One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon

Zaoxing Liu†, Antonis Manousis⋆, Gregory Vorsanger†, Vyas Sekar⋆, Vladimir Braverman†
† Johns Hopkins University ⋆ Carnegie Mellon University

ABSTRACT

Network management requires accurate estimates of metrics for many applications including traffic engineering (e.g., heavy hitters), anomaly detection (e.g., entropy of source addresses), and security (e.g., DDoS detection). Obtaining accurate estimates given router CPU and memory constraints is a challenging problem. Existing approaches fall in one of two undesirable extremes: (1) low fidelity general-purpose approaches such as sampling, or (2) high fidelity but complex algorithms customized to specific application-level metrics. Ideally, a solution should be both general (i.e., supports many applications) and provide accuracy comparable to custom algorithms. This paper presents *UnivMon*, a framework for flow monitoring which leverages recent theoretical advances and demonstrates that it is possible to achieve both generality and high accuracy. UnivMon uses an application-agnostic data plane monitoring primitive; different (and possibly unforeseen) estimation algorithms run in the control plane, and use the statistics from the data plane to compute application-level metrics. We present a proof-of-concept implementation of UnivMon using P4 and develop simple coordination techniques to provide a "one-big-switch" abstraction for network-wide monitoring. We evaluate the effectiveness of UnivMon using a range of trace-driven evaluations and show that it offers comparable (and sometimes better) accuracy relative to custom sketching solutions across a range of monitoring tasks.

CCS Concepts

•Networks → Network monitoring; Network measurement;

Keywords

Flow Monitoring, Sketching, Streaming Algorithms

SIGCOMM '16, August 22–26, 2016, Florianopolis, Brazil
© 2016 ACM. ISBN 978-1-4503-4193-6/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2934872.2934906

1 Introduction

Network management is multi-faceted and encompasses a range of tasks including traffic engineering [11, 32], attack and anomaly detection [49], and forensic analysis [46]. Each such management task requires accurate and timely statistics on different application-level metrics of interest; e.g., the flow size distribution [37], heavy hitters [10], entropy measures [38, 50], or detecting changes in traffic patterns [44].

At a high level, there are two classes of techniques to estimate these metrics of interest. The first class of approaches relies on *generic flow monitoring*, typically with some form of packet sampling (e.g., NetFlow [25]). While generic flow monitoring is good for coarse-grained visibility, prior work has shown that it provides low accuracy for more fine-grained metrics [30, 31, 43]. These well-known limitations of sampling motivated an alternative class of techniques based on *sketching* or *streaming* algorithms. Here, custom online algorithms and data structures are designed for specific metrics of interest that can yield provable resource-accuracy trade-offs (e.g., [17, 18, 20, 31, 36, 38, 43]).

While the body of work in data streaming and sketching has made significant contributions, we argue that this trajectory of crafting special-purpose algorithms is untenable in the long term. As the number of monitoring tasks grows, this entails significant investment in algorithm design and hardware support for new metrics of interest. While recent tools like OpenSketch [47] and SCREAM [41] provide libraries to reduce the implementation effort and offer efficient resource allocation, they do not address the fundamental need to design and operate new custom sketches for each task. Furthermore, at any given point in time the data plane resources have to be committed (a priori) to a specific set of metrics to monitor and will have fundamental blind spots for other metrics that are not currently being tracked.

Ideally, we want a monitoring framework that offers both *generality* by delaying the binding to specific applications of interest but at the same time provides the required *fidelity* for estimating these metrics. Achieving generality and high fidelity simultaneously has been an elusive goal both in theory [33] (Question 24) as well as in practice [45].

In this paper, we present the *UnivMon* (short for Universal Monitoring) framework that can simultaneously achieve both generality and high fidelity across a broad spectrum of monitoring tasks [31, 36, 38, 51]. UnivMon builds on and extends recent theoretical advances in *universal streaming*, where a single universal sketch is shown to be provably accurate for estimating a large class of functions [15, 16, 19, 21, 22]. In essence, this generality can enable us to delay the binding of the data plane resources to specific monitoring tasks, while still providing accuracy that is comparable (if not better) than running custom sketches using similar resources.

While our previous position paper suggested the promise of universal streaming [39], it fell short of answering several practical challenges, which we address in this paper. First, we demonstrate a concrete switch-level realization of UnivMon using P4 [12], and discuss key implementation challenges in realizing UnivMon. Second, prior work only focused on a single switch running UnivMon for a specific feature (e.g., source addresses) of interest, whereas in practice network operators want a panoramic view across multiple features and across traffic belonging to multiple origin-destination pairs. To this end, we develop lightweight-yet-effective coordination techniques that enable UnivMon to effectively provide a "one big switch" abstraction for network-wide monitoring [34], while carefully balancing the monitoring load across network locations.

We evaluate UnivMon using a range of traces [1, 2] and operating regimes and compare it to state-of-art custom sketching solutions based on OpenSketch [47]. We find that for a single network element, UnivMon achieves comparable accuracy, with an observed error gap $\leq 3.6\%$ and average error gap $\leq 1\%$. Furthermore, UnivMon outperforms OpenSketch in the case of a growing application portfolio. In a network-wide setting, our coordination techniques can reduce the memory consumption and communication with the control plane by up to three orders of magnitude.

Contributions and roadmap: In summary, this paper makes the following contributions:

- A practical architecture which translates recent theoretical advances to serve as the basis for a general-yet-accurate monitoring framework (§3, §4).
- An effective network-wide monitoring approach that provides a one-big switch abstraction (§5).
- A viable implementation using emerging programmable switch architectures (§6).
- A trace-driven analysis which shows that UnivMon provides comparable accuracy and space requirements compared to custom sketches (§7).

We begin with background and related work in the next section. We highlight outstanding issues and conclude in §8.

2 Background and Related Work

Many network monitoring and management applications depend on sampled flow measurements from routers (e.g., NetFlow or sFlow). While these are useful for coarse-grained metrics (e.g., total volume) they do not provide good fidelity unless these are run at very high sampling rates, which is undesirable due to compute and memory overhead.

This inadequacy of packet sampling has inspired a large body of work in data streaming or sketching. This derives from a rich literature in the theory community on streaming algorithms starting with the seminal "AMS" paper [9] and has since been an active area of research (e.g., [19, 24, 26, 28]). At the high level, the problem they address is as follows: Given an input sequence of items, the algorithm is allowed to make a single or constant number of passes over the data stream while using sub-linear (usually poly-logarithmic) space compared to the size of the data set and the size of the dictionary. The algorithm then provides an approximate estimate of the desired statistical property of the stream (e.g., mean, median, frequency moments). Streaming is a natural fit for network monitoring and has been applied to several tasks including heavy hitter detection [31], entropy estimation [38], change detection [36], among others.

A key limitation that has stymied the practical adoption of streaming approaches is that the algorithms and data structures are tightly coupled to the intended metric of interest. This forces vendors to invest time and effort in building specialized algorithms, data structures, and corresponding hardware without knowing if these will be useful for their customers. Given the limited resources available on network routers and business concerns, it is difficult to support a wide spectrum of monitoring tasks in the long term. Moreover, at any instant the data plane resources are committed beforehand to the application-level metrics and other metrics that may be required in the future (e.g., as administrators start some diagnostic tasks and require additional statistics) will fundamentally not be available.

The efforts closest in spirit to our UnivMon vision is the minimalist monitoring work of Sekar et al. [45] and OpenSketch by Yu et al., [47]. Sekar et al. showed empirically that flow sampling and sample-and-hold [31] can provide comparable accuracy to sketching when equipped with similar resources. However, this work offers no analytical basis for this observation and does not provide guidelines on what metrics are amenable to this approach.

OpenSketch [47] addresses an orthogonal problem of making it easier to implement sketches. Here, the router is equipped with a library of predefined functions in hardware (e.g., hash-maps or count-min sketches [26]) and the controller can reprogram these as needed for different tasks. While OpenSketch reduces the implementation burden, it still faces key shortcomings. First, because the switches are programmed to monitor a specific set of metrics, there will be a fundamental lack of visibility into other metrics for which data plane resources have not been committed, even if the library of functions supports those tasks. Second, to monitor a portfolio of tasks, the data plane will need to run many concurrent sketch instances, which increases resource requirements.

In summary, prior work presents a fundamental dichotomy: generic approaches that offer poor fidelity and are hard to reason about analytically vs. sketch-based approaches that offer good guarantees but are practically intractable given the wide range of monitoring tasks of interest.

Our recent position paper makes a case for a "RISC" approach for monitoring [39], highlighting the promise of recent theoretical advances in universal streaming [19,21]. How-

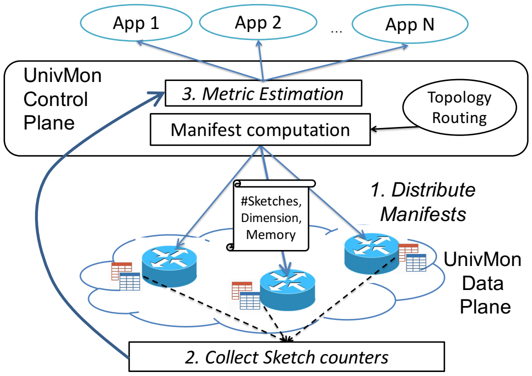


Figure 1: Overview of UnivMon: The data plane nodes perform the monitoring operations and report sketch summaries to the control plane which calculates application-specific metric estimates.

ever, this prior work fails to address several key practical challenges. First, it does not discuss how these primitives can actually be mapped into switch processing pipelines. In fact, we observe that the data-control plane split that they suggest is impractical to realize as they require expensive sorting/sifting primitives (see §6). Second, this prior work takes a narrow single-switch perspective. As we show later, naively extending this to a network-wide context can result in inefficient use of compute resources on switches and/or result in inaccurate estimates (see §5). This paper develops network-wide coordination schemes and demonstrate an implementation in P4 [12]. Further, we show the fidelity of UnivMon on a broader set of traces and metrics.

3 UnivMon architecture

In this section, we provide a high-level overview of the UnivMon framework. We begin by highlighting the end-to-end workflow to show the interfaces between (a) the UnivMon control plane and the management applications and (b) between the UnivMon control and data plane components. We discuss the key technical requirements that UnivMon needs to satisfy and why these are challenging. Then, we briefly give an overview of the control and data plane design to set up the context for the detailed design in the following sections.[1]

Figure 1 shows an end-to-end view of the UnivMon framework. The UnivMon data plane nodes run general-purpose monitoring primitives that process the incoming stream of packets it sees and maintains a set of counter data structures associated with this stream. The UnivMon control plane assigns monitoring responsibilities across the nodes. It periodically collects statistics from the data plane, and estimates the various application-level metrics of interest.

Requirements and challenges: There are three natural requirements that UnivMon should satisfy:

- **[R1.]** *Fidelity for a broad spectrum of applications:* Ideally UnivMon should require no prior knowledge of the set of metrics to be estimated, and yet offer strong guarantees on accuracy.
- **[R2.]** *One-big-switch abstraction for monitoring:* There may be several network-wide management tasks interested in measuring different dimensions of traffic; e.g., source IPs, destination ports, IP 5-tuples. UnivMon should provide a "one big switch" abstraction for monitoring to the management applications running atop UnivMon, so that the estimates appear as if all the traffic entering the network was monitored at a giant switch [34].
- **[R3.]** *Feasible implementation roadmap:* While pure software solutions (e.g., Open vSwitch [42]) may be valuable in many deployments, for broader adoption and performance requirements, the UnivMon primitives used to achieve [R1] and [R2] must have a viable implementation in (emerging) switch hardware [12, 13].

Given the trajectory of prior efforts that offer high generality and low fidelity (e.g, packet sampling) vs. low generality and high fidelity (e.g., custom sketches), [R1] may appear infeasible. To achieve [R2], we observe that if each router acts on the traffic it observes independently, it can become difficult to combine the measurements and/or lead to significant imbalance in the load across routers. Finally, for [R3], we note that even emerging flexible switches [3, 12, 13] have constraints on the types of operations that they can support.

Approach Overview: Next, we briefly outline how the UnivMon control and data plane designs address these key requirements and challenges:

- *UnivMon data plane:* The UnivMon plane uses sketching primitives based on recent theoretical work on *universal streaming* [19, 21]. By design, these so-called universal sketches require no prior knowledge of the metrics to be estimated. More specifically, as long as these metrics satisfy a series of statistical properties discussed in detail in §4, we can prove theoretical guarantees on the memory-accuracy tradeoff for estimating these metrics in the control plane.
- *UnivMon control plane:* Given that the data plane supports universal streaming, the control plane needs no additional capabilities w.r.t. [R1] once it collects the sketch information from the router. It runs simple estimation algorithms for every management application of interest as we discuss in §4 and provides simple APIs and libraries for applications to run estimation queries on the collected counters. To address [R2], the UnivMon control plane generates *sketching manifests* that specify the monitoring responsibility of each router. These manifests specify the set of universal sketch instances for different dimensions of interest (e.g., for source IPs, for 5-tuples) that each router needs to maintain for different origin-destination (OD) pair paths that it lies on. This assignment takes into account the network topology and routing policies and knowledge of the hardware resource constraints of its network elements.

In the following sections, we begin by providing the background on *universal streaming* that forms the theoretical ba-

[1] We use the terms routers, switches, and nodes interchangeably.

sis for UnivMon. Then, in §5, we describe the network-wide coordination problem that the UnivMon control plane solves. In §6, we show how we implement this design in P4 [7, 12].

4 Theoretical Foundations of UnivMon

In this section, we first describe the theoretical reasoning behind universal streaming and the class of supported functions [19, 21]. Then, we present and explain the underlying algorithms from universal streaming which serve as a basis for UnivMon. We also show how several canonical network monitoring tasks are amenable to this approach.

4.1 Theory of Universal Sketching

For the following discussion, we consider an abstract stream $D(m, n)$ of length m with n unique elements. Let f_i denote the frequency of the i-th unique element in the stream.

The intellectual foundations of many streaming algorithms can be traced back to the celebrated lemma by Johnson and Lindenstrauss [27]. This shows that N points in Euclidean space can be embedded into another Euclidean space with an exponentially smaller dimension while approximately preserving the pairwise distance between the points. Alon, Matias, and Szegedy used a variant of the Johnson-Lindenstrauss lemma to approximately compute the second moment of the frequency vector = $\sum_i f_i^2$ (or the L_2 norm = $\sqrt{\sum_i f_i^2}$) in the streaming model [9], using a small (polylogarithmic) amount of memory. The main question that *universal streaming* seeks to answer is whether such methods can be extended to more general statistics of the form $\sum g(f_i)$ for an arbitrary function g. We refer to this statistic as the $G\text{-}sum$.

Class of *Stream-PolyLog* Functions: Informally, streaming algorithms which have polylogarithmic space complexity, are known to exist for $G\text{-}sum$ functions, where g is monotonic and upper bounded by the function $O(f_i^2)$ [14, 19].[2] Note that this only guarantees that *some* (possibly custom) sketching algorithm exists if $G\text{-}sum \in$ *Stream-PolyLog* and does not argue that a single "universal" sketch can compute all such $G\text{-}sum$s.

Intuition Behind Universality: The surprising recent theoretical result of universal sketches is that for any function $g()$ where $G\text{-}sum$ belongs to the class *Stream-PolyLog* defined above can now be computed by using a *single universal sketch.*

The intuition behind universality stems from the following argument about heavy hitters in the stream. Informally, item i is a heavy hitter w.r.t. g if changing its frequency f_i significantly affects the $G\text{-}sum$ value as well. For instance, consider the frequency vector $(\sqrt{n}, 1, 1, \ldots, 1)$ of size n; here the first item is a L_2 heavy hitter since its frequency is a large fraction of the L_2 norm of the frequency vector. For function g, let $G\text{-}core$ be the set containing g-heavy elements. g-heavy elements can be defined as, for $0 < \gamma < 1$, any element $i \in [n]$ such that $g(f_i) > \gamma \sum_j g(f_j)$.

Now, let us consider two cases:

1. There is one sufficiently large g-heavy hitter in the stream: If the frequency vector has one (sufficiently) large heavy hitter, then most of mass is concentrated in this value. Now, it can be shown that a heavy hitter for the L_2 norm of the frequency vector is also a heavy hitter for computable g [14, 19]. Thus, to compute $G\text{-}core$, we can simply find L_2 heavy hitters (L2-HH) using some known techniques (e.g., [9, 24]) and use it to estimate $G\text{-}sum$.

2. There is no single g-heavy hitter in the stream and no single element contributes significantly to the $G\text{-}sum$:
When there is no single large heavy hitter, it can be shown that $G\text{-}sum$ can be approximated w.h.p. by finding heavy hitters on a series of sampled *substreams* of increasingly smaller size. The exact details are beyond the scope of this paper [19] but the main intuition comes from tail bounds (Chernoff/Hoeffding). Each substream is defined recursively by the substream before it, and is created by sampling the previous frequency vector by replacing each coordinate of the frequency vector with a zero value with probability 0.5. Repeating this procedure k times reduces the dimensionality of the problem by a factor of 2^k. Then, summing across heavy hitters of all these recursively defined vectors, we create a single "recursive sketch" which gives a good estimate of $G\text{-}sum$ [21].

4.2 Algorithms for Universal Sketching

Using the intuition from the two cases described above, we now have the following universal sketch construction using an online sketching stage and an offline estimation stage. The proof of the theorems governing the behavior of these algorithms is outside the scope of this paper and we refer readers to the previous work of Braverman et al [19, 21]. In this section, we focus on providing a conceptual view of the universal sketching primitives. As we will discuss later, the actual data plane and control plane realization will be slightly different to accommodate switch hardware constraints (see §6).

In the online stage, as described in Algorithm 1, we maintain $\log(n)$ parallel copies of a "L_2-heavy hitter" (L2-HH) sketch (e.g., [24]), one for each substream as described in case 2 above. For the j^{th} parallel instance, the algorithm processes each incoming packet 5-tuple and uses an array of j pairwise independent hash functions $h_i : [n] \to \{0, 1\}$ to decide whether or not to sample the tuple. When 5-tuple tup arrives in the stream, if for all h_1 to h_j, $h_i(tup) = 1$, then the tuple is added to D_j, the sampled substream. Then, for substream D_j, we run an instance of L2-HH as shown in Algorithm 1, and visualized in Figure 2. Each L2-HH instance outputs Q_j that contains L_2 heavy hitters and their estimated counts from D_j. This creates substreams of decreasing lengths as the j-th instance is expected to have all of the hash functions agree to sample half as often as the $(j-1)$-th instance. This data structure is all that is required

[2] This is an informal explanation; the precise characterization is more technically involved and can be found in [19]. While streaming algorithms are also known for $G\text{-}sum$ when its g grows monotonically faster than f_i^2 [17] they cannot be computed in polylogarithmic space due to the lower bound $\Omega(n^{1-2/k})$ where $k > 2$ [23].

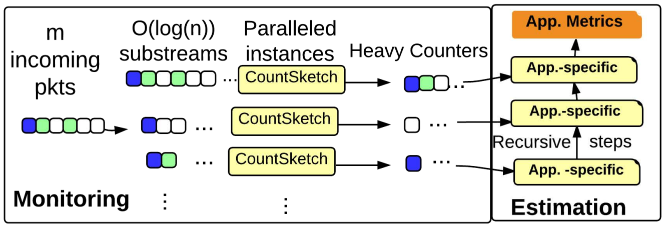


Figure 2: High-level view of universal sketch

```
Algorithm 1 UnivMon Online Sketching Step
Input: Packet stream $D(m, n) = \{a_1, a_2, \ldots, a_m\}$
    1. Generate $\log(n)$ pairwise independent hash functions $h_1 \ldots h_{\log(n)} : [n] \to \{0, 1\}$.
    2. Run L2-HH sketch on D and maintain HH set $Q_0$.
    3. For $j = 1$ to $\log(n)$, in parallel:
        (a) when a packet $a_i$ in $D$ arrives, if all $h_1(a_i) \times h_2(a_i) \cdots \times h_j(a_i) = 1$, sample and add $a_i$ to sampled substream $D_j$.[3]
        (b) Run L2-HH sketch on $D_j$ and maintain heavy hitters $Q_j$
Output: $Q = \{Q_0, \ldots, Q_{\log(n)}\}$
```

for the online portion of our approach.

In the offline stage, we use Algorithm 2 to combine the results of the parallel copies of Algorithm 1 to estimate different *G-sum* functions of interest. This method is based on the Recursive Sum Algorithm from [21]. The input to this algorithm is the output of Algorithm 1; i.e., a set of $\{Q_j\}$ buckets maintained by the L2-HH sketch from parallel instance j. Let $w_j(i)$ be the counter of the i-th bucket (heavy hitter) in Q_j. $h_j(i)$ is the hash of the value of the i-th bucket in Q_j where h_j is the hash function described in Algorithm 1 Step 1. It can be shown that the output of Algorithm 2 is an unbiased estimator of *G-sum* [19,21]. In this algorithm, each Y is recursively defined, where Y_j is function g applied to each bucket of Q_j, the L2-HH sketch for substream D_j, and the sum taken on the value of those buckets and all $Y_{j'}, j' > j$. Note that $Q_{\log(n)}$ is the set of heavy hitters from the sparsest substream $D_{\log(n)}$ in Algorithm 1, and we begin by computing $Y_{\log(n)}$. Thus, Y_0 can be viewed as computing *G-sum* in parts using these sampled streams.

The key observation here is that the online component, Algorithm 1, which will run in the UnivMon data plane is *agnostic* to the specific choice of g in the offline stage. This is in stark contrast to custom sketches where the online and offline stages are both tightly coupled to the specific statistic we want to compute.

4.3 Application to Network Monitoring

As discussed earlier, if a function *G-sum* ∈ *Stream-PolyLog*, then it is amenable to estimation via the universal sketch. Next, we show that a range of network measurement tasks can be formulated via a suitable *G-sum* ∈ *Stream-PolyLog*.

```
Algorithm 2 UnivMon Offline Estimation Algorithm
Input: Set of heavy hitters $Q = \{Q_0, \ldots, Q_{\log(n)}\}$
    1. For $j = 0 \ldots \log(n)$, call g() on all counters $w_j(i)$ in $Q_j$. After $g()$, the $i$-th entry in $Q_j$ is $g(w_j(i))$.
    2. Compute $Y_{\log(n)} = \sum_i g(w_{\log(n)}(i))$.
    3. For each $j$ from $\log(n) - 1$ to 0, compute:
        $Y_j$=$2Y_{j+1}$+$\sum_{i \in Q_j}$(1-$2h_{j+1}(i)$) $g(w_j(i))$
Output: $Y_0$
```

For the following discussion, we consider network traffic as a stream $D(n, m)$ with m packets and at most n unique flows. When referring to the definitions of Heavy Hitters, note that L_2 heavy hitters are a stronger notion that subsumes L_1 heavy hitters.

Heavy Hitters: To detect heavy hitters in the network traffic, our goal is to identify the flows that consume more than a fraction γ of the total capacity [31]. Consider a function $g(x) = x$ such that the corresponding *G-core* outputs a list of heavy hitters with$(1 \pm \epsilon)$-approximation of their frequencies. For this case, these heavy hitters are L_1-heavy hitters and $g(x)$ is upperbounded by x^2. Thus we have an algorithm that provides *G-core*. This is technically a special case of the universal sketch; we are not ever computing a *G-sum* function and using *G-core* directly in all cases.

DDoS Victim Detection: Suppose we want to identify if a host X is experiencing a Distributed Denial of Service (DDoS) attack. We can do so using sketching by checking if more than k unique flows from different sources are communication with X [47]. To show that the simple DDoS victim detection problem is solvable by the universal sketch, consider a function g that $g(x) = x^0$ and $g(0) = 0$. Here g is upper bounded by $f(x) = x^2$ and sketches already exist to solve this exact problem. Thus, we know *G-sum* is in *Stream-PolyLog* and we approximate *G-sum* in polylogarithmic space using the universal sketch. In terms of interpreting the results of this measurement, if *G-sum* is estimated to be larger than k, a specific host is a potential DDoS victim.

Change Detection: Change detection is the process of identifying flows that contribute the most to traffic change over two consecutive time intervals. As this computation takes place in the control plane, we can store the output of the universal sketches from multiple intervals without impacting online performance. Consider two adjacent time intervals t_A and t_B. If the volume for a flow x in interval t_A is $S_A[x]$ and $S_B[x]$ over interval t_B. The difference signal for x is defined as $D[x] = |S_A[x] - S_B[x]|$. A flow is a *heavy* change flow if the difference in its signal exceeds ϕ percentage of the total change over all flows. The total difference is $D = \sum_{x \in [n]} D[x]$. A flow x is defined to be a heavy change iff $D[x] \geq \phi \cdot D$. The task is to identify these heavy change flows. We assume the size of heavy change flows is above some threshold T over the total capacity c. We can show that the heavy change flows are L_1 heavy hitters on interval t_A $(a_1 \cdots a_{n/2})$ and interval t_B $(b_1 \cdots b_{n/2})$,

[3] In this way, we obtain $\log(n)$ streams $D_1, D_2 \ldots D_{\log(n)}$; i.e., for $j = 1 \ldots \log n$, the number of unique items n in D_{j+1}, is expected to be half of D_j.

where $L_1(t_A, t_B) = \sum |a_i - b_i|$. $G\text{-}sum$ here is L_1 norm, which belongs to *Stream-PolyLog*, and $G\text{-}core$ can be solved by universal sketch. The $G\text{-}sum$ outputs the estimated size of the total change D and $G\text{-}core$ outputs the possible heavy change flows. By comparing the outputs from $G\text{-}sum$ and $G\text{-}core$, we can detect and determine the heavy change flows that are above some threshold of all flows.

Entropy Estimation: We define entropy with the expression $H \equiv -\sum_{i=1}^{n} \frac{f_i}{m} \log(\frac{f_i}{m})$ [38] and we define $0 \log 0 = 0$ here. The entropy estimation task is to estimate H for source IP addresses (but could be performed for ports or other features). To compute the entropy, $H = -\sum_{i=1}^{n} \frac{f_i}{m} \log(\frac{f_i}{m}) = \log(m) - \frac{1}{m} \sum_i f_i \log(f_i)$. As m can be easily obtained,[4] the difficulty lies in calculating $\sum_i f_i \log(f_i)$. Here the function $g(x) = x \log(x)$ is bounded by $g(x) = x^2$ and thus its $G\text{-}sum$ is in *Stream-PolyLog* and H can be estimated by universal sketch.

Global Iceberg Detection: Consider a system or network that consists of N distributed nodes (e.g., switches). The data set S_j at node j contains a stream of tuples $< item_{id}$, c$>$ where $item_{id}$ is an item identity from a set $U = \{\mu_1 \ldots \mu_n\}$ and c is an incremental count. For example, an item can be a packet or an origin-destination (OD) flow. We define $fr_i = \sum_j \sum_{<\mu_i, c> \in S_j} c$, the frequency of the item μ_i when aggregated across all the nodes. We want to detect the presence of items whose total frequency across all the nodes adds up to exceed a given threshold T. In other words, we would like to find out if there exists an element $\mu_i \in U$ such that $fr_i \geq T$. (In §5, we will explain a solution to gain a network-wide universal sketch. Here, we assume here that we maintain an abstract universal sketch across all nodes by correctly combining all distributed sketches.) Consider a function $g(x) = x$ such that the corresponding $G\text{-}core$ outputs a list of global heavy hitters with$(1 \pm \epsilon)-$approximation of their frequencies. For this case, since g-heavy hitters are L_1 heavy hitters, we have an algorithm that provides $G\text{-}core$.

5 Network-wide UnivMon

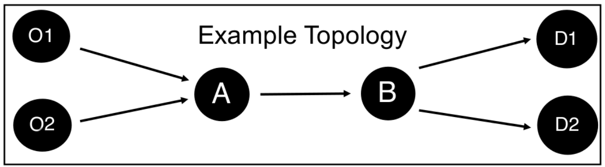
In a network-wide context, we have flows from several origin-destination (OD) pairs, and applications may want network-wide estimates over multiple packet header combinations of interest. For instance, some applications may want per-source IP estimates, while others may want characteristics in terms of the entire IP-5-tuple. We refer to these different packet header features and feature-combinations as *dimensions*.

In this section, we focus on this network-wide monitoring problem of measuring multiple dimensions of interest traversing multiple OD-pairs. Our goal is to provide equivalent coverage and fidelity to a "one big switch abstraction", providing the same level of monitoring precision at the network level as at the switch level. We focus mostly for the case where each OD-pair has a single network route and describe possible extensions to handle multi-pathing.

OD Pair	Label
O1,D1	P11
O1,D2	P12
O2,D1	P21
O2,D2	P21
All	P*

Sketch	Label
5-tuple	5t
Source IP	src
All	S*

Key
N – Number of Nodes
D – Number of Dimensions
Load – Sketches at a Node



Sketches Maintained at Each Node

Algorithms	O1	O2	A	B	D1	D2	Max Load	Total Sketches
RM	P*/S*	P*/S*	P*/S*	P*/S*	P*/S*	P*/S*	D	N*D
IM	P*/S*	P*/S*	0	0	0	0	D	~N*D
GDC	P11/src P12/src	P21/src P22/src	P*/5t	0	0	0	1	~D
Q&S	0	0	P*/src	P*/5t	0	0	1	D

Figure 3: Example topology to explain the one-big-switch notion and to compare candidate network-wide solutions

5.1 Problem Scope

We begin by scoping the types of network-wide estimation tasks we can support and formalize the one-big-switch abstraction that we want to provide in UnivMon. To illustrate this, we use the example in Figure 3 where we want to measure statistics over two dimensions of interest: 5-tuple and source-IP.

In this example, we have four OD-pairs; suppose the set of traffic flows on each of these is denoted by P_{11}, P_{12}, P_{21}, and P_{22}. We can divide the traffic in the network into four partitions, one per OD-pair. Now, imagine we abstract away the topology and consider the union of the traffic across these partitions flowing through one logical node representing the entire network; i.e., computing some estimation function $F(P_{11} \uplus P_{12} \uplus P_{21} \uplus P_{22})$, where $\uplus$ denotes the disjoint set union operation.

For this work, we restrict our discussion to network-wide functions where we can independently compute the F estimates on each OD-pair substream and add them up. In other words, we restrict our problem scope to estimation functions Fs such that:

$$F(P_{11} \uplus P_{12} \uplus P_{21} \uplus P_{22}) = F(P_{11}) + F(P_{12}) + F(P_{21}) + F(P_{22})$$

Note that this still captures a broad class of network-wide tasks such as those mentioned in section 4.3. One such example measurement is finding heavy hitters for destination IP addresses. An important characteristic of the UnivMon approach is that in the network-wide setting the output of sketches in the data plane can then be added together in the control plane to give the same results as if all of the packets passed through one switch. The combination of the separate sketches is a property of the universal sketch primitive used in the data plane and is independent of the final statistic monitored in the control plane, allowing the combination to work for all measurements supported by UnivMon. We do however acknowledge that some tasks fall outside the scope

[4] e.g., a single counter or estimated as a $G\text{-}sum$.

of this partition model; an example statistic that is out of scope would be measuring the statistical independence of source and destination IP address pairs (i.e. if a source IP is likely to appear with a given destination IP, or not), as this introduces cross-OD-pair dependencies. We leave extensions to support more general network-wide functions for future work (see §8).

The challenge here is to achieve *correctness* and *efficiency* (e.g., switch memory, controller overhead) while also *balancing the load* across the data plane elements. Informally, we seek to minimize the total number of sketches instantiated in the network and the maximum number of sketches that any single node needs to maintain.

5.2 Strawman Solutions and Limitations

Next, we discuss strawman strategies and argue why these fail to meet one or more of our goals w.r.t. correctness, efficiency, and load balancing. We observe that we can combine the underlying sketch primitives at different switches as long as we use the same random seeds for our sketches, as the counters are additive at each level of the UnivMon sketch. With this, we only need to add the guarantee that we count each packet once to assure correctness. In terms of resource usage, our goal is to minimize the number of sketches used.

Redundant Monitoring (RM): Suppose for each of k dimensions of interest, we maintain a sketch on every node, with each node independently processing traffic for the OD-pairs whose paths it lies on. Now, we have the issue of combining sketches to get an accurate network-wide estimate. In particular, adding all of the counters from the sketches would be incorrect, as packets would be counted multiple times. In the example topology, to correctly count packets we would need to either only use the sketches at A or B, or, conversely, combine the sketches for source IP at $O1$ and $O2$ or $D1$ and $D2$. In terms of efficiency, this RM strategy maintains a sketch for all k dimensions at each node and thus we maintain a total of kN sketches across N nodes. Our goal, is to maintain s total sketches, where $s << kN$.

Ingress Monitoring (IM): An improvement over the RM method is to have only *ingress nodes* maintaining every sketch. Thus, for each OD pair, all sketch information is maintained in a single node. By not having duplicate sketches per OD pair, we will not double count and therefore can combine sketches together. This gives us the correctness guarantee missing in RM. In Figure 3, IM would maintain sketches at $O1$ and $O2$. However, for N_i ingress nodes, we would run kN_i sketches, and if $N_i \approx N$ we spend a similar amount of resources to RM, which is still high. Additionally, these sketches woul be would all be present on a small number of nodes, where other nodes with available compute resources would not run any sketches.

Greedy Divide and Conquer (GDC): To overcome the concentration of sketches in IM above, one potential solution is to evenly divide sketch processing duties across the path. Specifically, each node has a priority list of sketches, and tags packets with the current sketches that are already maintained for this flow so that downstream nodes know which remaining sketches to run. For instance, in Figure 3, GDC would maintain the source IP sketch at $O1$ and $O2$, and the 5-tuple sketch at A. This method is correct, as each sketch for each OD pair is maintained once. However, it is difficult to properly balance resources as nodes at the intersection of multiple paths could be burdened with higher load.

Reactive Query and Sketch (QS): An alternative approach is to use the controller to ensure better sketch assignment. For instance, whenever a new flow is detected at a node, we query the controller to optimally assign sketches. In Figure 3, the controller would optimally put the source IP sketch at A and the 5-tuple sketch at B (or vice versa). With this method, we can be assured of correctness. However, the reactive nature means that QS generates many requests to the controller.

5.3 Our Approach

Next, we present our solution, which uses the UnivMon controller to coordinate switches to guarantee correctness and efficiency but without incurring the reactive query load of the QS strategy described above.

Periodically, the UnivMon controller gives each switch a *sketching manifest*. For each switch A and for each OD-pair's route that A lies on, the manifest specifies the dimensions for which A needs to maintain a sketch. When a packet arrives at a node, the node uses the manifest to determine the set of sketching actions to apply. When the controller needs to compute a network-wide estimate, we pull sketches from all nodes and for each dimension, combine the sketches across the network for that dimension. This method minimizes communication to the control plane while still making use of the controller's ability to optimize resource use.

The controller solves a simple constrained optimization problem that we discuss below. Note that maintaining two sketches uses much more memory than adding twice as many elements to one sketch. Thus, a key part of this optimization is to ensure that we try to reuse the same sketch for a given dimension across multiple OD pairs. In Figure 3, we would first assign A the source IP sketch, then B the 5-tuple sketch for the OD pair $(O1, D1)$. When choosing where to place the sketches for the OD pair $(O2, D2)$, the algorithm matches the manifests such that the manifest for $(O2, D2)$ uses the source IP sketch already at A and the 5-tuple sketch already at B.

We formulate the controller's decision to place sketches as an integer linear program (ILP) shown in Figure 4. Let s_{jk} be a binary decision variable denoting if the switch j is maintaining a sketch for dimension j. The goal of the optimization is to ensure that every OD-pair is suitably "covered" and that the load across the switches is balanced. Let r_k be the amount of memory for a sketch for dimension k and let R denote maximum amount of memory available on a single node. Note that the amount of memory for a sketch can be chosen in advance based on the accuracy required. As a simple starting point, we focus primarily on the memory resource consumption assuming that all UnivMon operations can be done at line-rate; we can extend this formulation to incorporate processing load as well.

Minimize: $N \times \mathit{Maxload} + \mathit{Sumload}$, subject to

$$\forall i, k : \sum_{j \in p_i} s_{jk} \geq 1 \qquad (1)$$

$$\forall j : \mathit{Load}_j = \sum_k r_k \times s_{jk} \qquad (2)$$

$$\forall j : \sum s_{jk} r_k \leq R \qquad (3)$$

$$\forall j : \mathit{Maxload} \geq \mathit{Load}_j \qquad (4)$$

$$\forall j : \mathit{Sumload} = \sum_j \mathit{Load}_j \qquad (5)$$

Figure 4: ILP to compute sketching manifests

Eq (1) captures our coverage constraint that we maintain each sketch once for each OD-pair.[5] We model the per-node load in Eq (2) and ensure that it respects the router capacity in Eq (3). Our objective function balances two components: the maximum load that any one node faces and the total number of sketches maintained.[6]

5.4 Extension to Multi-path

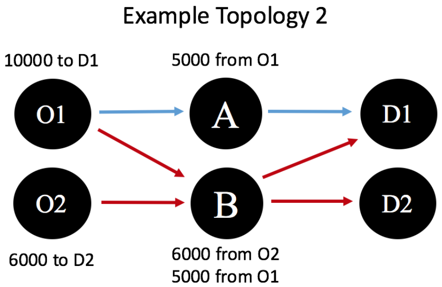


Figure 5: Example topology to showcase difficulty of multi-path

Adapting the above technique to multi-path requires some modification, but is feasible. For simplicity, we still assume that packets are routed deterministically (e.g., by prefix rules), but may have multiple routes. We defer settings that depend on randomized or non-deterministic routing for future work.

Even in this deterministic setting, there are two potential problems. First, ensuring packets are only counted once is important to avoid false positives, as in the single path case. Second, if the packets with a heavy feature (e.g., the destination address is heavy) are divided over many routes, it can increase the difficulty of accurately finding heavy hitters, removing false positives and preventing false negatives.

The first issue, guaranteeing packets are counted only once, is solvable by the ILP presented above. For each path used by an OD pair, we create a unique sub-pair which we treat as an independent OD pair. This is shown in Figure 5 by the red O1-D1 path and the blue O1-D1 path. By computing the ILP with multiple paths per OD pair as needed, sketches are distributed across nodes, and single counting is guaranteed. This method works best when the total number of paths per OD pair is constant relative to the total number of nodes, and larger numbers of paths will cause the sketches to concentrate on the source or destination nodes, possibly requiring additional solutions.

The second issue occurs when multi-path routing causes the frequency of an item to be split between too many sketches. In the single-path setting, if an OD pair has a globally heavy feature, then it will be equally heavy or heavier in the sketch where it is processed. However in the multi-path case, it is possible for some OD pairs to have more paths than others, and thus it becomes possible for items that are less frequent but have fewer routes to be incorrectly reported heavy, and in turn fail to report true heavy elements in the control plane. This problem is shown in Figure 5. In this case, we have 10,000 packets from node O1 to D1 split across two paths, and 6,000 packets from O2 to D2. For simplicity, assume we are only looking for the "heaviest" source IP, the source IP with the highest frequency, and that the nodes have a single IP address, (i.e. Packets go from IP_{O_1} to IP_{D_1} and IP_{O_1} IP_{D_2}). For this metric, the sketch at A will report IP_{O_1} as a heavy source address with count 5,000, and B will report IP_{O_2} as a heavy source address with count 6,000. At the data plane these values are compared again, and the algorithm would return IP_{O_2}, a false positive, and miss IP_{O_1}, a false negative. To solve this issue, instead of sending the heavy hitter report from individual sketches as described in Algorithm 1, the counters from each sketch must be sent directly to the control plane to be added, reconstructing the entire sketch and allowing the correct heavy hitters to be identified. In our example, the counters for the O1 at A and B would be added, and IP_{O_1}would be correctly identified as the heavy hitter. This approach is already used in the P4 implementation discussed below, but is not a requirement of UnivMon in general. We note that when using the method described below in Section 6.2, identifying the true IP address of the heavy item is harder in the multi-path setting, but is solved by increasing γ relative to the maximum number of sketches per true OD pair, which is naturally limited by the ILP. With these modifications, the heavy hitters are correctly found from the combined sketch, and the one big switch abstraction are maintained in a multi-path setting.

6 UnivMon Implementation

In this section, we discuss our data plane implementation in P4 [7, 12]. We begin by giving an overview of key design tradeoffs we considered. Then, we describe how we map UnivMon into corresponding P4 constructs.

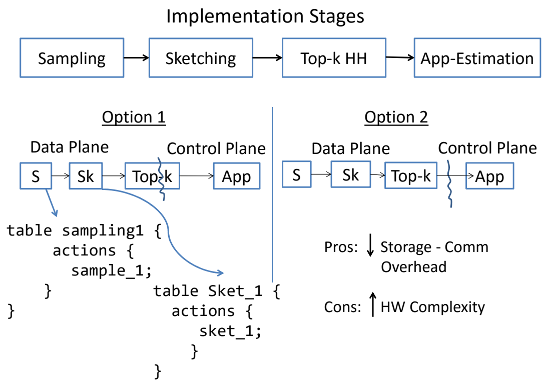
6.1 Implementation overview

At a high level, realizing the UnivMon design described in the previous sections entails four key stages:

1. A **sampling** stage which decides whether an incoming packet will be added to a specific substream.

[5] Our coverage constraint allows multiple sketches of the same kind to be placed in the same path. This is because in some topologies, it may not be feasible to have an equality constraint. In this case, the controller post-processes the solution and removes duplicates before assigning sketches for a given OD pair.

[6] The N term for $\mathit{MaxLoad}$ helps to normalize the values.



```
table sampling1 {
      actions {
        sample_1;
    }
}
```

```
table Sket_1 {
  actions {
    sket_1;
  }
}
```

Figure 6: An illustration of UnivMon's stages along with the two main implementation options.

2. A **sketching** stage which calculates sketch counters from input substreams and populates the respective sketch counter arrays.

3. A **top-k** computation stage which identifies (approximately) the k heaviest elements of the input stream.

4. An **estimation** stage which collects the heavy element frequencies and calculates the desired metrics.

Let us now map these stages to our data and control plane modules from Figure 1. Our delayed binding principle implies that the estimation stage maps to the UnivMon control plane. Since the sampling and sketching are processing packets, they naturally belong in the data plane to avoid control plane overhead.

One remaining question is whether the top-k computation stage is in the data or control plane (Figure 6). Placing the top-k stage in the data plane has two advantages. First, the communication cost between the data and control plane will be low, as only the top-k rather than raw counters need to be transferred. Second, the data plane only needs to keep track of the flowkeys (e.g., source IP) of the k heaviest elements at any given point in time, and thus not incur high memory costs. However, one stumbling block is that realizing this stage requires (i) sorting counter values and (ii) storing information about the heavy elements in some form of a priority queue. Unfortunately, these primitives may be hard to implement in hardware and are not supported in P4 yet. Thus, we make a pragmatic choice to split the top-k stage between the control and the data planes. We identify the top-k heavy flowkeys in the dataplane and then we use the raw data counters to calculate their frequencies in the control plane. The consequence is that we incur higher communication overhead to report the raw counter data structure, but the number of flowkeys stored in the data plane remains low.

UnivMon's raw counters and flowkeys are stored on the target's on-chip memory (TCAM and SRAM). We argue that in practice the storage overhead of UnivMon is manageable even for hardware targets with limited SRAM [4, 8, 47]. We show that for the largest traces that we evaluate and without losing accuracy, the total size of the raw counters can be less than 600 KB whereas the cost of storing flowkeys (assuming k is ≤ 20) is only a few KBs per measurement epoch. Thus, this decision to split the top-k between the two planes computation is practical and simplifies the data plane requirements.

6.2 Mapping UnivMon data plane to P4

Based on the above discussion, the UnivMon data plane implements sampling, sketching, and "heavy" flowkey storage in P4. In a P4 program, packet processing is implemented through Match+Action tables, and the control flow of the program dictates the order in which these tables are applied to incoming packets. Given the sketching manifests from the control plane, we generate a control program that defines the different pipelines that a packet needs to be assigned to. These pipelines are specific to the dimension(s) (i.e., source IP, 5-tuple) for which the switch needs to maintain a universal sketch. We begin by explaining how we implemented these functions and then describe a sample control flow.

Sampling: P4 enables programmable calculations on specific header fields using user-defined functions. We use this to sample incoming packets, with a configurable flowkey that can be any subset of the 5-tuple (srcIP, dstIP, srcPort, dstPort, protocol). We define l pairwise-independent hash functions, where l is the number of levels from §4. These functions take as input the flowkey and output a binary value. We store this output bit as packet metadata. A packet is sampled at level i if the outputs of the hash functions of all levels $\leq i$ is equal to 1. We implement sampling for each level as a table that matches all packets and whose action is to apply the sampling hash function of that level. The hash metadata in the packets are used in conditional statements in the control flow to append the packet to the first i substreams. Packets that are not sampled are not subject to further UnivMon processing.[7]

Sketching: The sketching stage is responsible for maintaining counters for each one of the l substreams. From these sketch counters, we can estimate the L2-HH for each stage and then the overall top-k heavy hitters and their counts. While UnivMon does not impose any constraints on the L2-HH algorithm to be used, in our P4 implementation we use Count Sketch [24]. The sketching computation for each level is implemented as a table that matches every packet belonging to that level's substream and its actions update the counters, stored in the sketch counter arrays. Similar to the sampling stage, we leverage user-defined hash functions that take as input the same flowkey as in the sampling stage. We use their output to retrieve the indexes of the sketch register arrays cells that correspond to a particular packet and update their value as dictated by the Count Sketch algorithm.

P4 provides a *register* abstraction which offers a form of stateful memory that can store user-defined data and that can be arranged into one dimensional arrays of user-defined length. Register cells can be read or written by P4 action statements and are also accessible through the control plane API. Given that our goal is to store sketch counter values

[7] There may be other routing/ACL actions to be applied to the packet but this is outside our scope.

which do not represent byte or packet counts, we use register arrays to store and update sketch counters. The size of the array and the bitlength of each array cell are user-defined and can be varied based on the required memory-accuracy tradeoff as well as on the available on-chip memory of the hardware target. Each sketch is an array of t rows and w columns. We instantiate register arrays of length $t * w$, and the bitlength of each cell is based on the maximum expected value of a counter.

The one remaining issue is storing flowkeys corresponding to the "heavy" elements since these will be needed by the estimation stage running in the control plane. One option is to use a priority queue to maintain the top k heavy hitters online, as it is probably the most efficient and accurate choice to maintain heavy flowkeys. However, this can incur more than constant update time for each element, which makes it difficult to implement on hardware switches. To address the issue, we use an alternative approach which is to maintain a fixed sized table of heavy keys and use constant time updates for each operation. It is practical and acceptable when the size of the table is small (e.g., 10-50) and the actual number of heavy flows doesn't greatly exceed this size. The lookup/update operations could be very fast (in a single clock cycle) when leveraging some special types of memory (e.g., TCAM) on hardware switches.

Another scheme we use is as follows, and we leave improved sketches for finding heavy flowkeys as future work. For γ-threshold heavy hitters, there are at most $1/\gamma$ of them. While packets are being processed, we maintain an up-to-date L_2 value (of the frequency vector), specifically $L_2 = (L_2^2 + (c_i + 1)^2 - (c_i)^2)^{1/2}$, where c_i is each flow's current count and we create $\log(1/\gamma)$ buckets of size k. In the online stage, when updating the counters in L2-HH, c_i is obtained by reading current sketch counters.

We then maintain buckets marked with $L_2/2, L_2/4, \ldots, \gamma L_2$. For each element that arrives, if its counter is greater than $L_2/2$, insert it into the $L_2/2$ bucket using a simple hash; otherwise, if its counter is greater than $L_2/4$, insert it into the $L_2/4$ bucket, and so forth. When the value of L_2 doubles itself, we delete the last γL_2 bucket and we add a new $L_2/2$ bucket. This scheme ensures that $O(k \log(1/\gamma))$ flowkeys are stored, and at the end of the stream we can return most top k items heavier than γL_2.

P4 Control Flow: As a simple starting point, we use a sequential control flow to avoid cloning every incoming packet l (i.e., number of levels) times. This means that every packet is processed by a sketching, a storage and a sampling table sequentially until the first level where it doesn't get sampled. More specifically, after a packet passes the parsing stage during which P4 extracts its header fields, it is first processed by the sketching table of level_0. The "heavy" keys for that stage are updated and then it is processed by the sampling table of level_1. If the packet gets sampled at level_1, it is sketched at this level, the "heavy" keys are updated and the procedure continues until the packet reaches the last level or until it is not sampled.

6.3 Control plane

We implement the UnivMon control plane as a set of custom C++ modules and libraries. We implement modules for (1) Assigning sketching responsibilities to the network elements, and (2) implementing the top-k and estimation stages. The P4 framework allows us to define the API for control-data plane communication. We currently use a simple RPC protocol that allows us to import sketching manifests and to query the contents of data plane register arrays

After the heavy flowkeys and their respective counters have been collected, the frequencies of the k-most frequent elements in the stream are extracted. The heavy elements along with the statistical function of the metric to be estimated are then fed to the recursive algorithm of UnivMon's estimation stage.

7 Evaluation

We divide our evaluation into two parts. First, we focus on a single router setup and compare UnivMon vs. custom sketches via OpenSketch [47]. Second, we demonstrate the benefits of our network-wide coordination mechanisms.

7.1 Evaluation setup

We begin by describing our trace-driven evaluation setup.

Applications and error metrics: We have currently implemented translation libraries for five monitoring tasks: Heavy Hitter detection (HH), DDoS detection (DDoS), Change Detection (Change), Entropy Estimation (Entropy), and Global Iceberg Detection (Iceberg). For brevity, we only show results for metrics computed over one feature, namely the source IP address; our results are qualitatively similar for other dimensions too.

For Heavy Hitters and Global Iceberg detection, we set a threshold $T = 0.05\%$ of the link capacity and identify all large flows that consume more traffic than that threshold. We obtain the average *relative error* on the counts of each identified large flow; i.e., $\frac{|True - Estimate|}{True}$. For Change Detection, whose frequency has changed more than a threshold ϕ of the total change over all flows across two monitoring windows. We chose this threshold to be 0.05% and calculate the average relative error similar to HH. For Entropy Estimation and DDoS, we evaluate the relative error on estimated entropy value and the number of distinct source IPs.

Configuration: We normalize UnivMon's memory usage with the custom sketches by varying three key parameters: number of rows t and number of columns w in Count-Sketch tables, and the number of levels l in the universal sketch. In total UnivMon uses $t \times w \times l$ counters. In OpenSketch, we configure the memory usage in a similar way by varying number of rows t and counters per row w in all the sketches they use. When comparing the memory usage with OpenSketch, we calculate the total number of sketch counters assuming that each integer counter occupies 4 bytes. Both UnivMon and OpenSketch use randomized algorithms; we run the experiment 10 times with random hash seeds and report the *median* cross these runs.

Trace	Loc	Date and Time
1. CAIDA'15	Equinix-Chicago	2015/02/19
2. CAIDA'15	Equinix-Chicago	2015/05/21
3. CAIDA'15	Equinix-Chicago	2015/09/17
4. CAIDA'15	Equinix-Chicago	2015/12/17
5. CAIDA'14	Equinix-Sanjose	2014/06/19

Table 1: CAIDA traces in the evaluation

Traces: For this evaluation, we use five different one-hour backbone traces (Table 1) collected at backbone links of a Tier1 ISP between (i) Chicago, IL and Seattle, WA in 2015 and (ii) between San Jose and Los Angeles in 2014 [1, 2]. We split the traces into different representative time intervals (5s, 30s, 1min, 5min). For example, each one hour trace contains 720 5s-epoch data points and we report *min*, *25%*, *median*, *75%*, and *max* on whisker bars. By default, we report results for a 5-second trace. Each 5s packet-trace contains 155k to 286k packets with ∼55k distinct source IP addresses and ∼40k distinct destination IP addresses. The link speed of these traces is 10 Gbps.

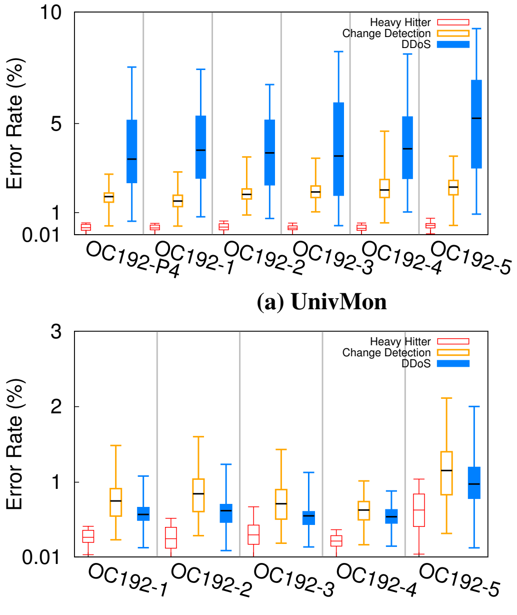
Experiment Setup: For our P4 implementation prototype, we used the P4 behavioral simulator, which is essentially a P4-enabled software switch [6]. To validate the correctness of our P4 implementation, we compare it against a software implementation of the data plane and control plane algorithms, written in C++. We evaluate P4 prototype on Trace 1 and run software implementation in parallel on Trace 1- 5. The results between the two implementations are consistent as the relative error between the results of the two implementations does not exceed 0.3%. To evaluate OpenSketch, we use its simulator written in C++ [5].

7.2 Single Router Evaluation

Comparison under fixed memory setting: First, we compare UnivMon and OpenSketch on the applications that OpenSketch supports: HH, Change, and DDoS. In Figures 7a and 7b, we assign 600KB memory and use all traces in order to estimate the error when running UnivMon and OpenSketch. We find that the absolute error is very small for both approaches. We observe that OpenSketch provides slightly better results for all three metrics. However we note that UnivMon uses 600KB memory to run three tasks concurrently while OpenSketch is given 600KB to run each task. Figure 7a and 7b confirm that this observation holds on multiple traces; the error gap between UnivMon and OpenSketch is ≤3.6%.

Accuracy vs. Memory: The previous result considered a fixed memory value. Next, we study the sensitivity of the error to the memory available. Figure 8a and 8b shows that the error is already quite small for all the HH and DDoS applications and that the gap is almost negligible with slightly increased memory $\geq$ 1MB.

Figure 8c shows the results for the Change Detection task. For this task, the original OpenSketch paper uses a streaming algorithm based on reversible k-ary sketches [44]. We implement an extension to OpenSketch using a similar idea as UnivMon.[8] Our evaluation results show that our extension offers better accuracy vs. memory tradeoff than OpenSketch's original method [44]. For completeness, we also report the memory usage of OpenSketch's original design (using the k-ary sketch). From Figure 8c, we see UnivMon provides comparable accuracy even though UnivMon has a much smaller sketch table on each level of its hierarchical structure. This is because the "diff" across sketches are well preserved in UnivMon's structure.



(a) UnivMon

(b) OpenSketch

Figure 7: Error rates of HH, Change and DDoS for UnivMon and OpenSketch

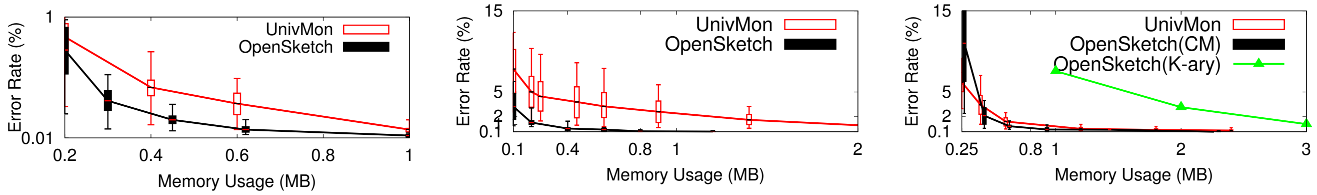
Fixed Target Errors: Next, we evaluate the memory needed to achieve the same error rates (≤1%). In Figures 9 and 10 as we vary the monitoring window, we can see that only small amount of memory increase is required for both UnivMon and OpenSketch to achieve 1% error rates. In fact, we find that UnivMon does not require more memory to maintain a stable error rate for increased number of flows in the traffic. This is largely because sketch-based approaches usually just take logarithmic memory increase in terms of input size to maintain similar error guarantees. Furthermore, the nature of traffic distribution also helps as there are only a few very heavy flows and the entire distribution is quite 'flat'.

Other metrics: We also considered metrics not in the OpenSketch library in Figure 11 to confirm that UnivMon is able to calculate a low-error estimate. Specifically, we consider the entropy of the distribution and the *second frequency moment* $F_2 = f_1^2 + f_2^2 \cdots + f_m^2$ for m distinct elements.[9] Again, we find that with reasonable amounts of memory ($\geq$ 500KB) the error of UnivMon is very low.

Impact of Application Portfolio: Next, we explore how UnivMon and OpenSketch handle a growing portfolio of

[8] We maintain two recent Count-Min sketches using the same hash functions; combine two sketches by one sketch "subtracts" the other; and use reversible sketch to trace back the keys.

[9] This is a measure of the "skewness" and is useful to calculate repeated rate or Gini index of homogeneity.



(a) HH **(b) DDoS** **(c) Change**

Figure 8: Error vs. Memory for HH, DDoS, Change

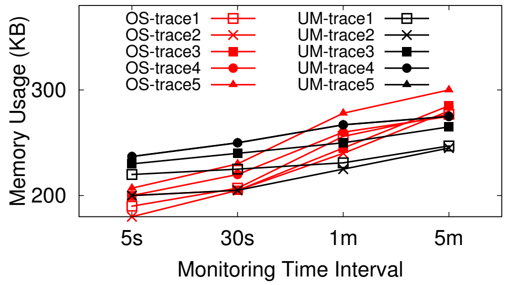


Figure 9: HH: average memory usage to achieve a 1% error rate for different time intervals

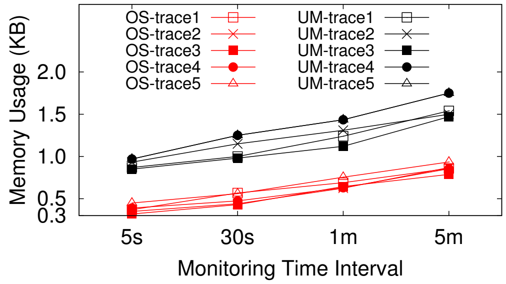


Figure 10: Change: average memory usage to achieve a 1% error rate for different time intervals

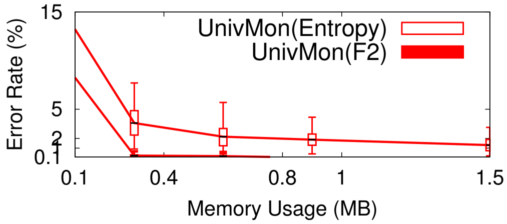


Figure 11: Error rates of Entropy and F2 estimation

monitoring tasks with a fixed memory. We set the switch memory to 600KB for both UnivMon and OpenSketch and run three different application sets: AppSet1={HH}, AppSet2 ={HH,DDoS}, and AppSet3={HH,DDoS,Change}. We assume that OpenSketch divides the memory uniformly across the constituent applications; i.e., in AppSet1 600KB is devoted to HH, but in AppSet2 and Appset3, HH only gets 300KB and 200KB respectively. Figure 12 shows the "error gap" between UnivMon and OpenSketch (UnivMon − OpenSketch); i.e., positive values imply UnivMon is worse and vice versa. As expected, we find that when running concurrent tasks, the error gap decreases as each task gets less memory in OpenSketch. That is, with more concurrent and supported tasks, UnivMon can still provide guaranteed results on each of the applications.

Choice of Data Structures: UnivMon uses a a sketching algorithm that identifies L_2 heavy hitters as a building block. Two natural questions arise: (1) How do different heavy hitter algorithms compare and (2) Can we use other popular heavy hitter identifiers, such as Count-Min sketch? We implemented and tested the Pick-and-Drop algorithm [20] and Count-Min sketch [26] as building blocks for UnivMon. Figure 13 shows that Pick-and-Drop and CM sketch lose the generality of UnivMon as they can provide accurate results only for HH and Change tasks. This is because, intuitively, only $L_p(p = 1 \text{ or } p \geq 3)$ heavy hitters are identified. The technical analysis of universal sketch shows that only L_2 heavy hitters contribute significantly to the $G\text{-}Sum$ when $G\text{-}Sum$ is upper bounded by some L_2 norm. As discussed in Section 4.3, the $G\text{-}Sum$ functions corresponding to HH and Change are actually L_1 norms. Therefore, the estimated L_1 heavy hitters output by Count-Min or Pick-and-Drop work well for HH and Change tasks, but not Entropy or DDoS. When combining heavy hitter counters in the recursive step of calculation, we will simply miss too many significant heavy elements for all tasks.

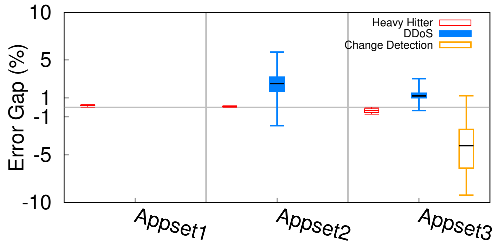


Figure 12: The impact of a growing portfolio of monitoring applications on the relative performance

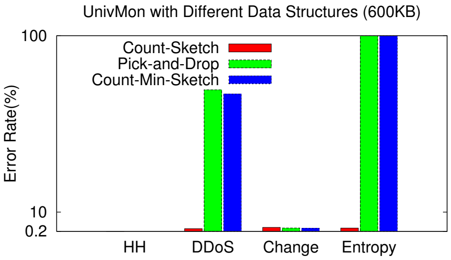
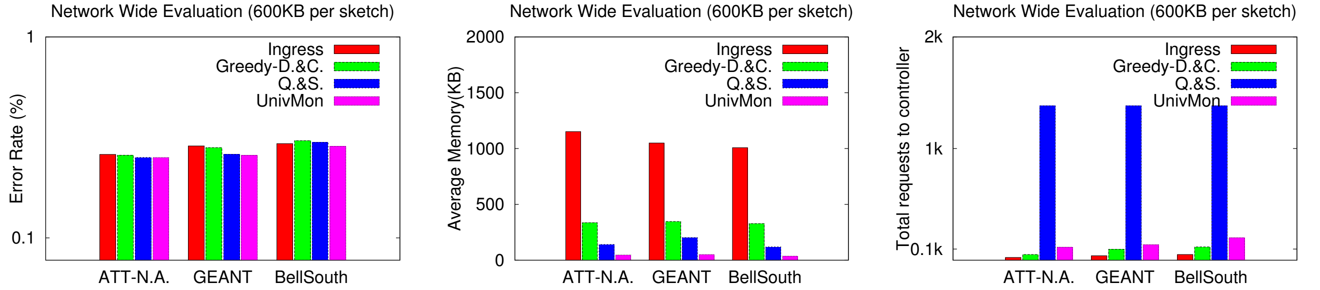


Figure 13: Analyzing different HH data structures

Processing Overhead: One concern might be the computational cost of the UnivMon vs. custom sketch primitives. We used the Intel Performance Counter Monitor [29] to evaluate compute overhead (e.g., Total cycles on CPU) on UnivMon and OpenSketch's software simulation libraries. For any given task, our software implementation was only 15% more expensive than OpenSketch. When we look at all three applications together, however, the UnivMon takes only half the compute cycles as used by OpenSketch in total. While



(a) Error rates of global iceberg detection **(b) Average memory consumption** **(c) Total number of requests to controller**

Figure 14: Network-wide evaluation on major ISP backbone topologies

Topology	OD Pairs	Dim.	Time (s)	Total Sketches
Geant2012	1560	4	0.09	68
Bellsouth	2550	4	0.10	60
Dial Telecom	18906	4	2.8	252
Geant2012	1560	8	0.22	136
Bellsouth	2550	8	0.28	120
Dial Telecom	18906	8	12.6	504

Table 2: Time to compute sketching manifests using ILP

we acknowledge that we cannot directly translate into actual hardware processing overheads, this suggests that UnivMon's compute footprint will be comparable and possibly better.

7.3 Network-wide Evaluation

For the network-wide evaluation, we consider different topologies from the Topology Zoo dataset [35]. As a specific network-wide task, we consider the problem of estimating source IP and destination IP "icebergs". We report the average relative errors across these two tasks.

Benefits of Coordination: Figure 14a, Figure 14b, and Figure 14c present the error, average memory consumption, and total controller requests of four solutions: Ingress Monitoring(IM), Greedy Divide and Conquer(GDC), Query and Sketch(QS), and our approach(UnivMon). We pick three representative topologies: AT&T North America, Geant, and Bell South. We see that UnivMon provides an even distribution of resources on each node while providing results with high accuracy. Furthermore, the control overhead is several orders of magnitude smaller than purely reactive approaches.

ILP solving time: One potential concern is the time to solve the ILP. Table 2 shows the time to compute the ILP solution on a Macbook Pro with a 2.5 GHz Intel Core i7 processor using `glpsol` allowing at most k sketches per switch, where k is the number of dimensions maintained. We see that the ILP computation takes at most a few seconds which suggest that updates can be pushed to switches with reasonable responsiveness as the topology or routing policy changes.

7.4 Summary of Main Findings

Our analysis of UnivMon's performance shows that:

1. For a single router with 600KB of memory, we observe comparable median error rate values between UnivMon and OpenSketch, with a relative error gap $\leq 3.6\%$. The relative error decreases significantly with a growing application portfolio.
2. When comparing sensitivity to error and available memory, we observe that UnivMon provides comparable accuracy with OpenSketch with similar, or smaller memory requirements.
3. The network-wide evaluation shows that UnivMon provides an even distribution of resources on each node while providing results with high accuracy.

8 Conclusions and Future Work

In contrast to the status quo in flow monitoring that can offer generality or fidelity but not both simultaneously, UnivMon offers a dramatically different design point by leveraging recent theoretical advances in universal streaming. By delaying the binding of data plane primitives to specific (and unforeseen) monitoring UnivMon provides a truly software-defined monitoring approach that can fundamentally change network monitoring. We believe that this "minimality" of the UnivMon design will naturally motivate hardware vendors to invest time and resources to develop optimized hardware implementations, in the same way that a minimal data plane was key to get vendor buy-in for SDN [40].

Our work in this paper takes UnivMon beyond just a theoretical curiosity and demonstrates a viable path toward a switch implementation and a network-wide monitoring abstraction. We also demonstrate that UnivMon is already very competitive w.r.t. custom solutions and that the trajectory (i.e., as the number of measurement tasks grows) is clearly biased in favor of UnivMon vs. custom solutions.

UnivMon already represents a substantial improvement over the status quo, That said, we identify several avenues for future work to further push the envelope. First, in terms of the data plane, while the feasibility of mapping UnivMon to P4 is promising and suggests a natural hardware mapping, we would like to further demonstrate an actual hardware implementation on both P4-like and other flow processing platforms. Second, in terms of the one-big-switch abstraction, we need to extend our coordination and sketching primitives to capture other classes of network-wide tasks that entail cross-OD-pair dependencies. Third, while the ILP is quite scalable for many reasonable sized topologies, we may need other approximation algorithms (e.g., via randomized rounding) to handle even larger topologies. Fourth, in terms of the various dimensions of interest to track, we currently maintain independent sketches; a natural question if we can avoid explicitly creating a sketch per dimension. Finally, while being application agnostic gives tremendous power, it might be useful to consider additional tailoring where operators may want the ability to adjust the granularity of the measurement to dynamically focus on sub-regions of interest [48].

Acknowledgments: We thank our shepherd Mohammad Alizadeh and the SIGCOMM reviewers for their comments that helped improve the paper. This work was supported in part by NSF awards CCF-1536002, IIS-1447639, Raytheon BBN Technologies, and by a Google Faculty Award.

9 References

[1] Caida internet traces 2014 sanjose. http://goo.gl/uP5aqG.

[2] Caida internet traces 2015 chicago. http://goo.gl/xgIUmF.

[3] Intel flexpipe. http://goo.gl/H5qPP2.

[4] Netfpga technical specifications. http://netfpga.org/1G_specs.html.

[5] Opensketch simulation library. https://goo.gl/kyQ80q.

[6] P4 behavioral simulator. https://github.com/p4lang/p4factory.

[7] P4 specification. http://goo.gl/5ttjpA.

[8] Why big data needs big buffer switches. https://goo.gl/ejWUIq.

[9] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc.*, STOC, 1996.

[10] N. Bandi, A. Metwally, D. Agrawal, and A. El Abbadi. Fast data stream algorithms using associative memories. In *Proc.*, SIGMOD, 2007.

[11] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine grained traffic engineering for data centers. In *Proc.*, CoNEXT, 2011.

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, July 2014.

[13] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proc.*, SIGCOMM, 2013.

[14] V. Braverman and S. R. Chestnut. Universal Sketches for the Frequency Negative Moments and Other Decreasing Streaming Sums. In *APPROX/RANDOM*, 2015.

[15] V. Braverman, S. R. Chestnut, R. Krauthgamer, and L. F. Yang. Streaming symmetric norms via measure concentration. *CoRR*, 2015.

[16] V. Braverman, S. R. Chestnut, D. P. Woodruff, and L. F. Yang. Streaming space complexity of nearly all functions of one variable on frequency vectors. In *Proc., PODS*, 2016.

[17] V. Braverman, J. Katzman, C. Seidell, and G. Vorsanger. An optimal algorithm for large frequency moments using o(nˆ(1-2/k)) bits. In *APPROX/RANDOM*, 2014.

[18] V. Braverman, Z. Liu, T. Singh, N. V. Vinodchandran, and L. F. Yang. New bounds for the CLIQUE-GAP problem using graph decomposition theory. In *In Proc., MFCS*, 2015.

[19] V. Braverman and R. Ostrovsky. Zero-one frequency laws. In *Proc.*, STOC, 2010.

[20] V. Braverman and R. Ostrovsky. Approximating large frequency moments with pick-and-drop sampling. In *APPROX/ROMDOM*, 2013.

[21] V. Braverman and R. Ostrovsky. Generalizing the layering method of indyk and woodruff: Recursive sketches for frequency-based vectors on streams. In *APPROX/RAMDOM*. 2013.

[22] V. Braverman, R. Ostrovsky, and A. Roytman. Zero-one laws for sliding windows and universal sketches. In *APPROX/RANDOM*, 2015.

[23] A. Chakrabarti, S. Khot, and X. Sun. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In *IEEE CCC*, 2003.

[24] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. 2002.

[25] B. Claise. Cisco systems netflow services export version 9. RFC 3954.

[26] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 2005.

[27] S. Dasgupta and A. Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Struct. Algorithms*, Jan. 2003.

[28] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, June 2002.

[29] R. Dementiev, T. Willhalm, O. Bruggeman, P. Fay, P. Ungerer, A. Ott, P. Lu, J. Harris, P. Kerly, P. Konsor, A. Semin, M. Kanaly, R. Brazones, and R. Shah. Intel performance counter monitor - a better way to measure cpu utilization. http://goo.gl/tQ5gxa.

[30] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *Proc.*, SIGCOMM, 2003.

[31] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proc.*, SIGCOMM, 2002.

[32] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Trans. Netw.*, June 2001.

[33] P. Indyk, A. McGregor, I. Newman, and K. Onak. Open problems in data streams, property testing, and related topics. 2011.

[34] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proc.*, CoNEXT, 2013.

[35] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, october 2011.

[36] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proc., ACM SIGCOMM IMC*, 2003.

[37] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proc.*, SIGMETRICS, 2004.

[38] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang. Data streaming algorithms for estimating entropy of network traffic. In *Proc.*, SIGMETRICS/Performance, 2006.

[39] Z. Liu, G. Vorsanger, V. Braverman, and V. Sekar. Enabling a "risc" approach for software-defined monitoring using universal streaming. In *Proc.*, ACM HotNets, 2015.

[40] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, Mar. 2008.

[41] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *Proc., CoNEXT*, 2015.

[42] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. In *Proc., HotNets*, 2009.

[43] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani. Fast monitoring of traffic subpopulations. In *Proc.*, IMC, 2008.

[44] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proc.*, IMC, 2004.

[45] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proc.*, IMC, 2010.

[46] Y. Xie, V. Sekar, D. A. Maltz, M. K. Reiter, and H. Zhang. Worm origin identification using random moonwalks. In *S&P*. IEEE Computer Society, 2005.

[47] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *Proc.*, NSDI, 2013.

[48] L. Yuan, C.-N. Chuah, and P. Mohapatra. Progme: towards programmable network measurement. *IEEE/ACM TON*, 2011.

[49] Y. Zhang. An adaptive flow counting method for anomaly detection in sdn. In *Proc.*, CoNEXT, 2013.

[50] H. C. Zhao, A. Lall, M. Ogihara, O. Spatscheck, J. Wang, and J. Xu. A data streaming algorithm for estimating entropies of od flows. In *Proc.*, IMC, 2007.

[51] H. C. Zhao, A. Lall, M. Ogihara, and J. J. Xu. Global iceberg detection over distributed data streams. In *Proc., ICDE*, 2010.