Accuracy-Aware Adaptive Traffic Monitoring for Software Dataplanes

Gioacchino Tangari, Marinos Charalambides, Daphne Tuncer, and George Pavlou

***Abstract*—Network operators have recently been developing multi-Gbps traffic monitoring tools on commodity hardware, as part of the packet-processing pipelines realizing software dataplanes. These solutions allow the execution of sophisticated per-packet monitoring using the processing power available on servers. Although advances in packet capture have enabled the interception of packets 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 monitoring framework, MONA, which is resilient to bottlenecks while maintaining the accuracy of monitoring reports above a user-specified threshold. MONA dynamically reduces the measurement task sets under adverse conditions, and reconfigures them to recover potential accuracy degradations. To quantify the monitoring accuracy at run time, MONA adopts a novel task-independent technique that generates accuracy estimates according to recently observed traffic characteristics. With a prototype implementation based on a generic packet-processing pipeline, and using well-known measurements tasks, we show that MONA achieves lossless traffic monitoring for a wide range of conditions, significantly enhances the level of monitoring accuracy, and performs adaptations at the time scale of milliseconds with limited overhead.**

***Index Terms*—Network monitoring; Software packet processing; Dynamic resource allocation**

I. Introduction

Recent advances in network management allow for dynamic 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 satisfied by existing techniques such as SNMP, packet sampling tools [2], or OpenFlow counters [3][4], given their 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 sampling or passive trace analysis, thus enabling timely reports of fine granularity [5]. In particular, 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* [42] [43]. This allows for enhanced flexibility at a low cost since traffic monitoring can exploit the processing power of servers to execute complex per-packet measurements.

Gioacchino Tangari, Marinos Charalambides, and George Pavlou are with the Department of Electronic and Electrical Engineering, University College London, London WC1E 7JE, UK. Daphne Tuncer is with the Department of Computing, Imperial College London, London SW7 2AZ, UK.

Achieving lossless packet processing while performing traffic monitoring in software 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 [5]) while performing advanced measurement tasks on a per-packet basis. Recent packet capture engines [10][9], hardware technologies such as Receive-Side Scaling (RSS)[11], and multi-core packet scheduling architectures [19][20] allow to cope well with packet capture at wire-speed. However, short-lived bottlenecks can still arise in the monitoring process, leading to potential loss of packets in the input buffer(s). This occurs when unbalanced packet rate spikes, affecting one or a subset of CPU cores, compress the available per-packet time, or when variations of traffic skew [12], or concurrent access to shared server resources [15] inflate the packet-processing latencies.

Our previous work in [1] proposed an adaptive traffic monitoring approach for software dataplanes, which responsively reconfigures the measurement operations in the face of bottlenecks so as to avoid packet loss. The main idea of [1] is to dynamically restrict the sets of monitoring operations for portions of the input packet stream, so that all packets can be inspected in time. This involves (i) frequent estimations of the available processing time, coupled with extensive offline analysis of the different per-packet latencies involved in the monitoring process, and (ii) timely reduction of the monitoring operation sets for portions of the active flows' population.

In this paper we extend the approach in [1] by proposing MONA (Adaptive **MON**itoring with **A**ccuracy-Awareness), a traffic monitoring framework based on software packet-processing, which is resilient to bottlenecks while keeping the *accuracy* of monitoring reports above a user-defined threshold. Jointly achieving zero packet loss and accurate reporting is a hard problem. In particular, it is difficult to know the impact of monitoring reconfigurations on the reports' accuracy *a-priori* [6][28], as this depends on traffic characteristics and on the monitoring operations logic.

MONA overcomes these issues by decoupling the *adaptation* functionality proposed in [1] from *accuracy control*. The latter progressively redistributes subsets of the active traffic flows between the *measurement tasks* running in the system, so that monitoring reports can be generated at the desired accuracy. To quantify accuracy degradations, MONA estimates at run time how many events (*e.g.,* heavy hitters, traffic bursts) remain undetected *after* the measurements sets

have been reduced in part of the flows. This is obtained through a novel, task-independent, technique which ensures high levels of confidence by computing estimates according to recently observed traffic characteristics.

MONA is not the first design where monitoring is dynamically configured to achieve accuracy goals, but it is the first that is fully tailored to a software dataplane. Existing solutions rely on approximate measurement techniques such as sketches [6][30][36] and top-k counting [39][13], mainly geared towards reducing memory usage, while for software packet-processing the stringent constraint is on CPU-time. In contrast to these approaches, MONA operates with simple hash-tables [5][12][38], with enough space to store (error-free) results for all active flows. This not only shifts the focus of the design from memory to CPU-time consumption, but also allows for more heterogeneous traffic analysis, *e.g.,* beyond volume and connectivity-based results of sketches [34].

In [1] we showed that the *adaptation* functionality reduces the risk of packet loss under variations of traffic rate/skew, or due to concurrent access to memory and processor caches. Compared to [1], in this paper we further investigate the benefits of MONA focusing on the monitoring report accuracy. To this end, we implement MONA along with a set of widely-used measurement tasks that adopt the same reporting period used in [5] ($10ms$). Our evaluation shows a general improvement on the accuracy level. In particular, MONA enhances the monitoring task accuracy – in terms of the task *satisfaction* metric proposed in [28] – by a factor of 2 compared to the traditional *static* monitoring approach, and by a factor of 3 compared to the initial adaptive solution in [1]. Finally, we demonstrate that MONA can operate in short time-scales while incurring only a small CPU-time overhead (≈1-2%).

In summary, the contributions of this paper beyond [1] are the following:

- A novel design introducing adaptive monitoring functionalities for software-packet processing, which involves online estimation techniques coupled with timely reconfigurations of the measurement operation sets.
- A task-generic monitoring accuracy estimation technique, which guarantees high levels of confidence by providing estimates tailored to recently-observed traffic characteristics.
- An extensive evaluation of MONA focusing on throughput and resilience in the face of bottlenecks, accuracy performance of representative measurement tasks, and computational overhead.

The remainder of this paper is organized as follows. We provide background information on software-based traffic monitoring in Sec.II. We describe our proposal 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. Traffic Monitoring in Software Dataplanes

The research community has 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.

Compared to monitoring operations in hardware switches, where memory availability (TCAM space [44] in particular) is the main shortage, traffic measurements on commodity hardware are constrained by the CPU-time and the *working set*, *i.e.*, the data most frequently accessed for the measurements [12]. This clearly reflects on the design choices for such monitoring tools. Instead of using more complex measurement techniques like *heap-based* [13] solutions and *sketches* [35][7][14][34], 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 [12][5].

A. Measurement Tasks

We target well-known measurement tasks that monitor the packets stream and collect traffic statistics over short time intervals (*e.g.,* $10ms$), called *time-windows*. Each task analyzes the packets and collects per-flow results in the flow (hash) table. Flows are defined based on the packet's *5-tuple*[1]. At the end of a window, each task generates a report containing all the *events* found in the flows it processed. We focus on four measurement tasks extensively studied in the literature [28][6][34][40][41][5].

Heavy Hitter detection (**HH**): discovers traffic aggregates exceeding a bytes threshold, where each aggregate is the sum of all flows with same source IP.

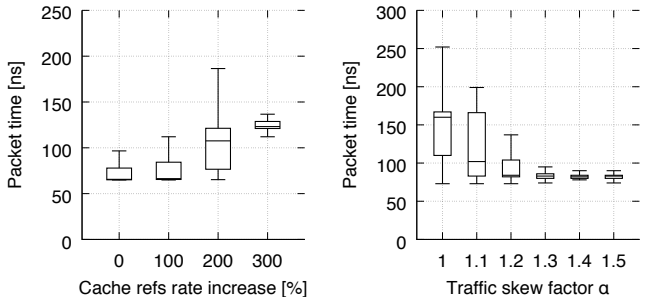
Bursty flow detection (**Bursty**): checks if at least $x\%$ of the packets of a flow arrived in bursts, *i.e.,* with an interarrival time below y milliseconds. If so, the flow is tagged as *bursty*.

Latency Change detection (**LatChange**): checks if the current round-trip-time (RTT) of a flow falls outside the interval $[mean(rtt) - \beta \cdot stddev(rtt), mean(rtt) + \beta \cdot stddev(rtt)]$, where β is a small integer value. If so, it increases the count of latency changes for that flow.

ReTransmission detection (**RTx**): counts for each flow the number of retransmissions (repeated acknowledgment/sequence numbers).

A wide range of analyses can be performed with the statistics collected by these tasks. Heavy hitters are used for anomaly detection and for supporting load balancing decisions. Bursty flows serve the diagnosis of congestion, while retransmissions can reveal reachability problems between two virtual machines (VM) or servers. The results can also be combined to detect network misbehaviors or to guide network management decisions, for traffic engineering or VM migration, for instance. An operator can identify TCP flows with significant loss (high retransmissions) and correlate them with heavy hitters in order to detect short-lived congestions [17]. Alternatively, LatChange can be used to track unresponsive servers and Bursty results are analyzed to check if latency changes are due to bursts (*e.g.,* spikes of requests).

[1] 5-tuple fields include source and destination IP, source and destination port, and protocol



(a) *vs.* cache references rate (b) *vs.* traffic skew

Fig. 1: Packet processing time

B. Analysis of Potential Bottlenecks

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 [5]), achieving zero packet loss and supporting diverse, sophisticated forms of analysis. Collectively meeting 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, events such as fast variation of user demand [16], sub-second congestion [17], or DoS attacks [5] can result to traffic rate spikes affecting one or more cores, even for deployments with multi-queue packet capture (such as RSS). Variations of the input packet rate affects the monitoring process. Intuitively, if the rate increases by $x\%$, the available time for processing each packet drops by $x\%$, or more if additional overheads are included for packet acquisition.

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 [15] 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 [15]. 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 in a higher miss ratio. As depicted in Fig.1a[2], the increase of cache reference rate on cores 2 to n (initially ≈ 0) can double the per-packet latency.

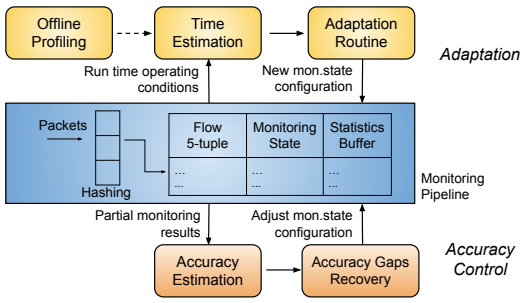


Fig. 2: Overview of MONA

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.[3] As shown in Fig.1b[2], reductions of the skew factor α can double the per-packet latency and generally make it less predictable.

III. MONA

Bottlenecks in the monitoring process, resulting from the aforementioned conditions, translate into longer queues in the packet capture stack, which lead 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 [18], enough to absorb only less than one millisecond of traffic at 10 Gbps. To ensure resilience to potential bottlenecks, the operator can either count on resource overprovisioning, or restrict the available monitoring tasks to a minimal set, *e.g.*, packet and byte counting only. However, the former approach violates the requirement of only devoting a limited amount of resources for monitoring and the latter penalizes the granularity and expressivity of monitoring reports.

To overcome these limitations, we introduce MONA, a traffic monitoring framework for software dataplanes, that dynamically configures the measurement operations sets based on emerging conditions. From a logical view, MONA is the combination of two functions. The first function, referred to as *Adaptation*, is aimed at *lossless* traffic monitoring, *i.e.,* it adjusts the monitoring process so that all packets are processed in time. This is achieved through (i) online detection of changes in the operating conditions, and (ii) timely reconfigurations (in $10ms$) of the monitoring operations on part of the input packet stream, to obtain more *light-weight* processing under strict time constraints. In contrast to existing approaches that use multi-core packet scheduling [19][20], MONA tackles the challenge of lossless traffic monitoring from a different angle, by realizing adaptations in the monitoring process itself, at the level of a single CPU core.

[2]The server hardware and configurations used are detailed in Sec. VI.
[3]We use a Zipf distribution and 10^5 flows

Reductions of the packet-processing time are achieved by preventing certain measurement tasks to execute for subsets of the active flow population. For example, by moving the active flows from a configuration where all tasks presented in Sec.II-A are executed (approx. $200ns$ per packet consumption), to one where only *HH* and *RTx* are active (approx. $100ns$ per packet consumption), MONA can handle a packet rate increase from 5 to $10Mpps$. Adaptations of the monitoring operations can however result in missed *events*, *e.g.* undetected bursty flows, which penalizes the accuracy of the monitoring reports. To deal with accuracy degradations, MONA executes a second function, referred to as *Accuracy Control*, that re-adjusts the flow allocation after adaptations have been executed to ensure that a user-specified level of accuracy is satisfied for all tasks. This is achieved by (i) tracking the monitoring report accuracy at run-time by estimating the number of events *missing* for each task, and (ii) recovering the identified accuracy gaps by iteratively re-allocating flows so as to meet the desired accuracy objective for all tasks.

MONA Design An overview of MONA is shown in Fig.2. The traffic monitoring process is modelled as a 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 tasks to perform for each packet of the flow. The reason for aggregating tasks in *monitoring states* is two-fold. It simplifies the design of applications where transitions between monitoring configurations are decided within the dataplane, *e.g,* based on the outcome of recent measurements and using a finite state machine [31][25]. It also enables settings in which multiple tasks share part of their data to save CPU cycles.

The Adaptation and Accuracy Control functions operate together with the monitoring pipeline, as part of the same process, to avoid additional resource usage (*i.e.*, more cores) as well as synchronization overheads. As shown in Fig.2, the two are however decoupled. This design choice is a consequence of a well-known problem in software-defined measurements [28][6]: determining *a priori* the effects of the set of flows processed by a task on the accuracy of its report is hard[4]. This difficulty is further amplified by the fact that the impact of adaptation decisions on the report accuracy differ between tasks as this depends on the task-to-monitoring state mapping. Even in the case where the same decision is applied to all tasks, they can each experience different degrees of accuracy degradations. Take the example of an adaptation that prevents both tasks *Bursty* and *HH* to run on a flow subset containing small but very bursty flows. While this causes a large fraction of bursty flows to be *missed* (low accuracy for *Bursty*), it results in only few missed heavy hitters (high accuracy for *HH*). These differences are not only due to traffic characteristics but also depend on the thresholds used by measurement tasks to trigger new events, *e.g.,* the bytes threshold for a heavy hitter. In MONA, Adaptation and Accuracy Control are executed separately on the same time-window basis. At the end of each time-window, the former estimates the available packet-processing time and determines the new assignment of flows to Monitoring States, while the latter estimates accuracy degradations and decides how to best recover them. Reconfigurations, if any, are enforced over the following window(s) based on the functions' joint outcome.

[4]An optimization-based approach could be used otherwise to process all packets in time while minimizing accuracy losses.

The design of MONA addresses three main challenges. It first ensures that the overhead incurred at run time is limited. All the involved procedures time-share the CPU with packet processing since they operate on the same core. As such, all operations in MONA are designed so that (i) they generate limited CPU-time overhead ($\approx 1ms$), and (ii) they all run to completion in short times (no more than $10\mu s$) to avoid starvation in the packet capture queue. In addition, it enables the operational conditions to be accurately estimated by only employing light-weight tools that provide high levels of confidence. Finally, it supports reactive monitoring reconfigurations by making sure that each component of the proposed solution works with time-windows as small as $10ms$.

IV. Monitoring Adaptation

As shown in Fig.2, the Adaptation function relies on a three-phase procedure. The first phase, *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. 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 and 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 window. The *Adaptation Routine* takes this value as input and, if the monitoring configuration exceeds the target time, it adjusts the set of measurement tasks for subsets of the flow-table entries to ensure all packets can be processed on time.

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 the operations in the current Monitoring State of the matching flow-entry (*i.e.,* the associated measurement tasks). The total expected packet time T_{pkt} can then be estimated based on the following equation:

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

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 [15][12], 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:

$$T_r = T_r^H \cdot P + T_r^M \cdot (1 - P) \quad (2)$$

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) [5]. Combining equations (1) and (2), the time per packet T_{pkt} is given by:

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

As observed from equation (3), T_{pkt} can be obtained based on the estimation of six variables. 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.

A. 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 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 CPU time rather than 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. This is due to the lower variability of T_p values of each monitoring task. Computing the coefficient of variation for *HH*, *RTx*, *Bursty*, and *LatChange*, we obtain 0.023, 0.013, 0.028, 0.045, respectively, which is much lower than T_r^M (0.225) and T_r^H (0.11). 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 access, such as TLB (Translation Lookaside Buffer) misses, whose impact also depends on the server hardware and kernel configuration (*e.g.*, memory page size).

[5]Retrieving flow-table entries from processor cache or main memory has a substantial impact on the per-packet time, due to the difference between T_r^H and T_r^M. For example, on a 2.7Ghz processor with 3MB L3 cache, T_r^H is on average 90ns, while the mean value of T_r^M is close to 200 ns. Assuming 250B packet size, this can result in a 10Gbps difference in throughput.

	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 I: Model selection for T_r^H and T_r^M

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.

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).

Using a standard fitting strategy, the second step selects 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 as it shows more consistent results compared to the maximum likelihood estimation for very large sample sizes. It is based on $-2\log(likelihood)$, hence 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 [5] could be used in conjunction with our solution. We discuss this aspect in detail in Section IV.C.

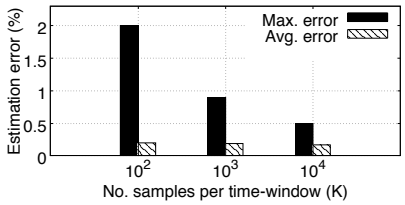


Fig. 3: Precision of P estimation

B. 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 λ_f, we use a simple strategy incurring negligible overhead that counts the flow-table insertions and divides 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, *i.e.*, each time a new packet burst is loaded by the packet acquisition library. In DPDK [10], for instance, this information can be retrieved using the counts in the rte_eth_stats and rte_ring API.

Several methods can be considered for estimating the value of the variable P. One possibility is to use an analytical model to predict the cache miss rate as proposed in [21]. However, this 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 [22]), while faster techniques, like the one presented in [23], rely on cache-related hardware counters that are restricted in current hardware [24].

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 sampling 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 r denote the vector of results with each element $r(i)$ equal to:

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

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

Algorithm 1: Adaptation Routine

1: **procedure** COMPUTENEWCONFIGURATION (T_p^{target},$\{n_i\}$,$\{s_{i-1}\}$)
2: $j = 0$
3: Compute avg processing time: $T^j = \sum_{i=1}^{k} n_i T_p^{s_i} / \sum_{i=1}^{k} n_i$
4: **while** ($T^j > T_p^{target}$) **do:**
5: **for each** $i \in (1, .., k)$ **do:** Shift all flows in s_i to s_{i-1}
6: j++
7: Update avg processing time: $T^j = \sum_{i=1}^{k} n_i T_p^{s_{i-1}} / \sum_{i=1}^{k} n_i$
8: Compute residual shift x: $x = (T^{j-1} - T_p^{target})/(T^{j-1} - T^j)$
9: **return** adaptation decision (j, x)

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. 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 $r(i)$ 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. For $K = 10^3$ the error is around 5%, whereas our method achieves $< 1\%$.

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:

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

C. Adaptation Routine

In case a monitoring configuration exceeds the target time T_p^{target}, the role of the Adaptation Routine is to decide which measurement task(s) should be avoided in the next time window, and for which flows. To this end, available knowledge on the monitoring pipeline can be exploited, such as the current average processing time and the time estimations described in Sec.IV. Based on this information, new monitoring configurations that satisfy the available time constraint can be derived and enforced in the next time window.

Ideally, a new configuration should be generated by a convex optimization that assigns each flow entry to a specific monitoring state, so as to maximize some objective in terms of monitoring accuracy. In practice, this is not viable for two reasons. The first is the overhead associated with solving such a problem per flow entry, and the second is that the impact of reconfigurations on the monitoring accuracy is not known *a priori*.

To overcome these limitations, the proposed Adaptation Routine takes a different path. Instead of dealing with individual 5-tuples or packets, it operates at the granularity of Monitoring States, as each one maps to a different subset of the flow-table. In addition, it follows a probabilistic approach, where random portions of the flow-table are re-allocated to more light-weight states such that T_p^{target} is achieved. These two features allow our solution to generate a new monitoring configuration within a few microseconds.

The pseudocode of the proposed approach is shown in Alg.1. When invoked at the end of a time-window, it takes

Fig. 4: Monitoring states graph

as input the count of packets processed at each Monitoring State s_i in the last time-window, and a graph mapping each s_i to a less time-consuming state s_{i-1} is generated. An example of such a graph is shown in Fig.4. The way this can be derived depends on the logic of the monitoring process. For example, in scenarios like [25][31], each Monitoring State incorporates the necessary logic in its code for transitions to other states once specific conditions on flow-entry statistics are met. s_{i-1} is therefore obtained for each s_i by backtracking in the state machine until a more light-weight monitoring state is found.

Our algorithm initially calculates the current average processing time $T^0 = (\sum_{i=1}^{k} m_i T_p^{s_i}) / \sum_{i=1}^{k} m_i$, where m_i is the number of packets in state s_i during the last time-window. Then, it iteratively re-allocates flow-entries to more light-weight Monitoring States. At each iteration j it computes the average processing time if all flows were forced to their previous Monitoring State, *i.e.*, from s_i to s_{i-1}. If this value, T^j, exceeds T_p^{target}, a new iteration is executed. Otherwise, the procedure takes the monitoring configuration for T^{j-1} and forces only a portion x of the flow-table to an additional step-back in the Monitoring State set, where $x = (T^{j-1} - T_p^{target})/(T^{j-1} - T^j)$. In practice, x is the ratio of flows to be further shifted so that the average processing time can match T_p^{target}.

When the next time-window starts, the new configuration is applied using the hash of the first *new* packet for each flow-entry, which ensures that x is a (pseudo)random portion of the flow-table. By comparing the hash with x (using a modulo operation), it decides to update the flow Monitoring State to j or $j-1$ steps back.

V. Monitoring Accuracy Control

While timely adaptations prevent packets from being dropped in the monitoring pipeline, they can penalize the accuracy of the reports computed by each measurement task, as explained in Section III. In this section, we present a Monitoring Accuracy Control function that aims at re-adjusting the flow allocation *after* adaptations have been executed, so that a global accuracy objective (maintain accuracy above a threshold) can be satisfied for all tasks. Its design addresses two non-trivial challenges. It first quantifies the effect of adaptations on the monitoring report accuracy. Given the lack of ground-truth information for the monitoring results, this is achieved using a *task-generic* solution that generates accuracy estimates at run-time; prior work *e.g.* [28][6] is limited to employing ad-hoc estimation techniques for each task. Based on the output of the online estimation procedure, an accuracy gap recovery process is then executed to re-adjust the flow allocation and alleviate accuracy degradation.

A. Online Accuracy Estimation

The objective of the Online Accuracy Estimation procedure is to generate accuracy estimates at run-time based on partial monitoring results. We quantify the accuracy of the reports computed by a measurement task according to the value of the *Recall* [35][39][28][6], *i.e.,* the ratio between the number of events identified by the task (*e.g.,* number of heavy hitters, bursty-flows, *etc.*) and the total number of events in the traffic. This metric is well-aligned with the logic of the adaptations: the accuracy degradation is in terms of *false negatives* (*e.g.*, missed heavy-hitters) rather than *false positives*. We define $Recall_{iw}$ as the recall of monitoring task i at time-window w:

$$Recall_{iw} = N_{iw}^{Found} / (N_{iw}^{Found} + N_{iw}^{Miss})$$

where N_{iw}^{Found} is the number of events identified by task i, while N_{iw}^{Miss} is the number of events *missing* with respect to the input traffic (*i.e.,* ground-truth), which is unknown. N_{iw}^{Miss} can be further expanded as:

$$N_{iw}^{Miss} = F_{iw}^{Miss} \cdot E[X_{iw}^{Miss}]$$

where F_{iw}^{M} is the number of *missing* flows for task i at time-window w (*i.e.*, flows for which task i has been dropped), which is measured from the output of the *Adaptation Routine*, and X_{iw}^{Miss} the number of events for a missing flow at w, which is unknown, *e.g.,* the number of retransmissions of a flow not processed by RTx at time-window w. The objective is to determine a reliable estimation of X_{iw}^{Miss}.

Formally, the estimation problem can be modeled using a decision-theoretic approach. In this work, we express the decisions as risk-taking decisions, where the risk is in generating poor estimations of X_{iw}^{Miss}. Let $\hat{x}^{Miss}$ represent a generic estimator for X_{iw}^{Miss}. We define a *Risk* function $R(\hat{x}^{Miss})$ associated with the choice of the estimator $\hat{x}^{Miss}$ as:

$$R(\hat{x}^{Miss}) = \sum_{l=0}^{\infty} L(x_l^{Miss}, \hat{x}^{Miss}) Prob(X_{iw}^{Miss} = x_l^{Miss}) \tag{5}$$

where the *Loss* function $L(X_{iw}^{Miss}, \hat{x}^{Miss})$ represents the loss incurred by replacing X_{iw}^{Miss} with $\hat{x}^{Miss}$. The best estimator of X_{iw}^{Miss} is the one with which $R(\hat{x}^{Miss})$ is minimized, *i.e.,*:

$$\hat{x}_{Best}^{Miss} = \underset{\hat{x}^{Miss}}{\text{argmin}}\ R(\hat{x}^{Miss}, L) \tag{6}$$

As can be observed, $\hat{x}_{Best}^{Miss}$ depends on the choice of the loss function L. In practice, different estimators can be used for X_{iw}^{Miss}. A possible approach is to replace X_{iw}^{Miss} with its worst-case value [28]. However, this solution is prone to significantly underestimating the accuracy if F_{iw}^{Miss} is large. In addition, it is not suitable for some tasks, *e.g.,* the worst-case number of missing flow retransmissions cannot be known.

In this paper we take a different approach and estimate X_{iw}^{Miss} based on the monitoring results observed over the most recent q time-windows. More specifically, let $X_{iw} = (x_{iw1}, x_{iw2}, ..., x_{iwm})$ be the vector of the results x_{iwj} of task i, for each flow j, at time-window w. x_{iwj} is equal to the number of events detected by task i for flow j at time-window

w, and undetermined if j is a missing flow for task i. We model the temporal dependence of X_{iw} over the most recent q time-windows as a moving average $MA(q)$ with order q [6]:

$$X_{iw} = \mu_i + z_w + \theta_1 z_{w-1} + ... + \theta_q z_{w-q}$$

where μ_i and θ_i are the mean and parameters of the model, respectively, and z_w ... z_{w-q} is Gaussian noise so that $E[X_{iw}] = \mu_i$.

The value of μ_i can easily be determined by noting that $E[X_{iw}] = E[\overline{X}_{iw}] = \mu_i$, where $\overline{X}_{iw}$ represents the mean of vector X_{iw}. In other words, to characterize the model, it is sufficient to track the values $\overline{X}_{i(w-q)}$ ... $\overline{X}_{iw}$ in the last q time-windows. Given that $\overline{X}_{iw}$ are unknown by definition, we estimate their value using the observed average monitoring results (*i.e.,* extracted from flows processed by task i), denoted as $\overline{x}^{Obs}_{i(w-q)}$ ... $\overline{x}^{Obs}_{iw}$. μ_i is then obtained by taking the weighted average of values $\overline{x}^{Obs}_{i(w-q)}$ ... $\overline{x}^{Obs}_{iw}$ as follows:

$$\mu_i = \frac{\sqrt{\frac{n_{iw}}{\sigma^2_{iw}}}\overline{x}^{Obs}_{iw} + \sqrt{\frac{n_{i(w-1)}}{\sigma^2_{i(w-1)}}}\overline{x}^{Obs}_{i(w-1)} + ... + \sqrt{\frac{n_{i(w-q)}}{\sigma^2_{i(w-q)}}}\overline{x}^{Obs}_{i(w-q)}}{\sqrt{\frac{n_{iw}}{\sigma^2_{iw}}} + \sqrt{\frac{n_{i(w-1)}}{\sigma^2_{i(w-1)}}} + ... + \sqrt{\frac{n_{i(w-q)}}{\sigma^2_{i(w-q)}}}}$$

where n_{iw} is the number of available results for task i at time-window w (*i.e.,* number of flows processed by task i) and σ^2_{iw} their sample variance. In practice, the more the results available at time w and the less their variance, the higher the contribution of time-window w.

Using the average values $\overline{x}^{Obs}_{iw}$ to compute the model parameters offers several advantages. The state that needs to be maintained for each monitoring task is drastically reduced and the computational overhead of the estimation is decreased.

We take into account the model of the evolution of $\overline{X}_{iw}$ values over the last q time-windows to set the loss function L, so as to make the accuracy estimation more or less conservative according to run-time conditions. In particular, these are assessed based on the probability of large accuracy estimation errors. If the probability is small, L is replaced with the *quadratic loss function*, widely-used in testing (*e.g.*, *least squares* techniques), that penalizes large errors more. In contrast, if the probability is large, L is defined so that only large deviations from X^{Miss}_{iw} are penalized. More specifically, the probability of large estimation errors depends on the variability of the distribution of the results of task i. This is captured by the coefficient of variation σ_i/μ_i of the mean values $\overline{X}_{iw}$ over the last q time-windows, where σ_i is the standard deviation of $\overline{X}_{iw}$ (measured based on $\overline{x}^{Obs}_{iw}$ values)[7]. We use the ratio $\sigma_i/\mu_i = 1$ as the threshold of high variability[8] and define the loss function L as follows:

$$L = \begin{cases} (X^{Miss}_{iw} - \hat{x}^{Miss})^2 & \frac{\sigma_i}{\mu_i} \leq 1 \\ \mathbb{I}[|X^{Miss}_{iw} - \hat{x}^{Miss}| > c] & \frac{\sigma_i}{\mu_i} > 1 \end{cases} \quad (7)$$

where c is an arbitrary value to decide when to penalize estimation errors in case $\frac{\sigma_i}{\mu_i} > 1$.

Based on (7), we determine the best estimator $\hat{x}^{Miss}$ of X^{Miss}_{iw} that depends on the value of σ_i/μ_i, as follows:

$$\hat{x}^{Miss} = \begin{cases} \mu_i & \frac{\sigma_i}{\mu_i} \leq 1 \\ \underset{\hat{x}^{Miss}}{\text{argmin}}\ Prob(|X^{Miss}_{iw} - \hat{x}^{Miss}| > c) & \frac{\sigma_i}{\mu_i} > 1 \end{cases} \quad (8)$$

The justification of (8) is provided in the supplementary material. In case $\frac{\sigma_i}{\mu_i} > 1$, the equation cannot be solved if no assumption is made on X^{Miss}_{iw}. Here we obtain an approximate bound for the Risk function by applying the Chebyshev's inequality to the probability of large estimation errors $Prob(|X^{Miss}_{iw} - \hat{x}^{Miss}| > c)$, *i.e.,* $Prob(|\overline{X}_{iw} - \mu_i| \geq \lambda\sigma_i) \leq \frac{1}{\lambda^2}$, from which we can further derive $Prob(\overline{X}_{iw} \geq \mu_i + \lambda\sigma_i) \leq \frac{1}{\lambda^2}$. This gives an estimator of X^{Miss}_{iw} equal to $\mu_i + \lambda\sigma_i$ in the case of large probability of large estimation errors, with guarantees that the Risk is no larger than $1/\lambda^2$. For instance, with $\lambda = 3$, the $Risk$ of poor estimation is bounded to 10%.

Summary: The estimation procedure for a generic task i operates as follows. At the end of time-window w, the number of available results n_{iw} and their sample variance σ^2_{iw} are updated. These values, together with those obtained in the $q-1$ previous time windows, are then used to compute μ_i. The sample mean of the available results $\overline{x}^{Obs}_{iw}$ are also extracted and used to update the coefficient of variation of $\overline{X}_{iw}$, *i.e.,* $\frac{\sigma_i}{\mu_i}$. Based on the value of $\frac{\sigma_i}{\mu_i}$, the loss function L is selected according to (7). The estimator is finally chosen as μ_i if $\frac{\sigma_i}{\mu_i} \leq 1$, and as $\mu_i + \lambda\sigma_i$ otherwise.

Performance evaluation and discussion: Fig.5a illustrates the performance of the proposed online accuracy estimation procedure in terms of *Accuracy Estimation Error* calculated as the absolute distance between the Estimated_Recall and Real_Recall (extracted from the ground-truth) normalized by the Real_Recall. Experiments are conducted using 100 ground-truth packet traces derived from [27], with dynamic rate in $[0Gbps, 10Gbps]$, and the four different monitoring tasks presented in Sec.II-A. To show the gain achieved by adapting $\hat{x}^{Miss}$ to the run-time conditions, the performance is also compared against two baselines approaches, one with an estimator always given by the mean (*mean*), and one only using the upper bound $\mu_i + \lambda\sigma_i$ (*u-bound*).

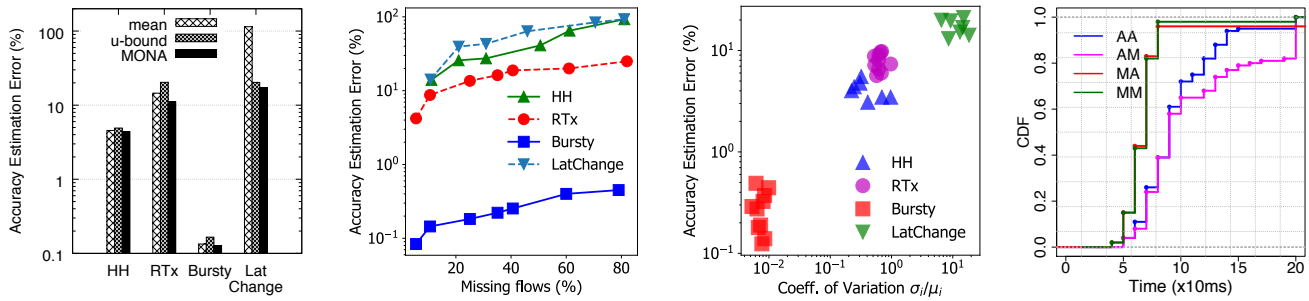
As can be observed, the estimation error is generally low, overall 8% on average, and never exceeds 20%. The highest error (around 17%) is obtained for *LatChange*, which is the task producing the least predictable monitoring results in our experiments. Compared to the two baseline cases, the proposed approach generally achieves lower error, with substantial gain in particular for *RTx* (2x reduction compared to *u-bound*) and *LatChange* (>5x reduction compared to *mean*).

The observed estimation error can be attributed to two main factors: *i)* the number of missing flows, *i.e.,* flows for which monitoring results are missing due to adaptations and/or bottlenecks, and *ii)* the variability of the monitoring results that can be quantified based on the coefficient of variation $\frac{\sigma_i}{\mu_i}$ of each monitoring task. The impact of the two factors is shown in Fig. 5b and 5c. As can be observed, for all tasks, the estimation error increases as the percentage of missing flows increases. The higher the percentage of missing flows, the less the confidence on the accuracy estimation, and the

[6] By default we use results from the last 10 time windows, so $q = 10$

[7] σ^2_i is the variance of the sample mean of X_{iw}

[8] This is a standard choice, based on the comparison with the exponential distribution



(a) Performance of monitoring accuracy estimation

(b) Accuracy estimation error vs missing monitoring results

(c) Accuracy estimation error vs monitoring result variability

(d) Recovery convergence time (A=additive, M=multiplicative)

Fig. 5: Accuracy Control

higher the average error. In contrast, the influence of the coefficient of variation is driven by the type of the monitoring tasks. As depicted in Fig. 5c, tasks with higher coefficients of variation (*i.e.,* LatChange and RTx) are associated with higher estimation errors. This effect can be imputed to how MONA's accuracy estimation operates. More specifically, the estimation procedure is designed such that a conservative estimator is selected when the results associated with a task manifest a high degree of statistical dispersion (high coefficient of variation), *i.e.,* the one providing an upper bound for the Risk function. This naturally favors under-estimation over over-estimation in situations of high uncertainty, where over-estimation can leave accuracy degradations unhandled.

In general, the task-generic nature of our solution largely compensates the impact of the introduced estimation errors, and provides key advantages compared to recent solutions for TCAM-based [28] and sketch-based [6] measurements, which define ad-hoc estimation methods for each monitoring task. In practice, our approach can be applied to all monitoring tasks whose output is a count (*e.g., how many retransmissions for flow x?*) or a classification (*e.g., is aggregate y a heavy-hitter?*). As shown in [5][28][6], tasks detecting network episodes or anomalies, or serving network management decisions, fall well in these categories. In addition, our solution can run on a limited time budget as it involves operations that can be executed at a low cost: all distributions are obtained by sampling and fast techniques can be used to compute mean and variance values, *e.g.,* [32].

B. Accuracy Gaps Recovery

An estimate, based on the value of the *Recall*, is generated by the Online Accuracy Estimation procedure for each monitoring task at the end of each time-window w. The estimates are further used by an Accuracy Gaps Recovery procedure to re-adjust the flow allocation so that accuracy degradation resulting from the *Adaption Routine* can be alleviated.

Let A_{iw} denote the accuracy estimate for task i at time-window w. A task is tagged as *poor* at time w if A_{iw} is below a threshold and as *rich* otherwise.[9] The objective of the Accuracy Gaps Recovery procedure is to recover the accuracy gap of a poor task by applying it to a larger set of flows and compensating the additional CPU time needed with controlled degradation of rich tasks. This is achieved using a *rebalancing* approach that reallocates subsets of flows to different *monitoring states* in an iterative fashion, with one iteration per time-window. Ideally, all accuracy gaps should be recovered within one time-window, but this is difficult to achieve in practice. Not only is the number of flows needed to meet an accuracy target not known *a priori*, it is also not possible to determine the maximum number of flows a task can *donate* while keeping the accuracy of its report above the desired threshold.

More specifically, at each iteration the algorithm presented in Alg.2 is executed. Let $A_w = (A_{1w}, A_{2w}, ..., A_{kw})$ denote the vector of estimates for each monitoring task $1, ..., k$ and M a binary $n \cdot k$ matrix, where n is the number of monitoring states and k the number of tasks, with the task/state *mapping* so that $M_{ij} = 1$ if state i includes task j, and 0 otherwise. In addition, let B be a $n \cdot n$ matrix indicating the transitions of flows between monitoring states due to previous adaptation decisions, as recorded in the current time-window. Each value B_{ij} is the ratio of flows that were previously moved by the *Adaptation Routine* from state i to j (with j being more light-weight than i) to save time.

Alg.2 initially selects, from the global set of monitoring states, a subset of *poor* states, containing one or more poor task(s), and a subset of *rich* states, in which all tasks have an accuracy higher than the desired threshold with a small headroom ϵ. The headroom is used to prevent rich states becoming poor after a single iteration. For each pair of states (s_{rich}, s_{poor}) a re-balancing action is taken based on a *step-size* parameter S. This involves shifting Δ^- flows from s_{rich} to the default state (*e.g.,* s_1 in Fig.4) and using the resulting gain in time to revert the monitoring adaptation for Δ^+ flows moved from s_{poor}. The value Δ^- is set to S normalized by the number of poor states, while Δ^+ is obtained from Δ^- by imposing the equilibrium condition *"constant CPU time consumption"*:

$$\Delta^-(t_s^{rich} - t_s^{default}) = \Delta^+(t_s^{poor} - E[\widetilde{t_s^{poor}}]) \qquad (9)$$

where the values t_s are the state execution times, and $E[\widetilde{t_s^{poor}}]$ is the expected execution time for flows moved away from s_{poor} by the *Adaptation Routine*, as indicated by B:

$$E[\widetilde{t_s^{poor}}] = \textstyle\sum_j B_{poor,j} t_s^j / \sum_j B_{poor,j}$$

The choice of the step-size S involves a trade-off between

[9]For simplicity, the same accuracy threshold was used for all tasks.

Algorithm 2: Recover Accuracy Gaps

function UPDATESTEPSIZE(x, S_x)
  Compute accuracy decrease $D = A_{x,w-1} - A_{x,w}$
  Update residual accuracy $H = A_{x,w} - threshold$
  if $D > H$ **then return** INCREASE(S_x)
  else return DECREASE(S_x)
function REBALANCEBYSTEP(s_{Rich}, s_{Poor}, S)
  Compute $\Delta^- = S/n_p$, where n_p number of poor states
  Retrieve $E[\widetilde{t_s^{Poor}}]$ from $T_{Poor,j}$, $j \in 1, .., n$
  Compute Δ^+ from equilibrium condition (9)
  return Δ^-, Δ^+
procedure RECOVERYGAPS(A_w, M, T_w)
  Find set of rich, poor states $\{s_{Rich}\}$, $\{s_{Poor}\}$ using A_w
  if $\{s_{Poor}\} == \emptyset$ or $\{s_{Rich}\} == \emptyset$ **then return**
  for each x in $\{s_{Rich}\}$ **do**:
    S_x = UPDATESTEPSIZE(x, S_x)
  for each (x, y) with $x \in \{s_{Rich}\}, y \in \{s_{Poor}\}$ **do**:
    REBALANCEBYSTEP(x, y, S)

stability and convergence time, which is a well-known challenge of any resource allocation algorithm. While using small steps may lead to high convergence times, big step sizes can make measurement tasks oscillate between *rich* and *poor* over consecutive time-windows. The best practice for setting the value of S is to use an increase/decrease policy [28]. In this work, we relate the change of S to the accuracy evolution of each *rich* state s_{rich}, with the objective to converge rapidly while preventing s_{rich} from dropping below the desired accuracy threshold. After each iteration, *i.e.,* at time-window *w+1*, the algorithm computes the decrease of s_{rich} accuracy $D = A_{rich,w+1} - A_{rich,w}$, as well as its residual accuracy $H = threshold - A_{rich,w+1}$. S is increased if $H > D$ and decreased otherwise.

We evaluate the impact of different standard increase-decrease policies on the convergence times of Alg.2. The results are depicted in Fig.5d for the same input traffic used in Fig.5a and initial S values in $[0.25\%-10\%]$ of the initial active flow set. The best performance in this scenario is achieved by the multiplicative increase-decrease policy (MM).

VI. Evaluation

We implemented MONA in the C 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 and then fetch packets with small bursts at run time so as to isolate the monitoring pipeline from the packet capture stack. One second of traffic is pre-loaded, which corresponds to approximately 1GB allocated memory for 10Gbps of traffic. The packet trace we use includes only TCP flows and is derived from recently published results on flow statistics in data centers [27]. In addition, packet retransmissions are injected in the trace using the Gilbert-Elliot model [33][8].

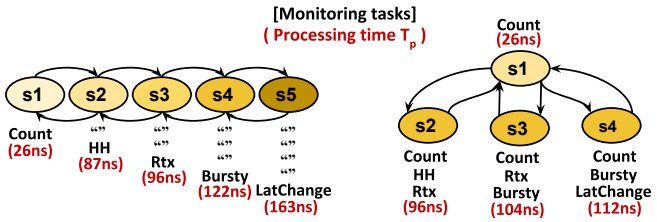


Fig. 6: Monitoring states Config1 (left) and Config2 (right)

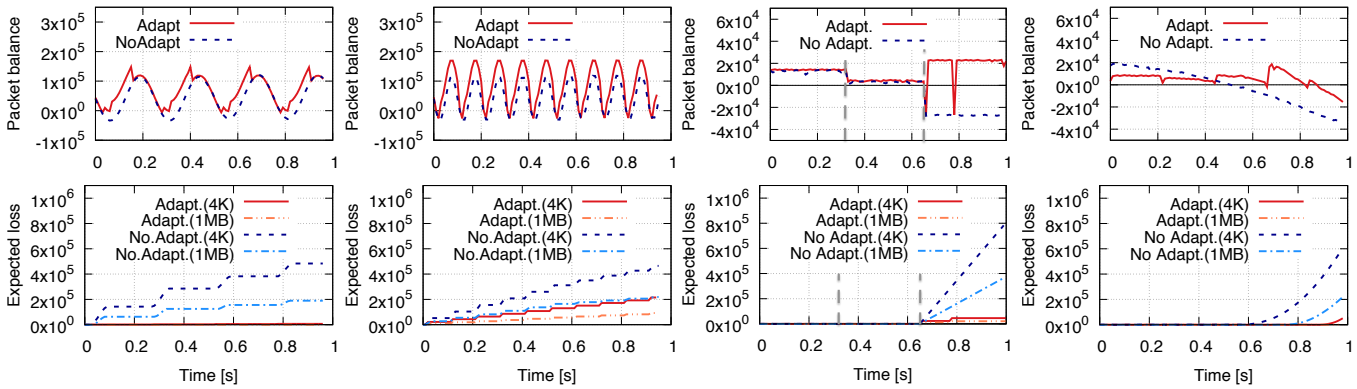
Measurement tasks setup The monitoring pipeline is based on the tasks described in Sec. II-A. For *HH*, we use a threshold of 5MB for a 10ms time-window. *Bursty* selects those flows for which at least 10% of packets come with inter-arrival below 1 ms. *LatChange* detects an anomaly when the RTT falls outside the range $[mean(rtt) - stddev(rtt), mean(rtt) + stddev(rtt)]$.

Monitoring states setup To experiment with the monitoring pipeline, we define two different monitoring state configurations, namely *Config.1* and *Config.2*. These are presented in Fig.6, along with their (estimated) processing times T_p. Both configurations include a default state s_1 where only simple packet counting (*Count*) is performed. In Config.1, additional tasks are progressively incorporated in each step of the monitoring states chain so that the depth of the traffic analysis increases with higher monitoring states. The process initially detects large traffic aggregates (s_2) and then starts monitoring the packet loss of these aggregates (s_3). For flows with a high loss, it incorporates burst detection in s_4 and finally looks for RTT changes in s_5. In Config.2 flows can be processed based on three possible monitoring states, inspired by the use-cases in [5]. The goal of s_2 is to identify the root cause of congestion by correlating lossy TCP flows with heavy hitters. State s_3 identifies loss as a result of bursty traffic, and s_4 detects server imbalance by collectively tracking latency changes and bursty flows. The two configurations are used to depict different examples of monitoring applications. While Config.1 provides a hierarchical model, with some tasks (*e.g., HH*) having higher priority compared to others (*e.g., LatChange*), Config.2 reproduces a "flat" configuration with three different monitoring objectives of equal importance.

The evaluation is conducted as follows. We first analyze the performance of adaptive traffic monitoring in terms of packet loss risk and adaptation responsiveness (Sec.VI.A). We then investigate the impact of monitoring adaptations and accuracy control on the measurement tasks (Sec.VI.B), explore the throughput limiting factors for MONA (Sec.VI.C) and extensively evaluate the overhead of the proposed solution (Sec.VI.D). Finally, we highlight differences with sketch-based measurement approaches by comparing MONA to SketchVisor [34] (Sec.VI.E). 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



(a) Rate variation ($T = 250ms$) (b) Rate variation ($T = 100ms$) (c) Shared resource contention (d) Traffic skew variation

Fig. 7: Packet balance and expected packet loss

risk of packet loss as a result of bottlenecks in the monitoring process. The first is *packet balance*, which indicates whether the number of packets in the trace can be processed during each time-window – a negative value signifies inability to 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) [18].

For these experiments we use *Config.1* and we initially assign $1/5$ of the flows to each state. The monitoring adaptation time-window is set to $10ms$. To study the performance of our solution under the emergence of bottlenecks we perform three types of experiments, which reproduce the three main conditions described in Sec. II.B.

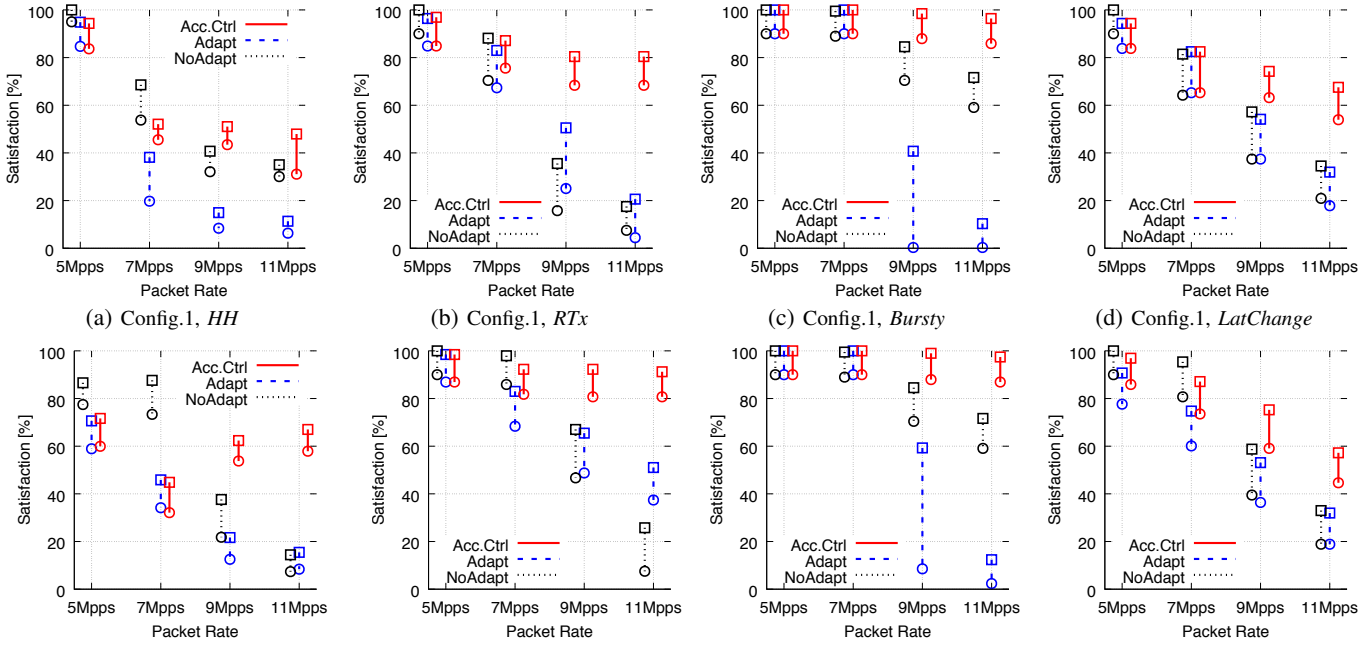
Traffic rate variations In these experiments we test our solution against multi-million packets per second (Mpps) variations of the input rate, which are realized by tuning the rate of exponential packet inter-arrivals in the trace. In particular, we evaluate the responsiveness of monitoring adaptations 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.7a and 7b, respectively. For $T = 250ms$, monitoring adaptations always provide a response to emerging bottlenecks in time, before packet loss occurs. In the case of $T = 100ms$, large 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. 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 proposed in [15]: 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 number of L3 cache references per second. As depicted in Fig.7c, 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.7c, 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 (see Sec.IV-B) 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.7d, 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 $\alpha \approx 1.25$.

B. Monitoring Report Accuracy

We now evaluate the impact of our solution on the monitoring report accuracy to investigate how the report quality is preserved under bottleneck conditions. To this end, we measure the real accuracy, in terms of *Recall*, of each task running in the system. This is obtained by comparing the



(a) Config.1, *HH* (b) Config.1, *RTx* (c) Config.1, *Bursty* (d) Config.1, *LatChange*

(e) Config.2, *HH* (f) Config.2, *RTx* (g) Config.2, *Bursty* (h) Config.2, *LatChange*

Fig. 8: Measurement task satisfaction (Median ○, Max □)

monitoring results of the system against the respective *ground-truth*, which is extracted from offline analysis of the traces.

To control the accuracy we use a threshold of 75% for all tasks, which is inline with the one used in [28]. For simplicity, traffic flows are assigned randomly to any available monitoring state with equal probabilities. Finally, to generate bottlenecks and thus trigger adaptations, we use a dynamic input packet rate in the range $[5, 11]$ Mpps, which is obtained by tuning packet inter-arrivals in the trace.

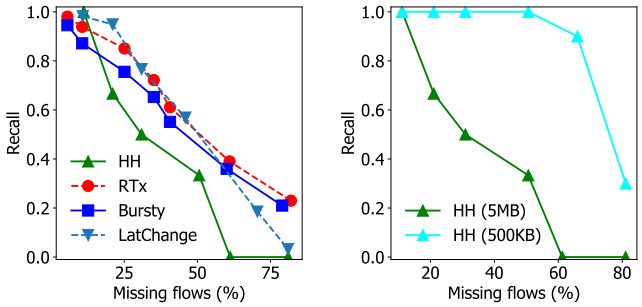
Fig.8 presents the monitoring accuracy results in terms of a *Satisfaction* metric similar to the one used in [6] and [28]. This represents the fraction of time a task has its *Recall* above the threshold. Intuitively, 100% satisfaction for all tasks means that the global accuracy goal is fully achieved. The minimum and median satisfaction is shown from a set of 100 experiments, for the two monitoring states configurations (Config.1, Config.2). As depicted in Fig.8, we compare the performance obtained with accuracy control (*Acc.Ctrl*) against two baselines: *No-Adapt* which is the standard setup without dynamic adaptations and accuracy control; and *Adapt* which executes adaptations without accuracy recovery in place.

As shown in Fig.8, both *Adapt* and *No-Adapt* incur serious accuracy degradations, with the satisfaction dropping even below 20% for some tasks. For the *Adapt* case this is due to accuracy-unaware adaptations, while in the *No-Adapt* case, the reduced satisfaction depends on the amount of packets never entering the monitoring pipeline, as they are starved in the buffers under increasing packet rates. In contrast, *Acc.Ctrl* drastically improves the report accuracy, raising the overall median satisfaction to 75%; 2x and 3x impovement compared to *No-Adapt* (38%) and *Adapt* (23%), respectively.

The satisfaction generally demonstrates similar trends for the two configurations. When comparing different measurement tasks, although *Acc.Ctrl* always achieves a considerable gain, the results show that the performance of MONA in terms of satisfaction depends on the monitoring task. For instance, in the case of Config.2 we observe the median satisfaction at the maximum packet rate oscillating between 60% and 100%. The reason of these differences is two-fold.

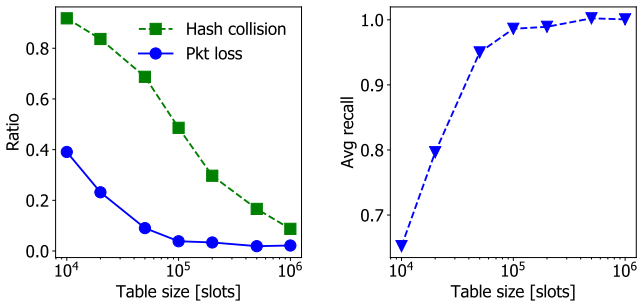
The accuracy estimation errors can have an influence on the satisfaction of a monitoring task. Considering *RTx* and *LatChange*, with errors $\approx 10\%$ and $\approx 20\%$, respectively (see Fig.5a), we obtain a satisfaction of 80/95% for the former and 70/60% for the latter. It is also evident that the very low estimation error for *Bursty* (less than 1% in Fig.5a) relates to its very high median satisfaction. In addition to this, the joint effect of tasks with under- and over-estimated accuracy can penalize the fairness of the Accuracy Gap recovery process as it may result in competition for resources. This issue specifically emerges in situations where under-estimation is calculated for rich tasks while the accuracy of poor tasks is over-estimated. In this case, rich tasks are less prone to redistribute their resources to the poor ones, preventing part of the resources to be reassigned when the accuracy falls below a threshold, and thus leading to lower satisfaction.

The different satisfaction ranges also reflect the different resource/accuracy trade-offs of different tasks. The amount of monitoring information losses due to missing flows is generally not the same for the different monitoring tasks, which affects the Recall and hence the satisfaction. Fig. 9a shows the effect of the percentage of missing flows on the Recall for each task in the form of *curves of diminishing returns* [28]. As can be observed, tasks are associated with curves depicting different slope and concavity properties. These properties cannot be known *a priori* as they not only depend on the characteristics of a task, *i.e.,* aggregation function used to trigger a new monitoring event (*e.g.,* SUM of 5-tuple flows by src IP address in the case of HH) and associated threshold



(a) All monitoring tasks (b) *HH* using different thresholds

Fig. 9: Curves of diminishing returns



(a) Hash collision and pkt loss ratio (b) Average task accuracy (recall)

Fig. 10: Impact of hash collisions

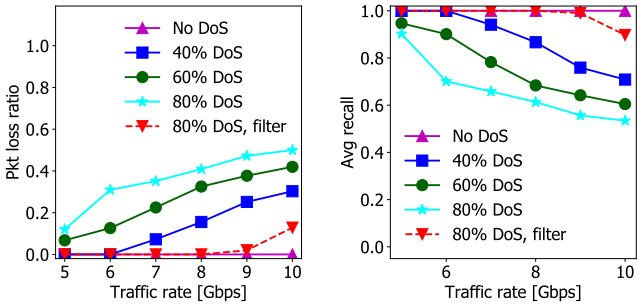
(*e.g.,* aggregate size threshold in Bytes for HH - see Fig. 9b), but also on the current traffic. This effect explains for instance the difference of *HH* compared to other tasks. When a high threshold is used (5Mb for 10ms), the concavity of the curve is such that few missing flows can translate to a high ratio of undetected heavy hitters (*i.e.,* low Recall) given that these events are evaluated based on the total size of multiple flows.

C. MONA Throughput Limiting Factors

In this section, we investigate throughput limiting factors not captured by MONA's adaptation logic, as opposed to the conditions discussed in Section VI.A. Specifically, we evaluate the effects of hash collisions and *uniform* traffic distributions.

Impact of hash collisions Hash table collisions in MONA are resolved through chaining with the use of additional linked lists. This prevents measurement data stored in the flow table to be corrupted or lost in case of collisions. However, high collision ratios result into larger per-packet processing times due to inflated table look up and flow insertion times, which may translate into increasing risk of packet starvation/loss. To address this issue, MONA keeps collision rate low by over-provisioning the flow table in terms of hash-table slots/buckets. Fig.10 shows the effects of collisions on packet loss probability and accuracy reductions (using the Recall metric) under *challenging* traffic conditions, *i.e.,* low skew ($\alpha = 1.1$), high speed (10Gbps), and 1000 new flow arrivals every 10 ms. By choosing an appropriate hash table size ($5 \cdot 10^5$ slots), MONA avoids both collision-induced packet loss (Fig.10a) and accuracy drops (Fig.10b). In our implementation, this bound corresponds to approx. 100 MB total memory consumption. While it is significantly more than what consumed by sketch-based approaches, *e.g.,* [34], this does not constitute a main limiting factor given the large memory availability in today's multi-core servers and the fact that packet-processing in software is mainly constrained by available CPU-time.

Impact of uniform traffic As discussed in Section VI.A, MONA can cope with much lower levels of traffic skew compared to a non adaptive monitoring designs. However, there are cases where traffic shows a "uniform" distribution (*i.e.,* with extremely low skewness), especially when denial of service attacks aim at exhausting the resources of the monitoring system [5]. Uniform traffic can produce bottlenecks in the monitoring pipeline due to high sparsity in data



(a) DoS-induced packet loss (b) Accuracy (recall) reductions

Fig. 11: Impact of DoS traffic

accesses ($P \approx 0$) and, more importantly, to increasing flow insertion rates. To evaluate the performance of MONA under uniform traffic, we apply a SYN flood (DoS) attack – a worst-case scenario since each new packet triggers a new flow insertion. In particular, we mix DoS traffic with legitimate traffic (from the trace in use) using different ratios. As shown in Fig.11, high percentages of DoS traffic induce significant packet loss (Fig.11a) especially for high traffic speeds, with average accuracy reductions reaching 40% (Fig.11b). This is because MONA's formulation of the available per-packet time assumes no more than 10% of packets triggering a new flow-insertion (Sec.IV.A). The effects of uniform traffic can be mitigated by applying the resilience mechanism proposed in Trumpet [5] in conjunction with MONA. This mechanism involves the matching of packets against an additional *filtering table*, a small buffer accessed before the "main" flow-table. The filtering table tracks a new flow while its size is below x, where x is a small user-defined threshold in bytes. If the flow exceeds x, it is tagged as *legitimate* and accepted in the main table, otherwise discarded (*i.e.,* in the case of malicious SYN flows). As depicted in Fig.11, adding this filter to MONA allows to tolerate high DoS ratios (*e.g.,* 80%). This can however sacrifice up to 10% of MONA throughput due to the additional overhead introduced by the filtering table, and it can also penalize the accuracy since flows smaller than x are discarded. The latter can be mitigated by activating the filter only when extremely adverse traffic conditions are detected or by dynamically updating x based on available (online) time estimations [5].

TABLE II: Total overhead (x = % of CPU time)

#Tasks	#States	#A.Flows	Traffic (bps/pktSize)	$Overhead < x$ $x = 1\%$	$x = 2\%$
10	10	1000	10G/64B	✓	✓
20	20	1000	10G/250B	✓	✓
40	20	1000	10G/250B	✗	✓
40	40	1000	10G/250B	✗	✗
40	40	500	10G/250B	✗	✓

D. Monitoring Adaptation Overhead

We evaluate the cost of our solution in terms of run time overhead. To this end, we consider (i) the execution time of the main procedures running at the end of a time window, and (ii) the additional CPU time consumed throughout a time-window for the estimations presented in Sec.IV-B,V-A and for applying adaptation and accuracy control decisions. The latter implies a reduction in the sustainable packet-rate, while the former also translates to spikes of packets waiting at the input buffers.

Adaptation execution time This time corresponds to the adaptation routine completion. As depicted in Fig.12a, the worst-case (*i.e.,* 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 to only 100-150 packets being temporarily held at the input queue for 10 Gbps (much below the packet capture buffer size).

Accuracy control execution time This time is dominated by the completion of the accuracy recovery procedure. As shown in Fig.12b, this increases linearly with the number of measurement tasks k ($O(k)$), with the slope depending on the number of monitoring states defined. For a large number of tasks (*e,g.,* 50) and using 10 different monitoring states, this time is kept within the range of $10\mu s$.

Estimation overhead This metric groups different overhead components: (i) the time for counting the number of packets processed according to each monitoring state; (ii) the online estimation of P (probability of fast flow-entry retrieval) and (iii) the time consumed to compute μ_i and the loss function L. Results are shown in Fig.12c, where the consumed CPU time is expressed as a percentage of the $10ms$ time-window. While an increasing trend can be observed, the overhead is generally low and it exceeds 1% only for a large number of monitoring states (*e.g.,* 64) with constant 10Gbps traffic of 64B packets.

Reconfiguration overhead Fig.12d shows the additional time required to apply reconfigurations of the monitoring pipeline due to the adaptation routine or the accuracy recovery procedure. This overhead linearly depends on the number of monitoring states. Also, it relates to the number of active flows in the $10ms$ time window since the reconfigurations are enforced only once per flow. We can observe that this overhead exceeds 1% only for the maximum level of flow concurrency (1000), and for a large number of monitoring states (> 40). It should be noted that the range considered here for the number of active flows matches the statistics reported in [26] and [27].

Overall, the total overhead is a function of the number of measurement tasks and monitoring states, the packet rate and size, and the flow concurrency. Table III summarizes the feasibility range of our solution given two maximum overhead constraints, 1% and 2% of the CPU time. As shown in the table, the total overhead exceeds 1% only under very large numbers of tasks and monitoring states, such as 40-40, and maximum traffic intensity. However, the requirement can be still met by relaxing the overhead constraint from 1% to 2%, which is acceptable, or by slightly reducing the hypothesis on traffic characteristics.

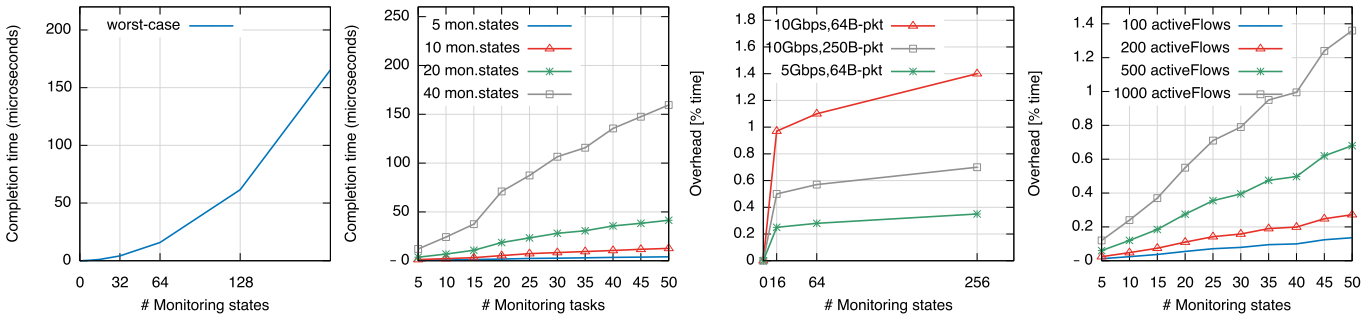
E. Comparison with a sketch-based approach

Recent research [34][45] has applied sketch-based measurements to software packet-processing pipelines in order to achieve high throughputs with reduced memory consumption. We evaluate here the extent to which MONA matches the performance of state-of-the art sketch-based traffic measurements. We compare against SketchVisor [34] due to its adaptive nature and performance-oriented design. SketchVisor augments sketch-based measurements in the software dataplane with a *fast path*, activated under high traffic load, which reduces the tracking of small flows to increase throughput. For the comparison, we restrict the measurement task set to HH detection only – this task is common to MONA and SketchVisor. This setup is particularly adverse for MONA: (i) HH detection is a perfect fit for SketchVisor fast-path as small flows contribute less to HHs; and (ii) executing a single monitoring task penalizes MONA as it exacerbates the impact of flow-entry retrievals (time T_r in Section IV) on the total per-packet time.

We run over 100 experiments and set the HH threshold to 0.05% of the input rate multiplied by the epoch length (10ms) as in [34]. Fig. 13a shows that SketchVisor achieves a high throughput (17.5 Gbps median). The performance of MONA is 25% lower in case of maximum accuracy, and 7% lower when meeting 100% task satisfaction. With regards to accuracy, MONA achieves maximum accuracy for traffic rates close to 10 Gbps (thus satisfying the goal of 10 Gbps traffic for a CPU core), while for rates above 15Gbps SketchVisor performs better. In general, the throughput reductions obtained by MONA are largely compensated by its key advantages in terms of adaptive traffic monitoring capability. While sketches-based approaches are mainly restricted to size-based (*e.g.*, HH detection) and cardinality-based (e.g., superspreader detection) measurements, MONA allows for a much larger range of monitoring operations including packet interarrival analysis (*e.g.*, Bursty) and TCP diagnosis (e.g., LatChange, RTx). In addition, MONA adaptations are not only triggered by traffic rate increases, but also follow shared-resource concurrency and traffic distribution. Overall, MONA provides a more flexible alternative for network measurements, at the cost of (limited) packet-processing efficiency reductions.

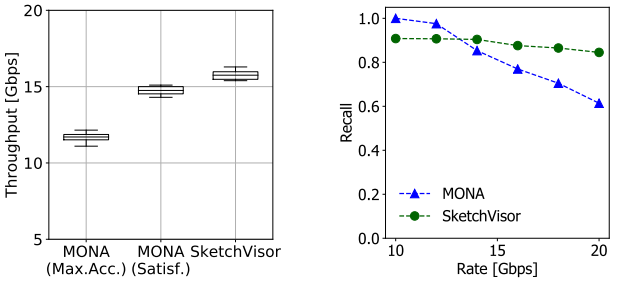
VII. Related work

A number of recent proposals [29][28][6] 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 [29] to enable anomaly detection with low overhead. Dynamic resource allocation solutions for traffic monitoring have been proposed in [28] and [6]



(a) Adaptation execution time (b) Accuracy control execution time (c) Estimation overhead (d) Reconfiguration overhead

Fig. 12: Overhead evaluation



(a) Throughput performance (b) Monitoring accuracy (recall)

Fig. 13: Comparison between MONA and SketchVisor

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 [19][20] investigated how to take advantage of multi-core architectures to minimize the packet processing times for sophisticated measurement tasks. While [19] relies on RSS and parallel threads to analyze multi-Gbps traffic, [20] 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 [18], the authors propose to temporary store traffic in large buffers (1GB), which improves resilience at the cost of additional resource usage. The approach in [15] uses adaptive scheduling to mitigate performance drops due to resource contention.

Orthogonal to scheduling and packet capture enhancements, recent solutions [34][36][12][5] address the problem of sustainable packet-processing in software by directly reconfiguring the monitoring process, as in the case of MONA. However, [34] and [36] focus on sketch-based measurements instead of hash tables where the main goal is to process all packets with fixed-size memory. [34] augments the dataplane with a separate *fast path* that provides fast but slightly less accurate measurements under high traffic load, and recovers missing information via compressive sensing. [36] learns the statistical distribution of the sketch and uses it to separate large and small flows so as to reduce the impact of hash collisions. Instead of focusing on sketches, MONA relies on simple hash tables for various reasons. One is the increased flexibility, as it enables more heterogeneous measurement tasks [5] and facilitates the design of stateful monitoring applications.

In a similar fashion to our solution, the methods presented in [12] and [5] also address the case of monitoring systems using simple hash tables to store flow statistics. In [12] 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 [5] monitors only flows whose size exceeds a dynamic threshold, so as to handle denial of service attacks. While in [5] 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 lead to packet loss when changes in the operating conditions create bottlenecks. In this paper we proposed MONA, an adaptive monitoring framework that guarantees resilience to bottlenecks while preserving the accuracy of monitoring reports according to user-specified accuracy thresholds. We showed that MONA achieves lossless traffic monitoring under various conditions such as packet rate spikes and skew variations, and guarantees significant enhancements of the accuracy ranges for widely-used measurement tasks. Moreover, we demonstrated that MONA is able to compute new monitoring configurations every 10 ms, without needing additional processor core(s) and with minimal CPU-time overhead ($\approx$ 1-2%), even for 10 Gbps traffic of small packets. Future work will extend our evaluation to investigate possible bottlenecks induced by the NIC (*e.g.*, bursty packet arrivals, polling inefficiencies, etc.) and potential performance degradations arising from end-to-end testing.

References

[1] G. Tangari, M. Charalambides, D. Tuncer and G. Pavlou. Adaptive traffic monitoring for software dataplanes. In *Proc. International Conference on Network and Service Management (CNSM)*, Tokyo, 2017, pp. 1-9.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

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

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

[11] Receive Side Scaling. Available: https://www.kernel.org/doc/ Documentation/networking/scaling.txt

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

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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

[26] M. Alizadeh et al. Data Center TCP (DCTCP). In *Proc. ACM SIGCOMM*, New Delhi, 2010.

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

[28] 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.

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

[30] X. Yu et al. CountMax: A lightweight and cooperative sketch measurement for Software-Defined Networks. In *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2774-2786, Dec. 2018.

[31] J. Boite et al. Statesec: Stateful monitoring for DDoS protection in software defined networks. In *Proc. IEEE NetSoft*, Bologna, IT, 2017, pp. 1-9.

[32] D.Knuth. The Art of Computer Programming. Vol 2, page 232, 3rd edition.

[33] G.Hasslinger, O.Hohlfeld. The Gilbert-Elliott Model for Packet Loss in Real Time Services on the Internet. In *Proc.* GI/ITG Measurement, Modelling and Evalutation of Computer and Communication Systems

[34] Q. Huang et al. Sketch Visor: Robust Network Measurement for Software Packet Processing. In *Proc. ACM SIGCOMM*, Los Angeles, CA, USA, Aug. 2017.

[35] G.Cormode and M. Hadjeleftheriou. Finding Frequent Items in Data Streams. In *Proc. PVLDB*, Aug. 2008.

[36] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug. 2018.

[37] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. 2016. MOZART: Temporal Coordination of Measurement. In *Proc. ACM SOSR*, Santa Clara, USA, Mar. 2016.

[38] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. 2005. Fast hash table lookup using extended bloom filter: an aid to network processing. SIGCOMM Comput. Commun. Rev. 35, 4 (August 2005), pp. 181-192.

[39] J. Moraney and D. Raz. On the Practical Detection of the Top-k Flows. In *Proc. International Conference on Network and Service Management (CNSM)*, Rome, Italy, 2018, pp. 81-89

[40] A. Gupta et al. Sonata: query-driven streaming network telemetry. In *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug. 2018.

[41] S. Narayana et al. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. ACM SIGCOMM*, Aug. 2017.

[42] D. Barach et al. Batched packet processing for high-speed software data plane functions. In *Proc. IEEE INFOCOM*, Honolulu, HI, 2018.

[43] M. Dobrescu and K. Argyraki. Software Dataplane Verification. In *Proc. Usenix NSDI*, Seattle, WA, USA, 2014.

[44] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks. In *Proc. of ACM SOSR*, 2016.

[45] Z.Liu et al. NitroSketch: Robust and General Sketch-based Monitoring in Software Switches. To appear in *Proc. ACM SIGCOMM*, 2019.

Gioacchino Tangari is a PhD student in the Department of Electronic and Electrical engineering at University College London. His main research interests include network monitoring in the context of programmable networks, and high-speed packet processing on commodity hardware. He has been working as a research intern in Nokia Bell Labs in 2014, Paris, and Telefonica Research, Barcelona, in 2017.

Marinos Charalambides is a senior researcher at University College London. He received a BEng in Electronic and Electrical Engineering, a MSc in Communications Networks and Software, and a Ph.D. in Policy-based Network Management, all from the University of Surrey, UK, in 2001, 2002 and 2009, respectively. His current research interests include network programmability, adaptive resource management, content distribution,and network monitoring.

Daphne Tuncer is a Research Fellow in the Department of Computing at Imperial College London, UK. She received her Ph.D. from University College London (UK) in 2013 and a Diplome d'ingenieur de Telecom SudParis (France) in 2009. Her research interests are in the areas of software-defined and programmable networks, adaptive network resource management and multimedia content distribution.

George Pavlou is Professor of Communication Networks in the Department of Electronic and Electrical Engineering, University College London, UK. He received MSc and PhD degrees in Computer Science from University College London, UK. His research interests focus on networking and network management, including aspects such as traffic engineering, quality of service management, information-centric networking, and software-defined networks.