Towards Software Defined Measurement in Data Centers: A Comparative Study of Designs, Implementation, and Evaluation

Zili Zha, An Wang, Yang Guo, and Songqing Chen

Abstract—Cloud data centers are increasingly adopting the Software-Defined Networking (SDN) technologies for their underlying connection and communications. However, as a critical part of daily operations and management of such data centers, the network measurement is essential but has often been constrained by the available resources in the traditional network devices. Thus, how to properly balance the resource consumption while maintain timely and accurate measurement remains a challenge to data center systems. Recent advances in Software-Defined Networking (SDN) have enabled flexible and programmable network measurement, which is referred to as Software Defined Measurement (SDM). A promising trend for SDM is to conduct network traffic measurement on widely deployed Open vSwitches (OVS) in data centers. However, little attention has been paid to the design options for conducting traffic measurement on the OVS. In this study, we set to explore different designs and investigate the corresponding trade-offs among resource consumption, measurement accuracy, implementation complexity, and impact on switching speed. Through extensive experiments and comparisons, we quantitatively show the various trade-offs that the different schemes strike to balance, and demonstrate the feasibility of instrumenting OVS with monitoring capabilities. These results provide valuable insights into which design will best serve different measurement and monitoring needs.

Index Terms—SDN; Open vSwitch; eBPF; network measurement and monitoring.

1 Introduction

The recent years have witnessed the increasing adoption of Software Defined Networking (SDN) technologies in cloud data centers. For example, Google uses SDN for both intra data center and inter data center connection management.[1]. Critical to the various operations and management of such data centers, the network measurement has been playing an essential role in a wide variety of network management tasks, ranging from traffic engineering, anomaly detection, to QoS provisioning, etc. Traditional monitoring tools, e.g., Netflow, sFlow, IPFIX, are usually deployed across in-network hardware devices to collect real-time traffic statistics. Nonetheless, due to the underlying hardware resource constraints, they only provide coarse-grained statistics that could not meet the monitoring demands of the diversified network applications. How to properly implement monitoring to achieve timely and accurate results while minimizing the corresponding resource consumption has remained as a challenging issue.

The rapid development of SDN and network function virtualization (NFV) [1] techniques has motivated a series of research [2], [3], [4], [5] to enhance the existing measurement schemes. However, they are either not generic by requiring to implement multiple sketches for each measurement task [2], [3], [4], or too expensive to deploy in hardware devices [5]. In recent years, the emerging programmable dataplanes have spawned great opportunities for innovation in integrating monitoring solutions into the switching hardware. This trend enables Software Defined Measurement (SDM) where users can flexibly manage the monitoring rules via programming APIs.

Such flexibility can be achieved via both hardware-based approaches and software-based approaches. For hardware-based approaches, programmable switch ASICs and SmartNICs are often leveraged. However, programmable switch ASICs are constrained by hardware resources, such as fixed hardware stages, limited per-stage actions and restricted stateful memory (e.g., Registers, Counters), making per-flow traffic statistics collection a non-trivial task. SmartNICs could also be leveraged to perform generic packet and flow-filtering. But they often have limited compute and memory capabilities, making it even more difficult to completely offload network measurement tasks. For that reason, some prior work focused on monitoring only heavy hitter flows [6], [7], [8]. Additionally, the hardware-based monitoring frameworks all utilize sketch-based streaming algorithms aiming to minimize the memory consumption, since memory is the primary concern.

On the other hand, software-based approaches have become more and more important in building network monitoring functionalities as inspired by the following observations [9], [10], [11]. There are two reasons for this trend. First, commodity servers are in possession of plentiful CPU and memory resources. Compared to the hardware routers that often have limited computing and memory resources, data centers and clouds often have redundant resources in terms of computing power and memory capacity that are not fully utilized or idle. Second, software-based approaches are much more scalable since each end host only needs to process much smaller amount of traffic as opposed to that of in-network hardware devices. This sheds light on SDM by using software

- *Z. Zha is with George Mason University. zzha@gmu.edu*
- *A. Wang is with Case Western Reserve University. axw474@case.edu*
- *Y. Guo is with NIST. yang.guo@nist.gov*
- *S. Chen is with George Mason University. sqchen@gmu.edu*

1. Certain commercial equipment, instruments, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose

switches, such as Open vSwitches (OVS), since they have become the building blocks of virtualization software and widely deployed in data center systems.

One such example is UMON [9], where a set of traffic monitoring interfaces are developed to enable user-defined monitoring rules in OVS. Despite of the abundant hardware resources in commodity servers, building efficient monitoring frameworks into data center end hosts remains unexplored. Existing software-based monitoring solutions [11], [12], [13], [14], mostly focus on the designs of sophisticated sketching algorithms to achieve both high monitoring accuracy and network performance. In certain circumstances, accuracy of small flows is sacrificed in order to keep up with the high packet rate, which is undesirable for security related network applications. Furthermore, their monitoring frameworks are not readily applicable to virtualization environment since they are completely independent of the existing software stack in the end hosts. On the contrary, building monitoring solutions into the software switches and re-using the same high level APIs makes it much easier to manage and program from upper layer applications.

Incorporating traffic monitoring capability into a software switch offers the opportunity to share the key functionalities required by monitoring that have been implemented in a software switch. However, the design of such an integration is challenging in order to achieve minimal forwarding-monitoring function interference, optimal code sharing, and efficient CPU/memory resource usage. So far, there is no comprehensive investigation regarding how to properly and efficiently conduct measurements leveraging these potentials.

To this end, in this study, we aim to explore different approaches for gaining a comprehensive understanding of various trade-offs in SDM using software-based approaches. For this purpose, we set to empirically investigate the different design trade-offs using OVS [15] as a representative software switch. We start with an intuitive design, called FCAP (*Flow CAPture scheme*), where the forwarding and monitoring forms a pipeline in the OVS kernel. In FCAP, a packet traverses through the forwarding module before going through the monitoring module. The flow stats of interested traffic flows are first collected in the OVS kernel and then transferred to the user space for further processing. To reduce the memory consumption, we further design SMON, a *Sketch [16] based MONitoring* scheme that compresses the flow stats using sketches. Sketches are probabilistic data structures that trade off query accuracy for space efficiency and widely employed in a multitude of various applications. Since flow identifiers and per-flow traffic stats both need to be collected, we use an advanced sketch design, namely, invertible bloom lookup tables [16], in SMON, which allows us to easily recover the complete flow details in upper layer applications.

However, both FCAP and SMON place monitoring in the same pipeline with forwarding. Such a design may cause the switch to operate below line rate. To minimize such impact, we propose to design off-path counterparts by decoupling the monitoring from the forwarding. This is achieved via a ring buffer in the kernel. The ring buffer temporally caches the packet headers of the interested traffic flows, which can then be processed independently by the monitoring module. In this way, the ring buffer effectively decouples the monitoring from the packet forwarding at the kernel data path.

FCAP/SMON designs all require extensive instrumentation into the OVS code base, which is not backward compatible. For practical deployment, we either need extra patches or re-install a modified version of OVS with the customized monitoring functionalities. Moreover, the SMON/FCAP monitoring modules are implemented within the kernel data path in order to achieve full visibility and minimize the impact on the forwarding performance. However, a single software flaw could crash the entire system. To address these challenges, we further propose an ~~eBPF (~~extended Berkeley Packet Filter (eBPF) enhanced [17] monitoring design that is completely independent of OVS. While its ancestor BPF is mostly used for in-kernel packet filtering, eBPF extends its architecture by integrating more features to support more types of events and actions other than filtering. eBPF offers the possibility to dynamically generate, load and execute code into the kernel using the `bpf()` system call, thus obviating the need to install customized kernel modules. Many eBPF-based tools are developed for performance debugging and troubleshooting, e.g., tracing the TCP sessions lifespan and the block device I/O latency, etc. Furthermore, BPF maps provide an asynchronous communication channel for sharing data between the userspace/kernel and across multiple runs of the kernel program. In our work, to gain full visibility of both inbound/outbound traffic, our monitoring programs are attached onto the Linux Traffic Control layer, while the monitoring filter and flow stats table are both implemented using eBPF maps.

We conduct extensive experiments to explore the various trade-offs under the metrics of throughput, latency, CPU overhead, memory overhead etc. The results show that (1) From the performance aspect, the off-path designs achieve the minimum measurement delay compared to on-path counterparts, including eBPF. (2) On the aspect of accuracy, all the designs can achieve almost full measurement accuracy, but at different costs. FCAP and eBPF need to leverage linked lists to resolve collisions in hash tables; SMON consumes more CPU cycles for sketch decoding; While the off-path designs have higher memory consumption. (3) For implementations, UMON is most flexible since it does not require modifications of OVS kernel code base. eBPF requires minimal maintenance efforts as it is independent of the development of OVS. (4) Overall, it is feasible to instrument OVS with monitoring capabilities without affecting the switching performance significantly.

The remainder of the paper is organized as follows. Section 2 describes some related work. We present our new designs and implementations in Section 3. We evaluate the proposed designs in Section 4 with more discussions in Section 5. Finally, we make concluding remarks in Section 6.

2 Related Work

Traditional Monitoring. Different network measurement frameworks have been investigated both in software and hardware switches. Traditional hardware-based solutions utilize tools such as Netflow [18], sFlow [19] and IPFIX [20], to collect IP Nework traffic. Other similar solutions include Jflow [21], Cflowd [22] and NetStream [23] etc. The drawbacks of these solutions are twofold: they are more expensive to deploy and they do not provide enough programmability for network management tasks.

SDN-enabled Monitoring. One of the earliest efforts is proposed by Yu et al. [2] called OpenSketch. In OpenSketch, different types of sketches are utilized to achieve different measurement goals. Furthermore, the controller optimizes the sketch allocation to balance the accuracy and the memory consumption. A followup prototype called DREAM [3] is proposed to dynamically assign TCAM counters to different measurement tasks across multiple

hardware switches in the network. But the users could not customize measurement tasks other than the counter-based ones. In these earlier works, sketches are designed and implemented for specific monitoring tasks, which means that the monitoring devices must instantiate multiple sketches in order to support a variety of concurrent monitoring tasks. This places enormous burden on resource-constrained hardware devices and drastically degrades the network performance. To address this limitation, UnivMon [24] proposes a single universal sketch to support multiple measurement tasks simultaneously. Nonetheless, it requires to update multiple components for each packet, which also introduces noticeable overhead. Yu et al. also proposed FlowRadar [5] to improve the NetFlow based network measurement by encoding and decoding counters with the invertible bloom filter lookup table (IBLT). In this way, the communication overhead could be reduced. However, extra components are necessary to implement on hardware devices. Also, the decoding may introduce redundant overhead to the controller.

Monitoring within Programmable Dataplanes. *Space Saving* [25] is a widely known top-*k* algorithm to identify the first top-*k* frequent items in data streams. Compared to other counter-based streaming algorithms, *Space Saving* is much more resource efficient since it only needs to maintain *O(k)* counters. Despite of its memory efficiency, *Space Saving* is not readily applicable for heavy flow detection within the emerging programmable hardware due to the underlying complexities in its data structure and algorithm design. Upon each new flow, the algorithm requires to find and replace the hash table entry with the minimum packet count, which cannot be easily implemented considering the hardware constraints of the hardware programming model. To adapt the classical algorithm into a hardware-friendly design, HashParallel and HashPipe [6] refactor the algorithm into a pipeline of hash tables that can fit in the programmable switches. This pipelined design helps to ensure that each stage only incurs a limited amount of processing in order to keep up with the line-rate switching throughput. Nonetheless, Precision [7] re-examines the problem and concludes that HashPipe is challenging to realize in the Reconfigurable Match Tables (RMT) [26] switch programming model since it does not satisfy the limited branching rule and single stage memory access rule imposed by the RMT model. To overcome the hardware limitations, Precision further improves the design by recirculating a small fraction of the packets at the cost of packet forwarding performance. Orthogonal to this direction, Memento [8] examines the problem from a different perspective by proposing a sliding window based heavy hitter detection model. It argues that sliding window models are more accurate and more efficient in terms of detection delay compared to the traditional interval based detection solutions. Further, it extends the algorithm to detect hierarchical heavy hitter (HHH) and network-wide scenarios. Sliding window based models have also been extensively studied in many earlier works [27], [28]. Marple [29] and Sonata [30] tackle the problem from a different perspective. Instead of designing new sketches to minimize the memory consumption in the hardware devices, Marple proposes a performance query language and designs new switch hardware primitives to support the language, which allows network operators to program their performance queries that collects customized fine-grained traffic statistics at a low processing overhead. Different from Sonata, it performs aggregations directly in the switch hardware, further reducing the data volume streamed to collection servers.

Edge-based Software Monitoring. Over the past few years, with the data center networks evolving to larger scales and the ever increasing line speeds, the resource constraints of hardware devices have become considerably more stringent. Comparatively, the edge servers are typically equipped with much more powerful hardware resources, e.g., CPU and memory. Motivated by this, there has been continuous efforts aiming to migrate monitoring functionalities from hardware devices to edge servers. Generally, existing software-based solutions can be broadly classified into two categories: passive monitoring system [11], [12], [13] and active monitoring system [9], [10]. The former category strives to collect traffic stats for all flows with minimal memory consumption and provable accuracy guarantees, by designing sophisticated sketches and algorithms. However, in order to keep up with high line-rates, they focus more on the accuracy of heavy flows while sacrificing that of the small flows, considering that heavy flows are usually more important than small flows in typical monitoring tasks. In contrast, active monitoring systems provide programmability that allows users to define their own monitoring tasks and only monitoring the traffic the network operators are interested in. This efficiently lessens the monitoring workload, further minimizing resource usage and impacts on the forwarding performance.

SketchVisor [11] focuses on accurate and timely network measurement under high traffic load. It proposes to combine a sketch based normal path and a top-*k* based fast path to achieve both high throughput and high accuracy. Under high traffic load, the fast path is activated to absorb the excessive traffic overflowed from the fast path with slight accuracy degradation. Further, it employs compressive sensing [31] to recover the flow stats information that serves as input for higher level monitoring applications. Following this work, Elastic Sketch [12] enhances SketchVisor by designing an elastic sketch with two components, a heavy part and a light part where the former maintains elephant flows and the latter records the mouse flows. Under heavy traffic load, only the heavy part is updated and the mouse flow information is lost. Compared to SketchVisor, Elastic Sketch achieves much higher performance since only one memory access is needed at high packet rate. In NitroSketch [13], it is pointed out that Elastic Sketch falls short in performance and accuracy when the number of flows increases to a certain point. In comparison, NitroSketch proposes a generic sketching framework that addresses the bottlenecks of existing sketched designs and minimizes per-packet CPU and memory overhead. HeavyKeeper [14] further improves heavy hitter detection accuracy of Elastic Sketch via a new strategy, *count-with-exponential-decay*, to actively evict small flows through decaying. It reduces the error by 3 orders of magnitude compared to the state-of-the-art detection schemes. However, in certain applications, such as anomaly detection, small flows play an equally important role as heavy flows but cannot be captured by existing passive monitoring systems that focus on heavy flows.

Following this trend, Trumpet [10] is proposed to collect data from end-host machines to detect network-wide events. Though Trumpet is optimized to run on hardware network devices, it is independent of the existing network management framework.

Although existing software based measurement designs all function well under particular circumstances, an in-depth investigation about the resource-accuracy trade-offs is still lacking in the literature. Our work aims to fill this void by looking into the various software-based monitoring designs from a systematic view.

eBPF-based Monitoring. eBPF has enabled the high performance datapath in Linux since it was first merged into the Linux kernel. The development of eBPF has fully unlocked network programmability and allowed for a diverse community to form around it, spanning networking, tracing, security, profiling, and observability. Recently, Abranches *et al.* designed and implemented a network monitoring architecture based on eBPF [32]. Their proposed framework differs from ours in two aspects. First, their framework attaches itself to a different hook point than ours. Secondly, their framework is designed to perform analytics over application traffic, while our framework is mainly used for capturing abnormal network behaviors. eBPF has also been leveraged to mitigate DDoS attacks. For example, Cassagnes *et al.* proposed a framework to monitor containerized user-space applications and prevent DDoS attacks [33]. Similar efforts have also been conducted by Miano *et al.* in [34].

3 Design and Implementation

To empirically explore the various design trade-offs among multiple factors, including server resource consumption, monitoring overhead, and implementation complexities, in this section, we propose five novel monitoring designs, namely, on/off-path FCAP/SMON [35] and eBPF. Among them, four are incorporated into OVS and an eBPF-based monitoring framework in parallel with OVS. Following a brief discussion about the design challenges arising from building monitoring logic into OVS, we walk through the design and implementation details of each monitoring framework.

3.1 OVS and Design Challenges

OVS consists of two major components: a userspace daemon (`ovs-vswitchd`), and a kernel datapath [36]. They work together to forward packets, with the userspace daemon as a full but slow path while the kernel datapath serving as a forwarding cache. Such a design aims to optimize the forwarding performance of the switch. More specifically, incoming packets are firstly matched against the flow table in the kernel datapath. If the packet encounter a flow miss in the kernel, it will be forwarded to the userspace by injecting a *upcall*. In the userspace, *upcall*s are handled by the *handler* threads, and a flow rule is generated and installed into the kernel flow cache. As a result, subsequent packets belonging to the same flow do not need to make detours through the userspace. Finer-grained kernel flow rules undoubtedly result in a larger number of flow misses and *upcall*s. This not only undermines the switch performance, but also introduces heavier workloads, thus higher CPU overhead for *handler*s. Fortunately, due to the locality of the network traffic, most packets are processed in the fast path.

Each flow entry provides built-in monitoring capabilities via fields, such as packet and byte counters. These counters record the total number and bytes of packets processed by the corresponding flow entry. As aforementioned, the userspace does not have full visibility into all packets. Thereby, the packet/byte counts in the userspace table entries need to be updated by polling the kernel cache entries. These are managed by the *revalidator* threads, which periodically poll the kernel cache for each flow's packet and byte counts and aggregates them into the userspace flow table. In addition, *revalidator*s are also responsible for maintaining the kernel cache entries. Similarly as *handler*s, a larger kernel cache introduces heavier workloads for *revalidator*s.

Nevertheless, this built-in feature in OVS flow tables is neither flexible nor sufficient for the dynamic monitoring needs, since flows that are relevant for monitoring and forwarding might not be overlapped. For example, forwarding rules might specify actions over destination IP addresses, while monitoring applications need fine grained flow statistics for each 5-tuple subflow. Relying only on packet and byte counts of the flow rule could not achieve the desired monitoring granularity. To cope with this limitation, some works propose to dynamically install flow forwarding rules for each subflow into OVS. As a consequence, the first packet of each subflow has to be sent to the centralized controller for further handling. This drastically degrades the forwarding performance of the data plane and causes potential control path congestion. A more efficient programmable monitoring solution is imperative. UMON is one of the earliest efforts in this direction.

However, UMON has some limitations on the switch performance. The fundamental idea of UMON is to decouple monitoring from forwarding logic in the userspace, while the kernel datapath remains intact. To achieve this, the cached entries in the kernel need to be much more fine-grained than the native OVS, which is elaborated in Section 3.2. As explained earlier, this inevitably incurs heavier consumption of system resources. Motivated by this, our investigations of different monitoring designs mainly consider the following aspects: monitoring accuracy, resource consumption, switching performance, and portability.

Overall, to build monitoring capabilities into OVS, there are a number of challenges we need to address: (1) The added monitoring logic should introduce minimal interference to the forwarding path in OVS to guarantee the forwarding and monitoring efficiency; (2) Due to the resource constraints, it is necessary to strike a balance among efficiency, resource consumption, and monitoring accuracy; (3) To maximize feasibility and compatibility, the monitoring function should be as portable as possible so that minimal effort would be required to accommodate the monitoring function.

Taking these into consideration, the monitoring function is decoupled from the forwarding function by maintaining an additional monitoring table, as illustrated in Figure 1. The default forwarding process is performed by OVS, while the additional monitoring table supports the added monitoring functionalities. Contrarily, in the kernel datapath, five monitoring designs are proposed, which differ in multiple aspects, including interaction between the monitoring and forwarding functions, the placement of the monitoring module (Challenge 1), the stats collection data structures and algorithms (Challenge 2). Beyond these, we also develop an eBPF-based monitoring framework that runs in parallel and independently with OVS (Challenge 3).

Unlike other existing work that utilize streaming algorithms and compact data structures, we mainly leverage simple yet efficient algorithms in our designs for two reasons. First, streaming algorithms typically require encoding and decoding processes, where encoding is generally much simpler than decoding. In most existing work, such as Trumpet [10], the workload of complex decoding is often offloaded to the control plane, making the software-based measurement less scalable. Second, streaming algorithms can only provide estimate values, making it difficult for network operators to troubleshoot when something goes wrong. In the following sections, we delve into the details of each specific design following a brief overview of UMON.

3.2 Recap of UMON

As introduced in Section 3.1, the monitoring programmability of UMON is facilitated through the introduction of an independent monitoring table in the userspace daemon. The overall architec-

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3181890

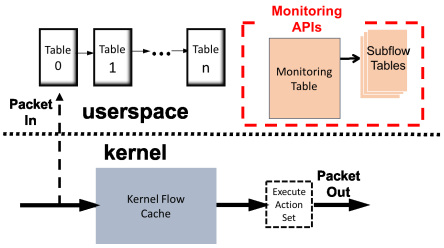


Fig. 1: The OVS architecture.

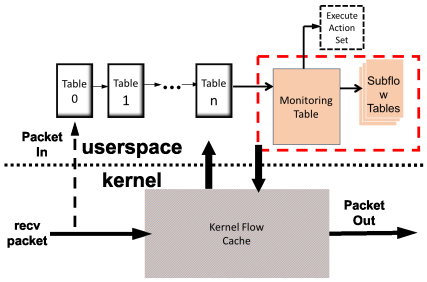


Fig. 2: UMON architecture.

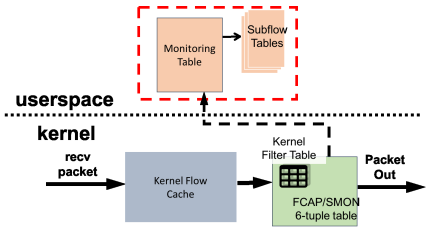


Fig. 3: FCAP/SMON architecture.

ture of UMON is depicted in Figure 2. Comparing this figure with Fig. 1, we can observe that UMON preserves the original architecture of OVS. It simplifies the monitoring design by only instrumenting the OVS userspace module, leaving the kernel datapath untouched. Specifically, the monitoring table maintains rules that monitor specific TCP traffic, such as TCP SYN packets, or/and collects subflows in the subflow tables. Besides, the monitoring table provides APIs for the controller to install and update monitoring rules via an extended OpenFlow protocol.

To support the user-defined monitoring granularities, UMON compiles the forwarding and monitoring flow rules together to generate cache entries in the kernel. For example, a flow rule forwards packets destined to host B to port 1 while the monitoring rule needs to collect the packet/byte counts originated from host A. UMON combines the two rules to generate a more fine-grained rule that forwards packets with source IP of A and destination IP of B to port 1 instead. Following this design, an incoming packet with *(srcIP=C, dstIP=B)* cannot find a match in the kernel cache and will raise a flow miss that needs to be sent to the userspace for further processing. As aforementioned, due to the lack of visibility in the userspace, flow statistics need to be properly populated from the kernel space to the userspace flow table. In the meanwhile, the flow stats in the monitoring table should also be updated. In UMON, this credit logic is piggybacked in the *revalidator* thread of OVS because it maintains the flow statistics periodically. As in the above example, the *revalidator* threads polls the kernel space for packet/byte counts in the cached flow entries and aggregates the micro-flows by different fields, e.g., dstIP for the flow table and srcIP for the monitoring table.

The downside of UMON is that the kernel flow table might get inflated with a vast amount of fine-grained flows during peak traffic. Furthermore, a significantly larger amount of flow misses are generated, thereby, both *handler* and *revalidator* threads will potentially experience much heavier workloads. On the other hand, statistics collection of monitored flows is integrated into the native OVS operations without any changes of its current workflow. Therefore, the latency caused by monitoring interruptions will be reduced. Moreover, UMON does not require any modifications in the kernel, thus it could be easily ported to other edge devices and platforms, such as DPDK and NetFPGA. More details are discussed in [9].

In this study, we only use UMON as a comparison against the other designs. These designs, including UMON, embody different trade-offs between resource (e.g., CPU, memory) consumption, monitoring efficiency, forwarding efficiency (throughput and latency). One has to strike a balance among these considerations. In the following, we discuss the specific design considerations in greater detail.

3.3 Design of Flow Capture (FCAP)

The performance impact of the monitoring functions is primarily dependent on where the functions are placed. Intuitively, a separate monitoring function in the userspace provides better isolation and allows better interaction with the users. However, we have learned two lessons from the development of UMON. (1) the complexity of userspace rules would introduce extra overhead to the kernel; (2) the collection of monitoring stats should be prompt to preserve accuracy. Based on these considerations, we propose to build a separate monitoring phase in the kernel datapath in OVS.

To guarantee the accuracy of monitoring tasks, we need to maintain statistics of all the related packets efficiently. The micro-flow information is more preferable than the mega-flow information because monitoring tasks often have dynamic granularity requirements and micro-flows simplify the aggregation operations. Thus, we first design two different schemes, FCAP (`Flow CAPture`) and SMON (`Sketch based MONitoring`), to collect micro-flows. In our current designs, we use 6-tuples (source/destination IP addresses, source/destination ports, protocol and TCP flags) to represent each micro-flow.

Fig. 3 shows the architecture of FCAP and SMON. In this figure, since userspace pipeline is similar as UMON, the forwarding pipeline in the userspace is omitted here for clarity. To facilitate user-defined monitoring tasks, an additional kernel filter table is utilized to classify packets in both FCAP and SMON. The workflow of FCAP is described in Algorithm 1. Once the packet is determined to be relevant to a monitoring task (*line 1*), the 6-tuple information will be stripped off and kept in the custom 6-tuple flow stats tables (*line 5-8*).

However, the way for FCAP and SMON to store such information is different. FCAP employs a straightforward mechanism by storing the 6-tuple flow stats in a hash table. In order to maintain full accuracy, linked lists are used to resolve hash collisions. With the hash index, the monitoring thread scans through the linked list to find the flow entry with the same 6-tuple identifier as the incoming packet(*line 4*). Due to its ability to preserve the complete 6-tuple information, the aggregation operations required by monitoring tasks are simplified. Furthermore, the collected statistics are accurate without any loss.

As illustrated in Fig. 3, the monitoring pipeline consists of two stages, including a kernel filter table and a 6-tuple table. Only packets finding a match in the filter table are counted towards the latter. The entries in the kernel filter table are populated from the userspace monitoring table. Note that the kernel table differs from the userspace table from two aspects. First, the kernel table employs longest prefix matching to find any rule that matches against the header. Instead, the monitoring rules in the userspace table are matched against one by one since the monitoring rules may overlap. Secondly, it is not necessary to maintain stats of the headers in this table.

To aggregate the collected 6-tuple flows, we implement a thread that employs similar mechanism with that of the *revalidator*

thread in OVS [36]. The thread retrieves the flow stats from the 6-tuple stats table at fixed time intervals and updates the counters associated with the rules in the userspace monitoring table. The credit function is implemented in a similar way as UMON, which credits both flow stats and subflows to the monitoring table. To enhance the efficiency, we further cache the matching results of the 6-tuple information with an extra hash table, where entries expire with the default timeout *ofproto_max_idle* value. The extra hash table is a simple data structure that maps the hash values of 6-tuple information to its corresponding bucket.

Algorithm 1 FCAP algorithm

```
Input: FlowStatsTable, flowTuple
isMonitored ← LookupMonitorFilter(flowTuple)
if isMonitored = True then
    hash ← Hash(flowTuple)
    bucket ← FindBucket(FlowStatsTable, hash, flowTuple)
    if bucket ≠ null then
        UpdateFlowStats(FlowStatsTable, bucket, flowTuple)
    else
        InsertFlowTuple(FlowStatsTable, bucket, flowTuple)
    end if
end if
```

3.4 Design of Sketch based Monitoring (SMON)

Although FCAP provides highly accurate statistics, it is not always affordable and also sometimes may not be necessary. Inspired by previous work, sketches have great potential in reducing memory consumption on end hosts. Sketches are space-efficient probabilistic data structures that are extensively used in streaming applications to process and store summary information. Examples include bitmaps, bloom filters, and count-min sketch, which serve diverse purposes. They provide provable guarantees on the storage usage and error bounds. In previous work, sketches have been used for traffic monitoring in hardware network devices, where memory is a primary concern. Nonetheless, the performance of sketch monitoring built into software entities remains unexplored. Intuitively, to achieve higher memory efficiency, sketches involve more complex computation logic, e.g., more hash computations, to compress the memory. To investigate the trade-offs between memory/CPU consumption and monitoring accuracy, we propose SMON, a sketch-based mechanism to maintain the compressed 6-tuple flow information. As just mentioned, many research works have focused on utilizing various sketch mechanisms to perform monitoring [37], [38], [39], [40]. Among all existing solutions, the bloom filter has a strong space advantage over other data structures. However, the primary drawback of bloom filter is that it does not store the data elements themselves. Therefore, we cannot retrieve the item based on its key, which limits the capability to collect subflows.

Goodrich *et al.* proposed invertible bloom lookup tables (IBLT), which consists of three components in each bucket to store a key/value pair and the corresponding count [16]. In this way, the 6-tuple flow IDs that are hashed into the same bucket are *XOR*ed and stored in a single bucket, as depicted in Algorithm 2. Instead of using linked lists as in FCAP, flows that fall into the same bucket are compressed in order to save memory space (*line 4-6*). In this algorithm, H represents the number of different hash functions that are pairwise independent. To reduce hash collisions, an intuitive solution is to use a lot of space to make collisions unlikely enough to get accurate results. However, space is typically limited in network switches. To make the algorithm more efficient, we instead use different hash functions so that flows have chance to be mapped to buckets that have fewer hash collisions. Similar approaches have been widely adopted by existing streaming algorithms, such as the Count-Min sketch [41], FM sketch [42] and the Tug-of-War sketch [43]. In the user space, a customized thread periodically retrieves the bloom filter from the kernel datapath via Netlink socket interface and recovers all flows from the sketch. The size of the sketch structure is adjusted according to the estimated number of flows in each time interval to guarantee successful decoding of flows at a high rate while ensuring a minimum amount of memory usage. The detailed decoding process is explained as follows. It iteratively finds the elements in the bloom filter that contain a single flow and remove its stats from all the other encoded cells that the flow is hashed to, until all the buckets are decoded. Ideally, if the size of the bloom filter is sufficiently large, and exported to the user space at a high frequency, all flows could be successfully recovered. In a cloud system, OVSes are often deployed at the edge, thus can only observe a moderate amount of flows. This suggests that we can achieve a high decoding rate with a moderate amount of memory. Apparently, the flow decoding time grows with the size of the sketch and the number of flows in each measurement epoch. Similarly with FCAP, the decoded flows are aggregated into in the user space monitoring table. Besides, the filter table in SMON has the same designs as FCAP. Later we will show in Section 4 that with a small amount of memory consumption, we manage to preserve highly accurate statistics.

Algorithm 2 SMON algorithm

```
Input: IBLT, flowTuple
isMonitored ← LookupMonitorFilter(flowTuple)
if isMonitored = True then
    foreach k ∈ [1..H] do
        h_k ← Hash_k(flowTuple)
        if IsNewFlow(flowTupe) then
            CompressFlowID(IBLT, h_k, flowTuple)
        end if
        UpdateFlowStats(IBLT, h_k, flowTuple)
    end for
end if
```

3.5 Off-path Designs of FCAP/SMON

The intuitive designs of FCAP and SMON place monitoring logic on the normal packet forwarding path in OVS kernel, thereby are called on-path designs. Such on-path designs introduce extra processing delay to the OVS forwarding, since monitoring usually requires more complicated processing logic than forwarding, which may further reduce the forwarding throughput. To reduce the negative impact, we further embrace a buffering mechanism in order to take the monitoring function off the forwarding path. By using a ring buffer, we aim to decouple the monitoring functions from the forwarding path. We consider this mechanism for both FCAP and SMON, and thus design off-path FCAP and off-path SMON.

The overall architecture of off-path designs is demonstrated in Fig. 4. The ring buffer is conceptually a circular FIFO queue with pre-defined size. The main difference between a circular queue and a linear one is that a circular queue has the maximum size or capacity that allows it to continue to loop back over itself in a circular motion. With a linear queue, it is more difficult to adjust its size during operation. This can be replaced with a dynamic buffer that grows and shrinks automatically based on packet rates. But circular buffer is selected for simplicity and efficiency. The size of this circular queue is set to be **X** bytes by default. This value is subject to change and can be determined by system operators based on their experience. The ring buffer has two pointers, head pointer for the consumer thread and tail pointer for the producer thread, as depicted in Fig. 5. As long as the distance between

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3181890

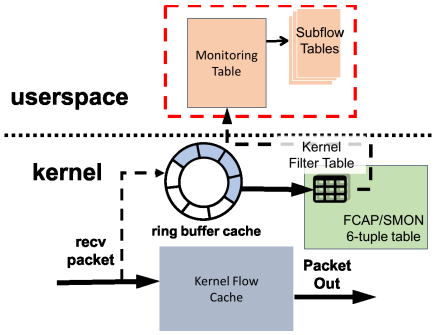


Fig. 4: Off-path FCAP/SMON architecture.

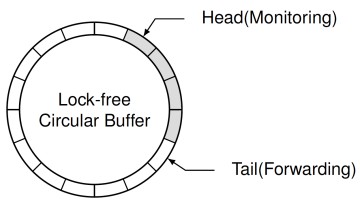


Fig. 5: Lock-free single-producer single-consumer circular buffer.

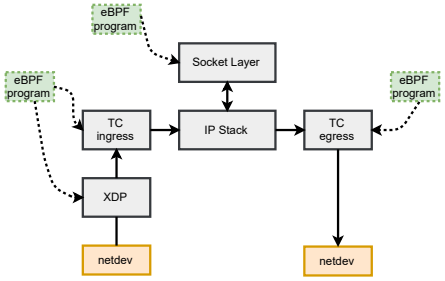


Fig. 6: Locations of eBPF hooks where monitoring programs can be attached.

the two pointers does not shrink to zero nor expand to the full buffer size, both the producer and the consumer could operate on the data in the queue. In our case, the forwarding process is the producer by making a copy of the incoming packet header and appending it to the tail of the buffer, while the monitoring thread, as the consumer, fetches the headers for further processing from the head of the buffer. Therefore, the ring buffer provides a communication channel for the asynchronous interactions between the two functions.

In our implementation, we employ a lock-free mechanism when writing to the ring buffer to minimize the performance overhead. Compared to the exclusive lock mechanism, it brings a negligible probability of overwriting unprocessed data. In this way, forwarding will not be delayed if the monitoring thread cannot keep up with the forwarding process. While reading will be blocked if the buffer is empty, this aims to guarantee the high performance of the forwarding function by making constant-time operations. The lock-free nature helps to ensure that no waiting is involved in adding or deleting data in the buffer. Since we employ overwriting to handle full queues, the collected statistics might not be highly accurate. In order to achieve precise measurement results, the ring buffer has to be sufficiently large in order to keep up with the ever-increasing packet rates and flow bursts. These will be evaluated in our experiments.

In practice, with the built-in support for cpuset in Linux kernel, one can confine processes to certain processors and memory node subsets so that the monitoring thread and the forwarding thread do not compete for CPU resources. Such optimizations are feasible and practical for data center edge devices where abundant computing resources are available.

3.6 eBPF-based Monitoring

eBPF was originated from BPF, the Berkeley Packet Filter. BPF allows to capture and filter network packets that match specific rules, where filters can be implemented as programs and run on a register-based virtual machine. eBPF extends the support to 64-bit registers, among others, and represents an effort to make programmable Linux kernel. That is, one can run sandboxed programs in the Linux kernel without changing kernel source code or loading kernel modules, and thus can be leveraged for monitoring and security, etc.

The eBPF code is executed in an in-kernel virtual machine using a custom 64-bit RISC instruction set, with 11 64-bit registers, a program counter and a 512-byte stack space. eBPF supports running the code as Just-in-Time compiled bytecode, which is verified by an in-kernel verifier to guarantee security (e.g., forbidding loops to ensure program termination and type checks) before loading the code into the kernel. Internally, various mechanisms enable communication between in-kernel eBPF code and user space processes asynchronously, such as eBPF maps and perf events (FIFO queues). eBPF maps are efficient key-value stores that allow data to be shared within the kernel (i.e., among multiple eBPF programs) or between the kernel and user space.

Figure 6 illustrates how eBPF can be used for network traffic monitoring. Specifically, eBPF programs can be attached to different hook points in the networking data path, such as Traffic Control (TC) or eXpress DataPath (XDP) [44], thereby enabling flexible processing on the intercepted packets. As shown in the figure, ingress traffic can be intercepted in XDP or TC ingress hooks, while the egress traffic can only be intercepted at the TC egress hook, as XDP is not available in egress. Furthermore, eBPF programs can also be attached in the socket layer. Unfortunately, this does not meet the need of monitoring both local and non-local traffic. Therefore, the TC layer can serve our purpose the best because it allows us to investigate both ingress and egress traffic. In the following, we discuss the specific design details of our eBPF-based monitoring framework.

eBPF-based Monitoring Framework. As illustrated in Figure 7, the framework consists of multiple components, including a monitoring pipeline in the kernel space and a monitoring application in the userspace. The communication between the kernel and user applications is facilitated through a shared eBPF map that maintains the real time traffic statistics. The userspace application retrieves the flow stats from the map and clears the entries at fixed time intervals. Similar to the FCAP/SMON designs, the interval is determined based on the users monitoring demands.

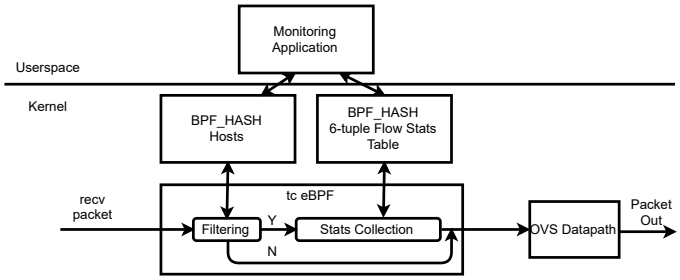


Fig. 7: Design of eBPF-based monitoring framework.

As discussed above, our eBPF program is attached to both the TC ingress and the egress to gain full visibility of all inbound/outbound traffic. The eBPF program is executed and the flow stats are updated for each incoming packet. The monitoring workload varies along with the number of hosts to be monitored. For each monitored host, we need to track the number of packets for each 6-tuple flow associated with it. To filter out the hosts, each incoming packet has to go through a monitoring filter before it is counted towards the flow status hash table. More specifically, the monitoring filter examines the destination IP address of the

packet and filters out packets that are not monitored (*line 12*). Only packets that find a match in the filter will be counted towards the following flow stats hash table (*line 13-14*). The workflow of eBPF monitoring is outlined in Algorithm 3. Intuitively, such processing may introduce extra performance penalty, which comes in the form of map lookups and updates. However, programmability is mainly achieved through this table. Since eBPF maps allow sharing data between kernel and userspace programs, the monitoring application can dynamically update the entries in this table on demand. For example, monitoring can progress along flow of different granularity – from course-grained flows to fine-grained flows – to improve monitoring efficiency. The eBPF maps provide similar but more efficient programmable APIs than OVS's match+action tables. In this work, we will conduct in-depth investigation about the performance of the eBPF-based monitoring module and compare it to the aforementioned monitoring alternatives.

Algorithm 3 eBPF Workflow

```
1: procedure eBPF Userspace Component
2:   LoadBPFProgram()
3:   PopulateMonitorTable()
4:   while true do
5:     Sleep(T)
6:     flowStats ← BPFMapRead()
7:     BPFMapClear()
8:   end while
9: end procedure
10: procedure eBPF Kernel Monitoring
11:   flowTuple ← ParseHeaders(packet))
12:   isMonitored ← BPFMapLookup(monitorTable, flowTuple)
13:   if isMonitored = True then
14:     BPFMapUpdate(flowStatsTable, flowTuple)
15:   end if
16: end procedure
```

To understand the root cause of the processing overhead, we first examine the underlying implementation of the eBPF maps. Currently, eBPF is featured with fifteen types of maps to maintain the states across the invocations of the eBPF program and share data among multiple programs or between the kernel and the user space. Two of the most commonly used types are hash maps and arrays, while the other variants serve more complex purposes. As aforementioned, our eBPF-based monitoring design involves multiple data structures, including a monitoring filter and a key-value store for recording flow stats. Note that eBPF programs cannot process packets in an off-path fashion.The design of the data structures for each functional component is critical since the incurred overhead directly affects the network throughput/latency. In the following, we discuss the designs in more details.

Similar to FCAP/SMON, the monitoring workload is specified in terms of the set of the destination IP addresses of the monitored hosts. BPF_ARRAY and BPF_HASH can both be used for this purpose, while BPF_HASH achieves better performance due to its hash-based design. Therefore, our monitoring filter is implemented based on BPF_HASH, which can be populated with the host IP addresses from the user space. Furthermore, it can be updated at runtime in accordance with changes in the monitoring tasks. Specifically, in our userspace program, the hash map is initialized with the IP addresses as keys whereas the value is set to 1, before it is loaded into kernel. For each incoming packet, if the destination IP address has a value 1 in the hash table, the corresponding flow stats will be updated; otherwise, control flow will follow the original packet processing path. In this way, the host filtering stage can be performed in O(1) time.

On the other hand, to maintain the 6-tuple flow stats information, a hash map (BPF_HASH) is used to maintain the flow identifiers (e.g., 6-tuple) and the corresponding values that refer to the packet/byte counts per flow. Since eBPF maps are instantiated inside the kernel, it is critical to keep the size within a reasonable limit to avoid the exhaustion of kernel memory. In the meanwhile, to achieve the desired monitoring accuracy, the actually requested size should be determined based on an sensible estimation of the total number of flows in the monitored network. By default, BPF_HASH has 10240 entries. Since in our monitoring workload, the total number of flows far exceeds this value, the table size has to be explicitly specified during initialization. Unfortunately, eBPF map cannot be resized after it is created. In the latest kernel implementation, eBPF hash maps use pre-allocation by default. The maximum memory size is bounded by the *max_entries* defined by the userspace program during map initialization. Once the map is full, insertions of new keys will fail in order to make sure that the eBPF programs will not exhaust kernel memory. In other words, an underestimated flow count will result in inaccurate measurement results. Therefore, *max_entries* must be carefully chosen in order to accommodate all 6-tuple flows. The actual parameter setting is workload-dependant and will be discussed in detail in Section 4.

As a consequence, the performance penalty is mostly incurred by the hash map related operations, including hash computation and hash map updates. Also note that updates to eBPF hash map elements are atomic, which are more expensive. Eventually, the exact amount of overhead should be directly correlated with the actual monitoring workload. A closer scrutinization of the underlying implementation of the eBPF hashmap APIs further reveals that the kernel hash table is consistently reused. In the eBPF hashmap implementation, linked lists are used to resolve hash collision. Due to this design, the measurement results of the flow stats are accurate as long as there is no packet loss. In the meanwhile, the underlying implementation is optimized for lookup speed. Given the *max_entries*, the hashmap size is always set to the next power of 2. The total memory allocation is *n_buckets * bucket_size + max_entries * element_size*, where *n_bucket* is actual hashmap size and *max_entries* is the maximum number of entries estimated by the user. We will conduct experiments to measure and compare the throughput/latency under various monitoring workloads. A detailed analysis and comparison with other monitoring designs will be presented in our evaluations.

4 Performance Evaluation

Our test-bed consists of three Lenovo ThinkServer machines equipped with Intel Xeon 4-Core 3.20GHz CPU and 4GB memory that run Ubuntu 14.04. One machine is dedicated to run the instrumented Open vSwitch (OVS). The second machine serves both as the packet generator and the data sink that receives the data from the OVS. These two machines are connected with two 10Gbps Ethernet cables. As shown in [45], the native OVS can achieve $\sim$3Gbps switching speed. Thus 10G NIC is sufficient to make it not the bottleneck. We host the packet generator and the data sink on the same machine to facilitate the delay and throughput measurement. The third machine serves as the SDN controller running Ryu [46]. We perform the trace driven evaluation using a CAIDA trace [47] that contains about 30 million packets. The packet trace is replayed using TCPReplay [48] and is fed into the instrumented OVS. The packets are routed and measured by the OVS, and received by the data sink.

4.1 FCAP vs. SMON vs. eBPF

In this section we compare the performance of all the monitoring designs. For FCAP and SMON, both on-path and off-path versions are considered. This measurement is conducted under a

packet rate of 160 Kpps with 1400 hosts being monitored. We examine the performance in terms of forwarding latency incurred in the kernel data forwarding path, kernel memory usage, and measurement accuracy. The results are averaged over 10 runs and reported in Table 1.

TABLE 1: Comparison of different monitoring designs.

Kernel Monitoring Design	*eBPF*	*On-Path*		*Off-Path*	
		SMON	*FCAP*	*SMON*	*FCAP*
Forwarding Latency	344ns	848ns	515ns	182ns	182ns
Measurement Accuracy	100%	$> 99\%$	100%	$> 99\%$	100%
Memory Usage	396KB	96.97KB	149.58KB	2.116MB	2.168MB

The kernel data-path forwarding latency measures the extra delay introduced by the monitoring modules in the kernel datapath. For off-path FCAP and SMON, we only measure the delay introduced by the ring buffer. The processing delay of the actual stats collection by the monitoring module is ignored since they work off-path. We also measure the overall performance in Section 4.3 which shall reflect off-path modules' impact.

As shown in Table 1, in general, on-path FCAP/SMON incur longer delays than their off-path counterparts. It takes $182ns$ to put each packet into the ring buffer for off-path monitoring, while on-path SMON and on-path FCAP incur $848ns$ and $515ns$ of delay, respectively. In addition, the eBPF processing takes $344ns$ for each packet. Apparently, the off-path design is the most efficient among all since it only involves a single memory copy operation. Comparatively, on-path designs consume a significantly amount of CPU cycles from hash computation and counter updates, resulting in much longer processing delays. Among the three on-path designs, eBPF outperforms the other two due to its highly performant underlying implementation. As discussed in Section 3.6, eBPF hashmap size is kept sufficiently large in order to minimize the length of the linked lists. Thereby, the average per-packet latency is considerably smaller than FCAP. Compared to eBPF and FCAP, on-path SMON incurs much long processing delay since sketch encoding requires multiple hash computations and memory access to multiple counters for each incoming packet.

In terms of memory consumption, on-path SMON consumes less memory ($96.97KB$) than on-path FCAP ($149.58KB$) and eBPF ($396KB$). By utilizing sketches, SMON is the most memory-efficient by compressing multiple flow information into a single sketch counter at the cost of slight accuracy degradation. Between FCAP and eBPF, the latter requires more memory usage due to its large hash table size and the memory pre-allocation mechanism. Compared to the on-path designs, off-path FCAP and SMON consume the largest amount of memory since they require a ring buffer to store all incoming packets.

We next examine if the use of ring buffer and sketch reduces the measurement accuracy. As shown in Table 1, the results show that the off-path measurements achieve comparable accuracy as the on-path measurements as long as the ring buffer is sufficiently large to accommodate incoming packets. We find that in order to avoid packet losses, for a packet rate of 160 Kpps, the memory allocated for the ring buffer must be over 2MB. The size of the ring buffer can be configured from the user space through the Netlink interface according to the estimated packet rate and the desired accuracy. The measurement accuracy of SMON is over 99%, which suggests the use of sketches does not lead to large accuracy loss. Moreover, on-path/off-path FCAP and eBPF provide fully accurate measurement results since they both employ linked lists to resolve hash collisions in their implementation.

4.2 Impact of Monitoring Workloads

Here we examine the impact of monitoring workloads on the CPU utilization and memory usage of instrumented OVSes. We vary the number of monitored hosts, i.e. IP addresses, which directly leads to a varying number of monitored micro-flows, as listed in Table 2 and Table 3. We also experiment with two different packets rates, 80 Kpps and 160 Kpps, replayed by TCPReplay to represent different OVS switching workloads.

TABLE 2: Memory usage (MB)(packet rate = 160 Kpps).

#hosts	200	400	600	800	1000	1200	1400	1600
#flows	346	703	1067	1402	1698	2082	2550	3043
off-path SMON	2.039	2.049	2.063	2.079	2.087	2.100	2.116	2.132
off-path FCAP	2.075	2.091	2.107	2.118	2.133	2.149	2.168	2.191
eBPF	0.105	0.105	0.160	0.211	0.262	0.324	0.387	0.449
UMON	4.556							

TABLE 3: Memory usage (MB)(packet rate = 80 Kpps).

#hosts	200	400	600	800	1000	1200	1400	1600
#flows	217	487	641	811	980	1219	1456	1712
off-path SMON	1.347	1.358	1.360	1.366	1.371	1.379	1.387	1.395
off-path FCAP	1.386	1.393	1.402	1.410	1.418	1.426	1.438	1.469
eBPF	0.043	0.063	0.121	0.152	0.184	0.215	0.246	0.281
UMON	1.589							

Figure 8 and Figure 9 depict the CPU utilization overhead caused by the monitoring activities against the number of monitored hosts at two packet rates. The reported results represent the total CPU utilization of all related threads including handlers, revalidators, and new threads created for monitoring purposes. In FCAP and SMON, we create a user-space thread called *collector* to collect the flow stats from the data structures exported from the kernel datapath at fixed time intervals. Moreover, for off-path FCAP/SMON, there is a kernel thread that retrieves packets from the ring buffer. Differently, eBPF monitoring threads are not incorporated into OVS, so we measured their CPU usage separately. In all experiments, the stats collection interval is set to 0.5 second.

In UMON, the flow aggregation functionality is integrated into the existing revalidator threads. CPU utilization of the two on-path designs is not shown in the figure since the implementation of the monitoring modules is similar to their off-path counterparts, resulting in similar CPU utilization.

As illustrated in Figure 8 and Figure 9, the CPU utilization overhead increases as the number of monitored hosts and the packet rate increase. In addition, off-path FCAP incurs the least amount of CPU utilization overhead, while eBPF incurs comparable but slightly larger CPU utilization overhead in the average case. The difference between the two is attributed to the different underlying implementation schemes for the communication between kernel and userspace. As discussed in Section 3, in FCAP and SMON, the kernel/userspace communication is facilitated via the Netlink socket interface, following the same communication mechanism as OVS. Comparatively, with eBPF, at fixed time intervals, the entries are accessed and cleared from the userspace via system calls. Compared to FCAP and eBPF, SMON requires complex sketch decoding operations performed by the *collector* thread, resulting higher CPU utilization. The overhead introduced by UMON is significantly larger than those of off-path FCAP/SMON, which is mainly due to two reasons. First, since UMON installs fine-grained forwarding rules in the kernel flow table, there are more frequent packet misses such that its userspace handler threads are busy with handling upcalls. Second, with a large kernel flow table, the revalidator threads in UMON are also heavily loaded with updating the flow stats into the userspace monitoring table. On the contrary, in off-path FCAP/SMON, the

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3181890

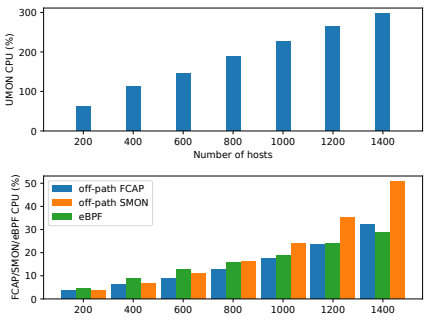


Fig. 8: CPU overhead under various monitoring workloads with packet rate 160 Kpps.

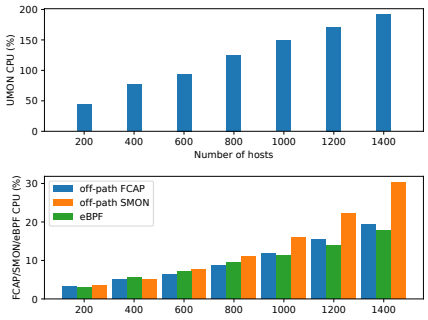


Fig. 9: CPU overhead under various monitoring workloads with packet rate 80 Kpps.

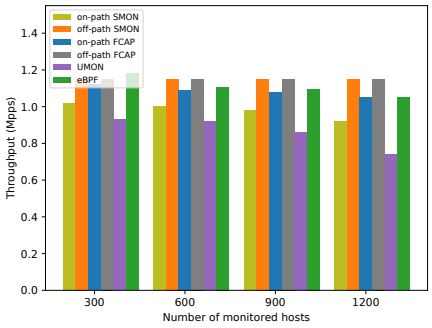


Fig. 10: Throughput(Mpps) of different schemes under various monitoring workloads.

kernel flow table only contains two flow rules in our experimental setup, thus imposing negligible CPU overhead to the userspace handler and revalidator threads. The CPU utilization in these two cases is mainly attributed to the flow stats aggregation performed by our custom *collector* thread.

We next evaluate the memory overhead for monitoring. Since on-path FCAP/SMON use less memory than their off-path counterparts, we focus on the two off-path schemes and the eBPF-based approach and compare them to UMON. The results are shown in Table 2 and Table 3 for two different packet rates. The memory size in the table is the amount of memory used in the kernel for the monitoring purpose. As the memory usage dynamically changes with incoming flows for all schemes, we take the peak usage in comparison. The memory consumption in the off-path FCAP/SMON is caused by two data structures: a kernel ring buffer that caches incoming packets, and the actual data structures (i.e., hash table in FCAP and sketch in SMON) to maintain the flow stats. As previously mentioned, sufficient amount of memory needs to be allocated to the ring buffer in order to avoid packet losses. The experimental results show that off-path FCAP consumes about 3% more memory than off-path SMON.

For eBPF, the memory consumption is mostly incurred by the BPF map to record flow statistics. For the sake of memory pre-allocation, the userspace program needs to specify the approximate maximum number of flows in each time interval at map initialization. Since this number varies across workloads and packet rates, the memory consumption of eBPF also varies accordingly. In Table 1, it indicates that eBPF requires more memory than on-path FCAP, largely due to the underlying eBPF hashmap implementation. UMON uses much larger amount of memory than all other monitoring designs. This is because in order to support comparable measurement accuracy, the kernel flow table in UMON has to maintain an individual forwarding entry for each 6-tuple flow. Consequently, the memory usage greatly exceeds the other solutions.

4.3 Switching Throughput and Latency

To study the throughput and latency of our designs under high packet rates, we use DPDK based packet generator MoonGen [49] for traffic generation and measure the maximum achievable throughput for each measurement framework when there is no packet loss and the monitoring stats are highly accurate. Since only those packets with a match in the monitoring table will be counted towards the hash table/sketch, the ratio of monitored packets to the total number of packets directly affects the throughput of the entire system. To study this impact, we conduct experiment by varying the number of hosts in the kernel space monitoring table. The throughput and latency results under different workloads are shown in Figure 10 and Figure 11, respectively.

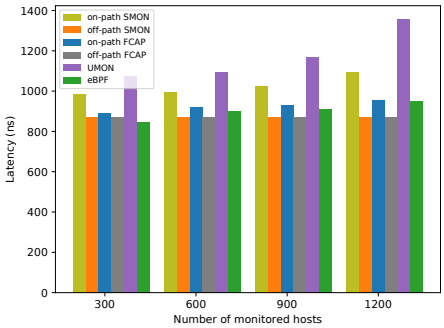


Fig. 11: Average latency(ns) of different schemes under various monitoring workloads.

First and foremost, the switching throughput of UMON is the lowest among all monitoring designs, as can be seen in Figure 10. This could be explained by the fact that UMON follows the traditional design of OVS kernel datapath, which requires the first packet of each new flow to traverse the slow path through the userspace. The userspace of UMON introduces an extra monitoring table in the forwarding pipeline. The forwarding rule and monitoring rule are combined to generate a more fine-grained flow that would be installed into the kernel cache table. Due to this design, an enormous amount of flow cache misses are introduced when we need to collect the flow statistics for each 6-tuple flow. On-path FCAP/SMON both outperform UMON but lag behind compared with eBPF and their off-path counterparts. As explained previously, this is because the monitoring modules (filtering, stats collection) are placed in the switch forwarding path. But on-path FCAP achieves higher throughput than on-path SMON, since the latter requires complex sketch encoding operations, while the former implements more light-weight hash tables.

On the other hand, off-path FCAP/SMON achieves higher throughput because monitoring logic is decoupled from forwarding and the overhead only involves memory copies from the ring buffer. This also explains why FCAP and SMON achieve the same throughput in the off-path paradigm. Finally yet importantly, despite the fact that eBPF and on-path FCAP both are on-path and implemented based on hash tables, eBPF achieves slightly better performance than on-path FCAP as a result of its optimized hash table implementation within the Linux kernel. Due to the large size of the hash table, it has a shorter linked list for each bucket in the hash table in the average case, resulting in higher efficiency for hash lookups/updates. Overall, from the network performance perspective, off-path FCAP/SMON is a preferable solution among

all the designs.

With the same experiment setup, we also measure the average switching latency and investigate the impact from various monitoring workloads. Figure 11 reveals that off-path designs incur minimum delay, while UMON significantly degrades the forwarding efficiency. Further, the switching performance worsens along the increase of the workloads. Likewise, eBPF yields smaller latency than on-path FCAP/SMON for the same reason as throughput. Off-path options outperform all other alternatives with more memory consumption. More specifically, the memory required (15MB ring buffer) scaled linearly (7.5x) with the increase in throughput to 1.2 Mpps from our earlier experiments at 160 Kpps. We conclude roughly one MB is needed for each 80 Kpps of throughput.

5 Discussion

TABLE 4: Comparison of different frameworks.

Designs	eBPF	*On-Path*		*Off-Path*		UMON
		SMON	*FCAP*	*SMON*	*FCAP*	
CPU Overhead	low	moderate	low	moderate	low	high
Memory Consumption	low	low	low	moderate	moderate	high
Measurement Accuracy	precise	high	precise	high	precise	precise
Forwarding Latency	high	high	high	low	low	high
Implementation Complexity	low	high	high	high	high	low

Based on the evaluation results, it is clear that building monitoring capabilities into software entities on the edge servers, either software routers or independent monitoring modules, is feasible without significantly sacrificing performance overhead. Nevertheless, each design bears its own pros and cons. Although it is a seemingly daunting task to determine which design achieves the overall best performance, we have attained several insights, as sketched in Table 4, into the design of software-based measurement framework.

First, UMON requires the least implementation efforts with no modifications to the OVS kernel datapath. However, it derives fine-grained forwarding rules by combining the forwarding and monitoring functionality, which leads to the heaviest CPU load in the user space. In addition, it necessitates significantly more memory consumption.

Second, FCAP outperforms both SMON and eBPF in all aspects except that it needs to instrument OVS kernel code. The off-path FCAP is a particularly better option than its on-path version on servers with abundant memory resources. Among all other alternatives, it introduces the minimal impact on OVS throughput and latency.

Third, SMON is the most memory-efficient option, owing to the strong space-efficiency of bloom filter sketches. Although sketches have proved to be efficient in memory-constrained hardware devices, it turns out concerns have shifted away for building monitoring logic into the end hosts. SMON has a higher demand on CPU, thus making it less ideal for servers with insufficient CPU resource or fierce CPU competition.

Fourth, eBPF-based monitoring design achieves comparable performance with on-path FCAP from the perspective of CPU utilization, measurement accuracy and switching performance. In the meantime, it requires minimal maintenance efforts since it executes independently of OVS and can be configured and updated without interrupting the system operations. However, it requires more memory than on-path FCAP due to the underlying hash table implementation. Nevertheless, its overall performance falls behind the off-path designs since the eBPF program lies in the packet processing path.

Since all these schemes are designed for software defined measurement, we can see that: In terms of the switching throughput and latency, off-path designs offer the best performance, regardless of the monitoring algorithms, since throughput and latency are only affected by the ring buffer write operations. Without ring buffers, on-path FCAP and eBPF achieve comparable throughput/latency, which demonstrate that hash tables could suffice in a software monitoring system;

In terms of implementation complexity and portability, eBPF is the best since monitoring programs could be loaded and updated at runtime, while OVS-embedded designs require to recompile and reinstall the OVS binary whenever there is an update.

By comparing the results across all the experiments, we observe that

- by removing the monitoring functionality from the kernel forwarding path, off-path schemes can achieve better switching performance than on-path schemes in terms of network throughput and latency, while achieving the same measurement accuracy at the cost of higher memory consumption.
- in the design of flow stats collection module, our results demonstrate that hash table is a more efficient solution compared to sketch due to its lower computational cost, which is a major factor in the evaluation of CPU utilization.

While there is no scheme that outperforms in all aspects, building monitoring frameworks into edge servers requires us to carefully examine the interplay of multiple key factors, including memory and CPU consumption, measurement accuracy, impact on switching throughput and latency, maintenance complexity, and so on so forth. Our empirical study demonstrates that hash tables are a better fit than sketches in a software monitoring framework despite of the strong memory-efficiency and wide utilisation of the sketches in hardware environment. The difference lies in the fact that hardware devices have much tighter memory constraints than commodity servers. In terms of the placement of monitoring functionalities, off-path outperforms on-path since the latter introduces noticeable latency to the traditional packet forwarding pipeline. Such impact becomes even more evident under high packet rate.

6 Conclusion

In recent years, cloud data centers have undergone through a underlying network diagram change from the traditional network to software defined networking. Software defined networking has provided more flexibility for network measurement and monitoring, and enabled software defined measurement. However, properly achieving timely and accurate measurement results while minimizing the resource consumption in data centers remains a critical challenge. But little is known in the current literature. In this paper, we have investigated various design options to implement software based measurement using Open vSwitches and eBPF-based solution. Enabling monitoring capability on the widely deployed OVSes in data centers requires us to take into a number of factors into consideration during design and implementation, including resource consumption, impact on the forwarding, measurement accuracy, implementation complexity, portability etc. In this study, we have empirically explored the various trade-offs among these factors by designing, implementing, and evaluating five different monitoring schemes and quantitatively shown their advantages and disadvantages. These results provide insightful guidelines for conducting network traffic measurement on the OVS as well as software defined measurement in data centers in general. A preliminary version of this paper appears as “Instrumenting Open vSwitch with monitoring capabilities: designs and challenges” in

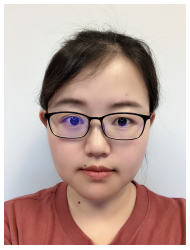
SOSR2018 [35].

ACKNOWLEDGEMENT

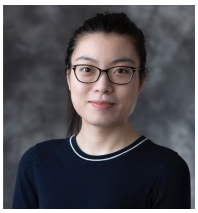
We appreciate the constructive comments from the reviewers. This work was supported in part by the NSF grants CNS-2007153, CNS-2008468, a Commonwealth Cyber Initiative grant and a Google Faculty Research Award.

REFERENCES

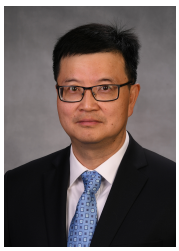
[1] B. Han *et al.*, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, 2015.
[2] M. Yu *et al.*, "Software defined traffic measurement with opensketch." in *USENIX NSDI*, 2013.
[3] M. Moshref *et al.*, "Dream: dynamic resource allocation for software-defined measurement," in *ACM CCR*, 2014.
[4] ——, "Scream: Sketch resource allocation for software-defined measurement," in *ACM CoNEXT*, 2015.
[5] Y. Li *et al.*, "Flowradar: A better netflow for data centers." in *USENIX NSDI*, 2016.
[6] V. Sivaraman *et al.*, "Heavy-hitter detection entirely in the data plane," in *ACM SOSR*, 2017.
[7] R. Ben-Basat *et al.*, "Efficient measurement on programmable switches using probabilistic recirculation," in *IEEE ICNP*, 2018.
[8] R. B. Basat *et al.*, "Memento: Making sliding windows efficient for heavy hitters," in *ACM CoNEXT*, 2018.
[9] A. Wang *et al.*, "Umon: Flexible and fine grained traffic monitoring in open vswitch," in *ACM CoNEXT*, 2015.
[10] M. Moshref *et al.*, "Trumpet: Timely and precise triggers in data centers," in *ACM SIGCOMM*, 2016.
[11] Q. Huang *et al.*, "Sketchvisor: Robust network measurement for software packet processing," in *ACM SIGCOMM*, 2017.
[12] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *ACM SIGCOMM*, 2018.
[13] Z. Liu *et al.*, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *ACM SIGCOMM*, 2019.
[14] T. Yang *et al.*, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, 2019.
[15] "Open vSwitch," http://openvswitch.org/, 2017.
[16] M. T. Goodrich *et al.*, "Invertible bloom lookup tables," in *IEEE Allerton Conference*, 2011.
[17] M. Fleming, "A thorough introduction to ebpf," 2017.
[18] B. Claise, "Cisco systems netflow services export version 9," 2004.
[19] S. Panchen *et al.*, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," 2001.
[20] B. Claise, "Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information," 2008.
[21] Juniper, "Juniper flow monitoring," 2011.
[22] Alcatel-Lucent, "Cflowd," https://tinyurl.com/2p9c24dp, 2017.
[23] HP, "Hp netstream monitoring module," 2012.
[24] Z. Liu *et al.*, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *ACM SIGCOMM*, 2016.
[25] A. Metwally *et al.*, "Efficient computation of frequent and top-k elements in data streams," in *Springer ICDT*, 2005.
[26] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM CCR*, 2013.
[27] M. Datar *et al.*, "Maintaining stream statistics over sliding windows," *SIAM journal on computing*, 2002.
[28] R. Ben-Basat *et al.*, "Heavy hitters in streams and sliding windows," in *IEEE INFOCOM*, 2016.
[29] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *ACM SIGCOMM*, 2017.
[30] A. Gupta *et al.*, "Sonata: Query-driven streaming network telemetry," in *ACM SIGCOMM*, 2018.
[31] E. J. Candès *et al.*, "Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information," *IEEE Trans. Inf. Theory*, 2006.
[32] M. Abranches *et al.*, "Efficient network monitoring applications in the kernel with ebpf and xdp," in *IEEE NFV-SDN*, 2021.
[33] C. Cassagnes *et al.*, "The rise of ebpf for non-intrusive performance monitoring," in *IEEE/IFIP NOMS*, 2020.
[34] S. Miano *et al.*, "A framework for ebpf-based network functions in an era of microservices," *IEEE TNSM*, 2021.
[35] Z. Zha *et al.*, "Instrumenting open vswitch with monitoring capabilities: Designs and challenges," in *ACM SOSR*, 2018.
[36] B. Pfaff *et al.*, "The design and implementation of open vswitch." in *USENIX NSDI*, 2015.
[37] X. Li *et al.*, "Detection and identification of network anomalies using sketch subspaces," in *ACM IMC*, 2006.
[38] R. Schweller *et al.*, "Reversible sketches for efficient and accurate change detection over network data streams," in *ACM IMC*, 2004.
[39] G. Cormode *et al.*, "Finding hierarchical heavy hitters in data streams," in *VLDB*, 2003.
[40] Y. Zhang, "An adaptive flow counting method for anomaly detection in sdn," in *ACM CoNEXT*, 2013.
[41] G. Cormode *et al.*, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, 2005.
[42] P. Flajolet *et al.*, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, 1985.
[43] N. Alon *et al.*, "The space complexity of approximating the frequency moments," *Journal of Computer and system sciences*, 1999.
[44] T. Høiland-Jørgensen *et al.*, "The express data path: Fast programmable packet processing in the operating system kernel," in *ACM CoNEXT*, 2018.
[45] P. Emmerich *et al.*, "Performance characteristics of virtual switching," in *IEEE CloudNet*, 2014.
[46] "Ryu SDN controller," https://osrg.github.io/ryu/, 2017.
[47] "Caida internet traces 2012," https://tinyurl.com/5fcwerv7, 2012.
[48] "Tcpreplay," http://tcpreplay.appneta.com/, 2017.
[49] P. Emmerich *et al.*, "Moongen: A scriptable high-speed packet generator," in *ACM IMC*, 2015.



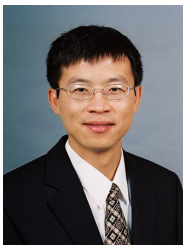
Zili Zha is a PhD student in the Computer Science Department at George Mason University. She received her MS degree from the College of William and Mary and BS degree from University of Science and Technology of China. Her research interests include data center traffic measurement, network programmability and security.



An Wang is currently an assistant professor in the Computer and Data Science Department of Case Western Reserve University. Before joining Case, she received her Ph.D. in Computer Science from George Mason University in 2018. Her research interests lie in the areas of security for networked systems and network virtualization, focusing on Software-Defined Networking (SDN) and cloud systems, and large-scale network attacks. She is also interested in the security and privacy issues in the Internet-of-Things (IoT) environment.



Yang Guo is a computer scientist in the Computer Security Division, National Institute of Standards and Technology (NIST). His research interests span broadly over the distributed systems and networking, with a focus on Software Defined Networking (SDN), Cybersecurity, and AI and Machine Learning. Before joining the NIST, he was a member of technical staff with Bell Labs (Crawford Hill, NJ) from 2010 to 2015, and was a Principal Scientist at Technicolor (formerly Thomson) Corporate Research from 2005 to 2010. He received multiple NIST Information Technology Lab's Building the Future awards, Bell Labs' team work award, and was on Technicolor's Fellowship Network as a technical leader.



Songqing Chen is currently a professor of computer science at George Mason University. His research interests mainly focus on design, analysis, and implementation of algorithms and experimental systems in the distributed and networking environment, particularly in the areas of Internet content delivery systems, Internet measurement and modeling, mobile and cloud computing, network and system security, and distributed system. He is a recipient of the US NSF CAREER Award and the AFOSR YIP Award. Currently, he serves as the chair of IEEE Technical Committee on the Internet (TCI), and on the editorial boards of IEEE TPDS, IEEE IC, IEEE IoT-J and ACM TOIT. He also serves in various capacities in conference organization committees, and most recently as the General Chair of IEEE ICDCS 2021.