well as two distributions characterized by a heavy tail, namely the Weibull and the Gumbel distributions. We use the Bayesian Information Criterion (BIC) for the model selection criterion as it shows more consistent results compared to the maximum likelihood estimation for very large sample sizes. It is based on $-2\log(likelihood)$, so the lower its value the better the fit.

Table II shows the results of the model fitting strategy based on the considered distributions for the setup of Table I. As observed, the best fit is obtained with the Gumbel model for both T_r^H and T_r^M.

Estimating the flow-insertion time The objective of this procedure is to extract the total execution time for packets of new flows T_i. The estimate of T_i is taken as the average of all T_i values collected from a large packet trace, (*e.g.*, approximately $210ns$ for the setup in Table I) under the assumption that new-flow packets form a small subset of the total traffic (*e.g.*, no more than 10%). In cases like SYN attacks, where T_i becomes dominant, existing resilience mechanisms [4], that are out of the scope of this paper, could be used in conjunction with our solution.

C. *Online Estimation*

The objective of the Online Estimation phase is to determine at each time-window the value of λ_f and P, as well as the packet arrival rate at the capture engine queue λ_{pkt}. The result is an estimate of the available per-packet time for the next time-window, which is based on the run time conditions of traffic monitoring and the value of T_p, T_r and T_i computed during the Offline Profiling phase.

To estimate the value of λ_f, we use a simple strategy incurring negligible overhead that consists in counting the flow-table insertions and dividing this value by the total number of processed packets during each time window. To derive the packet capture rate λ_{pkt}, we periodically update the count of packets that have been written in the queue, *e.g.*, each time a new packet burst is loaded by the packet acquisition library. In DPDK, for instance, this information can be retrieved using the counts in the rte_eth_stats and rte_ring API.

[2] Extensive analysis performed during this study confirmed this behavior.

[3] We used a 2.7 GHz CPU with 3MB L3 cache

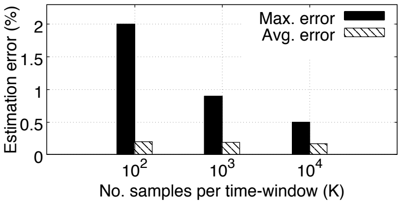


Fig. 3: Precision of P estimation

Several methods can be considered for estimating the value of the variable P. One method is to use an analytical model to predict the cache miss rate as proposed in [20]. However, this method does not apply well to our solution as (i) it requires that the temporal behavior of the application, in terms of reuse of addresses, has a single profile, which does not apply in our case; and (ii) it does not consider the effect of co-runner processes on the cache hit ratio. Other approaches involving online Miss Rate Curve generation generally incur substantial overheads (*e.g.*, an additional 230 ms is reported in [21]), while faster techniques [22] rely on cache-related hardware counters that are restricted in current hardware [23].

In this paper, we propose a simpler approach which is based on T_r sampling and uses the models of T_r^H and T_r^M obtained from the Offline Profiling phase. In each time-window, the proposed approach periodically samples the flow retrieval time with a high precision timer. Denoting K as the number of samples to collect, the sample period can be approximated by λ_{pkt}/K. For each sample t_r^i, the approach first computes $Prob(T_r > t_r^i|hit)$, *i.e.*, the probability for the retrieval time to be greater than t_r^i assuming that the retrieval was from a hit, and $Prob(T_r \leq t_r^i|miss)$, *i.e.*, the probability for the retrieval time to be lower than t_r^i under a miss. Given the model of T_r^H and T_r^M computed through profiling, the value of $Prob(T_r > t_r^i|hit)$ is obtained by $1 - CDF_{T_r^H}(t_r^i)$ and $Prob(T_r \leq t_r^i|miss)$ is obtained by $CDF_{T_r^M}(t_r^i)$. Let denote as c the vector of results so that all elements $c(i)$ are equal to:

$$c(i) = \begin{cases} 1 & 1 - CDF_{T_r^H}(t_r^i) \geq CDF_{T_r^M}(t_r^i) \\ 0 & otherwise \end{cases}$$

The value of P is approximated as the percentage of non-zero values in the vector c.

To illustrate the performance of the proposed approach in terms of estimation accuracy, we use it to classify 10^3 different ground-truth traces (for which P is known - P_{truth}) that we obtained by modulating the traffic skew as explained in Sec.II.[3] The estimation error is measured as the difference in percentage between the value of P_{truth} and the value of P computed by the algorithm. As depicted in Fig.3, our method achieves very high accuracy on average. We also compare its performance to the one obtained using a naive classification where each $c(i)$ is set to 1 or 0 by simply using the distance of each t_r^i from the sample mean of T_r^M and T_r^H. In this case, for $K = 10^3$ the error is around 5%, whereas with our method it is below 1%.

Available processing time Given the values of λ_f, λ_{pkt} and P, and the variables T_p, T_r^H, T_r^M and T_i, the Online Estimation procedure finally extracts the average processing time for the next time-window T_p^{target} by setting:

$$(1-\lambda_f)[T_r^H P + T_r^M(1-P) + T_p^{target}] + \lambda_f T_i = 1/\lambda_{pkt} \quad (4)$$

V. Adaptation Routine

If the current monitoring configuration exceeds the target time T_p^{target} from equation (4), the task of the Adaptation Routine is to decide which monitoring operations should be avoided next, and for which flows.

There are various ways in which this functionality can be realized. A *straw-man* approach would solve a convex optimization that re-assigns an amount of CPU-time to each monitoring task, *i.e.*, to each flow-entry, so that a specific accuracy objective is maximized. This approach is not viable for two reasons. First, it would require a well defined resource-accuracy relation, which cannot be easily characterized as it jointly involves the monitoring process logic (transitions between Monitoring States) and the traffic characteristics. Second, the execution time of the optimization would likely incur additional packet loss as the monitoring pipeline needs to stop until a new configuration is generated. Another approach would be to take many smaller scale decisions, at the arrival of each new flow or packet, based on the current resource *head-room* from the initial T_p^{target}. However, this can introduce excessive run time overhead and can also result to unfairness between the first and last flows/packets within a window.

To overcome these limitations, instead of working with individual 5-tuples or packets, the proposed Adaptation Routine operates at the granularity of Monitoring States, as each one maps to a different subset of the flow-table. Once invoked at the end of a time-window, the developed algorithm retrieves the count of packets processed at each Monitoring State s_i in the last time-window. This provides an indication of the current share of each s_i. Portions of the flow-table are subsequently re-allocated to more light-weight states such that T_p^{target} is achieved. This adaptation can be performed with two different methods, based on how transitions between Monitoring States take place in the monitoring pipeline.

Method 1: Stateful dataplane We initially consider a scenario where transitions are decided within the packet-processing pipeline [24]. Each Monitoring State incorporates the necessary logic in its code for transitions to other states once specific conditions on flow-entry statistics are met. An example is depicted in Fig.4, where for each new packet a flow-entry can progress to a more advanced state involving more operations, stay in the current state, or roll-back to the previous one, which is more light-weight. Such a scheme is representative of complex monitoring applications where different events, such as network congestion and security threats, trigger ad-hoc monitoring configurations.

In this method the adaptation operates using an iterative procedure. First, it calculates the current average processing time $T^0 = (\sum_{i=1}^{k} n_i T_p^{s_i}) / \sum_{i=1}^{k} n_i$, where n_i is the number of packets in state s_i during the last time-window. Then, the adaptation starts re-allocating flow-entries to more light-weight Monitoring States, for which two strategies can be adopted.

The first strategy (*Greedy*) computes at each iteration j the expected average time if all flows were forced to their previous Monitoring State. If this value, T^j, exceeds T_p^{target}, a new iteration is executed except if all states are empty apart from the initial one (s_1 in Fig.4). If not, the procedure takes the monitoring configuration for T^{j-1} and forces a portion x of the flow-table to an additional step-back in the Monitoring

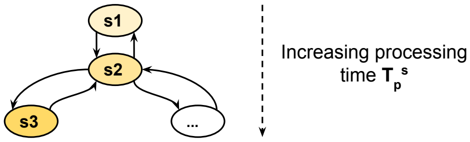


Fig. 4: Monitoring states in a stateful dataplane

State set, where $x = (T^{j-1} - T_p^{target})/(T^{j-1} - T^j)$. When the monitoring pipeline restarts, the new configuration is applied using the hash of the first *new* packet for each flow-entry. By comparing the hash with x (for example, a modulo operation can be used), it decides to update the flow Monitoring State to j or $j-1$ steps back.

The second strategy (*Low-States-First – LSF*) operates in a more selective fashion. Given the initial monitoring configuration, it considers the flow-entries in the most light-weight Monitoring State excluding the initial/default one (s_2 in Fig.4) and it moves them by a step-back (to s_1). It then updates the average processing time. If the new time estimation is below T_p^{target} it terminates, otherwise in the next iteration the same is applied to the flow-entries in the following Monitoring State, *i.e.* flow-entries in s_3 are moved to s_2, and so on. After the same action has been taken for all flow-entries, the following iterations move flow-entries by an additional step-back, starting again from the ones in the most light-weight Monitoring States. For instance, assume that the procedure terminates at iteration j and that the last processed flow-entries have been the ones in Monitoring State h. Then, the last operation is to force a portion $x = (T^{j-1} - T_p^{target})/(T^{j-1} - T^j)$ of the flow-entries in Monitoring State h to an additional step-back in the Monitoring States set.

As in the case of the *Greedy* strategy, the final configuration is applied using the first new packet of each flow-entry. The only difference is that, before comparing the packet hash with x, a preliminary check of the current Monitoring State is needed. Compared to *Greedy*, this strategy introduces a bias towards the more advanced Monitoring States, as for these states more detailed information on emerging episodes is collected, *e.g.*, for root cause analysis.

Method 2: Control-plane based monitoring We consider now a scenario where transitions between Monitoring States are explicitly driven by an external *controller*, which is representative of proposals such as [4]. The controller periodically issues monitoring *queries*, each triggering small sets of monitoring operations for a specific subset of flows, for example a specific IP source subnet. Therefore, at each time it is the set of queries affecting a flow-entry that determines its Monitoring State.

Compared to the previous scenario, this case poses two new requirements. First, in this case the controller can explicitly indicate the priority of monitoring tasks by providing a ranking of queries, which should be respected by the adaptation. In addition, the adaptation has to preserve query integrity, *i.e.*, it should reconfigure the flow Monitoring States such that each query is either discarded or satisfied with all required flows' statistics. Violating this constraint can lead to inconsistency of information at the controller.

As in Method 1, the adaptation operates iteratively by forcing flows to less advanced Monitoring States until the target processing time is met. However, given the additional requirements, this is achieved by deactivating the current controller queries, starting from the lowest ones in the ranking. At run time the new monitoring configuration is applied, again, using the first new packets matching each flow-entry, but this time a check of what query is still active or not is needed.

VI. Evaluation

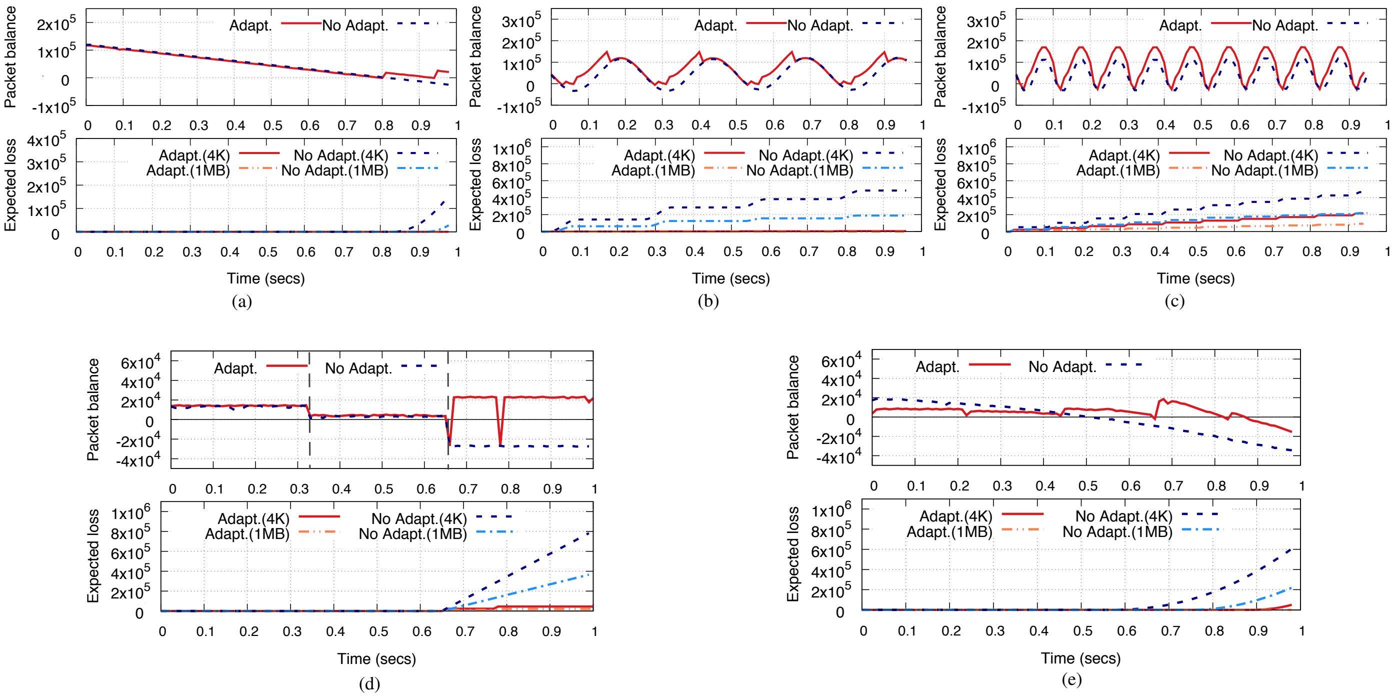
We implemented our solution using the C programming language as part of a generic monitoring pipeline based on a single flow-table. A hash table was used to realize the flow-table where collisions are handled by chaining – a table size of 2^{20} entries was chosen to limit the risk of hash collisions. We also set the flow-entry size to 64 bytes such that it can fit within a single cache line. To generate the input packet streams to the monitoring pipeline, we take the following approach. Since the focus is on the bottlenecks arising in the monitoring process, for each experiment we build a packet trace and pre-load it in memory (at initialization), and then fetch packets with small bursts at run time so as to isolate the monitoring pipeline from the packet capture stack. For each experiment we pre-load 1 second of traffic, which approximately translates to 1GB allocated memory for 10Gbps of traffic. The packet trace we use only includes TCP flows and is derived from recently published results on flow statistics in data centers [25].

We conducted the evaluation in two main steps. First, we analyze the performance of adaptive traffic monitoring in terms of packet loss risk and adaptation responsiveness. Then, we extensively evaluate the performance of monitoring adaptations focusing on their completion time and the associated run time overhead. The experiments have been conducted on an Intel i7-4790 CPU with 4 physical cores at 3.6 GHz and shared L3 cache of 8 MB.

A. Lossless Traffic Monitoring

We compare our approach (*Adapt*) with a more traditional setup where monitoring operations are not dynamically reconfigured (*No-Adapt*). We focus on two metrics that represent the risk of packet loss as a result of bottlenecks in the monitoring process. The first is *packet balance*, which indicates whether the system can process the number of packets in the trace during each time-window – a negative value signifies that the system cannot cope with the packet rate. The second metric is the *expected packet loss*, which quantifies the loss given the size of the input buffer. We compute this using a queue model for two buffer sizes: 4096 packets (saturation of a RSS queue) and 1 MB (maximum size range for circular buffers in packet acquisition libraries like DPDK and Netmap) [17].

We define 3 possible Monitoring States: the first ($T_p^{s_1} \approx 1ns$) only updates the packet count for the flow-entry, the second ($T_p^{s_2} \approx 30ns$) computes the flow rate and checks if a packet is part of a burst, and the third ($T_p^{s_3} \approx 60ns$) diagnoses TCP flows in depth. In the initial configuration $1/3$ of the flows is assigned to each of the Monitoring States at each time-window. Also, the monitoring adaptation time-window is set to $10ms$. To study the performance of our solution under the occurrence of bottlenecks we perform three types of experiments, which reproduce the three main conditions described in Sec.II.



(a) (b) (c)

(d) (e)

Fig. 5: Packet balance and expected packet loss

Traffic rate variations In these experiments we test our solution against multi-Mpps variations of the input rate, which are realized by tuning the rate of exponential packet inter-arrivals in the trace. We increase the rate linearly from 0 to 14.8 Mpps (10 Gbps of small packets) during a 1 second interval. As shown in Fig.5a, our solution achieves a significantly higher packet-rate compared to the non-adaptive approach (by 2 Gbps). In the *No-Adapt* case losses are observed after $t \approx 0.8s$, as soon as the rate becomes unsustainable, and a larger buffer (1MB) can only postpone losses by a few tens of milliseconds. To avoid the loss a second core would need to be allocated for monitoring the same input packet stream. In contrast, the two adaptations performed by our approach at $t \approx 0.8s$ and $t \approx 0.95s$ allow to sustain a maximum rate of 14.8 Mpps on a single core without any packet loss.

In addition, we evaluate the responsiveness of monitoring adaptations in the case of short rate spikes. To emulate the spikes, we generate packet-rate oscillation between 0 and 14.8 Mpps based on the function $sin(t/T)$, where T is the oscillation period. Two representative cases, $T = 250ms$ and $T = 100ms$ are depicted in Fig.5b and 5c, respectively. For $T = 250ms$, monitoring adaptations always provide a response to arising bottlenecks in time, before packet loss occurs. In the case of $T = 100ms$, huge packet rate variations, up to the equivalent of 3Gbps for 64-byte packets, are generated in the time-span of a single monitoring adaptation time-window ($10ms$). As such, some losses can occur before the new monitoring configuration is applied, *i.e.*, in the $10ms$ time-window preceding the adaptation. However, even for such intense and short-lived spikes our approach significantly outperforms *No-Adapt* in terms of loss, with a reduction of more than 50% for both buffer sizes.

Shared resource contention The objective of the next experiments is to assess how monitoring adaptations handle variations of the operating conditions in terms of concurrent access to shared resources. To emulate concurrency we use the approach in [14]: we run our solution on core 1, and *co-run* other processes on cores 2, 3 and 4. Each co-runner is defined as a special monitoring process that only retrieves flow-entries, so as to maximize the L3 cache references per second. As depicted in Fig.5d, we split the 1-second experiment into three intervals of length $1/3s$ each. In the first interval we execute 1 co-runner (on core 2), in the second interval we execute 2 co-runners (on cores 2 and 3), and in the last interval we execute 3 co-runners (on cores 2, 3 and 4).

As shown in Fig.5d, increasing levels of concurrency lead to performance degradation in terms of packets processed per time-window, and considerable loss for the *No-Adapt* setup with 3 active co-runners, regardless of the input buffer size. This is due to the inflation of retrieval times T_r as a result of increasing L3 cache misses. In contrast, our solution achieves minimal loss since concurrency variations are detected at run time through P estimation (Sec.IV-C) and a new monitoring configuration is provided within $10ms$.

Change of traffic skew We finally evaluate the performance of our solution under variations of traffic skew. To reproduce these variations we split the input packet trace into smaller intervals of $20ms$ and for each interval we assign packets to flows (5-tuples) based on a Zipf distribution with parameter α (flow population size of $2.5 \cdot 10^5$). We start with $\alpha = 1.5$ (high skew) at $t = 0s$ and we gradually decrease the skew factor until $\alpha = 1$ at $t = 1s$ to obtain more *uniform* traffic. The packet rate has been fixed at 10 Mpps. As expected, smaller values of α lead to a significant performance drop since less packets are served with flow-entries from the L3 cache. As shown in Fig.5e, our solution can sustain 10 Mpps on a single core under considerable skew variations, and prevents losses at much lower skew factors, $\alpha \approx 1.1$ ($t = 0.9$), compared to the *No-Adapt* case, which starts starving packets in the input buffers for $\alpha \approx 1.25$.

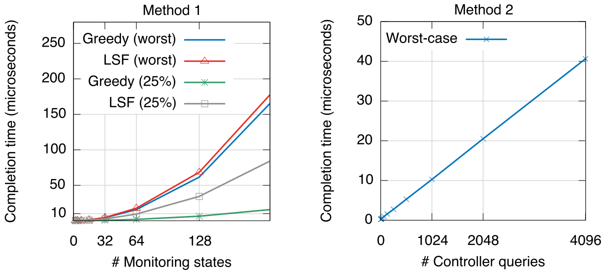


Fig. 6: Adaptation routine completion time

# Monitoring states	Overhead
16	0.5%
32	0.55%
64	0.57%

# Controller queries	Overhead
256	0.7%
1024	1.3%
4096	1.5%

TABLE III: Run time overhead

B. Monitoring Adaptation Overhead

We evaluate the cost of monitoring adaptations with respect to (i) the execution time of the Adaptation Routine and (ii) the run time overhead in terms of percentage of CPU time. While the former translates to packets waiting at the input buffers, the latter implies a reduction in the sustainable packet-rate.

Adaptation routine time Results on the completion time of the proposed adaptation routines are shown in Fig.6. Both strategies in Method 1, *Greedy* and *LFS*, have a worst-case (adaptation minimizes per-packet processing time) execution time in $O(k^2)$, with k being the number of Monitoring States. However, even for a large value of k, *e.g.* $k = 64$, the routine can still run to completion within a short period, in the range of $10 \mu s$, resulting in only 100/200 packets temporary stopped at the input queue for 10 Gbps (by far below the packet capture buffer size). In the case of more light-weight adaptations, which reduce the processing time by 25%, a gap between the two strategies arises, as *LSF* operates in a more selective way. For Method 2, the worst-case time increases linearly with the number of controller queries, and a completion time as small as $10 \mu s$ is achieved for 1024 queries.

Run time overhead The overhead is dominated by the time for counting the number of packets that have been processed, by the Monitoring States in the case of Method 1, or for controller queries in Method 2. Results are shown in Table III, where the consumed CPU time is expressed as a percentage of the $10ms$ time-window, for the worst case of 10 Gbps small-packet traffic. While an increasing trend can be observed in the results, the overhead is generally low, even for Method 2 where this is in the range of 1% for the maximum number of queries (4096). The run time estimation of P also incurs some overhead, which is proportional to parameter K, *i.e.*, the number of T_r samples per time-window. This overhead is equivalent to 1% of a 10 ms time-window for $K = 10^3$. However, this cost can be drastically reduced by precomputing offline all necessary values of the CDF for the estimation of P. With this improvement, we obtain no more than 0.2% additional overhead for $K = 10^3$.

VII. Related work

Generating accurate and fine-grained monitoring information at a reduced cost is a critical task in today's networks, especially in resource constrained environments. A number of recent proposals [27][26][5] have focused on the development of adaptive monitoring frameworks with the objective of supporting measurements under dynamic traffic patterns and resource availability. A novel adaptive flow counting approach was introduced in [27] to enable anomaly detection with low overhead. Dynamic resource allocation solutions for traffic monitoring have been proposed in [26] and [5] based on OpenFlow counters and sketch-based measurements, respectively. Our approach also reconfigures monitoring parameters at run time to achieve efficient resource usage, but in contrast to the aforementioned solutions it focuses on software deployments.

With the advent of packet processing on commodity hardware, previous efforts such as [18][19] investigated how to take advantage of multi-core architectures to minimize the packet processing times for sophisticated monitoring tasks. While [18] relies on RSS and parallel threads to analyze multi-Gbps traffic, [19] uses multiple cores with the support of GPUs to perform complex intrusion detection operations. In addition, the recent packet-rate increase at the NIC raises new challenges, especially with respect to zero-loss guarantees under jitter in packet processing and unbalanced or unexpected traffic bursts. In contrast to our solution, existing approaches mainly address these challenges by enhancing either the packet capture or the packet scheduling. In [17], the authors propose to temporary store traffic in large buffers (1GB), which improves resilience at the cost of additional resource usage. The approach in [14] uses adaptive scheduling to mitigate performance drops due to resource contention.

In a similar fashion to our solution, the methods presented in [11] and [4] also perform reconfigurations directly on the monitoring pipeline. In [11] the authors propose to adjust the size of the monitoring data-structure according to changes in the traffic properties. This can, however, incur significant time overhead for large flow-tables (structure size). The adaptive approach in [4] monitors only flows whose size exceeds a dynamic threshold, so as to handle DoS attacks. While in [4] adaptations affect only new flows, reconfigurations in our work are applied to all operations in the monitoring process, responding thus to a wider range of emerging conditions.

VIII. Conclusion

Traffic monitoring on commodity hardware can starve packets at the packet capture buffers and thus leads to packet loss when changes in the operating conditions create bottlenecks. In this paper we proposed an adaptive approach with which the operations of the monitoring process are timely reconfigured under such conditions, so as to ensure lossless packet processing. We showed that considerable benefits in terms of packet loss reductions can be achieved under various conditions such as packet rate spikes, concurrency-induced performance degradation, and changes in traffic skew. Moreover, we showed that our solution can compute new monitoring configurations every 10 ms, without requiring additional processor cores and with minimal CPU-time overhead ($\approx 1\%$), even for 10 Gbps traffic of small packets. In the future, we plan to address the more complex case of monitoring operating in a chain of network functions on the same core. This would impose new requirements on the available resource estimation since the CPU-time is shared between multiple tasks. We will also enhance our solution to cope with sub-millisecond packet bursts that can arise at different layers of the server's network stack.

Acknowledgment

This research was funded by the UK EPSRC KCN project (EP/L026120/1), web: https://www.ee.ucl.ac.uk/kcn-project/.

References

[1] V. Sekar, M. K Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proc. ACM IMC*, Melbourne, Australia, Nov. 2010, pp 328-341.

[2] L. Hendriks, R. Schmidt, R. Sadre, J. Bezerra and A. Pras. Assessing the quality of flow measurements from OpenFlow devices. In *Proc. TMA*, Louvain La Neuve, Belgium, Apr. 2016.

[3] J. C. Mogul et al. Devoflow: cost-effective flow management for high performance enterprise networks. In *Proc. ACM Hotnets*, Monterey, CA, USA, Oct. 2010.

[4] M. Moshref, M. Yu, R. Govindan, A. Vahdat. Trumpet: timely and precise triggers in data centers. In *Proc. ACM SIGCOMM*, Florianopolis, Brasil, Aug. 2016, pp 129-143.

[5] M. Moshref, M. Yu, R. Govindan, A. Vahdat. SCREAM: sketch resource allocation for software-defined measurement. *Proc. ACM CoNEXT*, Heidelberg, Germany, Dec. 2015.

[6] Z. Liu et al. One sketch to rule them all: rethinking network flow monitoring with UnivMon. In *Proc. ACM SIGCOMM*, Florianopolis, Brasil, Aug. 2016, pp 101-114.

[7] M. Ghasemi, T. Benson, J. Rexford. Dapper: data plane performance diagnosis of TCP. In *Proc. ACM SOSR*, Santa Clara, CA, USA, Apr. 2017, pp. 61-74.

[8] L. Rizzo. NETMAP: A novel framework for fast packet I/O. in *Proc. Usenix ATC*, Boston, MA, USA, Jun. 2012, pp. 1?9.

[9] DPDK. Available: http://dpdk.org/.

[10] Receive Side Scaling, Microsoft, Redmond, WA, USA, Feb. 15, 2015. Available: http://msdn.microsoft.com/en-us/library/windows/ hardware/ff567236(v=vs.85).aspx

[11] O. Alipourfard, M. Moshredf, M. Yu. Re-evaluating measurement algorithms in software. In *Proc. ACM Hotnets*, Philadelphia, PA, USA, Nov. 2015.

[12] A. Metwally, D. Agrawal, A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. ICDT,* Edinburgh, UK, Jan. 2005.

[13] M. Yu et al. software defined traffic seasurement with OpenSketch. In *Proc. USENIX NSDI*, Lombard, IL, USA, pp. 29-42, Apr. 2013.

[14] M. Dobrescu, K. Argyraki, S. Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Proc. USENIX NSDI*, San Jose, CA, USA, Apr. 2012, pp 11-24.

[15] B. Atikoglu et al. Workload analysis of a large-scale key-value store. In *Proc. ACM Sigmetrics*, London, UK, Jun. 2012, pp. 53-64.

[16] Y. Chen, R. Griffith, J. Liu, R. H. Katz, A. D. Joseph. Understanding TCP Incast throughput collapse in datacenter. In *Proc. ACM WREN*, Barcelona, Spain, Aug. 2009, pp. 73-82.

[17] M. Trevisan, A. Finamore, M. Mellia, M. Munafo, D. Rossi. Traffic analysis with off-the-shelf hardware: challenges and lessons learned. In IEEE Communications Magazine, Vol 55, Mar 2017, pp. 163-169.

[18] F. Fusco, L. Neri. High speed network traffic analysis with commodity multi-core systems. In *Proc. ACM IMC*, Melbourne, Australia, Nov. 2010, pp.218-24.

[19] M Jamshed et al. Kargus: a highly-scalable software-based intrusion detection system. In *Proc. ACM CCS*, Raleigh, NC, USA, Oct. 2012, pp. 317-328.

[20] F. Guo, Y. Solihin. An Analytical Model for Cache Replacement Policy Performance. In *Proc. ACM Sigmetrics*, Saint Malo, France, June 2006, pp. 228-239.

[21] D. Tam, R. Azimi, L. Soares, M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proc. APLOPS*, Washington DC, USA, Mar 2009, pp. 121-132.

[22] R. West, P. Zaroo, C. Waldspurger, X. Zhang. Online cache modeling for commodity multicore processors. In *Proc. ACM SIGOPS Operating System Review*, vol 44, Dec. 2010, pp.19-29.

[23] L. Zhao et al. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In Proceedings of the conference on Parallel architectures and compilation techniques (PACT), 2007.

[24] G. Bianchi et al. Open Packet Processor: a programmable architecture for wire speed platform-indipendent stateful in-network processing. Available: https://arxiv.org/pdf/1605.01977.pdf, 2016

[25] A. Roy et al. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM*, London, UK, Aug 2015, pp. 123-137.

[26] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Dream: dynamic resource allocation for software-defined measurement. In *Proc. ACM SIGCOMM*, Chicago, IL, USA, Aug. 2014, pp. 419-430.

[27] T. Zhang. An adaptive flow counting method for anomaly detection in SDN. In *Proc. ACM CoNEXT*, Santa Barbara, CA, USA, Dec. 2013, pp. 25-30.